

Simulation von Rauch

Bachelorarbeit

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

vorgelegt von
Sebastian Gaida

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Zweitgutachter: Bastian Kray MSc.
(Institut für Computervisualistik, AG Computervisualistik)

Koblenz, im September 2019

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. ☐ ☐

.....
(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1	Vorwort	2
2	Einleitung	3
3	State of the Art	3
3.1	Vektorfeldverfahren	4
3.2	Partikelsystem	5
4	Rauchsimulation	6
4.1	Gewichtungsfunktionen	9
4.2	Dichte	11
4.3	Druck	12
4.4	Viskosität	13
4.5	Auftrieb	13
4.6	Wirbelkraft	14
4.7	Update Kräfte	15
5	Implementierung	16
6	Beschleunigung	17
6.1	gridbasierte Nachbarschaftssuche	17
6.2	Speicherverfahren	19
6.3	Sortiervverfahren	20
6.4	Countingsort	21
6.5	Vergleich	25
7	Ergebnis	26
8	Fazit	26

Abstract

In dieser Arbeit wird auf die realistische Simulation von Rauch eingegangen. Dabei bezieht sich die Arbeit hauptsächlich auf die Simulationen von Müller et al.[MCG03] und Ren et al.[RYY⁺16]. Die Simulation wurde mittels C++, der OpenGL und Compute-Shadern erstellt. Hierbei wurde das SPH-Verfahren genutzt und die Möglichkeiten zur Beschleunigung auf der GPU untersucht.

This paper deals with the realistic simulation of smoke. The work refers mainly to the simulations of Müller et al.[MCG03] and Ren et al.[RYY⁺16]. The simulation was created using C++, the OpenGL and compute shaders. Here the SPH method was used and the possibilities to accelerate it on the GPU were investigated.

1 Vorwort

Vor dem Beginn der vorliegenden Bachelorarbeit möchte ich mich zunächst bei einigen Personen bedanken die mich während der Arbeit unterstützt haben.

Zunächst einmal bedanke ich mich bei Prof. Dr.-Ing. Stefan Müller und Bastian Kray MSc. für die großartige Betreuung meiner Arbeit.

Außerdem möchte ich mich bei Pascal Bendler bedanken, der mich tatkräftig beim debugging unterstützt hat.

Ein großes Dankeschön geht auch an den Freund, der mich immer wieder dazu motiviert hat weiter zu arbeiten und nach alternativen Möglichkeiten zu suchen.

2 Einleitung

Das Ziel dieser Arbeit ist es eine möglichst physikalisch korrekte Rauchsimulation zu implementieren. Dazu nutzen wir, dass sich Rauch wie ein Fluid verhält [Sta03], dabei wird die Simulation der physikalischen Basis von Fluiden angenähert. Hierbei wird in dieser Implementation ein Partikelsystem zur Berechnung der physikalischen Eigenschaften genutzt. Echtzeitanwendungen wie die Unity-Engine bieten eine Partikelsimulation an, jedoch beschränkt sich diese lediglich auf das Ausstoßen von Partikeln. Dabei können Partikelinteraktionen, sowie Verhaltensmuster nicht bearbeitet werden.

Die GPU eignet sich besonders gut zum berechnen parallelisierbarer Rechenoperationen, da sie im Vergleich zur CPU, die nur wenige Kerne besitzt, über tausend Kerne verfügt, die zwar nicht so leistungsfähig sind wie die der CPU, aber dennoch einen signifikante Steigerung der Leistung bieten.

In der Arbeit wird auch darauf eingegangen den genannten Aufwand zu minimieren, dazu wurden zwei Verfahren zur Beschleunigung des Partikelsystems, auf der GPU, implementiert und gegenübergestellt.

Für die Implementation wurde OpenGL genutzt, welches das programmieren auf der GPU deutlich vereinfacht und seit der Version 4.3 auch das verarbeiten von Daten mit Hilfe von Compute-Shadern unterstützt. Für den schnellstmöglichen Zugriff auf diese Daten wird Speicherplatz, in Form von SSBO, auf der GPU angelegt. Dabei sollte auch auf eine effiziente Nutzung des limitierten Speicherplatzes geachtet werden.

3 State of the Art

Die Simulation von einem Systemen, zur Darstellung von Fluiden, ist ein jahrelange Herausforderung für die Computergrafik. Dabei treten immer wieder die gleichen Problemstellungen auf. Zum einen soll die Simulation physikalisch korrekt sein, um eine für den Beobachter ein möglichst schönes, sowie nachvollziehbarer Ergebnis zu bieten. Schon kleinstes Fehlverhalten können die Immersion zerstören. Andererseits soll das System auch in Echtzeit berechnet werden und dabei auf mögliche Interaktionen reagieren können. Für eine möglichst effiziente Berechnung werden verschiedene Beschleunigungsverfahren verwendet, die ein gutes Ressourcenmanagement in Form der Laufzeit sowie Speicherplatz erfordern.

Die physikalische Grundlage, die Navier-Stokes-Gleichungen, basiert dabei auf den Gleichungen die von Claude Louis Marie Henri Navier und George Gabriel Stokes im 19. Jahrhundert aufgestellt wurden [Wik19]. Diese Gleichungen beschreiben die physikalischen Eigenschaften von Fluiden und werden für die Simulation dieser angewendet. Diese Formeln werden je nach Fluid noch angepasst um speziellere Eigenschaften darzustellen. Diese Simulation wird meist in Form eines rasterbasierenden Verfahren oder eines Partikelsystems implementiert.

3.1 Vektorfeldverfahren

Beim Vektorfeldverfahren wird die Umgebung in gleichgroße Voxels unterteilt, auch bekannt als Voxelgrid oder eulersches Grid. Bei diesem Vektorfeldverfahren betrachtet man die Partikel nicht direkt sondern einen Masse die in Form des Voxels generalisiert wird. Dabei werden Parameter wie Dichte, Druck und Geschwindigkeit in dem jeweiligen Voxel gespeichert. Die Berechnungen lassen sich in Advektion, Druck, Diffusion und Beschleunigung unterteilen. Die Advektion beschreibt dabei den Strömungstransport, das Übertragen der Bewegungskraft auf ein anliegendes Objekt. Druck wiederum beschreibt die Übertragung von Kräften an benachbarte Partikel, wodurch bei einem zu hohen Druck eine Kraft vom Zentrum weg entsteht und wiederum bei einem Unterdruck eine Kraft zum Zentrum hin. Die Diffusion beschreibt die Viskosität des Fluides. Je nach Anpassung der breitet sich das Fluid stark aus wie zum Beispiel Wasser oder weniger stark wie Lava aus. Bei Beschleunigung handelt es sich um externe Kräfte die auf das Fluid einwirken, dies ist vergleichbar mit der Schwerkraft oder einer Windgeschwindigkeit. Zur Beschleunigung zählt man bei den Vektorfeldern aber auch die Wirbelkraft, die bei Rauch die typischen Turbulenzen verursacht und damit einen signifikanten Einfluss auf die Erscheinung hat.

Wegen der physikalisch präziseren Ergebnisse eignet sich diese Verfahren besonders für Strömungsimulationen in Innenräumen [Pes09], da man Kraft dem System zuführt, diese Kraft wird daraufhin eingefärbt und spiegelt dabei das Fluid wieder.

Bei dieser Methode stellt das Lösen der Gleichungen und die Visualisierung der Ergebnisse die größte Schwierigkeit da. Die Visualisierung erweist sich als Hindernis, da in jedem Voxel Kräfte vorhanden sind. Dabei unterscheidet man in dem Grid unter einem gefärbten Teil und einem nicht sichtbaren Teil der meist Luft repräsentiert. Zur Darstellung des Fluides wird meist Volumerendering genutzt. Außerdem ist, wegen der Grid-Architektur des Verfahrens, der Rechenaufwand hoch und lässt sich nur schwer verbessern.

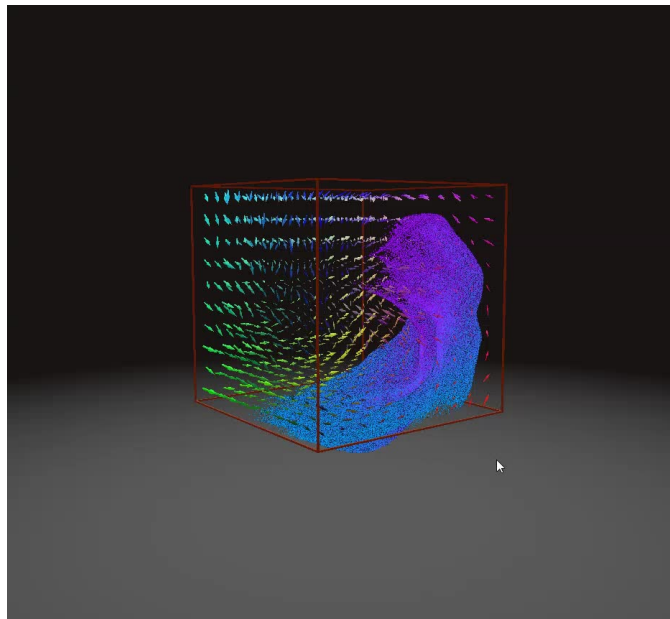


Abbildung 1: Fluidsimulation in Form des Vektorfeldverfahren

3.2 Partikelsystem

Bei dem Verfahren einer Partikelsimulation werden die Partikel einzeln betrachtet, dies bezeichnet man auch lagrangiansches Verfahren. Diese speichern Parameter wie Position und Geschwindigkeit selber ab. Die Berechnungen beschränken sich dabei auf die Dichte, Druck, Viskosität, Auftrieb und Wirbelkraft. Bei den Berechnungen werden die Nachbarpartikel mit einbezogen. Die Dichte beschreibt dabei wie viele Nachbarpartikel Einfluss auf dieses bestimmte Partikel haben und wird als Gewichtung für die Berechnung der Kräfte verwendet. Das typische Verhalten des Fluides, wird aber durch das Zusammenspiel der Kräfte Drucke und Viskosität erzeugt. Dabei sorgt der Druck dafür, dass Partikel sich voneinander wegbewegen und die Viskosität wirkt dem entgegen und führt das Anziehen von Partikel hervor. Der Auftrieb wiederum lässt sich über die Temperatur des Rauches bestimmen, welche je nach Dichte steigt oder sinkt. Zum anderen lässt sich aber die Wirbelkraft nicht so einfach berechnen und stellen somit ein Problem in der Forschung dar. Für die Berechnungen werden die Nachbarpartikel benötigt, welche aber nicht für jeden Partikel bekannt sind und es entsteht ein großer Aufwand, wenn man aus Einfachheit alle Partikel mit einbezieht. Hierbei entstehen viele Möglichkeiten das System zu beschleunigen.

Der Ansatz eines Partikelsystems eignet sich hervorragend zum einbinden in eine Echtzeitanwendung, wie ein Computerspiel oder einer Engine, da man mit der Partikelanzahl die Performance beeinflussen kann. Beim Re-

duzieren der Partikel sollte eine Anpassung der Parameter erfolgen, da dies sonst einen signifikanten Einfluss auf das Verhalten des Fluides hat. Die größten Schwierigkeiten bei einer Rauchsimulation in Form eines Partikelsystems entstehen durch Beschleunigung der Nachbarschaftssuche, sowie den Auftrieb und die Wirbelkraft.

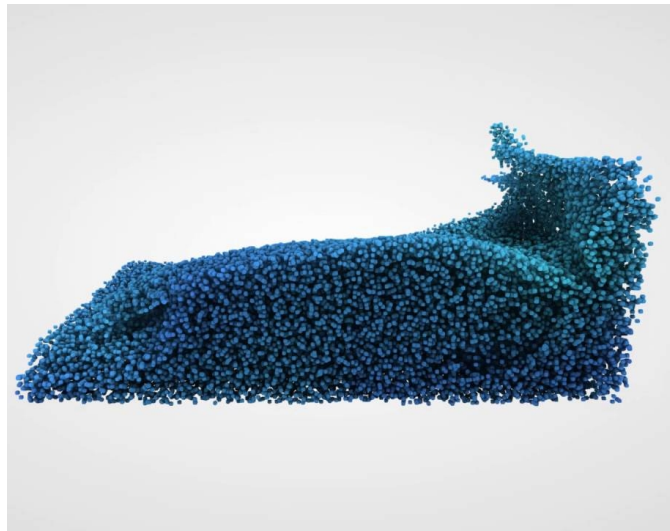


Abbildung 2: Partikelsimulation von Wasser

4 Rauchsimulation

Zur Simulation von Rauch wurde ein Partikelsystem implementiert, dessen physikalische Grundlage auf dem SPH-Verfahren basiert, welches Müller [MCG03] 2003 zur Simulation von Fluiden genutzt hat. Dabei handelt es sich um eine Abwandlung der Navier-Stokes-Gleichungen für die Berechnung der Dichte, Druckes und Viskosität. Diese wurden zur Verwendung in einem SPH angepasst. Der Auftrieb, sowie die Temperaturberechnung, stammen aus einem Paper von Ren [RYY⁺16]. Die größten Probleme bei der Implementation bereitete aber die Wirbelkraft, Ren und Macklin [MMCK14] boten eine Formel zur Berechnung dar, welche aber nicht das gewünschte Ergebnis lieferte.

Die Berechnung der Simulation bedarf vieler Schritte die von einander abhängig sind. In Abbildung 3 werden alle Rechenschritte in einer optimalen Reihenfolge dargestellt. Die Berechnungen lassen sich in 4 Compute-Shader unterteilen.

Zum Verständnis bildet Abbildung 4 alle Variablen da, sowie deren Bedeutung und Format.

```
1 computeshader 1
2     for all particle i do
3         for all neighbor of i do
4             calculate density
5         end for
6         calculate pressure
7     end for
8 end computeshader 1
9 computeshader 2
10    for all particle i do
11        for all neighbor of i do
12            calculate normal
13        end for
14        calculate vorticity
15    end for
16 end computeshader 2
17 computeshader 3
18    for all particle i do
19        for all neighbor of i do
20            calculate pressure force
21            calculate viscosity force
22            calculate vorticity force
23            calculate temperature
24        end for
25        calculate temperature cooldown
26        calculate buoyancy force
27        update velocity
28    end for
29 end computeshader 3
30 computeshader 4
31    for all particle i do
32        apply velocity on position
33    end for
34 end computeshader 4
```

Abbildung 3: Updateschleife der Physik

Symbol	Bedeutung	Format
m_i	Masse des Partikel i	float
r_i	Position des Partikel i	vec3
r_{ij}	Abstandsvektor von $r_i - r_j$	vec3
v_i	Geschwindigkeitsvektor des Partikel i	vec3
h	Radius	float
W_{ij}	Gewichtungsfunktion, kurz für $W(r_i - r_j)$	float
∇W_{ij}	Gradienten-Gewichtungsfunktion	vec3
$\nabla^2 W_{ij}$	Laplace-Gewichtungsfunktion	float
ρ_i	Dichte des Partikel i	float
ρ_0	Ruhedichte im Allgemeinen	float
k	Steifheit des Fluides	float
p_i	Druck des Partikel i	float
$f_i^{pressure}$	Druckkraft des Partikel i	vec3
μ	Viskosität des Fluides	float
ν_i	Viskosität des Partikel i	float
$f_i^{viscosity}$	Viskositätskraft des Partikel i	vec3
T_i	Temperatur des Partikel i	float
D_r	Zeit zum halbieren der Temperatur	float
b	Up-Vektor	vec3
D_c	Wärmeleitfähigkeitsfunktion des Fluides	float
c	Wärmeleitfähigkeit	float
C_b	Auftriebs-Koeffizient	float
$a_{b,i}$	Auftriebsbeschleunigung	vec3
n_i	Normale des Partikel i	vec3
C_N	nutzerdefinierter Schwellenwert	float
y	Zahl die nahezu 0 ist	float
$f_i^{buoyancy}$	Auftriebskraft des Partikel i	vec3
β	nutzerdefinierter Wert	float
ω_i	Wirbelstärke des Partikel i	vec3
f_i^{vortex}	Wirbelkraft des Partikel i	vec3
g	Gravitationskraft	vec3
ext	externe Kraft	vec3
δt	Zeit seit letzter Iteration	float
δ	veränderte Wert im Abstand von δt	

Abbildung 4: Bedeutung aller Symbole der Berechnungen

4.1 Gewichtungsfunktionen

Ein wichtiger Aspekt welchen Einfluss Nachbarpartikel auf den betrachteten Partikel haben sind die Gewichtungsfunktionen. Diese bestimmen den Einfluss über die Entfernung r_{ij} der Partikel zu einander, dabei spielt der Radius als Maximalabstand eine wichtige Rolle. Die Funktionen besitzen alle die Eigenschaft, dass sie symmetrisch sind und beim Erreichen des Radius gegen 0 konvergieren und damit haben weit entfernte Partikel keinen Einfluss mehr. Gewichtungsfunktionen dienen dazu um eine Stabilität in das Partikelsystem zu bringen. Dabei nähern sie sich üblich Ableitungen von bestimmten Funktionen an [MCG03]. Diese Funktionen wurden mit Hilfe von Bastian Kraymer implementiert.

Für das SPH werden die drei folgenden Gewichtungsfunktionen genutzt. Abhängig von den Berechnungen wird eine andere Funktion für die richtige Stabilität des Systems benötigt. In Abbildung 5 ist der Verlauf der Funktionen dargestellt.

Die W_{poly} -Funktion 1 bietet einen weichen abfallenden Verlauf bei ansteigendem Abstand, diese Funktion wird als Standard für jegliche Berechnung genutzt.

Es werden aber für die Berechnungen ebenfalls ein Gradient, sowie der Laplace benötigt. Der Gradient für die Druckberechnung lässt sich durch die Ableitung der für diese Berechnung vorgesehene Funktion berechnen. Mit der W_{poly} -Funktion beim Wert 1, bei der die Druckkraft am höchsten sein sollte, ein nahezu 0 Wert erreicht, welcher dazu führen würde, dass sehr nahe Partikel sich nicht mehr abstoßen würden. Deshalb nutzt Müller Desbrun [DG96] Funktion 2, da diese in der Ableitung 3 Werte liefert, die für die Berechnung entsprechend konvergieren.

$$W_{poly6}(r_{i,j}) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & otherwise \end{cases} \quad (1)$$

$$W_{spiky}(r_{i,j}) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \leq r \leq h \\ 0 & otherwise \end{cases} \quad (2)$$

$$\nabla W_{spiky}(r_{i,j}) = \frac{-45}{\pi h^6} \begin{cases} (h - r)^2 & 0 \leq r \leq h \\ 0 & otherwise \end{cases} \quad (3)$$

$$W_{visc}(r_{i,j}) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \leq r \leq h \\ 0 & otherwise \end{cases} \quad (4)$$

$$\nabla^2 W_{visc}(r_{i,j}) = \frac{45}{\pi h^6} \begin{cases} (h - r) & 0 \leq r \leq h \\ 0 & otherwise \end{cases} \quad (5)$$

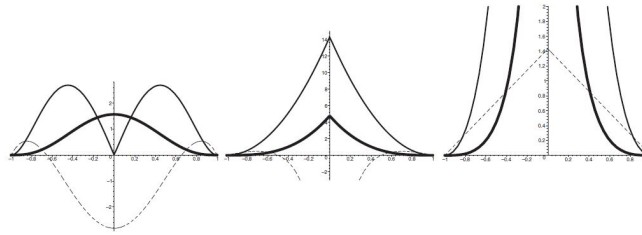


Abbildung 5: Die Gewichtungsfunktionen W_{poly6} , W_{spiky} , W_{visc} nach [MCG03] mit einem Radius von 1

Die W_{spiky} -Funktion, würde aber wiederum in der zweiten Ableitung 0 werden, bei anderen standardmäßigen Gewichtungsfunktionen könnte es sogar dazu führen, dass ein negativer Wert auftritt, der bei der Viskosität einen gegensätzlichen dem gewünschtem Effekt hätte. Deshalb wurde die W_{visc} -Funktion 4 entworfen, dessen Laplace 5 optimal für die Berechnung der Viskosität ist.

```

59 float Weight(vec3 relativePosition)
60 {
61     float relativePosition_2 = dot(relativePosition, relativePosition);
62     float radius_2 = radius * radius;
63     float temp = 315.0 / (64.0 * PI * pow(radius, 9));
64
65     return temp * pow(radius_2 - relativePosition_2, 3) * float(relativePosition_2 <= radius_2);
66 }
67
68 vec3 gradientWeight(vec3 relativePosition)
69 {
70     float temp = -45.0 / (PI * pow(radius, 6));
71
72     float vektorlength = max(length(relativePosition), 0.0001);
73
74     return (temp*relativePosition / vektorlength) * pow(abs(radius - vektorlength), 2.0)
75     *float(vektorlength <= radius);
76 }
77
78 float laplaceWeight (vec3 relativePosition)
79 {
80     float radius_2 = radius * radius;
81     float radius_3 = radius_2 * radius;
82     float radius_6 = radius_3 * radius_3;
83
84     float temp = 45.f / (PI * radius_6);
85     float vektorlength = length(relativePosition);
86
87     return temp * (radius - vektorlength) * float(vektorlength <= radius);
88 }

```

Abbildung 6: Implementation der SPH Kernel

In Abbildung 6 sind die Implementationen dargestellt. Dabei handelt es sich um die Funktionen W_{poly6} in Zeile 59, ∇W_{spiky} in Zeile 68 und $\nabla^2 W_{visc}$ in Zeile 78.

Die Abbruchfunktionen, wie in 1, 3 und 5 zu sehen, wurden dabei durch die Multiplikation mit einer boolesche Variable durchgeführt, welche bei einem Abstand höher als dem Radius dazuführt, dass das Ergebnis 0 wird.

4.2 Dichte

Die Dichte ist eine Variable die für jeden Partikel individuell berechnet wird. Sie beschreibt die Anzahl der Partikel in unmittelbarer Nähe. Außerdem wird sie bei fast jeder folgenden Berechnung benötigt, da sie als Gewichtung des eingenommenen Volumens dient.

$$A_s(r_i) = \sum_j m_j \frac{A_j}{\rho_j} W(r_i - r_j) \quad (6)$$

$$\rho_i(r_i) = \sum_j m_j \frac{\rho_j}{\rho_j} W(r_i - r_j) = \sum_j m_j W(r_{i,j}) \quad (7)$$

Die Dichte, wie in der Gleichung 7 zu sehen, errechnet sich aus der Summe über alle Nachbarn j . Dabei multiplizieren wir die Masse m_j mit der Dichte ρ_j durch die Dichte. Dies wird dann noch mit der Gewichtungsfunktion verrechnet. Die Gleichung ergibt sich aus der Berechnung für Skalaregrößen 6 von Müller [MCG03].

Wie in Abbildung 7 Zeile 90 zu sehen, ist die Masse der Partikel nicht vom jeweiligen Partikel abhängig. In dem Paper von Ihmsen [IOS⁺14] wird eine Formel $m_i = h^3 \rho_0$ für die Masse vorgestellt, diese wird initial durchgeführt und die Masse ist für den Rest der Simulation gleichbleibend. Diese Formel hat sich aber als nicht stabil, in der Implementation, herausgestellt. Deshalb wurde sie als Uniform-Variable implementiert die bei allen Partikel gleich ist. Der Vorteil dieser Implementation ist, dass man die Masse zu Beginn oder auch zur Laufzeit verändern kann. Wobei dem ändern während der Laufzeit abzuraten ist, da es zu einem instabilen Verhalten der Simulation führen kann.

In Zeile 87-89 ist ein If-Abfang eingebaut der es verhindert, dass die Partikel auf sich selbst Einfluss nehmen können. Auf die Begrenzung der Nachbarn, die in die Berechnung eingenommen werden, wird in Abschnitt 6 eingegangen.

```

79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95

for (int x = -1; x <= 1; x++){
    for (int y = -1; y <= 1; y++){
        for (int z = -1; z <= 1; z++){

            neighborGrid = cubeID(inParticle[id].position + vec4(x, y, z, 0));
            int count = 0;
            for (int j = grid[neighborGrid].currentSortOutPut; j < grid[neighborGrid].currentSortOutPut
+ grid[neighborGrid].particlesInGrid && count <= 16; j++){
                if(j == id){
                    continue;
                }
                density += mass * Weight(inParticle[id].position.xyz - inParticle[j].position.xyz);
                count++;
            }
        }
    }
}

```

Abbildung 7: Implementation der Dichtefunktion

4.3 Druck

Der Druckkraft $f_i^{pressure}$ beschreibt das Abstoßen von Partikeln von einander. Die standardmäßige Berechnung für Skalaregrößen 6, ist aber in diesem Fall nicht symmetrisch und würde zu einer instabilen Simulation führen. Da für Partikel i nur der Druck p des Nachbarpartikel j relevant wäre. Dies würde zu einem unterschiedlichen Druckkraft bei der Berechnung des Druckes zwischen den zwei Partikeln führen. Um dieses Problem zu umgehen stellt Müller [MCG03] eine alternative Formel 9 vor, die eine symmetrische Berechnung zwischen den Partikeln ermöglicht. Dabei werden beide Druckwerte addiert, um das Verhältnis aber beizubehalten wird auch die Dichte im Nenner verdoppelt. Zur Gewichtung wird dabei die Gradientenfunktion 3 genutzt und damit der Vektor vom anderen Partikel j weg zeigt wird dieser negiert.

Zum berechnen der Druckkraft muss aber zunächst der Druck berechnet werden. Die Formel 8 von Desbrun [DG96] diesbezüglich wurde angepasst, indem eine Ruhedichte von der eigentlichen Dichte abgezogen wird. Dabei wird die Ruhedichte p_0 als Offset verwendet und hat keinen mathematischen Einfluss auf die Druckkraft [MCG03]. Die Variable k beschreibt dabei die Steifheit und bestimmt wie stark sich der Rauch ausdehnt.

$$p_i = k(\rho - \rho_0) \quad (8)$$

$$f_i^{pressure} = -\nabla p(r_i) = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(r_{i,j}) \quad (9)$$

4.4 Viskosität

Ein wichtiger Aspekt der Physik ist auch die Viskosität. Diese beschreibt den Einfluss der Kraft der umliegenden Partikel, dabei haben aber nicht alle Partikel gleich viel Einfluss auf einander. Dies resultiert daraus, dass die Viskosität eine asymmetrische Kraft ist. Wäre sie jedoch eine symmetrische Kraft würde es dazu führen, dass viele Partikel die alle gleichzeitig in eine Richtung bewegen, sich ins unendliche beschleunigen würden. Dabei würde es bereits reichen, dass ein einzelner Partikel seine Kraft überträgt, da dieser wie bei einem Dominoeffekt, alle Partikel in seiner Umgebung in seine Bewegungsrichtung beschleunigen würde. Um dies zu umgehen modifiziert Müller [MCG03] die SPH-Formel 6 indem der Unterschied des Geschwindigkeitsvektor v in Betracht gezogen wird. Dies sorgt dafür, dass ein Partikel mit einem großen Geschwindigkeitsvektor trotzdem noch alle Partikel in seiner Nähe mit sich reißt, aber sich nicht mehr ins unendliche beschleunigen kann.

Dies lässt den Rauch wirken als würde er sich zusammen ziehen. Das resultiert daraus, dass die Partikel, mit der angepassten Formel, sich auch abbremsen können, da ein schnellerer Partikel beim berechnen der Viskosität mit einem langsamen Partikel eine Viskositätskraft entgegen seiner Bewegungsrichtung erhält.

Wie in 10 wird der Geschwindigkeitsvektor des eigenen Partikel abgezogen. Als Gewichtungsfunktion wird dabei der Laplace5 der W_{visc} 4 genutzt.

$$f_i^{visc} = \mu \sum_j m_j \frac{v_j - v_i}{\rho_j} \nabla^2 W(r_{i,j}) \quad (10)$$

Die Variable μ beschreibt die eigentliche Viskosität. μ ist bei Fluiden wie Honig hoch und bei Rauch sehr gering. In der Implementation hat μ deshalb einen Wert von 0.25.

4.5 Auftrieb

Auftrieb ist eine Kraft die abhängig von der Temperatur des Rauches ist. Oftmals wird diese Temperatur statisch implementiert und sich eine Berechnung dieser gespart. Dabei zu beachten ist, dass die Partikel sich bei einer hohen Dichte gegenseitig aufheizen und bei einer niedrigen abkühlen. Um dieses Verhalten umzusetzen wurde die Auftriebsvariante von Ren [RYY⁺16] implementiert.

Zum anpassen der Temperatur der Partikel wurde die Formel 11 verwendet. Diese betrachtet die Nachbarpartikel und heizt bzw. kühlt die Partikel abhängig der Nachbarn auf oder ab. Bei dieser Formel fehlte aber eine Definition der Funktion D_c , weshalb diese durch die in ?? ersetzt wurde. Diese multipliziert den Temperatur unterschied mit einer Konstanten C die den

Wärmestrom des Rauches beschreibt. γ ist dabei nur eine Variable, die positiv und nahe 0 ist, die das Teilen durch 0 verhindert.

Die Formel 13 beschreibt zudem das Abkühlen von Partikeln die kaum bis keine Nachbarpartikel haben. Dafür wird die Temperatur durch die Zeit geteilt die benötigt wäre um die Temperatur zu halbieren. Welche Partikel dabei betroffen sind wird über die Normale n_i ermittelt, dazu wird die Formel 12 verwendet. Die Normale wird bei einer geringen Dichte ρ größer und wenn diese einen nutzerdefinierten Schwellwert überschreitet führt dies zur Abkühlung des Partikel. Leider lässt sich diese Formel nur bei einer positiven Temperatur anwenden, welches das Abkühlen von bereits negativen Temperatur verhindert.

Wenn die Temperatur errechnet wurde lässt sich diese wie in 14 berechnen. Dabei beschreibt b einen Up-Vektor und C_b den Auftriebs-Koeffizienten.

$$\frac{\delta T_i}{\delta t} = \sum_j \frac{m_j}{\rho_i \rho_j} Dc(T_i - T_j) \frac{(r_i - r_j) \cdot \nabla W_{i,j}}{(r_i - r_j)^2 + \gamma^2} \quad (11)$$

$$n_i = \sum_j \frac{m_j}{\rho_j} \nabla W_{i,j} \quad (12)$$

$$\frac{\delta T_i}{\delta t} = -T_i / D_r \quad (13)$$

$$f_i^{buoyancy} = C_b T_i b \quad (14)$$

```

90 float thermalCon (float temperatureDiff){
91     return heatFlow * temperatureDiff;
92 }
93

```

Abbildung 8: Implementation der Wärmeleitfähigkeit

4.6 Wirbelkraft

Die Wirbelkraft ist die schwierigste zu berechnende Kraft in einem SPH. In dem Paper von Ren [RYY⁺16] wird eine beschrieben und ebenfalls die von Macklin [MMCK14] erwähnt.

Ein Bestandteil der Wirbelkraft ist die Wirbelstärke, die sich aus dieser und dem Geschwindigkeitsgradienten, sowie der Normalen und der Gravitation errechnen lässt 15. Bei dem Geschwindigkeitsgradienten ist aber zu beachten, dass dieser eine Jacobi-Matrix der Geschwindigkeit ist und dort eine Matrixmultiplikation mit der Wirbelstärke stattfindet. Um den Gradienten anzunähern wird wie von Macklin [MMCK14] empfohlen ein SPH-Gewichtungsfunktion 3 zu nutzen. Diese Formel beschreibt den Abstand des Partikel zu der Oberfläche des Rauches, dabei repräsentiert ein hoher

Wert die Oberflächenpartikel und ein kleiner die inneren Partikel. In der Formel 16 wird dann noch die Wirbelstärke der umliegenden Partikel mit dem Abstandsvektor als Vektorprodukt verrechnet, um einen Vektor zu erhalten der eine Verwirbelung innerhalb des Fluides verursachen soll.

$$\frac{\delta \omega_i}{\delta t} = \omega_i \cdot + \nabla v + \beta(n_i \times g) \quad (15)$$

$$f_j^{vortex} = \sum_j (\omega_j \times (r_i - r_j)) W_{i,j} \quad (16)$$

Eine Wirbelkraft zu errechnen die Turbulenzen verursacht ist leider nicht gelungen. Diese hatte lediglich die Wirkung die, dass der Rauch langsamer aufgetrieben ist.

4.7 Update Kräfte

Abschließend werden alle Kräfte zusammen gerechnet und zum Geschwindigkeitsvektor hinzugefügt 17. Ein Spezialfall dabei ist die Gravitation die mit der Dichte verrechnet wird. Zusätzlich können auch externe Kräfte *ext* hinzugefügt werden um bestimmte Umwelteinflüsse zu simulieren. Dabei beschränkt es sich lediglich auf einen Vektor der bei allen Partikeln gleichstark wirkt.

$$\frac{\delta v_i}{\delta t} = f_i^{pressure} + f_i^{viscosity} + f_i^{buoyancy} + f_i^{vortex} + ext + \rho_i g \quad (17)$$

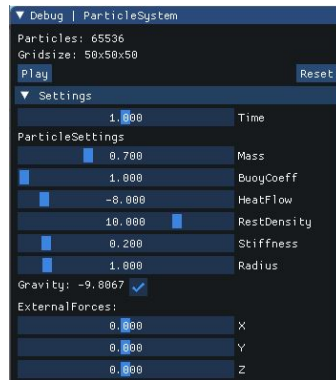


Abbildung 9: Anpassbare Variablen per ImGui

5 Implementierung

Bei der Implementierung wurde darauf geachtet, dass das System einfach und schnell anpassbar ist um das Testen zu vereinfachen. Dazu können einige Parameter während der Laufzeit angepasst werden. Hierfür wurde ImGui [Cor19] verwendet, welches eine UI zur Verfügung stellt um die Parameter anzupassen. Das Anpassen der Variablen während der Laufzeit kann aber zur Instabilität der Physik führen.

Da die Implementierung die GPU verwendet muss darauf geachtet werden, dass die angewendeten Verfahren auch parallel berechenbar sind. Da nicht alle Kerne der GPU gleich schnell arbeiten muss es verhindert werden, dass Werte die von anderen Prozessen noch gelesen wird nicht überschrieben werden. Um dieses Problem zu umgehen wurden zwei identische SSBOs für die Partikel angelegt, welches zwar mehr Speicheraufwand bedeutet, aber bei ca. 65.000 Partikel nur zusätzlich 12 KB beträgt. Bei den zwei SSBOs werden diese im Wechsel zum Lesen und Schreiben genutzt. Dadurch besitzt das lesbare SSBO die zuletzt berechneten Daten und schreibt die neuen in das beschreibbare SSBO, hierbei wird aber immer anfangs das beschreibbare zunächst mit dem lesbaren SSBO überschrieben, damit alle Daten aktuell sind. Es muss dabei aber auch auf eine gerade Anzahl der Aufrufe der Compute-Shader geachtet werden, da sonst im Beginn der Update-Schleife aus dem falschen SSBO gelesen wird.

```
27 layout(std430, binding = 0) readonly buffer buffer_inParticle
28 {
29     Particle inParticle[];
30 };
31
32 layout(std430, binding = 1) writeonly buffer buffer_outParticle
33 {
34     Particle outParticle[];
35 };
```

Abbildung 10: Partikel SSBOs zum Lesen und Schreiben

6 Beschleunigung

Um ein Partikelsystem in einer Engine zu integrieren, muss diese möglichst performant sein, da in einer Engine auch andere Prozesse berechnet werden. Es bestehen mehrere Möglichkeiten ein Partikelsystem zu beschleunigen, eine liegt darin die zu betrachtenden Nachbarpartikel zu begrenzen. Durch die Gewichtungsfunktionen 4.1 werden ohnehin Partikel die außerhalb des Radius h liegen nicht mehr miteinbezogen. Würde man einfach über alle möglichen Nachbarpartikel iterieren, würde ein $O(n^2)$ Aufwand anfallen, hierbei steht n für die Anzahl der Partikel. Dies würde bei 1000 Partikeln ein Aufwand von 1.000.000 Rechenoperationen pro Rechnung bedeuten pro Frame. Dies ließe sich nicht in Echtzeit berechnen geschweige denn in eine Engine integrieren. Deshalb wurden im folgenden Abschnitt die Möglichkeiten, diese Laufzeit zu reduzieren, untersucht und gegenübergestellt.

6.1 gridbasierte Nachbarschaftssuche

Eine Möglichkeit die Laufzeit zu verringern basiert darauf, dass man nur die Nachbarn mit in die Berechnung einbezieht die infrage kommen. Würde man aber alle Partikel miteinander vergleichen und die Nachbarn für einen einzigen Partikel herausfinden und für diese Iteration abspeichern würde dies mit einem immer noch hohem Aufwand so wie einem großen Speicherverbrauch verbunden sein, bei dem nicht klar ist wie viel Speicher benötigt wird, da die Anzahl der Nachbarpartikel nicht bekannt ist. Hoetzlein [Hoe14] stellt dabei eine auf einem Grid basierte Nachbarschaftssuche da. Diese lässt sich in die folgenden 3 Unterpunkte unterteilen.

1. Unterteilen der Welt in gleichgroße Gridbehälter
2. Hinzufügen der Partikel in die Gridbehälter
3. Suche der Partikel in den Nachbarbehältern

Der 1. Punkt wird bereits beim Initialisieren durchgeführt, dabei wird vorher vom Benutzer vorgegeben wie groß das Grid sein soll. Dieser Wert ist zu Laufzeit nicht mehr anpassbar, da das Grid als SSBO angelegt wird. Jedes der Gridbehälter(Grid) besitzt eine eindeutige ID die gleicht mit der `gl_GlobalInvocationID.x` in dem Compute-Shader ist.

Punkt 2 bedarf einer hohen Menge an Speicherplatz, da davon ausgegangen werden muss, dass im Worst-Case-Szenario sich alle Partikel in einem Grid aufhalten. Welches bei einer Gridgröße von $50x50x50$ und ca. 65.000 Partikel 8, 125 GB verbrauchen würde rein an Speicherplatz für die Partikel die in einem Grid vorhanden sind. Dabei handelt es sich bei diesen Zahlen um eine Simulation die später als Standardsimulation betrachtet wird.

Ein Problem dabei ist aber auch, dass die Partikel zunächst zugeordnet werden müssen. Dies erfolgt über die in Abbildung 21 in Zeile 46 zu findende Funktion *cubeID*, welche über die Position des Partikel eine eindeutige GridID ausrechnet. Das hinzufügen der Partikel in die Grids ist je nach Verfahren unterschiedlich und wird genauer in Abschnitt 6.4 und 6.2 erklärt. In Punkt 3 handelt es sich lediglich um ein iterieren über alle Partikel in den Nachbarbehältern, da diese bereits stark eingegrenzt wurden und durch die Gewichtungsfunktionen nur die Partikel betrachtet werden die in dem entsprechenden Radius sind.

In Abbildung 11 sieht man ein vereinfachtes 2D-Grid mit einem Partikel, welcher dem Grid in der Mitte zuzuordnen ist. Bei einem standardmäßigen Radius h von 1, werden alle möglichen Nachbarpartikel die infrage kommen abgedeckt. Hierbei werden alle 9 in 2D oder 27 in 3D Nachbarbehälter betrachtet und über die Partikel in diesen iteriert.

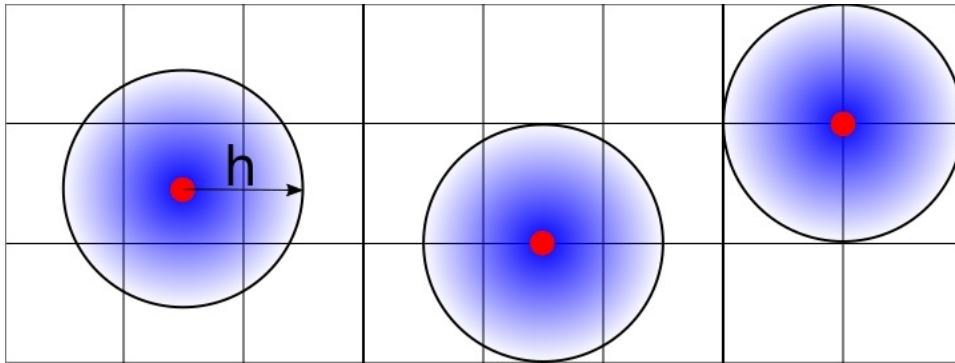


Abbildung 11: Partikel im Grid mit einem Radius von 1

Diese Art der Nachbarschaftssuche besitzt einen Aufwand von $O(nk)$ im optimal Fall, wobei k die Anzahl der Nachbarpartikel ist. Da aber der Speicheraufwand extrem hoch ist kommt diese Art der Nachbarschaftssuche leider nicht in Frage und es wird von Hoetzlein [Hoe14] eine alternative Suche mit einem Sortieralgorithmus vorgeschlagen.

Dabei werden die Partikel den Behältern zugeordnet, diese wiederum speichern die Anzahl der Partikel aller vorherigen Behälter und wie viele Partikel sie selbst beinhalten. Damit wird dann über einen Sortieralgorithmus bestimmt an welcher Stelle die Partikel zu welchem Grid im Speicher gehören. Dann muss nur noch von einem Grid die erste Speicheradresse, sowie die Anzahl der Partikel abgefragt werden und über diese iteriert werden. In Abschnitt 6.4 wird genauer auf ein solches Verfahren so wie die Implementation eingegangen.

6.2 Speicherverfahren

Da beim Speicherverfahren ein großer Aufwand an Speicherplatz auf der GPU benötigt wird, dieser aber begrenzt ist und deshalb nicht für große Simulationen verwendbar ist, wird eine Begrenzung des genutzten Speicherplatzes angewendet. Heinrich [Hei10] beschreibt, dass bei der Implementation von Müllers [MCG03] Fluidsimulation nur 32 Nachbarpartikel für eine korrekte Simulation von Nöten sind, die einen Einfluss auf den betrachteten Partikel haben. Da ein ausgeglichener Einfluss auf diesem Partikel herrschen soll, wurden die Nachbarn nicht auf 32 Partikel beschränkt, sondern auf 16 Partikel pro Grid. Dadurch können maximal 432 Partikel in die Berechnungen mit einbezogen werden. Wobei wie man in Abbildung 11 sehen kann nicht alle Grids durch die Gewichtungsfunktionen 4.1 mit einbezogen werden. Im Durchschnitt haben pro Nachbarschaftssuche ca. 138 Nachbarpartikel Einfluss auf die Berechnung.

Bei der Implementation wurde einfachheitshalber bei den Grids 16 unsigned int Variablen hinzugefügt, die die IDs der Nachbarpartikel beinhaltet. Wie in Abbildung 12 zu sehen, werden die zunächst die Zugehörigkeiten der Partikel zu dem Grid berechnet. Daraufhin wird mit der *count* Variable berechnet, an welche Speicherposition die IDs der Partikel gespeichert wird. Dafür wird wegen den parallelen Prozessen eine Atomic-Funktion genutzt, woraufhin Partikel die einen Wert unter 16 besitzen in dem Grid an der entsprechenden Position gespeichert. Es werden daraufhin bei der Nachbarschaftsbetrachtung nur noch die abgespeicherten IDs abgerufen.

```
45  uint cubeID(vec4 position){
46      return int(floor(position.x) * gridSize.x * gridSize.x + floor(position.y) * gridSize.y + floor(position.z));
47  }
48  void main(void) {
49      uint id = gl_GlobalInvocationID.x;
50      uint temp;
51      uint count;
52      if (id >= particleCount)
53      {
54          return;
55      } else
56      {
57          outParticle[id] = inParticle[id];
58          temp = cubeID(inParticle[id].position);
59          count = atomicAdd(grid[temp].particlesInGrid, 1);
60          outParticle[id].memoryPosition = count;
61          if(count < 16){
62              grid[temp].particles[count] = id;
63              atomicAdd(grid[temp].particleToUse, 1);
64          } } }
```

Abbildung 12: Speichern der Partikel-IDs im Grid

Diese Form der Nachbarschaftsbetrachtung besitzt in der Theorie einen Aufwand von $O(nk)$. Was aber dabei nicht beachtet wurde ist, dass die Partikel-IDs in den Grids sich sehr stark unterscheiden können, wie in Abbildung 13 zu sehen.

Durch diesen unsortierten Speicher entstehen scattered reads, diese Art den Speicher auszulesen ist um einiges langsamer als bei einem sortierten Speicher. Dies entsteht durch, dass lesen von Speicheradressen die nicht zusammenhängend sind sondern einen hohen Abstand besitzen.

Dieses Verfahren ist allerdings noch um einiges schneller als die Brute-Force-Methode.

Grid	20	16	13	20	20	13	16	18	15
Speicherposition	0	1	2	3	4	5	6	7	8

Abbildung 13: Unsortierter Speicher beim Speicherverfahren

6.3 Sortierverfahren

Um das Problem der scattered reads zu beheben, müssen die Speicherpositionen nach dem Grid sortiert werden 14, damit ein Speicherzugriff erfolgen kann der möglichst kompakt ist. Dafür wird aber ein Sortieralgorithmus benötigt. Ein sequentieller Algorithmus würde aufgrund der Architektur der GPU sich dafür nicht eignen, da diese zwar viele Kerne besitzt, die gemeinsam eine große Rechenleistung, aber einzeln nur eine geringe besitzen. Dort wäre es effizienter die Daten wieder auf die CPU zu übertragen und dort sortieren zu lassen.

Grid	13	13	15	16	16	18	20	20	20
Speicherposition	2	5	8	1	6	7	0	3	4

Abbildung 14: sortierter Speicher nach Sortieralgorithmus

Es wird ein paralleler Sortieralgorithmus benötigt, Hoetzlein [Hoe14] stellt dabei zwei Sortierverfahren gegenüber. Der erste ist RadixSort, dieser Sortieralgorithmus bezieht sich auf das betrachten der einzelnen Stellen einer Zahl und sortiert anhand dessen. Dadurch, dass jeder Kern eine einzelne Zahl betrachten und diese dann einordnen kann, mach diesen Sortieralgorithmus parallelisierbar. RadixSort hat einen Aufwand von $O(l \cdot n)$, wobei l für die Anzahl der Stellen steht. Laut diesem Aufwand ist er damit schneller als der zweite Algorithmus CountingSort mit einem Aufwand von $O(n \log(n))$. Jedoch benötigt laut Hoetzlein [Hoe14] RadixSort 15 Kernel calls und CountingSort nur 4 Kernel calls pro Frame. Dadurch ist CountingSort trotz größerem Aufwand der schnellere Algorithmus, auf wessen Theorie und Implementation in Abschnitt 6.4 näher eingegangen wird.

6.4 Countingsort

Countingsort ist ein nicht vergleichsbasierter Algorithmus, dies erlaubt eine parallele Implementation bei der die Laufzeit unabhängig von der Zuordnung des Arrays ist und damit einen festen Aufwand hat. Die einzelnen Schritte des Sortieralgorithmus erfolgen nach Demaine [DD11].

Für den Countingsort wird ein Array mit den zu sortierenden Zahlen wie in der Tabelle 15 benötigt, dabei muss die maximale Größe des Wertes im Array bekannt sein.

Index	0	1	2	3	4
Wert	8	2	5	3	5

Abbildung 15: Ausgangsarray beim Countingsort

Zunächst werden dann in einem Hilfsarray die Anzahl der vorkommenden Werte bestimmt 16. Daraufhin wird die Summe aus allen vorherigen Anzahlen bestimmt und in das Hilfsarray geschrieben 17.

Wert	0	1	2	3	4	5	6	7	8	9
Anzahl	0	0	1	1	0	2	0	0	1	0

Abbildung 16: Anzahl der Werte im Hilfsarray

Wert	0	1	2	3	4	5	6	7	8	9
Anzahl	0	0	1	2	2	4	4	4	5	5

Abbildung 17: Summe der Anzahl der Werte im Hilfsarray

Zu Letzt wird über die Werte des initialen Array iteriert und der Wert an die Anzahl der Summe minus 1 geschrieben, dies beruht darauf, dass die Arrays mit dem Index 0 starten und hat keinen Zusammenhang mit dem folgenden Schritt. Daraufhin wird die Summe der Anzahl bei diesem Wert um 1 verringert, aber bei den folgenden Zahlen nicht angepasst. Dadurch kann eine Zahl mehrfach vorkommen und wird trotzdem richtig einsortiert.

Index	0	1	2	3	4
Wert	2	3	5	5	8

Abbildung 18: vollständig sortiertes Array

Bei der Implementation mussten Anpassungen zwischen dem von Hoetzlein [Hoe14] vorgeschlagenen Algorithmus und der Theorie von Demaine [DD11]. Diese erfolgten für eine schnellere und stabilere Implementation des Sortieralgorithmus, außerdem liefert Hoetzlein bloß einen groben Aufbau des Algorithmus, welcher als Grundlage des in Abbildung 19 zu findenden Aufbau diene.

```

1  computeshader 1
2      for all grid i do
3          reset grid
4      end for
5  end computeshader 1
6  computeshader 2
7      for all particles i do
8          lable Particles to Grid
9      end for
10 end computeshader 2
11 computeshader 3
12     for all grid i do
13         init gridBuffer
14     end for
15 end computeshader 3
16 for log_2(ParticleCount) do
17     computeshader 4
18         for all grid i do
19             calculate PrefixSum
20         end for
21     end computeshader 4
22     computeshader 5
23         for all grid i do
24             update gridBuffer
25         end for
26     end computeshader 5
27 end for
28 computeshader 6
29     for all particles i do
30         rearrange Particles
31     end for
32 end computeshader 6

```

Abbildung 19: Aufbau des Implementierten Countingsort

Der Sortieralgorithmus beginnt mit dem zurücksetzen des Grid-SSBO 20. Dies dient dazu, dass die Berechnung des vorherigen Frames keinen Einfluss hat. Dabei werden alle Variablen bis auf die *id* zurückgesetzt, wobei die *id* und der *previousSortOutPut* Füllervariablen sind um die 16 Byte zu erreichen, sie haben aber noch einen Mehrwert im Debugging.

```

7 struct Grid{
8     unsigned int id;
9     unsigned int particlescount;
10    unsigned int previousSortOutPut;
11    unsigned int currentSortOutPut;
12 };

```

Abbildung 20: SSBO-Struct des Grids

Im Folgenden wird wie in Abbildung 21 in Zeile 60 zu sehen die Anzahl der Partikel in einem Grid gespeichert, die Zugehörigkeit zum Grid wird über die CubeID-Funktion ermittelt. Eine atomicAdd-Funktion wird dabei durchgeführt und es wird der Wert *grid[temp].particlesInGrid* um 1 erhöht. In *outParticle[id].memoryPosition* wird der Wert vor der Addition gespeichert und dieser sagt aus als wievielter sich dieser Partikel im Grid registriert hat. Die Abwandlung mit dieser Variable spart beim *rearrangeParticles*-Schritt kostbare Laufzeit.

```

44 uniform ivec4 gridSize;
45
46 uint cubeID(vec4 position){
47     return int(floor(position.x) * gridSize.x * gridSize.x + floor(position.y) * gridSize.y + floor(position.z));
48 }
49
50 void main(void) {
51     uint id = gl_GlobalInvocationID.x;
52     uint temp;
53     if (id >= particleCount)
54     {
55         return;
56     } else
57     {
58         outParticle[id] = inParticle[id];
59         temp = cubeID(inParticle[id].position);
60         outParticle[id].memoryPosition = atomicAdd(grid[temp].particlesInGrid, 1);
61     }
62 }

```

Abbildung 21: registrieren der Partikel im Grid mit CubeID-Funktion

Zum Berechnen der Prefixsum wird der parallele Scan von Navie nach Harris [HSO07] implementiert. Dieser berechnet die Prefixsum über die Summe der bisherigen Anzahl der $i - 2^d$ Nachbarn, wobei $d = 0$ to $\log_2(n) - 1$.

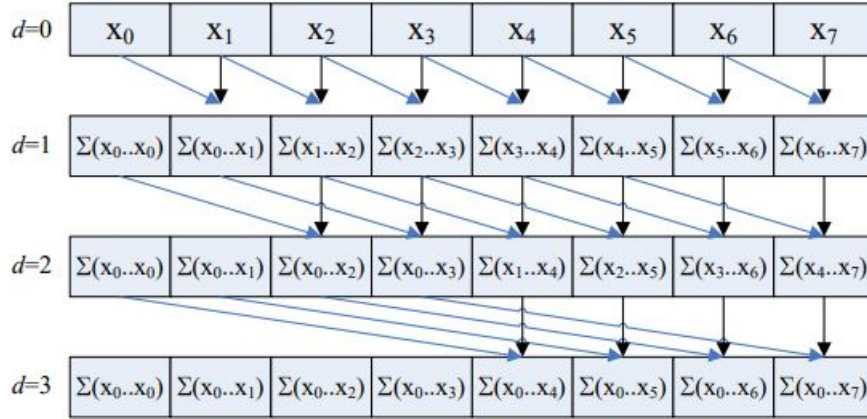


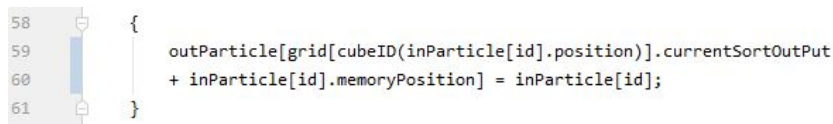
Abbildung 22: paralleler Prefixsum Scan von Navie nach Harris [HSO07]

Um den ersten Schritt zu vereinfachen und keinen Zugriff auf Variablen zu haben, die überschrieben werden könnten, wird der Algorithmus damit initialisiert, dass die Anzahl der Partikel die sich im Grid befinden in den *currentSortOutPut* sowie in den Gridbuffer geschrieben wird.

Der Gridbuffer dient in dem Fall als temporärer Buffer, der bloß eine Float-Variable speichert, damit nicht gleichzeitig aus einem Buffer die gleiche Variable gelesen und beschrieben wird. Dafür werden die ComputeShader zu Berechnung der Prefixsum und zum updaten des Gridbuffers im wechsell $\log_2(Particleanzahl) - 1$ durchgeführt. Bei der Prefixsum wird aus dem Gridbuffer die aktuelle Summe plus die Summe des $i - 2^d$ Nachbarn addiert und in dem *currentSortOutPut* des Grid-SSBOs gespeichert. Daraufhin wird der Gridbuffer aktualisiert und es wird die Summe die in der *currentSortOutPut* Variable steht übertragen.

Im letzten Schritt müssen die Partikel nur noch neu angeordnet werden. Dies müsste nach dem Algorithmus für jedes Grid sequentiell ablaufen, damit sich die Partikel nacheinander an die richtige Position schreiben. Dies entspricht dem Schritt von 17 zu 18.

Um diesen sequentiellen Prozess zum umgehen wurde aber im Vorhinein bereits in der *outParticle[id].memoryPosition* die Position des Partikels im Grid bestimmt. Deshalb können die Partikel wie in 23 zu sehen, sich bloß an die entsprechende Speicherstelle schreiben. Dadurch befinden sich nun alle Partikel die in einem Grid sind in aufeinanderfolgenden Speicherplätzen und verhindert damit scattered reads.



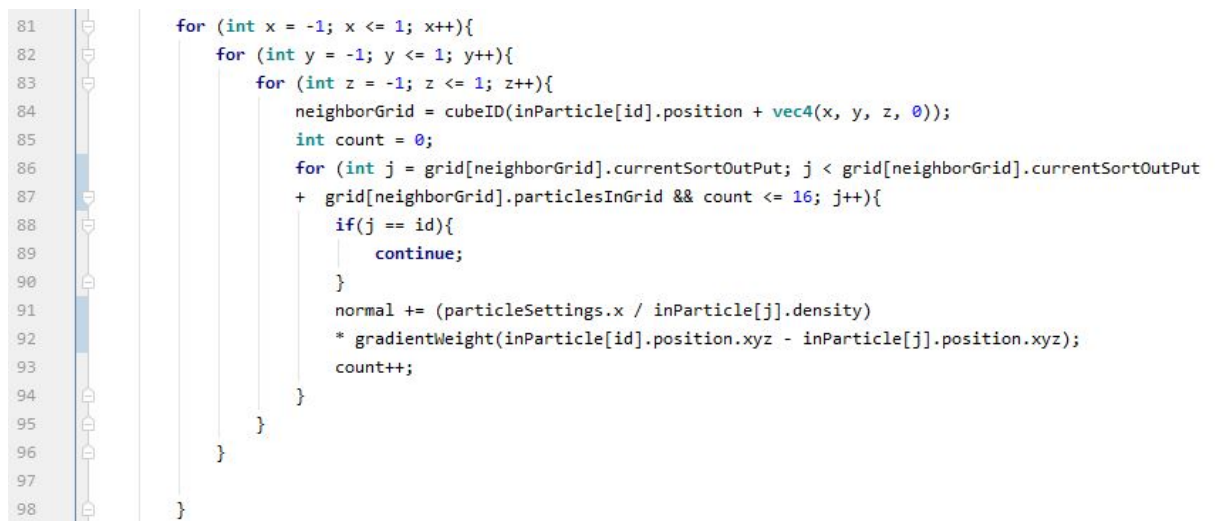
```

58 {
59     outParticle[grid[cubeID(inParticle[id].position)].currentSortOutPut
60     + inParticle[id].memoryPosition] = inParticle[id];
61 }

```

Abbildung 23: umorganisieren der Partikel im Speicher

Das Aufrufen der Nachbarpartikel erfolgt daraufhin wie zum Beispiel bei der Normalenberechnung 24. Dabei wird zunächst die *id* des Nachbargrids über die *CubeID*-Funktion ermittelt. Daraufhin wird der *currentSortOutPut* des Nachbargrids, welcher dem ersten Partikel in dem Grid entspricht, als initial Wert der For-Schleife genommen. Diese läuft bis die Summe des *currentSortOutPut* plus die *particlesinGrid* erreicht worden sind oder eine andere Abbruchbedingung eintritt.



```

81 for (int x = -1; x <= 1; x++){
82     for (int y = -1; y <= 1; y++){
83         for (int z = -1; z <= 1; z++){
84             neighborGrid = cubeID(inParticle[id].position + vec4(x, y, z, 0));
85             int count = 0;
86             for (int j = grid[neighborGrid].currentSortOutPut; j < grid[neighborGrid].currentSortOutPut
87             + grid[neighborGrid].particlesInGrid && count <= 16; j++){
88                 if(j == id){
89                     continue;
90                 }
91                 normal += (particleSettings.x / inParticle[j].density)
92                 * gradientWeight(inParticle[id].position.xyz - inParticle[j].position.xyz);
93                 count++;
94             }
95         }
96     }
97 }
98

```

Abbildung 24: Berechnung der Normalen einen Partikel

6.5 Vergleich

Die beiden Beschleunigungsverfahren bieten auf Kosten von Speicherplatz oder Laufzeit für das Sortieren eine sehr starke Verkürzung der Laufzeit. Im folgenden werden beide Beschleunigungsverfahren gegenübergestellt und mit einander auf Vor- und Nachteile untersucht.

Damit bei beiden Verfahren über gleich viele Partikel iteriert wird und dies damit keinen Einfluss auf die Beschleunigung hat wurde der Countingsort ebenfalls auf 16 Partikel pro Grid beschränkt 24. Als Standardwerte für die Partikelanzahl wird 65.536 und für die Gridgröße $50 \times 50 \times 50$ angesehen, welche bei speziellen Tests für die Partikel oder das Grid dann variieren können.

Die Tests wurden auf einem Desktop-Pc mit einer Intel(R) Core(TM) i5-

9600k CPU @ 4.0 GHz und einer Nvidia GeForce GTX 1660 Ti Grafikkarte durchgeführt.

7 Ergebnis

8 Fazit

Abbildungsverzeichnis

1	Fluidsimulation in Form des Vektorfeldverfahren Quelle: https://thumbs.gfycat.com/CelebratedElasticHartebeest-poster.jpg	5
2	Partikelsimulation von Wasser https://i.ytimg.com/vi/DhNt_A3k4B4/maxresdefault.jpg	6
3	Updateschleife der Physik	7
4	Bedeutung aller Symbole der Berechnungen	8
5	Die Gewichtungsfunktionen W_{poly6} , W_{spiky} , W_{visc} nach [MCG03] mit einem Radius von 1	10
6	Implementation der SPH Kernel	10
7	Implementation der Dichtefunktion	12
8	Implementation der Wärmeleitfähigkeit	14
9	Anpassbare Variablen per ImGui	15
10	Partikel SSBOs zum lesen und schreiben	16
11	Partikel im Grid mit einem Radius von 1	18
12	Speichern der Partikel-IDs im Grid	19
13	Unsortierter Speicher beim Speicherverfahren	20
14	sortierter Speicher nach Sortieralgorithmus	20
15	Ausgangsarray beim Countingsort	21
16	Anzahl der Werte im Hilfsarray	21
17	Summe der Anzahl der Werte im Hilfsarray	21
18	vollständig sortiertes Array	21
19	Aufbau des Implementierten Countingsort	22
20	SSBO-Struct des Grids	23
21	registrieren der Partikel im Grid mit CubeID-Funktion	23
22	paralleler Prefixsum Scan von Navie nach Harris [HSO07]	24
23	umorganisieren der Partikel im Speicher	25
24	Berechnung der Normalen einen Partikel	25

Literatur

- [Cor19] CORNUT, Omar: *ImGui: Bloat-free Immediate Mode Graphical User interface for C++ with minimal dependencies*. <https://github.com/ocornut/imgui>. Version: 2019. – [Online; Stand 09. September 2019]
- [DD11] DEMAINE, Erik ; DEVADAS, Srin: Introduction to Algorithms. In: *MIT OpenCourseWare 6.006* Massachusetts Institute of Technology, Herbst 2011
- [DG96] DESBRUN, Mathieu ; GASCUEL, Marie-Paule: Smoothed particles: A new paradigm for animating highly deformable bodies. In: *Computer Animation and Simulation'96*. Springer, 1996, S. 61–76
- [Hei10] HEINRICH, Alan: *Smoothed Particle Hydrodynamics Webcast*. <https://youtu.be/SQPCXzqH610>. Version: 2010. – [Online; Stand 10. September 2019]
- [Hoe14] HOETZLEIN, Rama C.: *FAST FIXED-RADIUS NEAREST NEIGHBORS: INTERACTIVE MILLION-PARTICLE FLUIDS*. <http://on-demand.gputechconf.com/gtc/2014/presentations/S4117-fast-fixed-radius-nearest-neighbor-gpu.pdf>. Version: 2014. – [Online; Stand 10. September 2019]
- [HSO07] HARRIS, Mark ; SENGUPTA, Shubhabrata ; OWENS, John D.: Parallel prefix sum (scan) with CUDA. In: *GPU gems 3* (2007), Nr. 39, S. 851–876
- [IOS⁺14] IHMSEN, Markus ; ORTHMANN, Jens ; SOLENTHALER, Barbara ; KOLB, Andreas ; TESCHNER, Matthias: SPH fluids in computer graphics. (2014)
- [MCG03] MÜLLER, Matthias ; CHARYPAR, David ; GROSS, Markus: Particle-based fluid simulation for interactive applications. In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* Eurographics Association, 2003, S. 154–159
- [MMCK14] MACKLIN, Miles ; MÜLLER, Matthias ; CHENTANEZ, Nuttapon ; KIM, Tae-Yong: Unified particle physics for real-time applications. In: *ACM Transactions on Graphics (TOG)* 33 (2014), Nr. 4, S. 153

- [Pes09] PESCHEL, Franz: *Simulation und Visualisierung von Fluiden in einer Echtzeitanwendung mit Hilfe der GPU*. http://aleph1.uni-koblenz.de/F?func=find-b&find_code=wr&request=franz+peschel. Version: 2009. – [Online; Stand 28. August 2019]
- [RYY⁺16] REN, Bo ; YAN, Xiao ; YANG, Tao ; LI, Chen-feng ; LIN, Ming C. ; HU, Shi-min: Fast SPH simulation for gaseous fluids. In: *The Visual Computer* 32 (2016), Nr. 4, S. 523–534
- [Sta03] STAM, Jos: Real-time fluid dynamics for games. In: *Proceedings of the game developer conference* Bd. 18, 2003, S. 25
- [Wik19] WIKIPEDIA: *Navier-Stokes-Gleichungen* — *Wikipedia, Die freie Enzyklopädie*. <https://de.wikipedia.org/w/index.php?title=Navier-Stokes-Gleichungen&oldid=191399294>. Version: 2019. – [Online; Stand 28. August 2019]