UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Simulation von stellenweise verflüssigten Festkörpern

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Dennis Ermtraud

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Gerrit Lochmann, M.Sc. Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Januar 2014

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Bachelorarbeit
Dennis Ermtraud
(Mat. Nr. 210 100 119)

Thema: Simulation von stellenweise verflüssigten Festkörpern

Bei der Schweißausbildung wird sehr viel Metall zu Übungszwecken verbraucht. Um die Ausbildungskosten für die Unternehmen zu mindern, bietet sich eine Computergestützte Simulation an. Diese Simulation sollte neben der realitätsnahen Steuerung auch eine realistische Darstellung bieten, wie sich das Metall verhält, wenn sich eine Wärmequelle nähert. Diese Darstellung ist auch auf weitere Gebiete wie z.B. das Löten übertragbar.

Ziel dieser Arbeit ist die Entwicklung der oben genannte Simulation. Dabei steht neben der computergrafischen Materialdarstellung von Metall die Frage im Mittelpunkt, wie eine stellenweise Verflüssigung eines Festkörpers dynamisch und physikalisch plausibel animiert werden kann. Zudem sollen Konzepte bezüglich der Benutzerinteraktion mit der Simulation untersucht werden.

Die inhaltlichen Schwerpunkte der Arbeit sind:

1. Recherche über Verfahren zur Darstellung der Verflüssigung
2. Konzeption der Anwendung und Benutzerinteraktion
3. Implementierung der Simulation
4. Evaluation der Anwendung
5. Dokumentation der Ergebnisse

Koblenz, 09.07.2013

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Die hier vorliegende Arbeit stellt eine Anwendung zur Simulation von Objekten vor, die zwischen den Aggregatzuständen fest und flüssig wechseln können. Dazu wird ein Temperatursystem verwendet. Dabei liegen die Schwerpunkte auf der Simulation von Fluiden, basierend auf einem Partikelsystem, der Generierung einer Oberfläche aus diesem und der Darstellung von Metall. Zusätzlich soll die Anwendung interaktiv sein und muss die Kriterien der Echtzeitfähigkeit erfüllen. Dazu werden verschiedene Shadertypen eingesetzt, um die Berechnungen auf der GPU zu parallelisieren. Weiterhin werden weitere Einsatzmöglichkeiten, sowie mögliche Verbesserungen der Anwendung aufgezeigt.

Abstract

This work presents an application for simulation objects, which can change their aggregate states between solid and liquid using a temperature system. The focal points are the simulation of fluids with a particle system, the generation of a surface and the visualization of metal. The application should be interactive and match the real time conditions. Different types of Shader are used for the parallelized computations on the GPU. Also more options to use the application and possible improvements are presented.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Fluidsimulationen[1] werden heute in vielen Bereichen für die Darstellung von z.B. Wasser, Rauch und Feuer verwendet. Von komplexen Simulationen für Spezialeffekte in Filmen bis hin zu einfachen, echtzeitfähigen Simulationen in modernen Computerspielen. Durch immer leistungsstärkere Grafikkarten und die Möglichkeit, mithilfe von Compute-Shadern die Berechnungen der Fluidsimulation auf diesen auszuführen, lassen sich inzwischen auch komplexere Simulationen in Echtzeit ausführen.

In dieser Arbeit soll eine Anwendung entwickelt werden, die auf Basis einer Fluidsimulation, eine stellenweise Verflüssigung bei Metallen simuliert. Die Verflüssigung bei Metallen spielt dabei eine große Rolle beim Löten und Schweißen. Auf Basis einer Simulation können Anwendungen erschaffen werden, die die Ausbildung in Berufen mit diesem Tätigkeitsfeld unterstützen. Zudem können Kosten reduziert werden, die sonst für Übungsmaterialien aufgewendet werden müssen. Die entwickelte Anwendung unterstützt dabei den Wechsel zwischen den beiden Aggregatzuständen fest und flüssig, durch den Einsatz einer Hitzequelle. Dabei müssen feste und flüssige Bereiche miteinander interagieren. Zusätzlich wird der Übergang dieser Bereiche mit einer Temperaturskala reguliert.

Da die verwendete Fluidsimulation auf einem Partikelsystem basiert, wird anhand der Partikel eine Oberfläche generiert. Damit die Oberfläche ein metallisches Aussehen bekommt, werden dieser mit einem entsprechenden Materialshader, Reflexionen und Beleuchtung hinzugefügt. Dabei wurde sich an dem Aussehen von Lötzinn orientiert. Weiterhin wird eine Möglichkeit geschaffen, damit Benutzer mit der Simulation interagieren können.

Die Hitzequelle kann durch den Benutzer beliebig bewegt und aktiviert werden. Zuletzt wird die Echtzeitfähigkeit der Anwendung auf unterschiedlichen Grafikkarten getestet und bewertet.

[1] **Fluid:** zusammenfassende Bezeichnung für Flüssigkeiten, Gase und Plasmen. Abgerufen am 04.01.2014 auf http://www.duden.de/rechtschreibung/Fluid

1.2 Verwandte Arbeiten

Die Simulation von schmelzenden und erstarrenden Materialien wird von Mark Carlson, Peter J. Mucha, R. Brooks Van Horn III und Greg Turk [MC02] beschrieben. Darin werden neben flüssigen Objekten, auch feste Objekte als Fluide betrachtet. Durch die Temperatur werden die Viskosität und damit die Zähflüssigkeit des Fluid angepasst.

In der Arbeit von Xiaoming Wei, Wei Li und Arie Kaufman [XWK03] wird das schmelzen von Objekten mit hoher Viskosität, wie z.B. Wachs, Plastik, Lava, Schokolade und Metall (Abb. 1), simuliert. Vor allem die Simulation von Metallen ist in Hinblick auf diese Arbeit von Bedeutung.

K. Iwasaki, H. Uchida, Y. Dobashi, und T. Nishita [KIN10] beschreiben den Schmelzvorgang von Eis. Dies geschieht dabei nur an der Oberfläche.

Bei allen genannten Simulationen von schmelzenden und erstarrenden Objekten, liegt eine Fluidsimulation als Grundlage vor. Matthias Müller, David Charypar und Markus Gross [MMG03] beschreiben in ihrem Paper eine echtzeitfähige Simulation von Fluiden, durch Verwendung der Smoothed-Particle-Hydrodynamics (SPH) Methode. Diese dient zusammen mit der Implementation von Rama Hoetzlein [Ram07] als Grundlage für diese Arbeit.

Zur Visualisierung des Partikelsystems wird der Marching-Cubes Algorithmus [LC87] verwendet. Cyril Crassin [Cyr06] beschreibt eine Implementation des Algorithmus mithilfe von Geometry-Shadern.

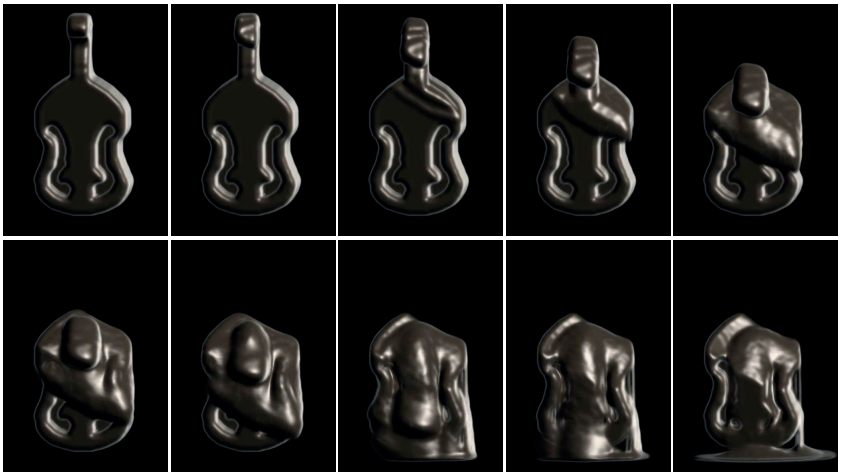


Abbildung 1: Schmelzendes Cello aus Zinn [XWK03]

2 Grundlagen

2.1 Partikelsysteme

Zum modellieren von Feuer, Rauch oder Flüssigkeiten können Partikelsysteme eingesetzt werden. Dabei wird das Volumen durch eine diskrete Anzahl von Partikeln angenähert. Im betrachteten Zeitintervall können sich Partikel im System bewegen und verändern. Dadurch besitzt das Model die Möglichkeit, Bewegung, Formveränderung und Dynamik darzustellen, was mit klassischen Oberflächenmodellen nicht möglich ist. [Ree83]

2.2 Smoothed-Particle-Hydrodynamics (SPH)

SPH wurde als Simulation von nicht achsensymmetrischen Phänomenen im Kontext der Astrophysik entwickelt (siehe Lucy [Luc77] und Gingold & Monaghan [GM77]). Es liefert eine hinreichende Genauigkeit und ist dabei einfach zu verwenden. SPH ist Partikel basierend und benötigt zur Berechnung kein Zellengitter. [Mon92]

Die Methode kann jedoch auch für jede andere Art von Fluidsimulation verwendet werden, wie im folgenden nach Müller, Charypar und Gross [MMG03] dargestellt wird.

SPH ist ein Interpolationsverfahren für Partikelsysteme. Dadurch können Feldwerte, die nur an diskreten Teilchen definiert sind, überall im Raum ausgewertet werden. Dafür werden die Werte in einer lokalen Nachbarschaft der Partikel mithilfe von radial symmetrischen Glättungskernen verteilt.

Der skalare Wert A an der Position $\mathbf{r}$ wird durch eine gewichtete Summe der Beiträge von allen Partikeln durch Interpolation berechnet (1).

$$A_S(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \tag{1}$$

Dabei iteriert j über alle Partikel, m_j ist die Masse des Partikel j, $\mathbf{r}_j$ die Position und ρ_j die Dichte. A_j ist der entsprechende Feldwert an Position $\mathbf{r}_j$. $W(\mathbf{r}, h)$ ist der verwendete Glättungskern mit Radius h.

Die Masse m_j bleibt über den gesamten Zeitraum der Simulation konstant und ist in dem Fall der Fluidsimulation für alle Partikel gleich. Die Dichte ρ_j dagegen muss nach jedem Schritt neu berechnet werden. Die Dichte an Position $\mathbf{r}$ kann durch Substitution in Gleichung (1) berechnet werden (2).

$$\rho_S(\mathbf{r}) = \sum_j m_j \frac{\rho_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) = \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h) \tag{2}$$

Die meisten Gleichungen für Fluidsimulationen benötigen Ableitungen der Feldwerte. Bei der SPH Methode beeinflussen die Ableitungen nur den Glättungskern. Der Gradient und der Laplace-Operator sind in (3) und (4) dargestellt.

$$\nabla A_S(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r}_j, h) \tag{3}$$

$$\nabla^2 A_S(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r}_j, h) \tag{4}$$

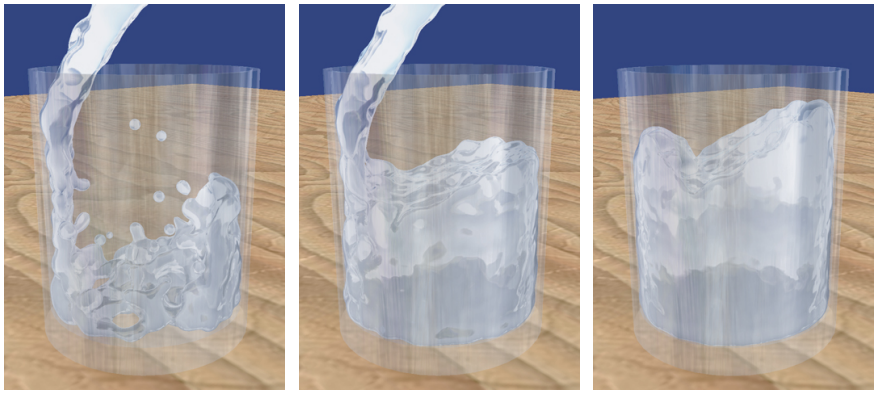
Um ein Fluid zu simulieren, werden ein Geschwindigkeits-, ein Dichte- und ein Druckfeld benötigt. Die Entwicklungen dieser Felder über einen bestimmten Zeitraum sind durch die Gleichungen zur Masseerhaltung und zur Impulserhaltung gegeben. Da es eine konstante Anzahl von Partikeln gibt, die alle die gleiche Masse besitzen, ist die Masseerhaltung gegeben. Für die Impulserhaltung wird die Navier-Stokes Gleichung verwendet (5).

$$\rho(\frac{\delta \mathbf{v}}{\delta t} + \mathbf{v} \cdot \nabla \mathbf{v}) = -\nabla p + \rho \mathbf{g} + \mu \nabla^2 \mathbf{v} \tag{5}$$

Dabei wird eine vereinfachte Form der Navier-Stokes Gleichung für inkompressible Fluide verwendet. $\mathbf{v}$ ist das Geschwindigkeitsfeld, μ die Viskosität, p der Druck und $\mathbf{g}$ sind externe Kräfte (z.B. Gravitation). Die Summe der Felder $\mathbf{f} = -\nabla p + \rho \mathbf{g} + \mu \nabla^2 \mathbf{v}$ ist die Änderung des Impuls der Partikel. Um die Beschleunigung von Partikel i zu erhalten, wird die Gleichung wie folgt umgestellt (6).

$$\mathbf{a}_i = \frac{d\mathbf{v}_i}{dt} = \frac{\mathbf{f}_i}{\rho_i} \tag{6}$$

Mit der Beschleunigung $\mathbf{a}_j$ und der Zeit t. Mit der Beschleunigung lässt sich nun die neue Position eines Partikels berechnen. In Anhang A sind die Formeln zur Berechnung des Drucks, der Druckkraft und der Viskositätskraft gegeben. In Anhang B sind mehrere Glättungskerne angegeben. In Abbildung 2 ist eine beispielhafte Anwendung von SPH dargestellt. [MMG03]



(a) **(b)** **(c)**

Abbildung 2: Einfüllen von Wasser in ein Glas mithilfe der SPH Methode [MMG03]

2.3 Marching-Cubes

Der Marching-Cubes Algorithmus wurde 1987 von William E. Lorensen und Harvey E. Cline entwickelt. Der ursprüngliche Zweck war die Visualisierung von medizinischen Volumendaten, die mithilfe von Computertomographie (CT), Magnetresonanztomographie (MRT) oder Einzelphotonen-Emissionscomputertomographie (SPECT) erzeugt wurden.

Der Algorithmus erzeugt dabei eine aus Dreiecken bestehende Oberfläche. Nach dem divide-and-conquer Ansatz, werden die Volumendaten in quaderförmige Zellen, bestehend aus acht Voxeln, unterteilt. Entlang eines vom Benutzer eingestellten Isowerts wird eine Oberfläche erzeugt. Der Isowert ist dabei der Schwellwert, ob ein Voxel innerhalb oder außerhalb einer Zelle liegt. Die daraus resultierenden 256 Fälle lassen sich aufgrund von Symmetrie in 15 verschiedene Fälle unterteilen (Abb. 3).

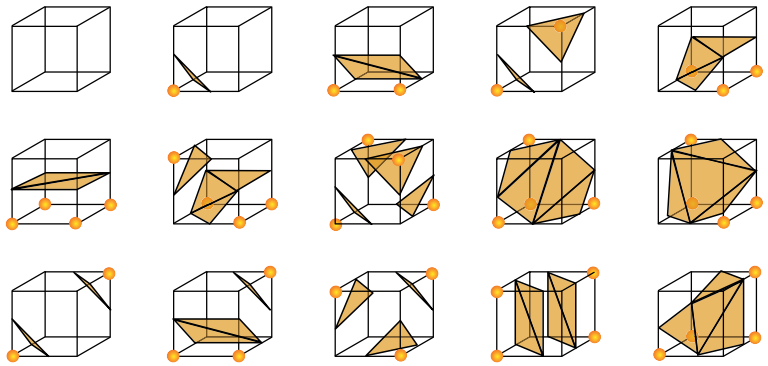


Abbildung 3: Marching-Cubes Lookup-Tabelle mit den 15 verschiedenen Möglichkeiten[2]

In einer Lookup-Tabelle wird für jeden Fall eine Liste von Nummern der Knotenpunkte hinterlegt, mit derer später die Dreiecke gezeichnet werden. Die Position der Knotenpunkte liegen dabei auf den Kanten der Zelle und können durch lineare Interpolation zwischen den Eckpunkten ermittelt werden. Für jede Zelle wird ein Index anhand der Eckpunkte bestimmt. Durch den Index können in der Lookup-Tabelle die entsprechenden Knotenpunkte entnommen werden. Anhand dieser Knotenpunkte werden die Dreiecke gezeichnet.

Um die daraus generierte Oberfläche zu beleuchten, wird mithilfe der Volumendaten ein Gradient berechnet. Durch Normalisierung des Gradienten erhält man die Normale. [LC87]

[2]Grafik nach dem Vorbild von Lorensen und Cline [LC87], abgerufen am 02.01.2014 auf http://en.wikipedia.org/wiki/File:MarchingCubes.svg

2.4 Enviroment-Mapping

Enviroment-Mapping bezeichnet ein Verfahren, Spiegelungen auf Objekten mithilfe von Umgebungstexturen darzustellen. Das Verfahren hat dabei einen viel geringeren Rechenaufwand als Ray-Tracing. In den ersten Schritten gleicht das Verfahren dem Ray-Tracing. Es wird ein Strahl vom Augpunkt zu einem Punkt auf der reflektierenden Oberfläche geschickt. Mit dem Normalenvektor kann daraus ein Reflexionsvektor berechnet werden. Im Gegensatz zum Ray-Tracing wird der Strahl nicht zur nächsten Objektoberfläche verfolgt. Stattdessen wird die Richtung des Strahls verwendet, um die Texturkoordinaten auf der Umgebungstextur zu bestimmen.

Die Umgebungstextur enthält dabei ein Bild der Umgebung aus dem Blickwinkel des Objekts. Es werden vor allem die Verfahren Sphere-Mapping und Cube-Mapping verwendet.[AN07, S. 262-263]

Sphere-Mapping Die sphärische Texturierung verwendet dabei ein Bild einer ideal verspiegelten Kugel, welches als Umgebungstextur dient. In dieser spiegelt sich die ganze Umgebung wieder. Zum Rand der Kugel nimmt dabei die Verzerrung zu. Das Bild der Kugel wird als Sphere-Map bezeichnet. [AN07, S. 263-266]

Cube-Mapping Bei der kubischen Texturierung werden sechs 2D-Texturen verwendet, die die Umgebung aus sechs verschiedenen Blickrichtungen darstellen. Diese bilden die Flächen eines Würfels. Das Objekt liegt im Zentrum des Würfels. Dabei müssen die Ränder der Texturen zusammen passen. Die so erhaltene Textur wird als Cube-Map bezeichnet. In OpenGL gibt es dafür den eigenen Texturtyp `GL_TEXTURE_CUBE_MAP`. [AN07, S. 266-270]

2.5 Geometry-Shader

Geometry-Shader wurden mit OpenGL 3.2 eingeführt und befinden sich in der Rendering-Pipeline hinter dem Vertex-Shader (bzw. hinter dem optionalen Tessellation-Shader) und vor der Vertex-Post-Processing-Fixed-Function-Pipeline.

Der Geometry-Shader ist dabei ein optionaler Shader, um neue Geometrie zu erzeugen. Der Eingabetyp der Primitiven (`points`, `lines`, `lines_adjacency`, `triangles`, `triangles_adjacency`) muss mit dem im Renderbefehl gesetzten Primitiventyp übereinstimmen. Die Ausgabe sind entweder keine Primitive, ein Primitiv oder mehrere Primitive (`points`, `line_strip`, `triangle_strip`). Die Ausgabe ist dabei auf den Wert `GL_MAX_GEOMETRY_OUTPUT_VERTICES` limitiert, der minimal 256 beträgt und von der verwendeten Grafikkarte abhängt.

Zusätzlich können mehrere Instanzen eines Geometry-Shaders für einen Vertex aufgerufen werden. [The97]

2.6 Compute-Shader

Compute-Shader wurden mit OpenGL 4.3 eingeführt und ermöglichen allgemeine Berechnungen (z.B. Partikelphysik). Die Compute-Shader sind dabei kein Teil der regulären Rendering-Pipeline, sondern können überall im Programm aufgerufen werden. Dies geschieht mit folgendem Befehl.

```
glDispatchCompute(GLuint num_groups_x, GLuint num_groups_y,
                  GLuint num_groups_z)
```

Der Parameter `num_groups_*` gibt die Anzahl der Work-Groups in drei Dimensionen an. Jede Work-Group besitzt zudem noch eine dreidimensionale Local-Size. Mit diesen beiden Größen lassen sich die Anzahl der aufgerufenen Instanzen regulieren. Mit Built-In Variablen kann auf die Work-Group-Size und Local-Size, sowie auf die aktuelle Work-Group-, Local- und Global-ID zugegriffen werden. Compute-Shader besitzen keine In- oder Outputs.

Daten können mithilfe von dem ebenfalls in OpenGL 4.3 eingeführten Buffertyp `GL_SHADER_STORAGE_BUFFER` an die Grafikkarte übergeben werden. Die Daten können, ohne die Grafikkarte verlassen zu müssen, auf dieser verändert werden. Damit weitere Shader die veränderten Daten verwenden können, müssen diese synchronisiert werden. Dies geschieht mit dem Einsatz von Barrieren, welche mit dem Befehl `glMemoryBarrier()` erzeugt werden.[The97][Bai13]

3 Umsetzung

3.1 Versuch

Um eine bessere Vorstellung von Metallen und deren Verflüssigung zu erhalten, wurde ein Versuch durchgeführt. Aufgrund seines geringen Schmelzpunktes, wurde Pb60Sn40Sb Stangenlötzinn als Metall verwendet. Dieses besteht aus 60% Blei und 40% Zinn. Das verwendete Lötzinn besitzt einen Schmelzbereich von $183° - 190°C$[3] und hat im festem Zustand eine silberweiße Farbe. Es wird in der Blechverarbeitung verwendet, ist jedoch nicht für Feinelektronik geeignet. Daraus wurde ein $64cm^3$ großer Würfel gegossen, welcher eine Seitenlänge von $40mm$ besitzt.

Der Lötzinnwürfel wurde mithilfe einer Metallvorrichtung in Position gebracht. Mit einer ca. $3000°C$ heißen Flamme, welche mit einem Sauerstoff-Propangas-Gemisch erzeugt wurde, konnte der Lötzinnwürfel an einer Ecke langsam verflüssigt werden. Dabei konnte festgestellt werden, dass sich das Lötzinn nur bei direkter Wärmeeinwirkung verflüssigt und in flüssigem Zustand stärker spiegelt. Sobald verflüssigtes Lötzinn wieder abgekühlt und somit erstarrt war, nahm es seine ursprüngliche, silberweiße Farbe wieder an. In Abbildung 4 sind einige Bilder des Versuchs dargestellt. Weitere Bilder werden in späteren Kapiteln als Vergleich verwendet.

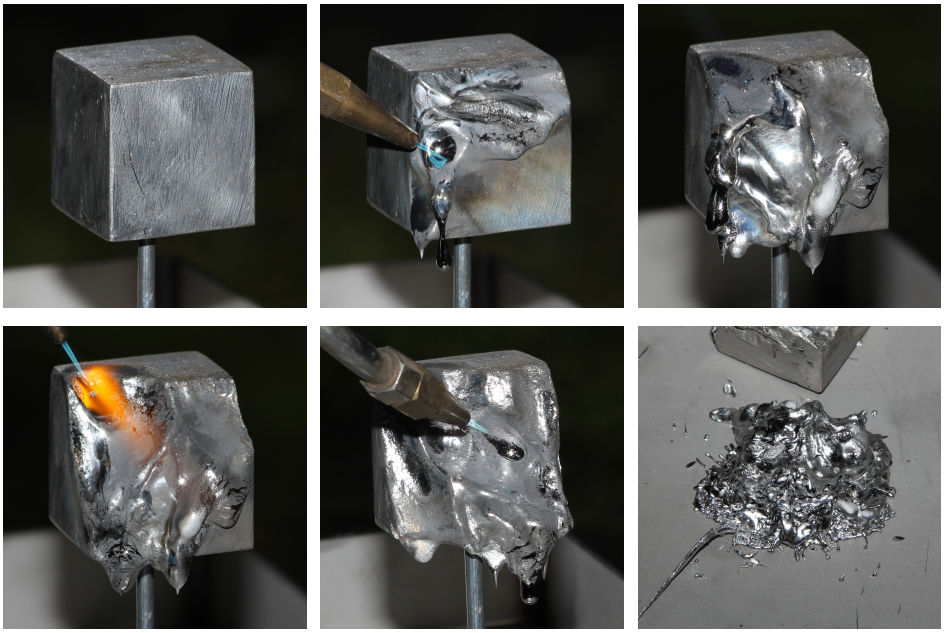


Abbildung 4: Lötzinnwürfel in mehreren Phasen des Verflüssigungsvorgangs

[3] Produktinformation FELDER-Stangenlötzinn S-Pb60Sn40Sb, abgerufen am 02.01.2014 auf http://www.felder.de/tl_files/felder/pdf/Produktinformationen/12-13/12-StangenlotS-Sn60Pb40Sb.pdf

3.2 Verwendete Programme und Bibliotheken

Die hier vorgestellte Anwendung wurde mit Visual Studio 2012 in der Programmiersprache C++ als Windows Konsolenanwendung entwickelt. Zur Darstellung der 3D Szene, wurde OpenGL und GLSL 4.3 verwendet. Die Version 4.3 ist erforderlich, da der Großteil der physikalischen Berechnungen, aufgrund der Parallelisierbarkeit, mithilfe von Compute-Shadern realisiert wurde. Diese sind erstmals ab dieser Version verfügbar. Zur Einbindung von OpenGL in C++ wurde die GLEW[4] Bibliothek verwendet. Des Weiteren wurde die OpenGL Mathematics (GLM)[5] Bibliothek verwendet, um verschiedene Vektor- und Matrizenberechnungen durchzuführen.

Zur Erzeugung des Fensters wurde GLFW 3.0.2[6] verwendet, welches auch den OpenGL Rendering-Context festlegt. Ebenso werden über diese Bibliothek die Maus- und Tastatursteuerung verwaltet. Um dem Benutzer eine Einstellmöglichkeit bestimmter Parameter zu ermöglichen, wurde die AntTweakBar 1.16[7] genutzt. Mit dieser lassen sich auf OpenGL basierte Menüs erstellen und mit wenig Aufwand in das Programm integrieren.

Um Bilder in der Anwendung als Texturen zu verwenden, wurde die Klasse stb_images.cpp[8] verwendet.

3.3 Aufbau der Anwendung

In disem Kapitel werden die erstellten Klassen und die zugehörige Ordnerstruktur der Anwendung vorgestellt. Der Quellcode, in Form eines Visual Studio 2012 Projekts, liegt auf DVD bei.

`Main` und `LiquidMetal` bilden den Einstiegspunkt in die Anwendung. In `Main` wird ein Objekt der Klasse `LiquidMetal` erstellt und initialisiert. Zusätzlich wird die Methode `run()` aufgerufen, welche die Hauptschleife der Anwendung ausführt. In der Klasse `LiquidMetal` werden auch alle GLFW Befehle ausgeführt, um ein Fenster zu erzeugen und den OpenGL-Rendering-Context an dieses zu binden. Weiterhin wird die verwendete OpenGL Version überprüft und gegebenenfalls die Anwendung mit einer Fehlermeldung beendet. Auch werden Objekte des Typs `Renderer` und `EventManager` angelegt.

Die Klasse `Renderer` erstellt die Skybox sowie die Bounding-Box für die Simulation und speichert verschiedene Grundeinstellungen, wie z.B. die Lichtquelle. Des Weiteren werden Objekte der Klassen `InterfaceManager`, `FileManager`, `Camera` und `ParticleSystem` erzeugt und verwaltet.

[4] http://glew.sourceforge.net/
[5] http://glm.g-truc.net/0.9.4/index.html
[6] http://www.glfw.org/
[7] http://anttweakbar.sourceforge.net/doc/
[8] http://nothings.org/

Die Klasse `Camera` speichert hierbei die Kamera Parameter und bietet verschiedene Methoden, um die Kamera mit wenig Befehlen im Raum zu navigieren. In diesen wird direkt die veränderte Projection- und Viewmatrix berechnet, auf die mit Get-Methoden zugegriffen werden kann.

Die Einstellmenüs werden mit der Klasse `InterfaceManager` verwaltet, die mithilfe der AntTweakBar Bibliothek erzeugt werden. Tastatur und Mauseingaben werden von der Klasse `EventManager` abgefangen und verarbeitet.

Damit eingestellte Parameter nach einem Reset oder Neustart erhalten bleiben, wurde noch die Klasse `FileManager` erstellt. Diese schreibt die Parameter in eine Textdatei und kann diese, wenn benötigt, wieder auslesen.

Um oft genutzte Befehle zu kapseln, existiert die Klasse `OpenGLManager`. Diese besitzt Befehle um Texturen, Cubemaps, Buffer und Shader mit wenigen Zeilen Code zu erstellen.

Als letztes existiert noch die Klasse `ParticleSystem`. In dieser werden alle Shader zur Berechnung der SPH Simulation, sowie die Shader zur Generierung der Oberfläche initialisiert. Alle Parameter für die Berechnungen werden mit verschiedenen Uniform-Variablen an diese übergeben und schließlich werden die Berechnungen ausgeführt.

Die Header-Datei `Resources` wird von allen Klassen eingebunden. In dieser werden häufig verwendete Header inkludiert und mehrere Structures sowie globale Variablen definiert. Zusätzlich existiert der Ordner **media** mit einer config.txt, die vom `FileManager` benötigt wird. Weiterhin befinden sich in diesem zwei Unterordner, in denen die GLSL Shader und die Bilder für die Cubemap liegen.

3.4 Fluidsimulation

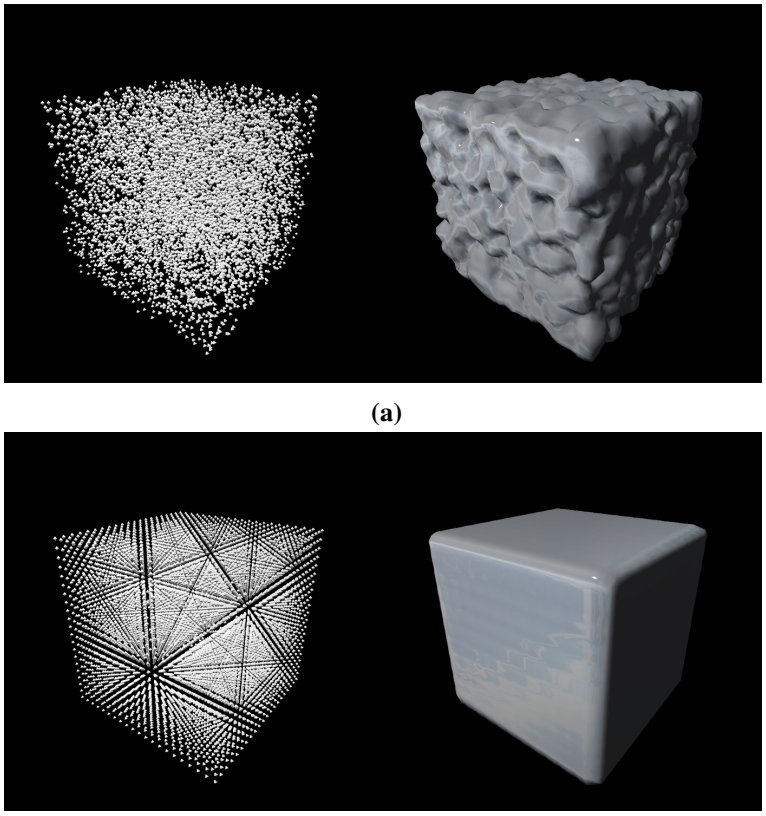
3.4.1 Initialisieren der Partikel

Die in dieser Anwendung verwendete Fluidsimulation basiert auf einem Partikelsystem. Dieses wird, nachdem der Benutzer die Partikelanzahl sowie die Anordnung bestimmt hat, initialisiert.

Diese Aufgabe wird von der Klasse `ParticleSystem` durchgeführt. Die Partikel werden dabei in dem Datentyp `struct PARTICLES` gespeichert. In diesem werden die Partikelanzahl und mehrere Partikeleigenschaften gespeichert. Mit der Partikelanzahl werden die verschiedenen Arrays für die Partikeleigenschaften angelegt. Dazu zählen Position, Geschwindigkeit und Beschleunigung, welche als `vec4` Array abgespeichert werden. Zusätzlich werden die Dichte, Druck und Temperatur als `GLfloat` Array gespeichert. Dabei wird die gesamte Simulation und Oberflächengenerierung um den Ursprung angelegt. Zusätzlich beschränkt sie sich

auf den Bereich [-1,1] in allen drei Dimensionen (Kapitel 3.5.1, Abb. 11).

Bei der Initialisierung werden alle Werte, mit Ausnahme der Position, auf 0 bzw. `vec4(0,0,0,0)` gesetzt. Die Position der Partikel wird je nach gewählter Anordnung festgelegt. Die Partikel werden im Bereich [-0.5, 0.5] angeordnet. Bei zufälliger Anordnung werden die Partikel mithilfe der C++ Funktion `rand()` um den Ursprung verteilt. Das Ergebnis ist nach erfolgter Oberflächengenerierung ein deformierter Würfel (Abb. 5a).



(a)

(b)

Abbildung 5: Partikel und daraus resultierende Oberfläche
(a) zufällige Verteilung, (b) geordnete Verteilung

Bei der geordneten Anordnung der Partikel, werden diese in diskreten Schritten verteilt. Aus der dritten Wurzel der Partikelanzahl kann bestimmt werden, wie viele Partikel in eine Dimension angeordnet werden müssen. Dabei wird das Ergebnis aus der Wurzel abgerundet. Alle Partikel die übrig bleiben, werden zufällig im Bereich [-0.25, 0.25] verteilt. Hieraus ergibt sich nach der Oberflächengenerierung ein gleichmäßiger Würfel (Abb. 5b), da die zufällig angeordneten Partikel im inneren liegen. Dies ist jedoch nur dann der Fall, wenn der Wert von Particle-Influence (Kapitel 3.5.1) nicht zu groß gewählt wird.

Quellcode 1: Funktion zum binden von Shader-Storage-Buffer

```
void OpenGLManager::createStorageBuffer(GLsizei n, GLuint* buffer,
       GLsizeiptr size, const GLvoid* data, GLuint index)
{
  glGenBuffers(n, buffer);
  glBindBuffer(GL_SHADER_STORAGE_BUFFER, *buffer);
  glBufferData(GL_SHADER_STORAGE_BUFFER, size , data, GL_STATIC_DRAW);
  glBindBufferBase(GL_SHADER_STORAGE_BUFFER, index, *buffer);
}
```

Nach erfolgter Initialisierung, werden alle Eigenschaften der Partikel, mithilfe von Shader-Storage-Buffer, an die Grafikkarte übergeben. Um dies zu vereinfachen, wurde die Funktion `createStorageBuffer()` angelegt.

3.4.2 Implementierung von SPH auf der GPU

Im vorigen Kapitel wurden die Partikel initialisiert, besitzen jedoch noch kein physikalisches Verhalten eines Fluid. Um dies zu gewährleisten wird die SPH-Methode verwendet, da sie sehr robust und einfach zu implementieren ist. Die Berechnungen von SPH werden dabei alle auf der Grafikkarte, mithilfe von OpenGL Compute-Shadern, durchgeführt. In jedem Frame werden nacheinander drei Compute-Shader benötigt. Alle drei Compute-Shader werden dabei mit folgenden Werten für die Anzahl der Work-Groups und die Local-Size initialisiert.

```
glDispatchCompute(particleNumber / 64, 1, 1)
layout(local_size_x = 64, local_size_y = 1, local_size_z = 1) in
```

Dabei berechnet jede Instanz der drei Compute-Shader ein Partikel. Durch eine Local-Size von 64 werden gute Ergebnisse erzielt [Bai13]. Zur Berechnung der Fluidsimulation werden mehrere Parameter benötigt, die vom Benutzer jederzeit geändert werden können.

- **Rest-Density ρ_0:** Dichte eines kleinen Teils des Fluid, wenn dieses sich im Ruhezustand befindet.
- **Mass m:** Masse eines Partikels, die für alle Partikel gleich ist.
- **Viscosity μ:** Viskosität des Fluid, welche ein Maß für die Zähflüssigkeit darstellt.
- **Gas-Stiffness k:** Ideale Gaskonstante, welche zur Berechnung des Drucks benötigt wird (Anhang A, Gleichung 11).
- **Support-Radius h:** Legt den Einflussbereich eines Partikels fest, wodurch die Anzahl der mit einzubeziehenden Nachbarn bei der Berechnung festgelegt wird.

Die Parameter werden mit Uniform-Variablen an die Compute-Shader übergeben. Zusätzlich arbeiten die Shader auf den zuvor angelegten Shader-Storage-Buffer, in denen die Partikeleigenschaften hinterlegt sind. Zusätzlich werden drei verschiedene Glättungskerne (Anhang B) verwendet.

Es werden drei Compute-Shader benötigt, da die Shader aufeinander aufbauen. Im ersten der drei Shader wird der Druck p (Anhang A, Gleichung 11) und die Dichte ρ (Kapitel 2.2, Gleichung 2) berechnet.

Mithilfe der Built-In Variable `gl_GlobalInvocationID` kann die ID der aktuellen Instanz des Compute-Shaders ermittelt werden. Da für jedes Partikel eine eigene Instanz aufgerufen wird, sind Instanz-ID und Partikel-ID identisch. Sobald die Partikel-ID bekannt ist, wird eine Schleife über alle Partikel ausgeführt (Quellcode 2). Dies ist nötig um die Summe der Dichtegleichung zu erhalten. Weiterhin wird die Masse m, der Support-Radius h und ein Glättungskern benötigt, um die Dichte ρ zu berechnen. Als Glättungskern kommt der Poly6-Kern (Anhang B, Gleichung 14) zum Einsatz.

Quellcode 2: Schleife für verschiedene SPH-Gleichungen

```
uint gid = gl_GlobalInvocationID.x;

//...

for(int j = 0; j < particleNumber; j++){
  //Berechnung der Distanz zwischen dem Partikel der aktuellen
  //Shader Instanz und dem aktuellen Partikel der Schleife
  vec4 temp = Positions[gid] - Positions[j];
  float distance = sqrt(dot(temp, temp));

  //Überprüfen ob der aktuelle Partikel der Schleife im
  //Einflussbereich liegt
  if(supportRadius > distance){
    //Berechnung
    //...
  }
}
```

Sobald die Dichte ρ des aktuellen Partikels berechnet wurde, kann mit dieser, der Ruhedichte ρ_0 sowie der Gaskonstante k, der Druck p des Partikels berechnet werden. Dichte und Druck werden in den entsprechenden Shader-Storage-Buffers gespeichert.

Im zweiten Compute-Shader wird die Beschleunigung der Partikel berechnet. Dies geschieht mit der vereinfachten Navier-Stokes Gleichung (Kapitel 2.2, Gleichung 6). Neben der Konstanten Gravitationskraft `vec3(0,-9.81,0)`, werden die Viskositäts- und die Druckkraft benötigt (Anhang A, Gleichung 12 und 13). Die Berechnung gestaltet sich ähnlich wie im ersten Shader, da dieselbe Schleife verwendet wird (Quellcode 2). Nur die Berechnung unterscheidet sich. Als Parameter werden, neben dem Support-Radius h und der Masse m, zusätzlich die Viskosität μ benötigt.

Für die Berechnung der Druckkraft wird die erste Ableitung des Spiky-Kerns (Anhang B, Gleichung 15) verwendet. Die zweite Ableitung des Viscosity-Kerns wird für die Berechnung der Viskositätskraft benötigt (Anhang B, Gleichung 16).

Aus den drei Kräften kann nun die Beschleunigung durch Addition berechnet werden. Zur Vermeidung von Fehlern durch Partikel mit zu hoher Beschleunigung wird ein Limit eingeführt. Die Beschleunigung eines Partikels kann dieses Limit dabei nicht überschreiten.

Der letzte Compute-Shader kann nun anhand der Beschleunigung die Geschwindigkeit und die neue Position des Partikels errechnen. Dazu wird zusätzlich die Zeit t benötigt. Diese ist mit einem konstanten Wert von $\frac{1}{60}$ gegeben. Sobald die neue Position des Partikels vorliegt, kann eine Kollisionserkennung durchgeführt werden. Wenn die neue Position des Partikels außerhalb der Bounding-Box liegt, wird dieser auf den Schnittpunkt zurückgesetzt. Die Geschwindigkeit wird umgekehrt und mit einem festgelegten Faktor verlangsamt. Dieser beträgt in der Anwendung 0,3.

In Abbildung 6 kann das Ergebnis mit Werten aus Tabelle 1 betrachtet werden. Dabei wird zur Visualisierung das Marching-Cubes Verfahren verwendet, welches in Kapitel 3.5 genauer erläutert wird.

Parameter	**Wert**
Particle-Number	20032
Rest-Density	70
Mass	0.02
Viscosity	3.5
Gas-Stiffness	5.0
Support-Radius	0.05

Tabelle 1: SPH Werte für die Simulation in Abb. 6. Gleichzeitig sind dies auch die im Programm vordefinierten Default-Werte.

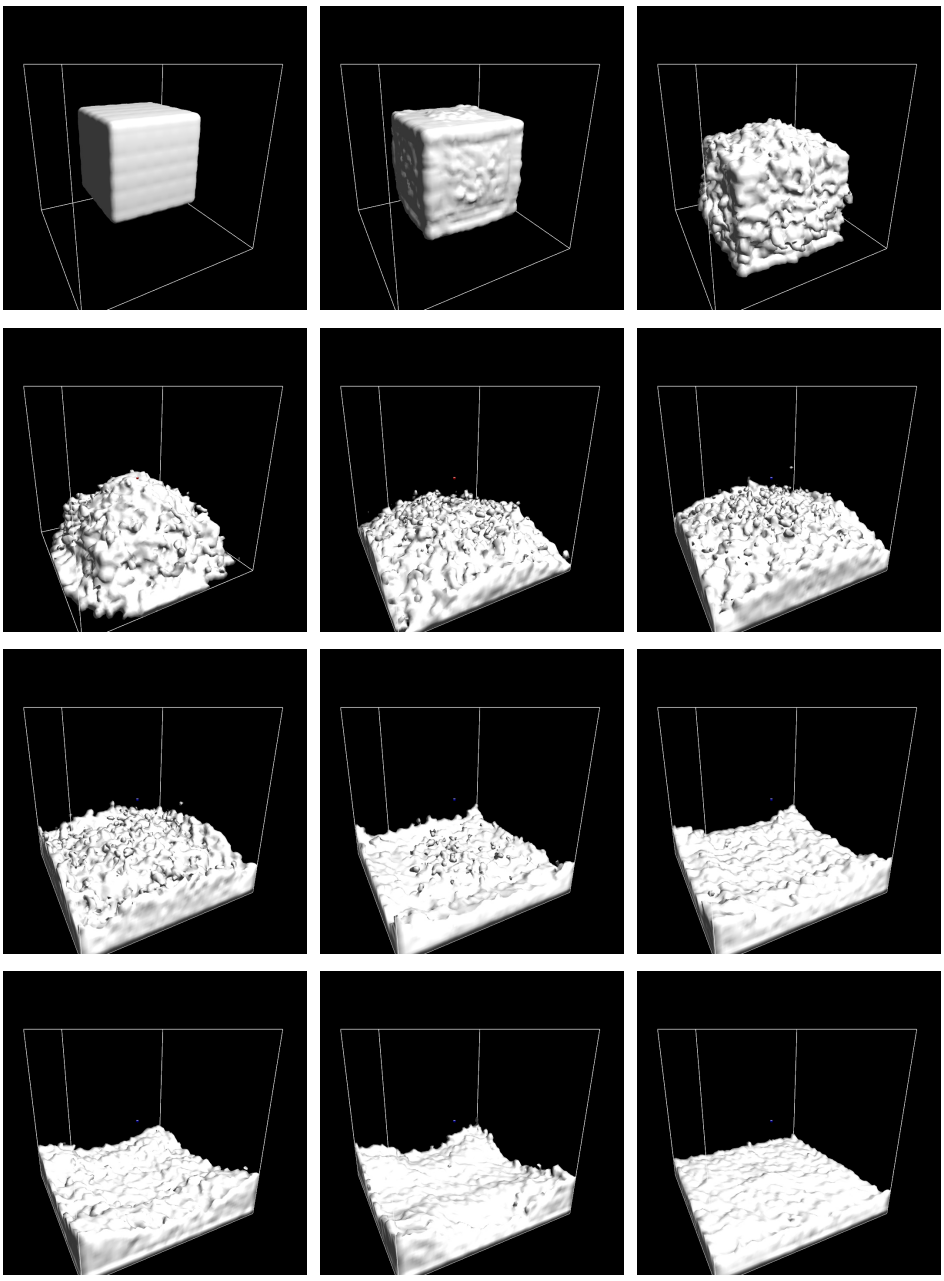


Abbildung 6: Zeitlicher Ablauf eines fallenden Würfels, der sich vollständig im flüssigen Zustand befindet

3.4.3 Integration des Temperaturverhaltens

Im vorigen Kapitel wurden die Partikel als Fluid simuliert. Um nun den festen Aggregatzustand und den ineinander übergehenden Wechsel zwischen fest und flüssig darzustellen, wird der Temperatur Parameter des Partikels verwendet. Dabei wird das Objekt zu jedem Zeitpunkt als Fluid betrachtet. Der Temperaturwert wird als `GLfloat` hinterlegt und wurde in Kapitel 3.4.1 eingeführt.

Initial wird die Temperatur auf 0 gesetzt. Bei diesem Wert ist das entsprechende Partikel bewegungslos und repräsentiert den festen Aggregatszustand. Das Maximum beträgt 1, bei dem das Partikel vollständig als Flüssigkeit agiert und sich so verhält, wie im vorigen Kapitel.

Um dieses Verhalten zu erreichen, wird die Geschwindigkeit im dritten SPH Compute-Shader, zur finalen Positionsberechnung, mit der Temperatur multipliziert. Dadurch wird bei sinkender Temperatur gleichzeitig auch die Geschwindigkeit der Partikel reduziert. Zusätzlich wird im zweiten und dritten Compute-Shader, zur Berechnung der Beschleunigung und der Position, die Bedingung (7) vorausgesetzt. Dadurch werden bei einem Temperaturwert von 0 Ressourcen gespart, da die beiden Berechnungen bei erstarrten Partikeln wegfallen.

$$Temperatur > 0 \tag{7}$$

Um ein erstarren der Partikel zu realisieren, wird ein weiterer vom Benutzer einstellbarer Wert eingeführt. Die Erstarrungsrate (in der Anwendung Solidify-Rate) wird dabei in jedem Frame, nach der Positionsberechnung, von der Temperatur subtrahiert. Dabei kann die Temperatur nicht negativ werden. Wenn die Erstarrungsrate auf den Wert 0 eingestellt wird, bleiben einmal erwärmte Partikel flüssig.

Da die Partikel sich initial im festen Zustand befinden, muss eine Möglichkeit geschaffen werden, diese zu erwärmen. Dazu steht dem Benutzer eine bewegliche Hitzequelle (in der Anwendung Heat-Source) zur Verfügung. Die Steuerung der Hitzequelle wird in Kapitel 3.7.3 erläutert. Wenn die Hitzequelle aktiviert ist, werden Temperaturwerte auf die Partikel im Einflussbereich der Hitzequelle in jedem Frame addiert. Der aufaddierte Wert ist im Zentrum des Einflussbereichs am größten und nimmt zum Rand hin ab.

Dies geschieht im ersten SPH Compute-Shader zur Berechnung der Dichte und dem Druck. Dabei ist der Einflussbereich (in der Anwendung Heat-Radius) wieder vom Benutzer einstellbar und beeinflusst auch den Temperaturwert, der auf einen Partikel addiert wird. Ein weiterer Parameter der den Temperaturwert beeinflusst, ist die Temperatur der Hitzequelle. Die Temperatur wird zusammen mit der Position der Hitzequelle als `vec4` als Uniform-Variable an den Shader übergeben. Ebenso wird der Einflussbereich an den Shader übergeben. Im folgenden Quellcode ist die Berechnung der Temperatur eines Partikels gezeigt.

Quellcode 3: Erwärmen der Partikel durch Addition

```
if(heatSource.w != 0.0f){
  //Berechnung der Distanz zwischen Hitzequelle und Partikel
  vec3 temp = Positions[gid].xyz - heatSource.xyz;
  float distance = dot(temp, temp);

  if(heatRadius > distance){
    float c = 1.0f - (distance / heatRadius);

    //Addition der Temperatur in Abhängigkeit zur Distanz
    Temperature[gid] += heatSource.w * c;
    if(Temperature[gid] > maxTemperature){
      Temperature[gid] = maxTemperature;
    }
  }
}
```

Die aktuelle Temperatur eines Partikels kann in der Anwendung angezeigt werden. Dazu wird die Temperatur mit einer, an eine Wärmebildkamera angelehnten, Farbskala dargestellt. Eine andere Möglichkeit wäre gewesen, die Temperaturwerte als Grauwerte darzustellen. Darauf wurde jedoch aus Gründen der Übersicht verzichtet, da eine Farbskala intuitiver ist. In Tabelle 2 sind die RGB Farbwerte mit entsprechender Temperatur aufgelistet. Die Zwischenwerte werden interpoliert.

Temperatur	Farbe	RGB Wert
0.00	Blau	(0, 0, 1)
0.25	Lila	(1, 0, 1)
0.50	Rot	(1, 0, 0)
0.75	Gelb	(1, 1, 0)
1.00	Weiß	(1, 1, 1)

Tabelle 2: Farbskala der Temperaturwerte

Eine Wärmeweiterleitung, die bei Metallen vorkommt, ist aktuell nicht in der Anwendung implementiert. Diese ist auch nicht unbedingt nötig, da durch Wärmeweiterleitung nicht genügend Wärme entsteht, um weitere Teile des Metalls zu verflüssigen.

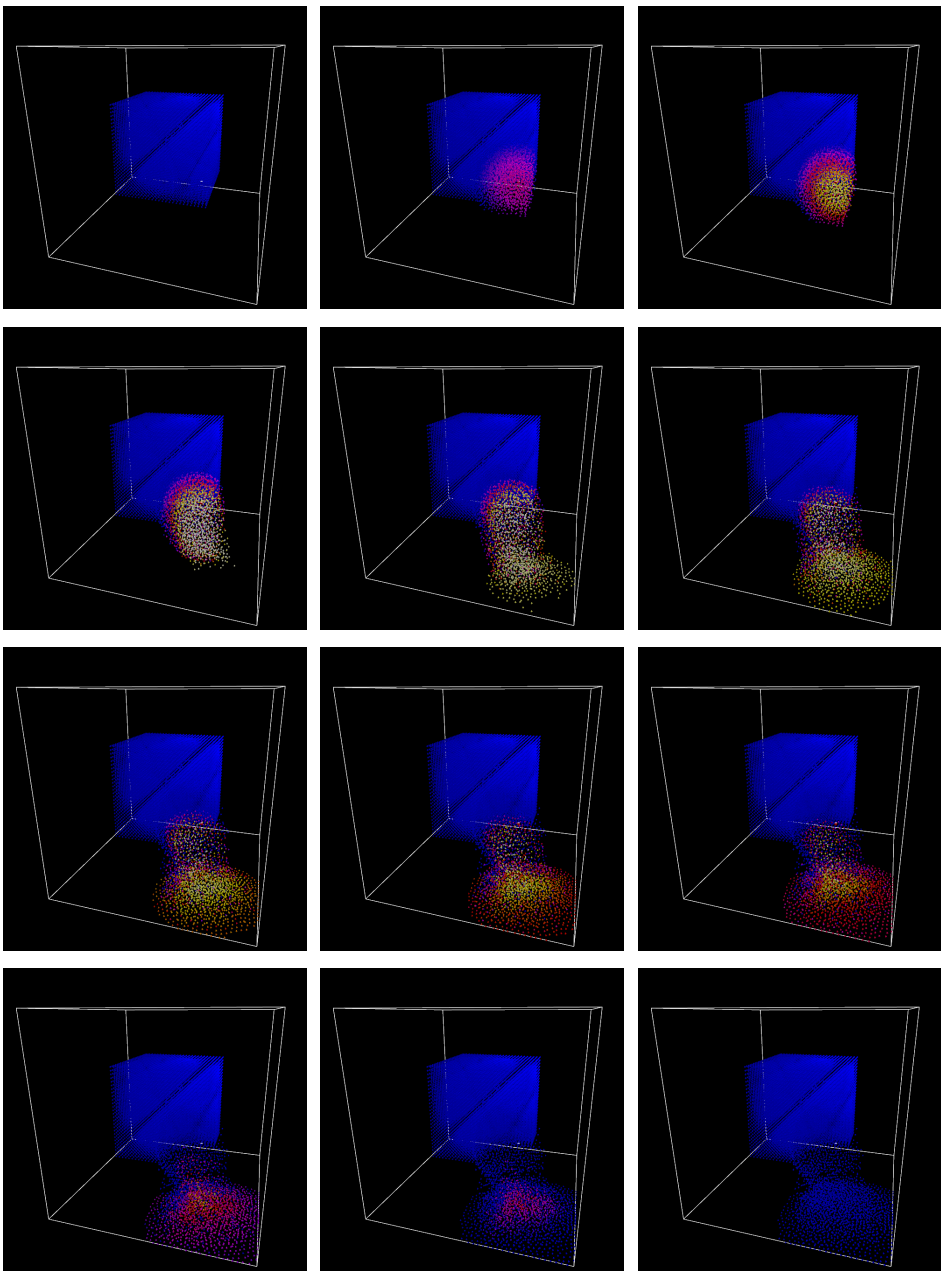


Abbildung 7: Zeitlicher Ablauf vom erwärmen bis zum erstarren der Partikel in Temperaturdarstellung

3.5 Visualisierung

3.5.1 Erstellung eines Volumen Datensatzes

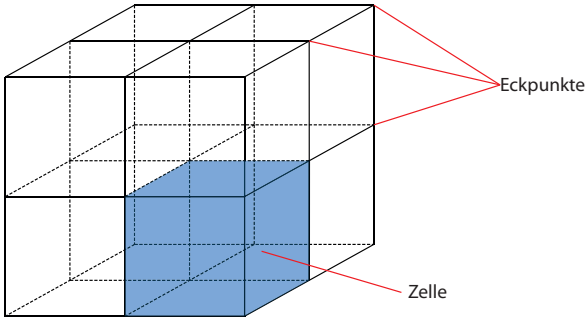
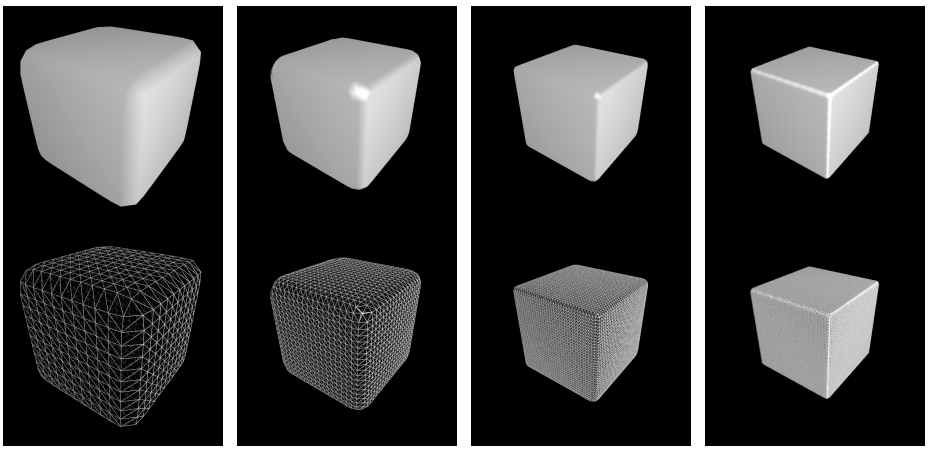


Abbildung 8: Beispielhaftes Zellengitter (Gridsize = 2)

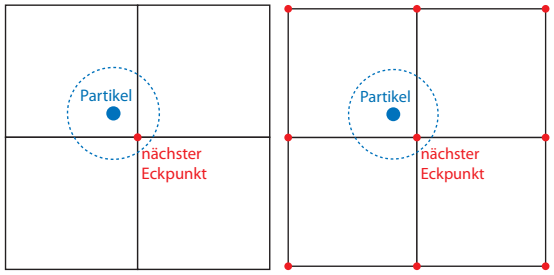
Zur Generierung einer Oberfläche, aus den im vorigen Kapitel erzeugten und animierten Partikeln, wird zuerst ein Volumendatensatz angelegt. Dieser wird in einer 3-Dimensionalen Textur gespeichert. Zuerst muss die Größe der 3D-Textur festgelegt werden. Dies kann der Benutzer nach dem Start der Anwendung festlegen. Ihm stehen Werte zwischen 2 und 128 in einer Dimension zur Verfügung. Dieser Wert wird in der Anwendung als Gridsize bezeichnet. Die anderen beiden Achsen werden auf denselben Wert eingestellt, um würfelförmige Zellen zu erhalten. Daraus ergibt sich, dass mindestens 2^3 und maximal 128^3 Zellen existieren. Je mehr Zellen zur Verfügung stehen, desto feiner wird die erstellte Oberfläche. In der 3D-Textur werden die Werte der Eckpunkte der Zellen gespeichert (Abb. 8). Aus diesem Grund hat die daraus generierte 3D-Textur die Größe $(Gridsize + 1)^3$. In Abbildung 9 werden die Auswirkungen für vier verschiedene Werte gezeigt.



(a) 16 **(b)** 32 **(c)** 64 **(d)** 128

Abbildung 9: Verschiedene Werte für Gridsize als Oberfläche und Drahtgittermodel

In der 3D-Textur wird die Partikelverteilung an den Eckpunkten des Gitters gespeichert. Dazu wird die Distanz zwischen jedem Partikel und einer vordefinierten Anzahl an Eckpunkten berechnet. Wie viele Eckpunkte um einen Partikel betrachtet werden, kann der Benutzer einstellen. Dieser Wert wird als Particle-Influence bezeichnet. Wenn Particle-Influence = 1 gilt, wird nur die Distanz zum nächsten Eckpunkt bestimmt (Abb. 10a). Bei Particle-Influence = 2 sind es dagegen schon 27 Eckpunkte. Zum einen der nächste Eckpunkt und zusätzlich die 26 umliegenden Eckpunkte (Abb. 10b).



(a) 1 **(b)** 2

Abbildung 10: Beispielhafte Darstellung für Particle-Influence im 2D mit unterschiedlichen Werten

In der Anwendung lässt sich dieser Wert von 1 bis 8 einstellen. Dadurch steigt die Anzahl der Eckpunkte stark an (Tabelle 3). Je höher der Einfluss eines Partikels ist, desto mehr wird die Oberfläche geglättet. Hohe Werte wirken sich dabei stark auf die Performance aus. Zum anderen können sich hohe Werte bei zu geringer Gridsize eher negativ auf die Darstellung auswirken. Daher kann der Wert auf 2 eingestellt bleiben.

Particle-Influence	**Anzahl der Eckpunkte**
1	1
2	27
3	125
4	343
5	729
6	1331
7	2197
8	3375

Tabelle 3: Werte für Particle-Influence und daraus resultierende Eckpunktanzahl

Zusätzlich kann durch den Wert von Particle-Influence die maximale Distanz zwischen einem Partikel und den Eckpunkten im Einflussbereich berechnet werden (8). Diese wird später benötigt, um die Distanzen auf den Wertebereich [0,1] zu skalieren. Die so berechneten Distanzen werden für jedes Partikel berechnet und auf die entsprechenden Werte in der 3D-Textur addiert.

$$MaxDistance = \frac{\sqrt{3}}{2} + \sqrt{3} * (ParticleInfluence - 1) \tag{8}$$

Um eine möglich effiziente Berechnung der Partikelverteilung zu gewährleisten, wird die Berechnung mithilfe von Compute-Shadern vollständig auf der GPU parallelisiert. Dabei sind bei diesem Schritt drei Compute-Shader im Einsatz. Diese werden in jedem Frame nacheinander ausgeführt.

In einem der Shader wird mit dem GLSL-Befehl `imageAtomicAdd()` gearbeitet. Dieser ist ein Teil von Image-Load-Store, welches mit OpenGL 4.2 eingeführt wurde. Der Befehl wird benötigt, um parallele Zugriffe auf die 3D-Textur im Compute-Shader zu ermöglichen. Da der Befehl aktuell Texturen der Datentypen `int` oder `uint` voraussetzt, wird für die 3D-Textur der Datentyp `GL_R32UI` verwendet.

Im ersten der drei Shader wird jeder Wert der Textur mit 0 initialisiert. Dabei wird für jeden Wert der 3D-Textur eine Instanz des Compute-Shaders aufgerufen.

```
glDispatchCompute(gridSize +1, gridSize +1, gridSize +1);
```

Sobald jeder Wert auf 0 gesetzt ist, kann die eigentliche Berechnung der Partikelverteilung beginnen. Dies geschieht im zweiten Compute-Shader. Von diesem wird für jedes Partikel eine Instanz aufgerufen. Dabei werden alle Partikel, wie schon in Kapitel 3.4.2 beschrieben, auf Work-Groups mit einer Local-Size von 64 aufgeteilt.

Um die Partikelverteilung im Compute-Shader berechnen zu können, wird die aktuelle Position der Partikel benötigt. Diese muss nicht nochmal an die Grafikkarte übergeben werden, da dies schon geschehen ist und in den vorigen Kapiteln zur Berechnung der Fluidsimulation beschrieben wurde.

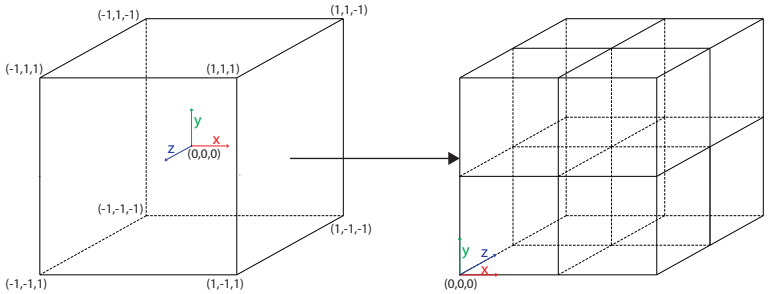


Abbildung 11: links: Bereich in dem Partikel liegen können in OpenGL-Koordinaten, rechts: Koordinatensystem der 3D-Textur

Die Partikelpositionen liegen zunächst in OpenGL-Koordinaten und im Bereich [-1,1] um den Ursprung verteilt vor. Diese müssen in das Koordinatensystem der 3D-Textur transformiert werden. In diesem liegt der Ursprung in der vorderen, linken, unteren Ecke (Abb. 11). Die dazugehörige Berechnung wird in (9)

dargestellt. $P_t(x_t, y_t, z_t)$ ist dabei die Partikelposition in Texturkoordinaten und $P_w(x_w, y_w, z_w)$ die Position in OpenGL-Koordinanten.

$$P_t(x_t, y_t, z_t) = \begin{cases} x_t = (x_w + 1) * \frac{gridSize}{2} \\ y_t = (y_w + 1) * \frac{gridSize}{2} \\ z_t = (1 - z_w) * \frac{gridSize}{2} \end{cases} \tag{9}$$

Sobald die Partikelpositionen in Texturkoordinaten vorliegen, kann mit einer dreifach verschachtelten for-Schleife die Berechnung der Distanz beginnen. Der Parameter `influence` ist dabei die zuvor vom Benutzer eingestellte Particle-Influence, welche mit einer Uniform-Variable an den Compute-Shader übergeben wird. In der Schleife werden nun die Distanzen zwischen Partikeln und Eckpunkten berechnet. Danach werden die berechneten Werte auf den Bereich [0,1] skaliert und invertiert. Je näher sich das Partikel am betrachteten Eckpunkt befindet, desto mehr nähern sich die Werte 1 an. Eckpunkte die sich außerhalb der maximalen Distanz befinden, werden nicht betrachtet. Da die Werte Integer bzw. Unsigned-Integer sein müssen, werden sie mit 10.000 multipliziert. Damit wird sichergestellt, dass die Werte bis zur vierten Kommastelle abgespeichert werden.

Mit dem schon erwähnten GLSL-Befehl `imageAtomicAdd()` werden diese auf die 3D-Textur aufaddiert.

Quellcode 4: Berechnung der Partikelverteilung

```
//Berechnung des am nächsten liegenden Eckpunktes
ivec3 nCorner = ivec3(particlePos.x + 0.5, particlePos.y + 0.5,
                      particlePos.z + 0.5);

for(int x = -(influence-1); x < influence; x++){
  for(int y = -(influence-1); y < influence; y++){
    for(int z = -(influence-1); z < influence; z++){
      //Berechnung des betrachteten Eckpunktes
      ivec3 corner = ivec3(nCorner.x+x, nCorner.y+y, nCorner.z+z);

      //Berechnung der Distanz
      vec3 temp = particlePos - vec3(corner.x, corner.y, corner.z);
      float distance = sqrt(dot(temp, temp));
      float normalDistance = 1.0f - (abs(distance) / maxDistance);
      int addData = int(normalDistance * 10000);

      //Verwendung von Image-Load-Store
      barrier();
      imageAtomicAdd(destTex, corner, addData);
    }
  }
}
```

Im dritten Compute-Shader wird ebenfalls, wie im ersten der drei Compute-Shader, für jeden Wert der 3D-Textur eine Instanz aufgerufen. Dabei werden die

Werte aus der Textur mit dem Datentyp `GL_R32UI` ausgelesen, mit 10.000 dividiert und in eine zweite 3D-Textur mit dem Datentyp `GL_R32F` gespeichert. Dadurch erhält man eine Textur mit Fließkommazahlen. Diese wird an den im folgenden Kapitel vorgestellten Geometry-Shader übergeben. Durch die Umwandlung in Fließkommazahlen können die Werte der Textur ohne Umwandlung in den folgenden Shadern verwendet werden. Zum anderen wird das vom Benutzer eingestellte Isolevel, welches im nächsten Kapitel vorgestellt wird, klein gehalten.

3.5.2 Generierung einer Oberfläche

Die Oberflächengenerierung wird mit Marching-Cubes realisiert, da verflüssigtes Metall eine klar definierte Oberfläche besitzt. Der komplette Marching-Cubes Algorithmus wird dabei auf der Grafikkarte mithilfe von Geometry-Shadern durchgeführt. Als Eingabe für den Vertex-Shader werden $Gridsize^3$ Vertices erzeugt. Jeder Vertex repräsentiert dabei eine Zelle im Gitter. Dadurch wird gewährleistet, dass der Vertex-Shader und der darauf folgende Geometry-Shader einmal für jede Zelle aufgerufen wird.

Quellcode 5: Pass-Through-Vertex-Shader

```
#version 420

in vec4 positionAttribute;

void main(){
    gl_Position = positionAttribute;
}
```

Der Vertex-Shader ist ein Pass-Through-Vertex-Shader und hat sonst keine Aufgabe. Neben den Vertices benötigt der Geometry-Shader noch weitere Eingaben, um eine Oberfläche zu generieren. Zum einen dient die 3D-Textur mit der Partikelverteilung aus dem vorigen Kapitel als Eingabe. Mithilfe dieser Werte wird bestimmt, welche Zelleckpunkte innerhalb und welche außerhalb der Oberfläche liegen.

Eine weitere Eingabe ist die Lookup-Tabelle, für die zu zeichnenden Dreiecke. Diese wird als fest definierte 2D-Textur an den Shader übergeben[9]. Da eine Zelle acht Eckpunkte besitzt und ein Eckpunkt entweder innerhalb oder außerhalb liegen kann, ergeben sich $2^8 = 256$ Möglichkeiten, die in der 2D-Textur definiert sind. Für jede Möglichkeit liegt eine Liste von Knotenpunktnummern vor, mit deren Hilfe die Dreiecke gezeichnet werden können.

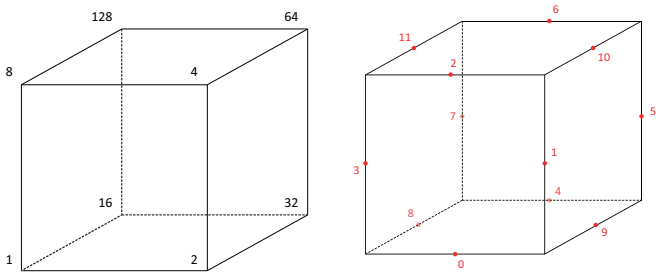
Weiterhin wird der Wert von Gridsize, sowie das Isolevel benötigt und mit Uniform-Variablen übergeben. Letzteres bestimmt den Schwellwert, ab wann ein Eckpunkt in der Oberfläche liegt. Das Isolevel kann dabei vom Benutzer während der Laufzeit dynamisch verändert werden. Verschiedene Werte von Isolevel sind in Abbildung 13 dargestellt.

[9] übernommen aus der Implementation von Cyril Crassin [Cyr06]

Im Geometry-Shader wird zuerst die Vertexposition in das Texturkoordinatensystem transformiert (Kapitel 3.5.1, Gleichung 9), um die betrachtete Zelle zu ermitteln. Als nächstes muss ermittelt werden, welche Eckpunkte der Zelle innerhalb und außerhalb liegen. Dazu werden die Werte der Eckpunkte in der entsprechenden 3D-Textur mit dem GLSL-Befehl `texelFetch3D()` ausgelesen und mit dem eingestellten Wert für Isolevel verglichen.

$$Eckpunktwert \geq Isolevel \tag{10}$$

Wenn (10) gilt, liegt der betrachtete Eckpunkt innerhalb der Oberfläche. Dabei werden die acht Eckpunkte mit 8-Bit, also für jeden Eckpunkt 1-Bit (Abb. 12a), repräsentiert. Wenn der Eckpunkt innerhalb liegt, wird das entsprechende Bit auf 1 gesetzt. Dies geschieht mit Bitweisem Oder, welches in GLSL durch den Operator „|“ realisiert wird. Durch den so erhaltenen Zellenindex zwischen 0 und 255, können direkt die Nummern der Knotenpunkte (Abb. 12b), der zu zeichnenden Dreiecke, aus der Lookup-Tabelle entnommen werden.



(a) **(b)**

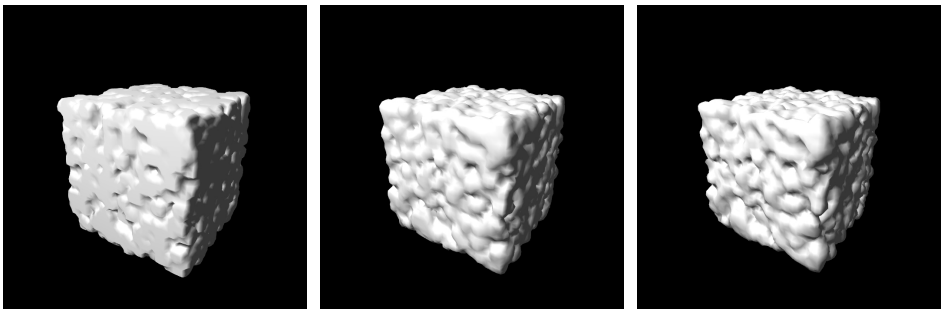
Abbildung 12: Darstellung einer Zelle im Gitter (a) Bitwerte für die Eckpunkte (b) Knotenpunkte für die Dreiecke mit entsprechenden Nummern

Die Knotenpunkte liegen dabei auf den Kanten der Zelle. Wenn genau einer der beiden Eckpunkte einer Kante die Bedingung (10) erfüllt, liegt auf der Kante ein Knotenpunkt. Dessen Position wird mithilfe von linearer Interpolation der beiden Eckpunktwerte und dem Isolevel berechnet.

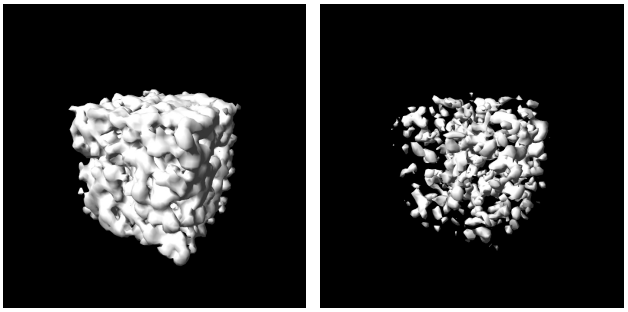
Quellcode 6: Berechnung der Knotenpunkte

```
vec3 vertexInterp(vec3 v0, vec3 v1, float v0Val, float v1Val){
  return mix(v0, v1, (isolevel - v0Val) / (v1Val - v0Val));
}
```

Im Quellcode 6 wird die dazu gehörige Funktion dargestellt. `v0` und `v1` sind die Positionen der Eckpunkte. `v0Val` und `v1Val` sind deren Werte. Wenn die Knotenpunkte nicht interpoliert, sondern immer in der Mitte einer Kante platziert werden (wie im Beispiel Abb. 12b), erhält man Ergebnisse wie in Abbildung 14. Die Positionen aller zwölf Knotenpunkte werden in einem `vec4()` Array gespeichert. Die Ausgabe des Geometry-Shaders ist ein `triangle_strip` bestehend aus maximal 16 Vertices pro Zelle.



(a) 0,1 **(b)** 0,5 **(c)** 1



(d) 3 **(e)** 5

Abbildung 13: Verschiedene Werte für das Isolevel (Gridsize = 64)

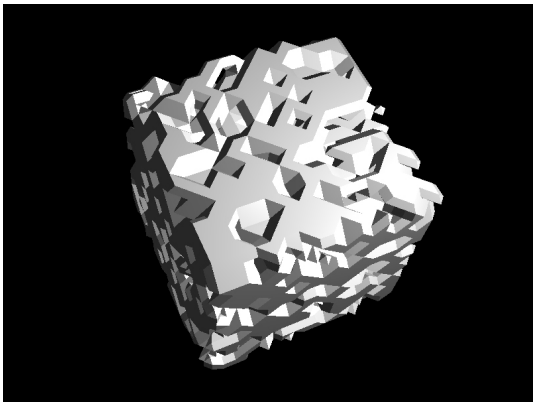


Abbildung 14: Ergebnis von Marching-Cubes ohne Interpolation (Beleuchtung: Flat-Shading)

3.5.3 Randbehandlung unter Verwendung von Marching-Squares

Im vorigen Kapitel wurde eine Oberfläche mithilfe von Marching-Cubes generiert. Es wird jedoch keine Oberfläche an den sechs Randflächen der Bounding-Box generiert. Um dies zu gewährleisten, wird Marching-Squares eingesetzt. Dies geschieht in einem eigenen Geometry-Shader. Das hat zum einen den Vorteil der Übersichtlichkeit und zum anderen kann die Randbehandlung sehr leicht wieder deaktiviert werden. Der Vertex-Shader ist dabei derselbe, wie bei der Oberflächengenerierung aus dem vorigen Kapitel. Als Eingabe für den Vertex-Shader, werden die gleichen Vertices verwendet. Es wird eine andere Lookup-Tabelle der zu zeichnenden Dreiecke für Marching-Squares verwendet. Diese wird ebenfalls als 2D-Textur an den Geometry-Shader übergeben und besitzt nur 16 verschiedene Kombinationen.

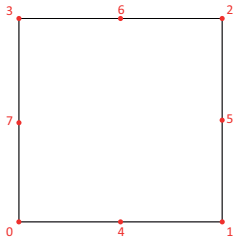


Abbildung 15: Marching-Squares Knotenpunkte mit entsprechenden Nummern

In dieser werden anstatt zwölf verschiedener Knotenpunkte nur noch acht benötigt, um die Dreiecke zu zeichnen (Abb. 15). Die Knotenpunkte 4-7, werden dabei ebenfalls durch Interpolation ermittelt (Kapitel 3.5.2 Quellcode 6).

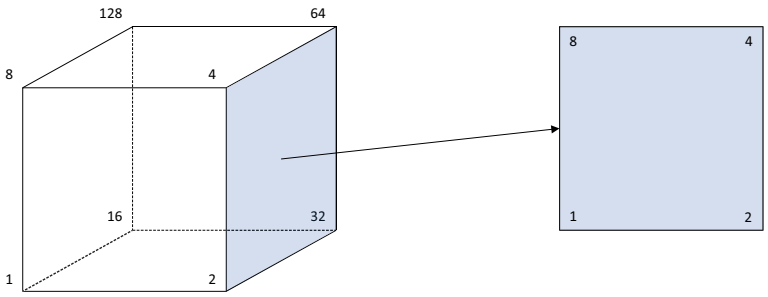


Abbildung 16: Beispiel für Zellen am rechten Rand der Bounding-Box

Im Geometry-Shader wird geprüft, ob die betrachtete Zelle an einer der sechs Randflächen der Bounding-Box liegt. Wenn dies der Fall ist, wird eine von sechs Funktionen für die entsprechende Fläche aufgerufen. Wenn sich die Zelle an einer Kante oder Ecke befindet, werden dementsprechend zwei oder drei der sechs Funktionen aufgerufen. In den Funktionen werden schließlich vier Eckpunkte, die zusammen ein Rechteck bilden, aus den 8 Eckpunkten der Zelle ausgewählt. Der Wertebereich wird von 8-Bit auf 4-Bit reduziert und die Werte der Eckpunkte ge-

ändert (Beispiel Abb. 16). Dadurch ergibt sich ein Zellenindex zwischen 0 und 16.

Danach werden, wie schon bei Marching-Cubes, durch Texturzugriffe auf die 3D-Textur mit der Partikelverteilung, die Werte der Eckpunkte ermittelt. Diese werden mit dem eingestellten Isolevel verglichen. Wenn Bedingung (10) aus Kapitel 3.5.2 gilt, kann durch Bitweises Oder die entsprechende Ecke auf 1 gesetzt werden.

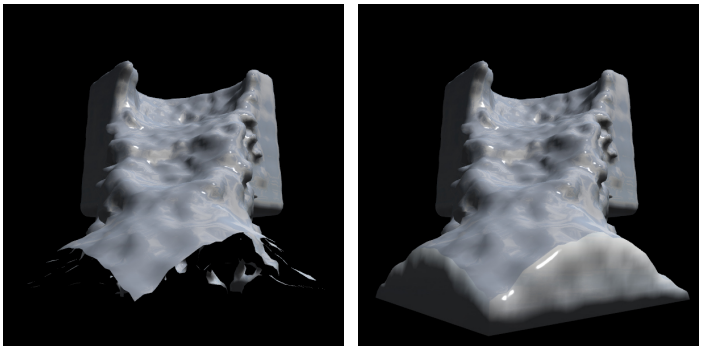
Quellcode 7: Berechnung des Zellenindex durch Bitweises Oder

```
int cellValue(int a, int b, int c, int d){
  int value = 0;
  if(cornerValue(a) >= isolevel) value = value | 1;
  if(cornerValue(b) >= isolevel) value = value | 2;
  if(cornerValue(c) >= isolevel) value = value | 4;
  if(cornerValue(d) >= isolevel) value = value | 8;

  return value;
}
```

Mithilfe des berechneten Zellenindex können die Nummern der Knotenpunkte an der richtigen Stelle in der Lookup-Tabelle entnommen werden. Daraus werden auch hier die Dreiecke generiert. Die Ausgabe ist ebenfalls ein `triangle_strip`.

In Abbildung 17 kann das Ergebnis mit und ohne Randbehandlung verglichen werden.

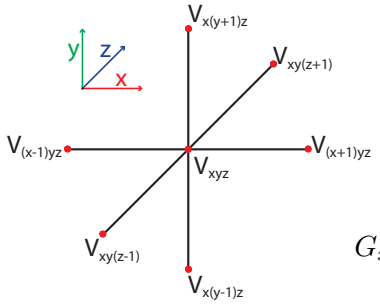


(a) ohne Randbehandlung **(b)** mit Randbehandlung

Abbildung 17: Darstellung der Oberfläche

3.5.4 Beleuchtung der erzeugten Geometrie

Die im vorigen Kapitel erzeugte Geometrie besitzt noch keine Normalen. Diese werden, innerhalb der beiden Geometry-Shader, zeitgleich mit den erzeugten Vertices berechnet. Hierzu wird für jeden erzeugten Vertex, ein Gradient mithilfe der Zentraldifferenz berechnet (Abb. 18).

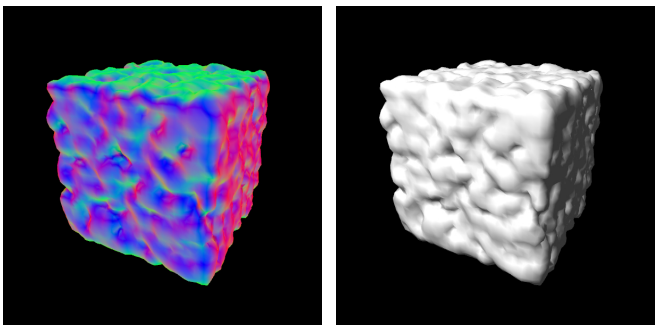


$$G_{xyz} = \begin{pmatrix} G_x \\ G_y \\ G_z \end{pmatrix} = \begin{pmatrix} V_{(x+1)yz} - V_{(x-1)yz} \\ V_{x(y+1)z} - V_{x(y-1)z} \\ V_{xy(z+1)} - V_{xy(z-1)} \end{pmatrix}$$

Abbildung 18: Berechnung des Gradienten G_{xyz} an Position V_{xyz}

Es wird hierfür auf die Volumendaten mit dem GLSL-Befehl `texture` zugegriffen. Im Gegensatz zu `texelFetch3D()`, können dadurch auch interpolierte Zwischenwerte erhalten werden. Der berechnete Gradient wird in einem weiteren Schritt normalisiert, um die Normale des betrachteten Vertex zu erhalten. In einem weiteren Schritt wird die Normale mit der Normalen-Matrix multipliziert und zusammen mit der Vertex- und Lichtposition, an den Fragment-Shader für die Beleuchtung weitergegeben.

Im Fragment-Shader kann die erzeugte Geometrie jetzt beleuchtet werden. Der Fragment-Shader ist dabei für beide Geometry-Shader der gleiche. Hierzu wird das Phong-Beleuchtungsmodell mit einer Lichtquelle verwendet. Dabei kann der Benutzer die ambiente, diffuse und spiegelnde Komponente der Lichtquelle anpassen. Diese werden als Uniform-Variablen an den Fragment-Shader übergeben (Abb. 19).



(a) **(b)**

Abbildung 19: (a) berechnete Normale mithilfe des Gradienten
(b) Beleuchtung mit dem Phong-Beleuchtungsmodell

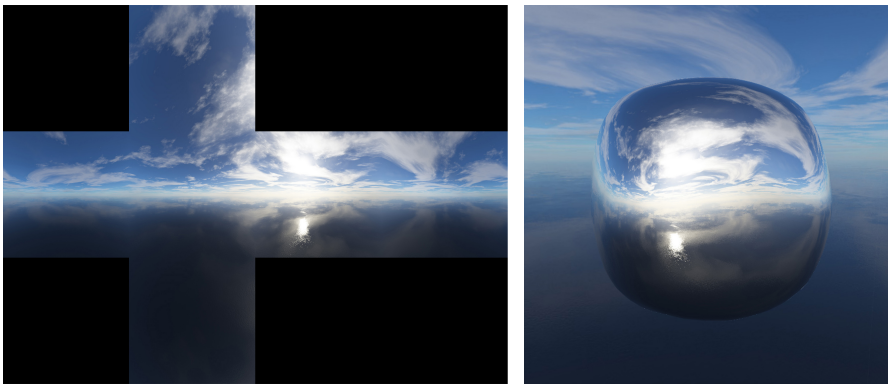
3.5.5 Shader zur Darstellung von Metall

Für eine realistische Darstellung von Metalloberflächen reicht eine einfache Beleuchtung jedoch nicht aus. Deshalb wird eine spiegelnde Oberfläche mit kubischem Environment-Mapping simuliert. Die sechs 2D-Texturen der Umgebung werden in einer Cubemap gespeichert. In den beiden Geometry-Shadern werden dabei für jeden Vertex die entsprechenden Reflektionsvektoren berechnet. Diese zeigen auf die zu reflektierenden Punkte der Cubemap. Um die Reflektionsvektoren zu berechnen, wird mit dem GLSL-Befehl `reflect` gearbeitet. Dieser berechnet, aus der Position der Kamera, der Position des Vertex und der Normalen des Vektors, automatisch den Reflektionsvektor.

Quellcode 8: Berechnung des Reflektionsvektors

```
out vec3 passReflection;
//...
vec4 position  = uniformModel * pos;
passReflection = reflect(vec3(position.xyz) -cameraPos,
                   vec3(normal.xyz));
```

Als nächstes werden der berechnete Vektor sowie die Cubemap an den Fragment-Shader übergeben. Mit dem Reflektionsvektor kann nun die zu reflektierende Pixelfarbe aus der Cubemap ermittelt werden. Wenn die so erhaltene Pixelfarbe unverändert für jeden Vertex festgelegt wird, erhalten wir ein spiegelndes Objekt. Punkte der Umgebung werden in der erzeugten Isofläche unverfälscht reflektiert (Abb. 20).

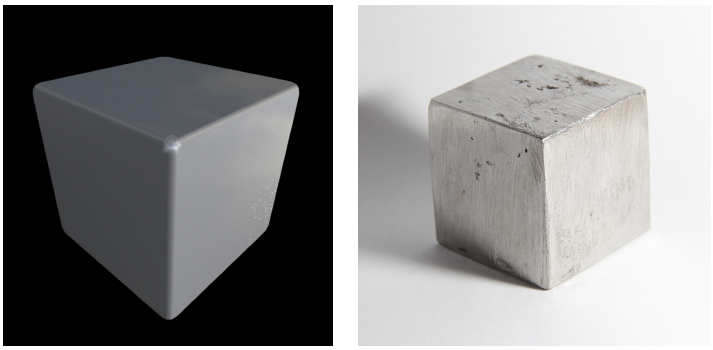


(a) **(b)**

Abbildung 20: (a) Cubemap der Umgebung (b) Ergebnis des Enviroment-Mapping

In den meisten Fällen gleichen jedoch Metalloberflächen keinem perfekten Spiegel. Deshalb wird im Fragment-Shader die Reflektionsfarbe mit einer fest definierten Materialfarbe vermischt. Diese Farbe ist in der Anwendung ein heller Grauton mit RGB(0.7, 0.7, 0.7). Mit dem GLSL-Befehl `mix` werden nun diese beiden Pixelwerte miteinander vermischt, wobei ein weiterer Parameter benötigt wird. Dieser bestimmt, in welchem Verhältnis dies geschieht. Dabei fließen hier

die Materialfarbe mit 70% und die Reflektionsfarbe mit 30% in die finale Farbe ein (Abb. 21). In der Anwendung wird der Parameter als Reflexionsfaktor bezeichnet.



(a) **(b)**

Abbildung 21: (a) finale Metalldarstellung, (b) Würfel aus Lötzinn

Da als Darstellungsgrundlage ein Würfel aus Lötzinn dient, gibt es einen weiteren Effekt der berücksichtigt werden muss. Erhitztes, flüssiges Lötzinn sieht anders aus als festes Lötzinn. Während das verwendete Lötzinn im festen Zustand nur leicht reflektiert hat und eher eine silbergraue Farbe besitzt, reflektiert das erhitzte und somit flüssige Lötzinn wesentlich stärker. Um diesen Effekt auch in der Simulation darzustellen, verändern sich die Materialfarbe und der Reflexionsfaktor in Abhängigkeit von der Temperatur der Partikel.

Um in den Darstellungs-Shadern auf die Temperatur zugreifen zu können, muss auch für die Temperatur ein Volumendatensatz angelegt werden. Dies geschieht auf dieselbe Weise, welche schon in Kapitel 3.5.1 für die Partikelverteilung beschrieben wurde. Um den Aufwand gering zu halten, wird auch dieser Volumendatensatz in den gleichen Shadern erzeugt.

Bedingt durch die Verwendung eines Zellengitters, wird die Temperatur nur angenähert. Dies hat zur Folge, dass die Temperaturdarstellung in Abhängigkeit des Isolevels variert. Bei einem Isolevel von 1.0 können dabei gute Werte erzielt werden. Gleichzeitig entsteht auch ein positiver Effekt durch die Addition von Temperaturwerten in den Volumendatensatz. Die Randbereiche eines Bereichs mit hoher Temperatur besitzen immer niedrigere Werte als das Zentrum. Dadurch wird der Anschein beim erstarren erweckt, als würde dies von außen nach innen geschehen. Die genauen Temperaturwerte können in der Partikeldarstellung betrachtet werden (wie z.B. in Kapitel 3.7.3 Abb. 7 gezeigt).

Sobald die Temperaturdaten vorliegen, können auch diese mithilfe von Uniform-Variablen an die Geometry-Shader übergeben werden. Durch Texturzugriffe werden die entsprechenden Temperaturwerte ausgelesen und an den Fragment-Shader übergeben. In diesem beeinflussen sie die Materialfarbe und den Reflexionsfaktor. Die so erhaltene Metallfarbe wird im letzten Schritt mit den drei Komponenten des Phong-Beleuchtungsmodells aus dem vorigen Kapitel verrechnet.

Quellcode 9: Berechnung der Metallfarbe

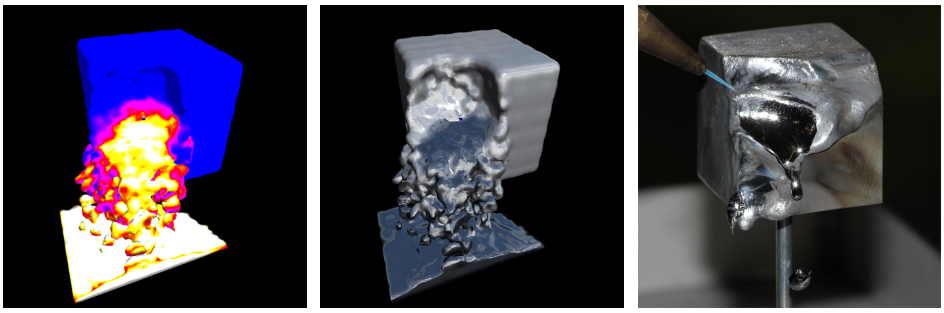
```
//Reflexionsgrad und Metallfarbe des festen Metalls
float reflectFactor = 0.3f;
vec4 materialColor = vec4(0.7, 0.7, 0.7, 1);

//Metallfarbe des flüssigen Metalls auf Basis der Temperatur
if(passTemperature > 0.3f) {
  float material = 1.0f - passTemperature;
  materialColor = vec4(material, material, material, 1);
  reflectFactor = passTemperature;
  if(reflectFactor > 0.6f) reflectFactor = 0.6f;
  if(material < 0.2f) materialColor = vec4(0.2f, 0.2f, 0.2f, 1);
}

//Berechnung der finalen Metallfarbe durch Interpolation
//der reflektierten Umgebung und der Metallfarbe
vec4 reflectColor = texture(CubeMap, passReflection);
vec4 metalColor = mix(materialColor, reflectColor, reflectFactor);

//Beleuchtung mit Phong-Shading
fragmentColor = ((diffuseColor + ambientColor) * metalColor)
                + specularColor;
```

In Abbildung 22 kann man die finale Darstellung sehen. Im linken Bild wurde die Temperatur dargestellt (Kapitel 3.4.3, Tabelle 2). Hier ist zu erkennen, dass die linke vordere Ecke des Würfels sich im heißen, flüssigen Zustand befindet, während der Rest kalt und fest ist. Im mittleren Bild wird das Ergebnis des vorgestellten Fragment-Shader dargestellt. In diesem Bild lässt sich gut der Unterschied in der Darstellung zwischen den beiden Zuständen erkennen. Das rechte Bild zeigt als Referenz einen realen Würfel aus Lötzinn, dessen vordere Ecke durch eine Sauerstoff-Propangas Flamme am Schmelzen ist.



(a) **(b)** **(c)**

Abbildung 22: (a) Temperaturdarstellung, (b) Darstellung des teilweise flüssigen Metalls in der Anwendung, (c) Darstellung in der Realität

3.6 Ablaufdiagramm

In Abbildung 23 ist der vollständige Ablauf der Shader dargestellt. Sobald die Daten nach der Initialisierung der Grafikkarte übergeben werden, werden diese nicht mehr von der CPU verwendet. Die vollständige Berechnung und Aktualisierung geschieht auf der Grafikkarte. Genauso wie die Generierung der Oberfläche mit Marching-Cubes.

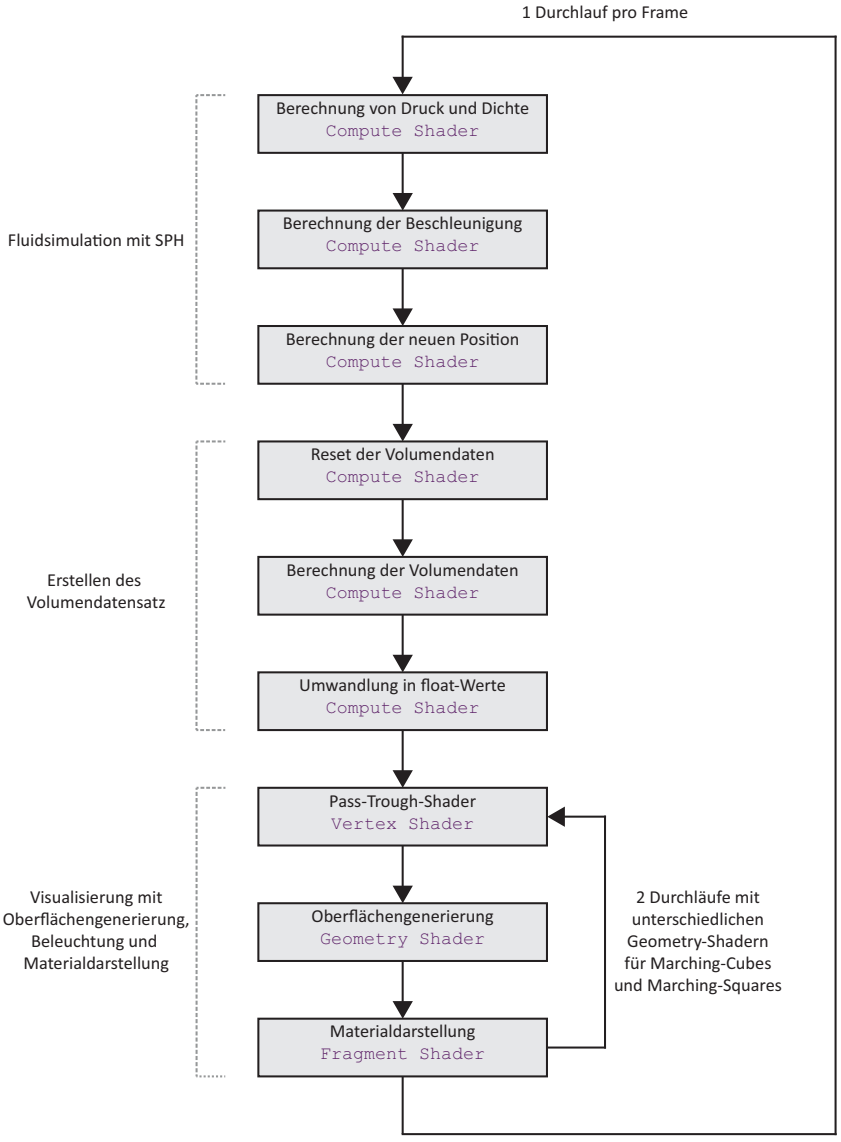


Abbildung 23: Ablaufdiagramm der verschiedenen Shader

3.7 Benutzerinteraktion

3.7.1 Einstellmenüs

Die Menüs wurden mit der freien C/C++ Bibliothek AntTweakBar erstellt. Mit dieser lassen sich schnell und einfach intuitive Menüs erstellen. Sobald die Anwendung gestartet wird, erscheint das erste Einstellmenü für die Grundeinstellungen (Abb. 24a). Diese können nicht während der laufenden Simulation geändert werden. Unter diesen befinden sich, wie in vorigen Kapiteln schon erwähnt, die Partikelanzahl, der Einfluss der Partikel, die Zellenanzahl der Volumendaten und ob die Partikel zufällig oder geordnet initialisiert werden sollen. Wenn der Benutzer mit den getroffenen Einstellungen zufrieden ist, hat er die Möglichkeit die Anwendung zu starten. Neben dem normalen Start-Button, gibt es noch einen weiteren Start-Default-Button. Dieser setzt alle vom Benutzer getroffenen Einstellungen auf einen vordefinierten Wert und startet ebenfalls die Anwendung.

(a) **(b)** **(c)**

Abbildung 24: Die drei verschiedenen Einstellmenüs

Nach betätigen einer der beiden Start-Button, befindet sich der Benutzer in der eigentlichen Simulation. In der Mitte des Fensters werden die Partikel standardmäßig mit dem aus Kapitel 3.5 vorgestellten Verfahren als Isofläche dargestellt. Um die Partikel wird die Bounding-Box mit weißen Linien angezeigt, in dieser sich die Simulation abspielt. Am linken Rand des Fensters befinden sich zwei weitere Menüs.

In dem oberen Menü kann der Benutzer allgemeine Einstellungen vornehmen (Abb. 24b), wie z.B. die Darstellung der Partikel. Weitere Möglichkeiten, neben der Darstellung als Isofläche, sind die direkt Darstellungen der Partikel oder die Darstellungen als Drahtgitter. Des Weiteren lässt sich die standardmäßige Metalldarstellung durch eine farbliche Temperaturdarstellung (z.B. Abb. 22a) austauschen. Auch hier sind mehrere Möglichkeiten vorhanden. Zusätzlich lassen sich verschie-

dene Parameter der Lichtquelle verändern, die Bounding-Box sowie eine Sky-Box ein- und ausblenden.

In dem darunter liegenden Menü lassen sich alle Einstellungen der Fluidsimulation durchführen (Abb. 24c). Darunter befinden sich die verschiedenen SPH-Parameter und die Parameter der Hitzequelle. Im Gegensatz zu dem ersten Menü, lassen sich diese Einstellungen während der Simulation verändern und werden auch sofort auf diese angewendet. Neben den Schaltflächen für die Parameter, befindet sich im unteren Menü ebenfalls ein Reset-Button. Dieser setzt die Partikel zurück ohne die Grundeinstellungen wie z.B. die Anzahl zu verändern. Als letzte Schaltfläche steht dem Benutzer ein Restart-Button zur Verfügung. Dieser setzt ebenfalls die Partikel zurück, ruft aber gleichzeitig das erste Einstellmenü wieder auf, um die Grundeinstellungen verändern zu können.

Die Parameter im Menü für die Grundeinstellungen, sowie die Parameter für die Fluidsimulation, werden dabei in einer Text-Datei gespeichert und stehen dem Benutzer auch beim nächsten Start der Anwendung wieder zur Verfügung.

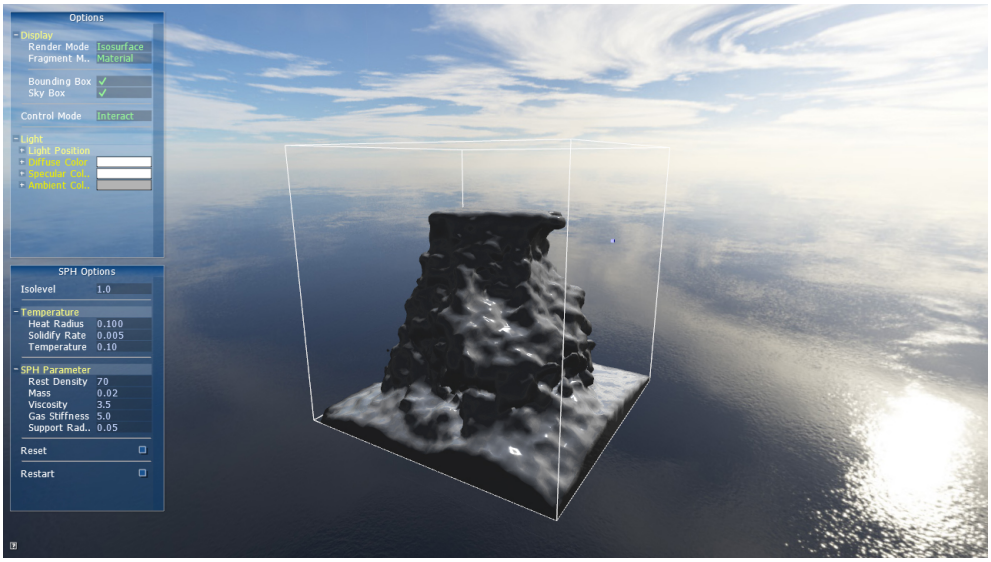


Abbildung 25: Komplettes Interface der Anwendung

3.7.2 Kamerasteuerung

Dem Benutzer stehen zwei mögliche Steuerungs-Modi zur Auswahl. Ein Betrachtungsmodus, in der Anwendung View genannt und der Interaktive Modus, in der Anwendung mit Interact bezeichnet. Zwischen diesen beiden Modi lässt sich mithilfe des Einstellmenüs beliebig oft umschalten. Zusätzlich hat der Benutzer die Möglichkeit, mit der V-Taste umzuschalten. Dies alleine reicht jedoch nicht aus, um mit der Simulation zu interagieren. Dazu genügt ein weiterer Tastendruck auf die C-Taste. Sobald dies geschehen ist, wird der Mauszeiger ausgeblendet und es können vorerst keine Einstellungen im Menü durchgeführt werden. Wie der Benutzer nun mit der Simulation interagiert, hängt vom gewählten Steuerungs-Modus ab. Weiterhin kann der Benutzer jederzeit mit einem weiteren Druck auf die C-Taste, den Mauszeiger wieder einblenden. Dabei wird die Kamerasteuerung deaktiviert. Dies funktioniert in beiden Steuerungs-Modi.

View-Mode In dem Betrachtungsmodus hat der Benutzer die Möglichkeit, mit einer fliegenden Kamera das Modell zu betrachten. Dabei pausiert die Fluidsimulation. Dies hat den Vorteil, dass die Grafikkarte entlastet wird, da die komplette SPH Berechnung wegfällt. Die fliegende Kamera wird mit Maus und Tastatur gesteuert. Mit der Maus kann der Benutzer die Betrachtungsrichtung ändern, wobei die Kameraposition unverändert bleibt. Mausbewegungen nach links und rechts ändern die Blickrichtung in der horizontalen Achse (engl. yaw), während eine Mausbewegung nach vorne und hinten die Blickrichtung in der vertikalen Achse verändert (engl. pitch). Um die Kameraposition zu verändern, wird auf die WASD-Tasten zurückgegriffen. Mit diesen lässt sich die Kamera entlang der Blickrichtung vor und zurück, sowie seitlich bewegen. Bei seitlichen Bewegungen wird die Blickrichtung in dieselbe Richtung und im selben Maß wie die Kameraposition verändert.

Interact-Mode Der zweite zur Verfügung stehende Modus ist die eigentliche Simulation. In diesem Modus kann daher auch die Wärmequelle bewegt und aktiviert werden. Die Funktionsweise hinsichtlich der Wärmequelle wird im folgenden Kapitel erläutert. In Bezug auf die Kamerasteuerung hat der Benutzer weniger Freiheiten als im Betrachtungsmodus. Die Kamera ist immer auf den Ursprung ausgerichtet. Mit den WASD-Tasten lässt sich die Kamera um diesen bewegen. Die Kamera wird dabei mit den Tasten A und D in einer horizontalen Kreisbahn um den Ursprung bewegt. Die Tasten W und S bewegen die Kamera auf einer vertikalen Halbkreisbahn. Zusätzlich hat der Benutzer die Möglichkeit, mit den Tasten Q und E die Kameraposition näher bzw. weiter an den Ursprung zu bewegen.

Das seitliche Neigen der Kamera (engl. roll) wird bei keinem der beiden implementierten Modi unterstützt. Die Bewegung der Blickrichtung in der vertikalen Ebene ist dabei auf einen Winkel kleiner als $180°$ (jeweils nach oben bzw. unten kleiner als $90°$) beschränkt, um zu vermeiden das der Lookat- und der Up-Vektor einen Gimbal-Lock verursachen.

3.7.3 Bewegung der Hitzequelle

Im Interact-Mode hat der Benutzer die Möglichkeit die Hitzequelle zu steuern. Diese wird als kleiner blauer Würfel dargestellt. Die Hitzequelle kann dabei auf zwei Arten gesteuert werden. Zum einen hat der Benutzer die Möglichkeit, den Würfel mit den Pfeiltasten zu navigieren. Die WASD-Tasten werden, wie schon beschrieben, für die Kamerasteuerung verwendet. Dabei wird die Hitzequelle, mit den Pfeiltasten vor und zurück, dementsprechend entlang des Lookat-Vektors bewegt. Mit den Tasten links und rechts wird die Hitzequelle entlang des Vektors bewegt, der sich orthogonal auf der Ebene befindet, die von dem Lookat-Vektor und dem Up-Vektor aufgespannt wird. Um eine Bewegung nach oben und unten zu ermöglichen, werden die beiden Bild-Tasten verwendet. Diese Bewegung findet auf dem Vektor statt, der sich orthogonal auf der Ebene des Lookat-Vektors und des eben erwähnten Vektors zur Bewegung nach links und rechts befindet.

Nachdem einige Benutzer die Steuerung testen konnten, wurde eine weitere Möglichkeit der Steuerung durch Verwendung der Maus implementiert. Die Steuerung der Hitzequelle mit der Tastatur hatte dabei die Nachteile, weniger innovativ zu sein. Des Weiteren ist die Maussteuerung komfortabler und es kann wesentlich schneller mit der Anwendung interagiert werden. Dies liegt auch daran, dass bei der Steuerung mit der Tastatur der Würfel in diskreten Schritten bewegt wird. Bei der Maussteuerung dagegen, kann durch die letzte und die aktuelle Position eine Geschwindigkeit errechnet werden, die in die Bewegung der Hitzequelle mit einfließt. Die Steuerung mit der Tastatur befindet sich jedoch weiterhin in der Anwendung, falls der Benutzer auf eine genauere Steuerung angewiesen ist.

Die Maussteuerung arbeitet mit den gleichen Richtungsvektoren wie schon bei der Tastatur erwähnt. Eine Bewegung nach links bzw. rechts gleicht den entsprechenden Pfeiltasten. Bei einer Mausbewegung nach vorne und hinten wurde eine Bewegung nach oben und unten verwendet, welcher der beiden Bild-Tasten entsprechen. Mit gedrückter rechter Maustaste wird die Bewegung nach oben bzw. unten durch die noch fehlende Bewegung entlang des Lookat-Vektors ersetzt.

Die Hitzequelle bleibt aktiv, solange die linke Maustaste gedrückt gehalten wird. Bei aktiver Hitzequelle verändert sich die Farbe des Würfels von Blau in Rot. Zusätzlich kann die Hitzequelle die Bounding-Box nicht verlassen.

Bei Start der Anwendung und nach jedem Reset wird die Hitzequelle auf den Punkt [0.9, 0.9, 0.9] zurückgesetzt.

4 Ergebnisse

4.1 Echtzeitfähigkeit

Ein wichtiges Kriterium für die vorgestellte Anwendung ist die Echtzeitfähigkeit, da es sich um eine interaktive Simulation handelt. Um eine gute Performance zu gewährleisten, werden alle Berechnungen auf der GPU parallelisiert. Dafür wird eine neuere Grafikkarte benötigt, die OpenGL 4.3 fähig ist. Trotzdem ist eine Fluidsimulation immer noch sehr rechenaufwendig und besitzt in dieser Anwendung einen Aufwand von $\mathcal{O}(n^2)$ mit der Partikelanzahl n.

Im Folgenden wurden zwei Versuche mit drei unterschiedlichen Grafikkarten von Nvidia durchgeführt. Dabei wurden eine leistungsstarke GTX 770 und die Einsteigergrafikkarte GTX 650, jeweils mit Kepler-Architektur, verwendet. Zusätzlich wurde die mobile Grafikkarte GT 525M mit Fermi-Architektur verwendet. Auf dem integrierten Grafikchip HD 3000 von Intel konnte die Anwendung aufgrund von fehlender OpenGL 4.3 Unterstützung nicht gestartet werden.

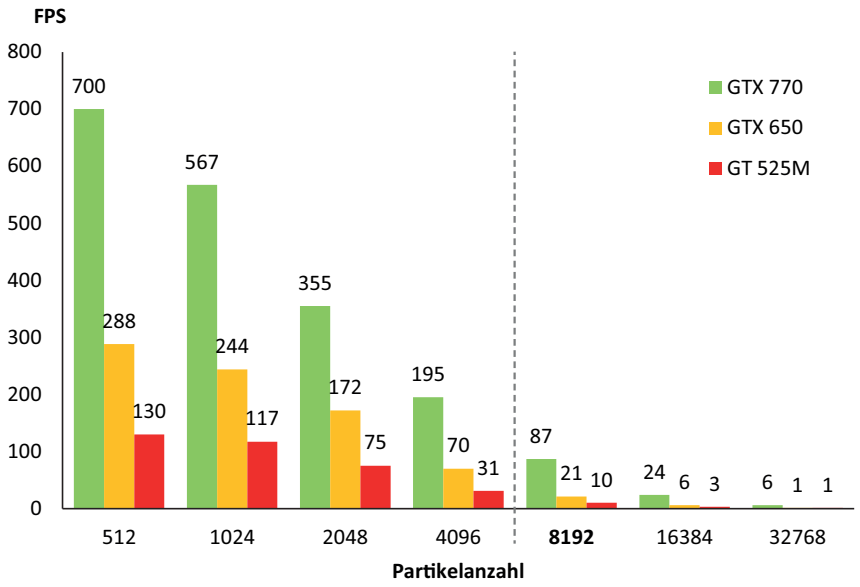


Abbildung 26: FPS in Abhängigkeit der Partikelanzahl auf verschiedenen Grafikkarten

Im ersten Versuch wurde die Anwendung mit verschiedenen Werten für die Partikelanzahl gestartet. Die Oberflächengenerierung wurde bei diesem Test ausgeschaltet (Einstellung in der Anwendung: Render Mode = Points). Es wurde der Steuerungsmodus Interact-Mode verwendet, die Solidify-Rate auf den Wert null gesetzt und alle Partikel verflüssigt. In der Nvidia Systemsteuerung wurde die Vertikale Synchronisation ausgeschaltet und mithilfe des Programms FRAPS[10] die

[10] http://www.fraps.com/

Bilder pro Sekunde (FPS) gemessen. Um in der Simulation gute Ergebnisse zu erhalten, sollte die Partikelanzahl nicht zu gering gewählt werden. Gute Ergebnisse können schon mit Werten von 8192 erzielt werden. Aus Abbildung 26 können die gemessenen FPS-Werte, für verschiedene Werte der Partikelanzahl, entnommen werden.

Die leistungsstarke GTX 770 hat mit einer Partikelanzahl von 8192 keine Probleme, aber schon bei der Einsteigergrafikkarte GTX 650 liegt die Performance unter 30 FPS. Die Notebook Grafikkarte schafft gerade noch 10 FPS und ist somit nur bedingt Echtzeitfähig. Die Folgerung daraus ist, dass eine neuere Grafikkarte über dem Einsteigerniveau benötigt wird, um die Anwendung im Hinblick auf die Fluidsimulation zu verwenden.

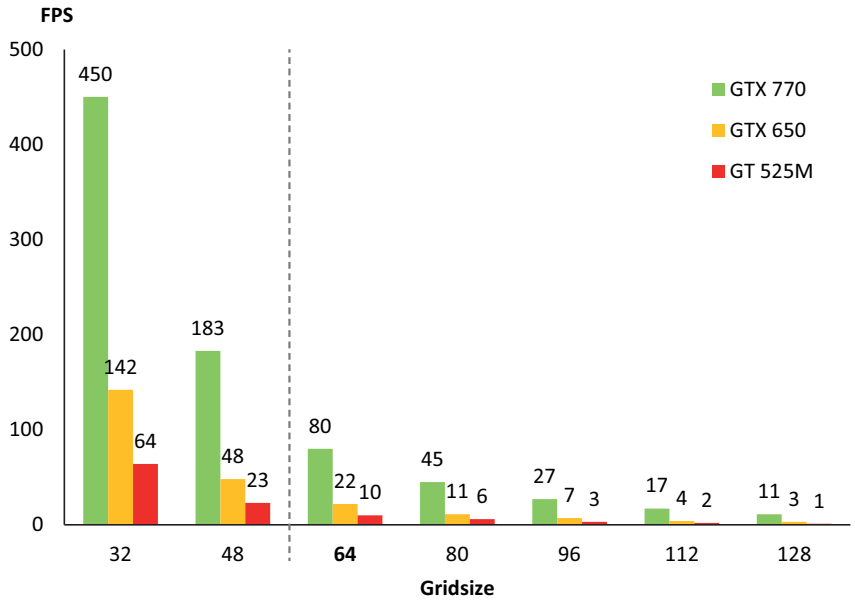


Abbildung 27: FPS in Abhängigkeit der Gridsize auf verschiedenen Grafikkarten

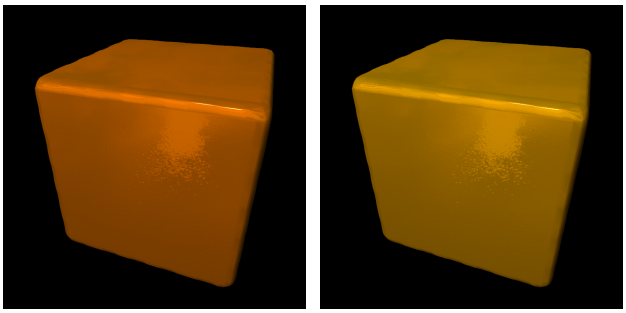
Im zweiten Versuch wurde die Performance der Oberflächengenerierung untersucht. Dabei wurde die Anwendung mit verschiedenen Werten für Gridsize im Steuerungsmodus View-Mode getestet, da hier die Fluidsimulation deaktiviert ist. Die Werte für Partikelanzahl und Isolevel wurden so gewählt, dass eine geschlossene Oberfläche entsteht, bei möglichst geringer Partikelanzahl. Die dadurch erhaltenen Ergebnisse, stellen die maximale Performance für einen Wert von Gridsize dar. Dies liegt daran, dass die Oberfläche so gering wie möglich gehalten wird. Unter realistischen Bedingungen können die Werte, je nach Größe der Oberfläche, abweichen. Die Particle-Influence wurde auf den Wert 2 gesetzt. Die FPS Werte werden wie beim ersten Versuch ermittelt. Hier können in der Simulation gute Ergebnisse mit einem Wert von 64 erhalten werden. Aus Abbildung 27 können die gemessenen FPS-Werte für verschiedene Werte von Gridsize entnommen werden.

Hier zeigen sich ähnliche Ergebnisse wie bei der Partikelanzahl. Die GTX 770 zeigt mit 80 FPS, bei einem Wert von 64, eine sehr gute Performance. Die GTX 650 liegt wieder bei ca. 20 FPS und die GT 525M nur noch bei 10 FPS. Auch in Hinblick auf die Oberflächengenerierung zeigt sich, dass eine Einsteigergrafikkarte wie die GTX 650 Probleme bekommt.

Da in der Anwendung, im Normalfall sowohl Fluidsimulation als auch Oberflächengenerierung aktiviert sind, liegen die FPS Werte deutlich tiefer. Die GTX 770 erreicht z.B. bei 8192 Partikeln und einer Gridsize von 64 noch 35 FPS.

4.2 Bewertung der Anwendung

Wie im vorigen Kapitel ersichtlich wird, kann die Anwendung mit sinnvollen Werten für die Partikelanzahl und die Gridsize nur auf leistungsstarker Hardware ausgeführt werden. Auf schwächeren Systemen können dagegen keine zufriedenstellenden Ergebnisse erzielt werden. Durch weitere Optimierungen der Anwendung (Kapitel 4.3) und den Einsatz von neuerer Technik, kann jedoch davon ausgegangen werden, dass die Anwendung zumindest in Zukunft auf Mittelklasse Systemen brauchbare Ergebnisse liefert. Gleichzeit kann dadurch auch von der Möglichkeit ausgegangen werden, mit leistungsstarken Systemen immer bessere Ergebnisse zu erzielen. Dies geschieht durch die Verwendung von mehr Partikeln und einem höher aufgelösten Modell, durch größere Werte von Gridsize.



(a) Kupfer **(b)** Gold

Abbildung 28: Mögliche Darstellung von anderen Metallen mit angepassten Werten

Des Weiteren ist es möglich, die Anwendung auch für die Darstellung von anderen Metallen zu verwenden. Dies kann durch eine einfache Anpassung des Materialshaders durchgeführt werden, wie z.B. in Abbildung 28 für die Metalle Kupfer und Gold gezeigt wird.

Neben der Darstellung von weiteren Metallen, kann die Anwendung auch zur Simulation von Materialien dienen, die nichts mit Metallen gemeinsam haben. In Abbildung 29 ist ein Beispiel für geschmolzenes Wachs gegeben. Weiterhin kann durch den Einsatz von transluzenten Oberflächen und dem einbinden des Brechungsgesetzes, schmelzendes Eis simuliert werden.

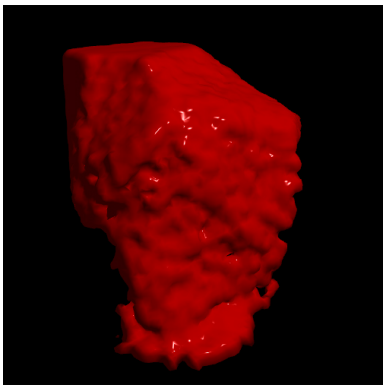


Abbildung 29: Mögliche Darstellung von Wachs mit angepassten Werten

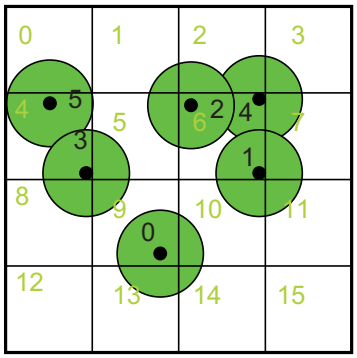
Durch Umkehrung verschiedener Temperaturparameter, wodurch aus der Hitzequelle eine Kältequelle wird, können zudem auch unterschiedliche Simulationen von erstarrenden Materialien dargestellt werden. Dadurch ist die Anwendung auf viele Arten erweiterbar und auch in anderen Bereichen nutzbar.

4.3 Verbesserungsmöglichkeiten

Eine Verbesserungsmöglichkeit der Anwendung ist die Optimierung von SPH. Aktuell beträgt der Aufwand $\mathcal{O}(n^2)$, da jeder Partikel über alle anderen Partikel iteriert. Der Aufwand kann jedoch stark reduziert werden, wenn jeder Partikel nur in seiner Nachbarschaft nach anderen Partikeln sucht. Dazu wird, neben dem Zellengitter von Marching-Cubes, ein weiteres Gitter angelegt. Das Gitter wird im Bereich der Bounding-Box angelegt und ist Uniform. Die Zellengröße muss dabei mindestens so groß sein, wie der Support-Radius h der Partikel. Damit wird gewährleistet, dass für jedes Partikel nur die eigene Zelle und die 26 Nachbarzellen dieser betrachtet werden müssen. In diesen 27 Zellen wird über alle Partikel iteriert, die sich in diesen befinden. Dadurch wird der Aufwand auf $\mathcal{O}(nm)$ reduziert, mit m gleich der durchschnittlichen Anzahl von Partikeln in einer Zelle. Viel größer als der Support-Radius sollte die Zellengröße nicht gewählt werden, da sonst unnötig viele Partikel betrachtet werden. [MMG03]

Um das Zellengitter anzulegen, wird jedem Partikel ein Hash-Wert basierend auf der Zellen-ID zugeordnet. Die einfachste Möglichkeit einen Hash-Wert zu generieren ist es, die lineare Zellen-ID direkt zu verwenden. Die Partikel-ID und der dazugehörige Hash-Wert, werden in einer Liste gespeichert. Diese Liste wird anhand des Hash-Werts sortiert. Dadurch erhält man eine Liste der Partikel-IDs nach Zellen geordnet. In einer weiteren Liste werden die Positionen gespeichert, an denen eine neue Zelle in der zuvor erstellten Liste beginnt. In Abbildung 30 ist ein Beispiel Gitter mit entsprechenden Listen gegeben. [Gre08]

Bei der späteren Berechnung der Fluidsimulation, können dadurch die aktuelle



Index	Unsorted list (cell id, particle id)	List sorted by cell id	Cell start
0	(9, 0)	(4, 3)	
1	(6, 1)	(4, 5)	
2	(6, 2)	(6, 1)	
3	(4, 3)	(6, 2)	
4	(6, 4)	(6, 4)	0
5	(4, 5)	(9, 0)	
6			2
7			
8			
9			5
10			
11			
12			
13			
14			
15			

Abbildung 30: Beispielhaftes Gitter mit Partikeln und zugehöriger Tabelle [Gre08]

Zelle des Partikels und deren Nachbarzellen bestimmt werden. Für die Zelle wird die Startposition mithilfe der zweiten Liste bestimmt, um mit derer in der ersten Liste die Partikel-IDs zu bestimmen, die betrachtet werden müssen.

Ein weiterer Term, der in die SPH Berechnung eingebunden werden kann, ist die Oberflächenspannung. Im inneren der Flüssigkeit sind alle Kräfte ausgeglichen. An der freien Oberfläche dagegen sind die Kräfte asymmetrisch. Diese wirken entlang der Oberflächennormale und minimieren die Krümmung der Oberfläche. Um die Oberfläche lokalisieren zu können, wird ein weiteres Feld angelegt. Dieses nimmt an den Positionen der Partikel den Wert 1 an. Alle anderen Werte werden auf 0 gesetzt. [MMG03]

Ein weiterer Punkt ist das erstarren von Partikeln in der Luft. Aktuell kann es passieren, dass Partikel in der Luft erstarren. Dadurch entsteht der Eindruck, dass die Partikel und die daraus resultierenden Oberflächen, in der Luft zu schweben scheinen. Durch Implementation einer Starrkörperphysik, kann dies verhindert werden. Diese muss auf Partikel wirken, die sich im festen Aggregatzustand befinden und darf keine Auswirkungen auf Partikel im flüssigen Zustand haben.

Bei der Erstellung der Einstellmenüs wird aktuell die AntTweakBar 1.16 Bibliothek verwendet. Die Version 1.16 ist derzeit nicht voll kompatibel mit GLFW 3.0.2. Mehrere Callbacks können nicht mit dieser Version der AntTweakBar verwendet werden. Um die Benutzerinteraktion zu verbessern, bleibt zum einen die Möglichkeit, auf eine neue Version zu warten oder eine andere Bibliothek zum erstellen von Menüs zu verwenden.

5 Fazit und Ausblick

Um Objekte zu simulieren, die zwischen den Aggregatzuständen fest und flüssig wechseln können, wurde gezeigt, dass die Objekte zu jedem Zeitpunkt als Fluid betrachtet werden müssen. Dabei wurde ein Temperatursystem vorgestellt, welches mit einer Hitzequelle arbeitet. Es wurden Möglichkeiten der Benutzerinteraktion, in Hinblick auf das Temperatursystem und die Kamerasteuerung, beschrieben.

Weiterhin wurde gezeigt, dass die Verwendung der GPU nötig ist, um qualitativ gute Ergebnisse echtzeitfähig darzustellen. Dabei stoßen aktuelle Grafikkarten, sowohl im Bereich der Fluidsimulation mit der SPH Methode, als auch in der Generierung einer Oberfläche mit dem Marching-Cubes Algorithmus, schnell an ihre Grenzen.

Mit Bezug auf die vorgestellten Verbesserungsmöglichkeiten und der steigenden Hardwareleistung nach Moore's Law[11], kann jedoch davon ausgegangen werden, dass der echtzeitfähige Einsatz von qualitativ hochwertigen Fluidsimulationen in Zukunft zunimmt. Die Darstellung von Metallen liefert dagegen, schon mit einfachen Ansätzen, gute Ergebnisse. Mit dem in dieser Arbeit verwendeten kubischen Enviroment-Mapping, zur Simulation von spiegelnden Oberflächen, hat eine aktuelle Grafikkarte keine Probleme mehr.

Die in dieser Arbeit vorgestellte Anwendung bildet dabei die Grundlage, Fluidsimulationen für diesen Zweck zu verwenden.

[11] Verdopplung der Transistoren alle zwei Jahre. Abgerufen am 02.01.2014 auf http://www.mooreslaw.org/

Anhang

A Gleichungen

1. Berechnung des Drucks [DG96]

$$p = k(\rho - \rho_0) \tag{11}$$

2. Berechnung der Druckkraft [MMG03]

$$\mathbf{f}_i^{\mathbf{pressure}} = -\nabla p(\mathbf{r}_i) = \sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{12}$$

3. Berechnung der Viskositätskraft [MMG03]

$$\mathbf{f}_i^{\mathbf{viscosity}} = \mu \nabla^2 \mathbf{v}(\mathbf{r}_a) = \mu \sum_j m_j \frac{\mathbf{v}_j + \mathbf{v}_i}{\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{13}$$

B Glättungskerne

1. Poly6-Kern [MMG03]

$$W_{poly6}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

2. Spiky-Kern [MMG03]

$$W_{spiky}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases}$$

$$\nabla W_{spiky}(\mathbf{r}, h) = -\frac{45}{\pi h^6}(h - r)^2 \tag{15}$$

3. Viscosity-Kern [MMG03]

$$W_{viscosity}(\mathbf{r}, h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases}$$

$$\nabla^2 W_{viscosity}(\mathbf{r}, h) = \frac{45}{\pi h^6}(h - r) \tag{16}$$

Abbildungsverzeichnis

Quellcodeverzeichnis

Tabellenverzeichnis

Literatur

[AN07] Max Fischer und Peter Haberäcker Alfred Nischwitz. *Computergrafik und Bildverarbeitung*. vieweg, März 2007.

[Bai13] Mike Bailey. Using gpu shaders for visualization. In *IEEE Computer Society*, May / June 2013.

[Cyr06] Cyril Crassin. Icare3d, 2006. URL: http://www.icare3d.org/codes-and-projects/codes/opengl_geometry_shader_marching_cubes.html.

[DG96] Mathieu Desbrun and Marie-Paule Gascuel. Smoothed particles: A new paradigm for animating highly deformable bodies. *Eurographics Workshop on Computer Animation and Simulation*, 1996.

[GM77] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics - theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, November 1977.

[Gre08] Simon Green. Cuda particles. *Technical Report contained in the CUDA SDK*, June 2008. URL: http://www.nvidia.com.

[KIN10] Y. Dobashi K. Iwasaki, H. Uchida and T. Nishita. Fast particle-based visual simulation of ice melting. *Pacific Graphics, Volume 29, Number 7*, 2010.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithmus. *Computer Graphics, Volume 21, Number 4*, July 1987.

[Luc77] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, December 1977.

[MC02] R. Brooks Van Horn III Greg Turk Mark Carlson, Peter J. Mucha. Melting and flowing. *Proc. ACM SIGGRAPH Symposium on Computer Animation*, 2002.

[MMG03] David Charypar Matthias Müller and Markus Gross. Particle-based fluid simulation for interactive applications. In *Eurographics/SIGGRAPH Symposium on Computer Animation*, 2003.

[Mon92] J. J. Monaghan. Smoothed particle hydrodynamics. *Annu. Rev. Astrophys*, 1992.

[Ram07] Rama Hoetzlein. r.c. hoetzlein, 2007. URL: http://www.rchoetzlein.com/eng/graphics/fluids.htm.

[Ree83] William T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *Computer Graphics, Volume 17, Number 3*, July 1983.

[The97] The Khronos Group. Opengl, 1997. URL: http://www.opengl.org.

[XWK03] Wei Li Xiaoming Wei and Arie Kaufman. Melting and flowing of viscous volumes. *CASA '03 Proceedings of the 16th International Conference on Computer Animation and Social Agents*, 2003.