

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)

Abstract

Diese Studienarbeit befasst sich mit der Simulation von Rauch in einem Partikelsystem. Hierbei werden die Möglichkeiten untersucht Rauch möglichst realistisch in einem Partikelsystem zu implementieren und in Echtzeit berechnen zu lassen. Die physikalische Simulation basiert dabei auf den Arbeiten von Müller [MCG03] und Ren [RYY⁺16], welche sich mit den physikalischen Eigenschaft von Fluiden und Gasen beschäftigen. Die Simulation wurde mittels C++, OpenGL und der in OpenGL verfügbaren Compute-Shader auf der GPU implementiert. Dabei wurde ein besonderes Augenmerk darauf gelegt, dass diese möglichst Performant ist. Hierfür werden Techniken von Hoetzlein [Hoe14] benutzt um das Partikelsystem zu beschleunigen. Daraufhin wurden zwei Beschleunigungsverfahren implementiert und gegenübergestellt. Es wird dabei auf die Laufzeit, aber auch auf den verbrauchten Speicherplatz der GPU eingegangen.

This student project deals with the simulation of smoke in a particle system. Here the possibilities are investigated to implement smoke as realistically as possible in a particle system and to calculate it in real time. The physical simulation is based on the work of Müller [MCG03] and Ren [RYY⁺16], who deal with the physical properties of fluids and gases. The simulation was implemented on the GPU using C++, OpenGL and the compute shaders available in OpenGL. Special attention was paid to the performance of the simulation. Hoetzlein [Hoe14] techniques are used to accelerate the particle system. Two acceleration methods were then implemented and compared. The runtime, but also the used memory space of the GPU is discussed.

Inhaltsverzeichnis

1	Vorwort	1
2	Einleitung	2
3	State of the Art	2
3.1	Vektorfeldverfahren	3
3.2	Partikelsystem	4
4	Rauchsimulation	5
4.1	Gewichtungsfunktionen	7
4.2	Dichte	10
4.3	Druck	11
4.4	Viskosität	12
4.5	Auftrieb	12
4.6	Wirbelkraft	13
4.7	Update Kräfte	14
5	Implementierung	14
6	Beschleunigung	16
6.1	gridbasierte Nachbarschaftssuche	16
6.2	Speicherverfahren	18
6.3	Sortiervverfahren	20
6.4	Countingsort	20
6.5	Vergleich	25
7	Ergebnis	28
8	Ausblick	32

1 Vorwort

Zur Simulation eines Fluids hat mich die Lehrveranstaltung Animation und Simulation inspiriert, da dort vermehrt Partikelsysteme implementiert wurden. Doch dort besaßen alle Simulationen eine schlechte Performance, weshalb nur wenige Partikel berechnet werden konnten. Mein persönliches Ziel war es deshalb das System weites gehen zu beschleunigen.

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. Danke an alle die sich die Zeit genommen haben meine Arbeit Korrektur zu lesen.

Außerdem möchte ich mich bei Pascal Bendler bedanken, der mich tatkräftig beim Debugging und Aufbau des Frameworks 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 in einem Partikelsystem zu implementieren. Dazu nutzen wir, dass sich Rauch wie ein Fluid verhält [Sta03], dabei wird die Simulation der physikalischen Basis von Fluiden angenähert. Es wird in dieser Implementation ein Partikelsystem zur Berechnung der physikalischen Eigenschaften genutzt. Echtzeit-Entwicklungsplattformen wie die Unity-Engine bieten zwar eine Partikelsimulation an, jedoch beschränkt sich diese lediglich auf das Ausstoßen von Partikeln. Dabei können Partikelinteraktionen, sowie Fluideigenschaften nicht angepasst 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. Diese sind zwar nicht so leistungsfähig 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 ein für den Beobachter möglichst schönes, sowie nachvollziehbares 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. Diese Gleichungen beschreiben die physikalischen Eigenschaften von Fluiden

und werden für die Simulation dieser angewendet. Diese Formeln werden je nach Fluid angepasst um speziellere Eigenschaften darzustellen[TC78]. 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 eine 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 dieser breitet sich das Fluid stark wie zum Beispiel Wasser oder weniger stark wie Honig aus. Bei Beschleunigung handelt es sich um eine externe Kraft die auf das Fluid wirkt, 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 Strömungssimulationen 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 dar. 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 **1**. Zur Darstellung des Fluides wird meist Volumerendering genutzt. Außerdem ist der Rechenaufwand hoch und lässt sich wegen der Grid-Architektur des Verfahrens nur schwer verbessern.



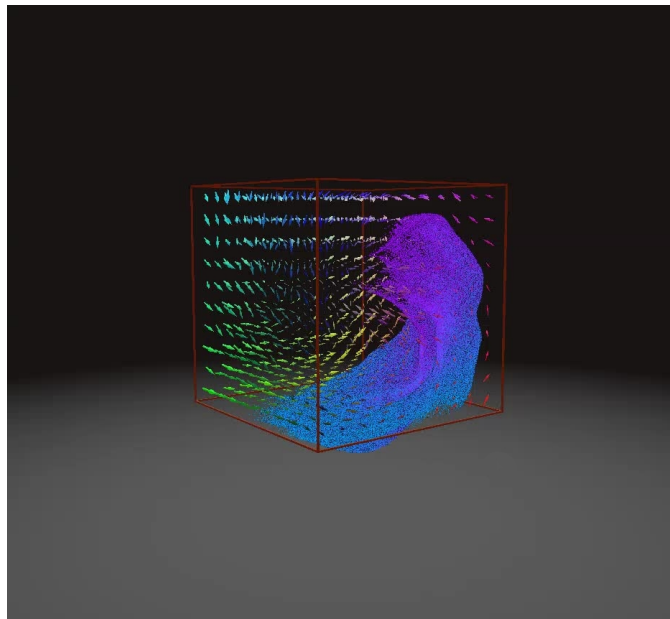


Abbildung 1: Fluidsimulation in Form des Vektorfeldverfahren

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. Die Nachbarpartikel werden dabei in die Berechnung mit einbezogen. Die Dichte beschreibt dabei wie viele Nachbarpartikel Einfluss auf den betrachteten 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 die Partikel sich voneinander weg bewegen. Die Viskosität wirkt dem entgegen und führt das Anziehen von Partikel hervor. Der Auftrieb wiederum lässt sich über die Temperatur des Raumes bestimmen, welche je nach Dichte steigt oder sinkt. Zum anderen lässt sich aber die Wirbelkraft nicht so einfach berechnen und stellt somit ein Problem in der Forschung dar. Für die Berechnungen werden die Nachbarpartikel benötigt, welche aber nicht für jeden Partikel bekannt sind. 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 beispielsweise ein Computerspiel oder einer Engine,** da man mit der Partikelanzahl die Performance beeinflussen

kann. Beim Reduzieren 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 die Beschleunigung der Nachbarschaftssuche, sowie den Auftrieb und die Wirbelkraft.

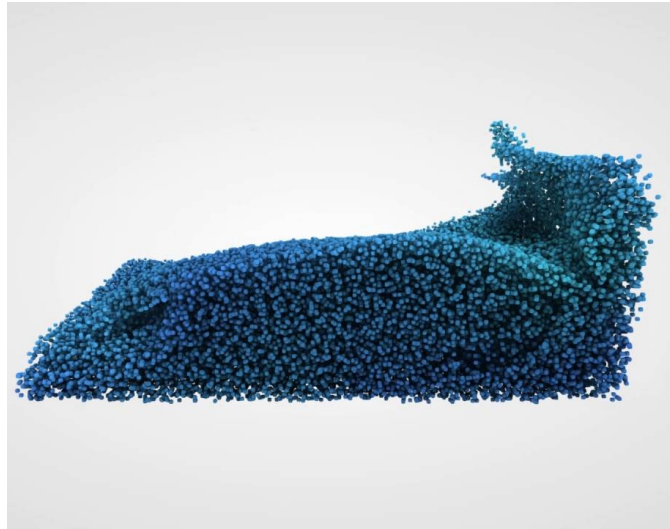


Abbildung 2: Partikelsimulation von Wasser

4 Rauchsimulation

Zur Simulation von Rauch wurde ein Partikelsystem implementiert, wessen physikalische Grundlage auf dem SPH-Verfahren basiert. Dieses Verfahren verwendet Müller [MCG03] 2003 zur Simulation von Fluiden. 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 vier Compute-Shader unterteilen, welche voneinander abhängig und deshalb nicht weiter reduzierbar sind.

Zum Verständnis stellt die Abbildung 4 alle Variablen dar, sowie deren Be-

deutung 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. Diese Funktionen besitzen alle die Eigenschaft, dass sie symmetrisch sind und beim Erreichen des Radius gegen 0 konvergieren. Damit haben zu weit entfernte Partikel keinen Einfluss mehr. Gewichtungsfunktionen dienen dazu 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 Krayer 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 ein Laplace benötigt. Der Gradient für die Druckberechnung lässt sich durch die Ableitung der für diese Berechnung vorgesehene Funktion berechnen. Mit der W_{poly} -Funktion wird beim Wert 1 ein nahezu 0 Wert erreicht. Dabei sollte dort die Druckkraft am höchsten sein. Dies würde dazu führen, dass sehr nahe Partikel sich nicht mehr voneinander abstoßen würden. Deshalb nutzt Müller Desbrun [DG96] die Funktion 2, da diese in der Ableitung als Funktion 3 Werte liefert, die für die Berechnung entsprechend konvergieren.

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

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

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

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

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

Die W_{spiky} -Funktion, würde aber wiederum in der zweiten Ableitung zu 0 werden. Bei anderen standardmäßigen Gewichtungsfunktionen könnte es sogar dazu führen, dass ein negativer Wert auftritt, der bei der

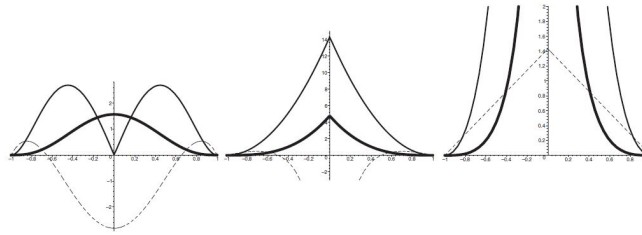


Abbildung 5: Die Gewichtungsfunktionen W_{poly6} , W_{spiky} , W_{visc} nach [MCG03] mit einem Radius von 1

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

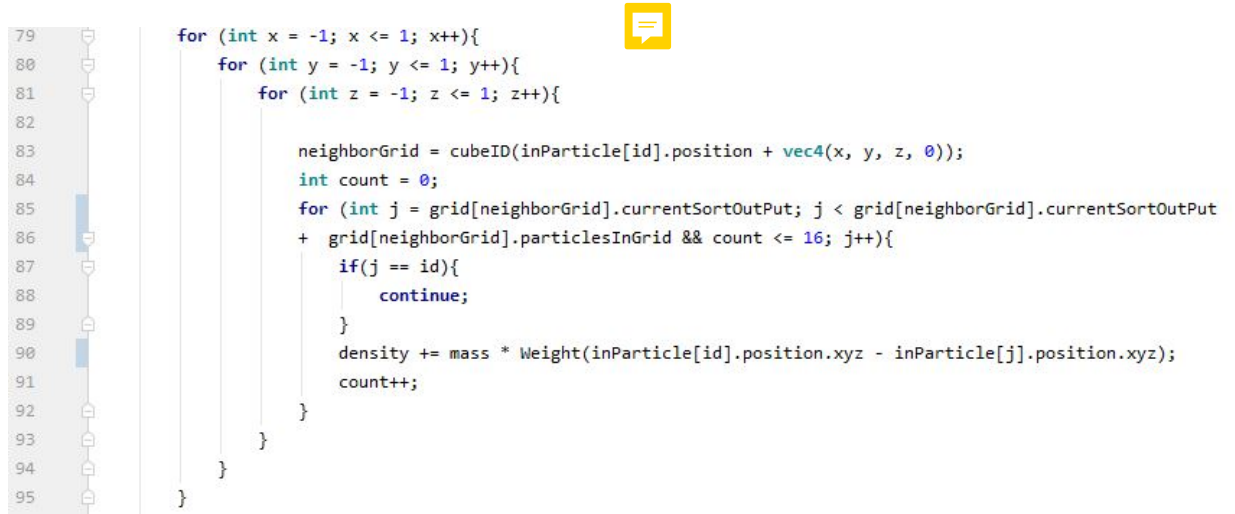
Die Dichte ist eine Variable, die für jeden Partikel individuell berechnet wird. Sie beschreibt die Anzahl der Partikel in unmittelbarer Nähe. Außerdem wird sie bei fast jeder folgenden Berechnung benötigt, da sie als Gewichtung des eingenommenen Volumens dient.

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

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

Die Dichte, wie in ~~der~~ Gleichung 7 zu sehen ist, errechnet sich aus der Summe ~~über~~ aller Nachbarn j . Dabei multiplizieren wir die Masse m_j mit der Dichte ρ_j durch die Dichte ρ_j selbst. 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, 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.



```

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

Die Druckkraft $f^{pressure}$ beschreibt das Abstoßen von Partikeln von einander. Die standardmäßige Berechnung für Skalaregrößen Gleichung 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 einer 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 beider Partikel addiert, um das Verhältnis aber beizubehalten wird auch die Dichte im Nenner verdoppelt. Zur Gewichtung wird dabei die Gradientenfunktion 3 genutzt. Damit der Vektor in die entgegengesetzte Richtung zum Partikel j zeigt, wird dieser negiert. Zum Berechnen der **Druckkraft muss aber zunächst der Druck** berechnet werden. Die Formel 8 von Desbrun [DG96] wurde diesbezüglich angepasst, indem eine Ruhedichte von der eigentlichen Dichte abgezogen wurde. Dabei wird die Ruhedichte p_0 als Offset verwendet und hat keinen mathematischen Einfluss auf die Druckkraft [MCG03], da es diese nur um das Offset verschiebt. 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. Daraus resultiert, dass die Viskosität eine asymmetrische Kraft ist. Wäre sie jedoch eine symmetrische Kraft würde es dazu führen, dass sobald sich eine große Anzahl an Partikel gleichzeitig in die selbe Richtung bewegen, sich diese 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. Um dies zu umgehen modifiziert Müller [MCG03] die SPH-Formel 6, indem der Unterschied des Geschwindigkeitsvektor v in Betracht gezogen wird. Dies hat zur Folge, 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 zusammenziehen. Daraus resultiert, dass die Partikel mit der angepassten Formel, sich auch abbremesen können, indem ein schnellerer Partikel beim Berechnen 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 12 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 11 ersetzt wurde. Diese multipliziert den Temperaturunterschied 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 14 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 13 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 negativer Temperatur verhindert.

Wenn die Temperatur errechnet wurde lässt diese sich wie in 15 berechnen. Dabei beschreibt b einen Up-Vektor und C_b den Auftriebs-Koeffizienten.

$$Dc(T) = C * T \quad (11)$$

$$\frac{\delta T_i}{\delta t} = \sum_j \frac{m_j}{\rho_i \rho_j} Dc(T_i - T_j) \frac{(r_i - r_j) \cdot \nabla W_{i,j}}{(r_i - r_j)^2 + \gamma^2} \quad (12)$$

$$n_i = \sum_j \frac{m_j}{\rho_j} \nabla W_{i,j} \quad (13)$$

$$\frac{\delta T_i}{\delta t} = -T_i / D_r \quad (14)$$

$$f_i^{buoyancy} = C_b T_i b \quad (15)$$

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 16. 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 17 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.

$$\frac{\delta \omega_i}{\delta t} = \omega_i \cdot + \nabla v + \beta(n_i \times g) \quad (16)$$

$$f_j^{vortex} = \sum_j (\omega_j \times (r_i - r_j)) W_{i,j} \quad (17)$$

Eine Wirbelkraft zu errechnen die Turbulenzen verursacht ist leider nicht gelungen. Diese hatte lediglich die Wirkung ~~die~~, dass der Rauch langsamer aufgetrieben ist. Das langsame Ausstoßen von Partikeln hatte leider keine positiven Auswirkungen auf die Physik. Dadurch entstanden eher Instabilitäten des Drucks, welcher sich zu Beginn anpassen lies, aber dann inkonstant wurde. Eine Anpassung der Variablen führte leider dabei zu keinem Erfolg. Es wurden auch andere Versuche unternommen die Wirbelkraft anzupassen, wie unter anderem die Formel zu modifizieren, welche aber erfolglos blieben.



4.7 Update Kräfte

Abschließend werden alle Kräfte zusammen gerechnet und zum Geschwindigkeitsvektor hinzugefügt 18. 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.

$$\frac{\delta v_i}{\delta t} = f_i^{pressure} + f_i^{viscosity} + f_i^{buoyancy} + f_i^{vortex} + ext + \rho_i g \quad (18)$$

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 8 zur Verfügung stellt um die Parameter anzupassen. Das Anpassen der Variablen während der Laufzeit kann aber zur Instabilität der Physik führen.

ImGui diene auch dazu die Laufzeit der einzelnen Berechnungsschritte dazustellen. Aus diesen Informationen konnten unter anderem darauf geschlossen werden, welche Berechnungen besonders viel Aufwand mit sich bringen. Diese Zeiten machten aber auch darauf aufmerksam, das ein Prozess der einen geringen Aufwand haben sollte trotzdem eine hohe Laufzeit

benötigte. Daraufhin wurden im Countingsort 6.4 beim Umorganisierungsschritt eine Anpassung vorgenommen, die sich positiv auf die Leistung ausgewirkt hat.

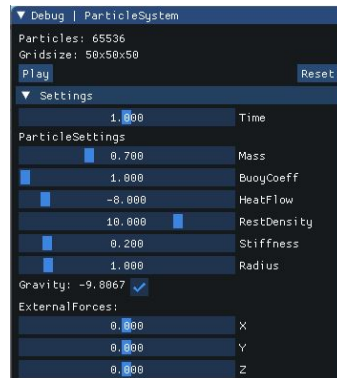


Abbildung 8: Anpassbare Variablen per ImGui

Damit bei dem Iterieren über das eigene Grid ein Partikel sich nicht selbst beeinflussen kann wurde wie in Abbildung 23 Zeile 88-90 zu sehen ist ein If-Abfang eingebaut. Diese verhindert es, dass ein Partikel mit derselben ID einen Einfluss, auf sich selbst, nehmen kann. Auf die Begrenzung der Nachbarn, die in die Berechnung eingenommen werden, wird in Abschnitt 6.5 eingegangen.

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 MB 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. Hierfür wurde ein simpler Compute-Shader implementiert, der nur das alte SSBO mit den Daten des neuen überschreibt.

Für das parallele Schreiben in das gleiche SSBO bedarf es aber wiederum speziellen Operationen, die verhindern, dass mehrere Zugriffe gleichzeitig passieren und eine Variable bearbeiten. Dieses Problem entsteht häufig beim Versuch der Beschleunigung, da dort viele parallele Prozesse der Partikel auf das SSBO des Grids zugreifen.

OpenGL bietet in GLSL dabei Atomic-Operationen an, diese verhindern das ein anderer Prozess ebenfalls auf die Variable zugreift und diese bearbeitet. Dies entsteht dadurch, dass diese Funktionen zunächst einmal überprüfen ob die Variable bereits bearbeitet wird, falls dies der Fall ist wird solange gewartet bis sie freigegeben wird. Durch viele Prozesse die auf eine Variable zugreifen wollen, können dadurch Warteschlangen für diese entstehen, woraus ein Zeitverlust resultiert.

```
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 9: 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 pro Frame bedeuten. 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, die Nachbarn für einen einzigen Partikel herausfinden und für diese Iteration abspeichern, würde dies mit einem immer noch hohem Aufwand, sowie einem großen Speicherverbrauch verbunden sein. Bei diesem wäre nicht klar, wie viel Speicher benötigt wird, da die Anzahl der Nachbarpartikel nicht bekannt ist.

Man müsste dabei vom Worst Case ausgehen und genug Speicherplatz für alle Partikel anlegen.

Hoetzlein [Hoe14] stellt dabei eine gridbasierte Nachbarschaftssuche vor. 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 die Griddimensionen sein sollen. Dieser Wert ist zu Laufzeit nicht mehr anpassbar, da das Grid als SS-BO angelegt wird. Jedes Grid besitzt eine eindeutige ID die gleich 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 $50 \times 50 \times 50$ und ca. 65.000 Partikel 32,5 GB verbrauchen würde, dies wäre noch ineffizienter als wenn alle Partikel genug Speicherplatz für alle potenziellen Nachbarn anlegen würden. 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 20 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 10 sieht man ein vereinfachtes 2D-Grid mit einem Partikel, welcher dem Grid in der Mitte zuzuordnen ist. Bei einem standardmäßigem Radius h von 1, werden alle möglichen Nachbarpartikel die infrage kommen abgedeckt. Hierbei werden alle 9 in 2D oder 27 in 3D Grids betrachtet und über die Partikel in diesen iteriert.



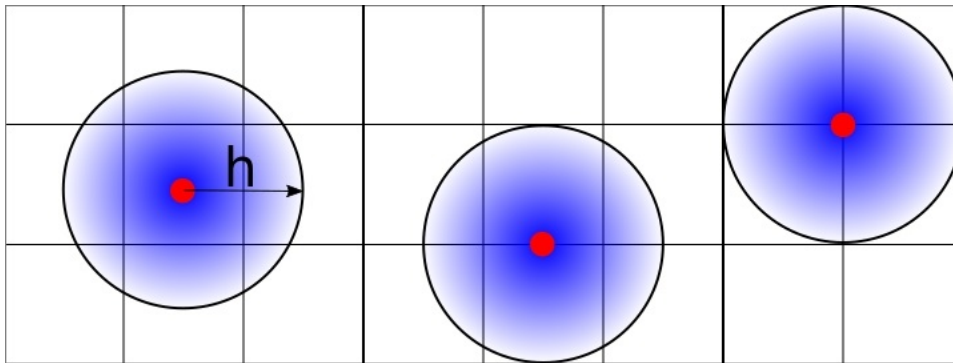


Abbildung 10: Partikel im Grid mit einem Radius von 1

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, sowie 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 Nachbarnpartikel 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 10 sehen kann, dass nicht alle Grids durch die Gewichtungsfunktionen 4.1 mit einbezogen werden. Im Durchschnitt haben pro Nachbarschaftssuche ca. 138 Nachbarpartikel einen 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 11 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 werden. 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 11: 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 12 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, den Zugriff auf Speicheradressen die einen großen 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 12: Unsortierter Speicher beim Speicherverfahren

6.3 Sortiervverfahren

Um das Problem der scattered reads zu beheben, müssen die Speicherpositionen nach dem Grid sortiert werden, 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 13: sortierter Speicher nach Sortieralgorithmus

Da das Sortieren auf der CPU aber sequentiell ist, würde es wertvolle Laufzeit verbrauchen, ebenfalls das Übertragen der Daten auf die CPU. Deshalb wird ein paralleler Sortieralgorithmus auf der GPU benötigt, Hoetzlein [Hoe14] stellt dabei zwei Sortiervverfahren 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, macht 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 dessen Theorie und Implementation im folgenden Abschnitt näher eingegangen wird.

6.4 Countingsort

Countingsort ist ein nicht vergleichsbasierter Algorithmus, sondern ist adressenbasiert. Dabei dürfen nur natürliche Zahlen, als Schlüsselwert, in dem Array vorkommen, welche mit einem maximalen Wert begrenzt werden. 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 14 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 14: Ausgangsarray beim Countingsort

Zunächst werden dann in einem Hilfsarray die Anzahl der vorkommenden Werte bestimmt 15. Daraufhin wird die Summe aus allen vorherigen Anzahlen bestimmt und in das Hilfsarray geschrieben 16.

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

Abbildung 15: 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 16: 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 17: 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 18 zu findenden Aufbau diente.

```

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 18: Aufbau des Implementierten Countingsort

Der Sortieralgorithmus beginnt mit dem Zurücksetzen des Grid-SSBOs 19. 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 19: SSBO-Struct des Grids

Im Folgenden wird die Anzahl der Partikel in einem Grid gespeichert, die Zugehörigkeit zum Grid wird über die CubeID-Funktion ermittelt, welches in Abbildung 20 in Zeile 60 zu sehen ist. 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 20: 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 $\log_2(n) - 1$.

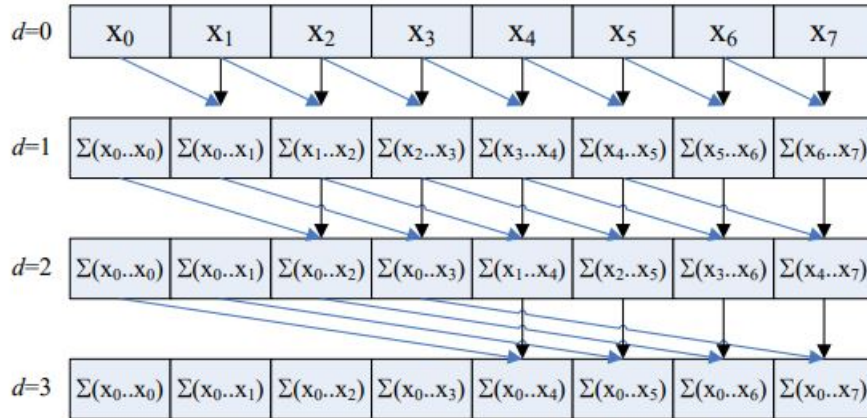


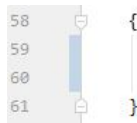
Abbildung 21: paralleler Prefixsum Scan von Navie nach Harris [HSO07]

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 Compute-Shader zu Berechnung der Prefixsum und zum updaten des Gridbuffers im wechsel $\log_2(Particleanzahl) - 1$ 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 16 zu 17.

Um diesen sequentiellen Prozess zum umgehen wurde aber im Vorndherein bereits in der *outParticle[id].memoryPosition* die Position des Partikels im Grid bestimmt. Deshalb können die Partikel wie in 22 zu sehen, sich bloß an die entsprechende Speicherstelle schreiben, diese Anpassung spart ca. 0,1ms pro Frame bei 65.000 Partikeln. Dadurch befinden sich nun alle Partikel die in einem Grid sind in aufeinanderfolgenden Speicherplätzen und verhindert damit scattered reads.



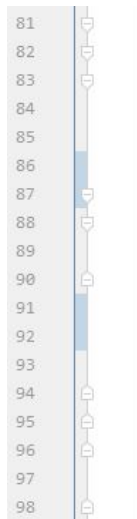
```

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

```

Abbildung 22: umorganisieren der Partikel im Speicher

Das Aufrufen der Nachbarpartikel erfolgt daraufhin wie zum Beispiel bei der Normalenberechnung 23. 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 23: Berechnung der Normalen einen Partikel

6.5 Vergleich

Die beiden Beschleunigungsverfahren bieten auf Kosten von Speicherplatz und Laufzeit 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 23. 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. Bei den Tests wurden die Durchschnittswerte von 20s Laufzeit pro Simulation, **die jeder 10 mal wiederholt wurden**, ausgewertet. Bei den zeitlichen Angaben handelt es sich um Millisekunden pro Frame. Beim Vergleich der Sortieralgorithmen bei steigender Partikelanzahl 24 haben beide eine ähnliche Berechnungsdauer bei einer niedrigen Anzahl, dabei ist aber der Countingsort bereits ein wenig schneller. Besonders bemerkbar macht sich der Unterschied bei hoher Partikelanzahl bei dem das Speicherverfahren fast die doppelte Laufzeit pro Frame besitzt. Dies ist ganz klar auf die scattered reads zurückzuführen, die das Speicherverfahren, trotz weniger Kern calls, erheblich verlangsamen. Beim Countingsort deutet es einen linearen Verlauf der Laufzeit an, wie es Hoetzlein [Hoe14] beschreibt. Diese Linearität **scheint** aber zwischen 525.000 und 1.000.000 nicht mehr gegeben zu sein. Dies **scheint** aber an der Datenübertragungsrate der Grafikkarte zu liegen. Da es bei einer mit höherer Datenübertragungsrate und vergleichbar vielen Kernen zwar ebenfalls keine perfekte Linearität aufwies, aber eine deutlich besseren Verlauf zeigte. Leider konnten keine eindeutigeren Tests auf dieser Grafikkarte durchgeführt werden, da diese nur kurz zur Verfügung stand.

Partikel	8.912	16k	32k	65k	131k	262k	524k	1kk
Countingsort								
Laufzeit(ms)	0,72	1,27	2,13	4,12	12,71	27,81	56,72	142,56
Speicher								
Laufzeit(ms)	0,9	1,88	3,69	7,34	15,15	38,71	107,45	270,65

Abbildung 24: Laufzeit in Relation zur Partikelanzahl

Einen Einfluss auf die Laufzeit hat auch die Griddimension, wobei dieser sehr viel geringer ausfällt als die der Partikelanzahl. Wie in Abbildung 25 zu sehen, ist auch hier der Countingsort schneller und bei diesem haben die Griddimensionen auch weniger Einfluss auf die Laufzeit, als beim Speicherverfahren. Bei dem Speicherverfahren steigt die Laufzeit stärker an, obwohl dieser Anstieg sich nicht signifikant von dem des Countingsort unterscheidet.

Partikel	20	30	40	50	60	70	80	90	100
Countingsort									
Laufzeit(ms)	5,85	4,43	3,92	4,04	4,27	4,54	4,85	5,37	6,36
Speicher									
Laufzeit(ms)	7,4	7,04	7,14	7,45	8,47	9,29	9,54	9,84	10,41

Abbildung 25: Laufzeit in Relation zur den Griddimensionen

Das Sinken und anschließende Steigen aus Abbildung 25 lässt sich durch das stärkere Verteilen der Partikel erklären, da sich in einem kleineren Raum mehr Partikel in einem Grid befinden und sich dadurch mehr atomicAdd-Funktionen bei den einzelnen Grids stauen.

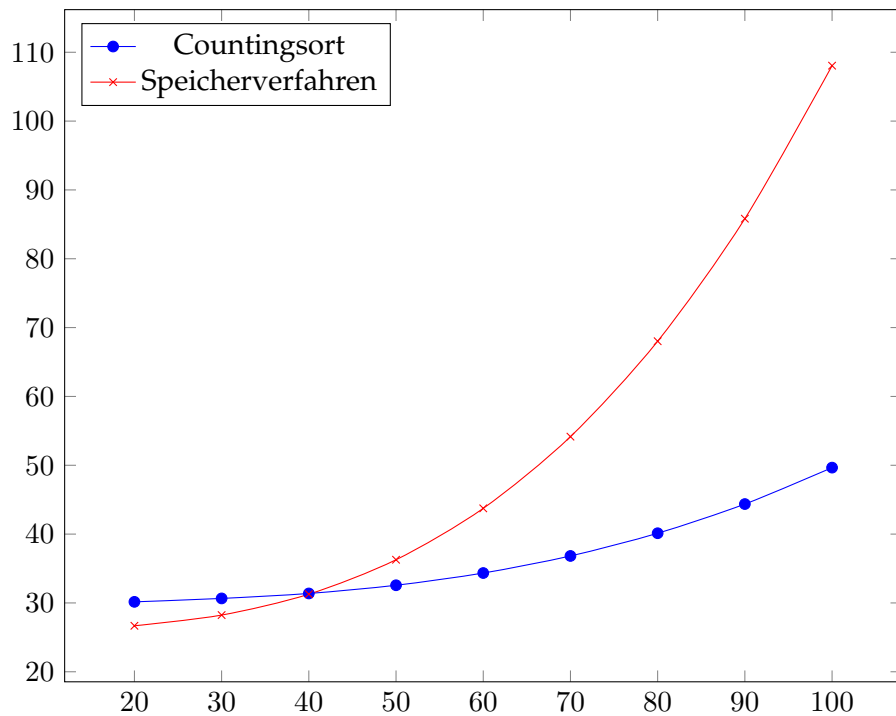


Abbildung 26: Speicherverbrauch in Kb auf der Y-Achse und Griddimensionen auf der X-Achse

Da der Speicherplatz der Grafikkarte nicht unbegrenzt ist, lohnt es sich ebenfalls einen Blick auf diesen zu werfen in Verbindung mit den Griddimensionen, da diese sich bei den Sortieralgorithmen im nutzenden Speicher unterscheiden 26.

Zunächst hat das Speicherverfahren einen geringeren Verbrauch, welcher daraus resultiert, dass für den Countingsort neben dem Grid-SSBO noch der temporäre Gridbuffer als SSBO existiert. Doch Beim Speicherverfahren steigt der benötigte Speicherplatz sehr viel stärker an als beim Countingsort. Bei diesem ist nur ein geringer Anstieg zu verzeichnen.

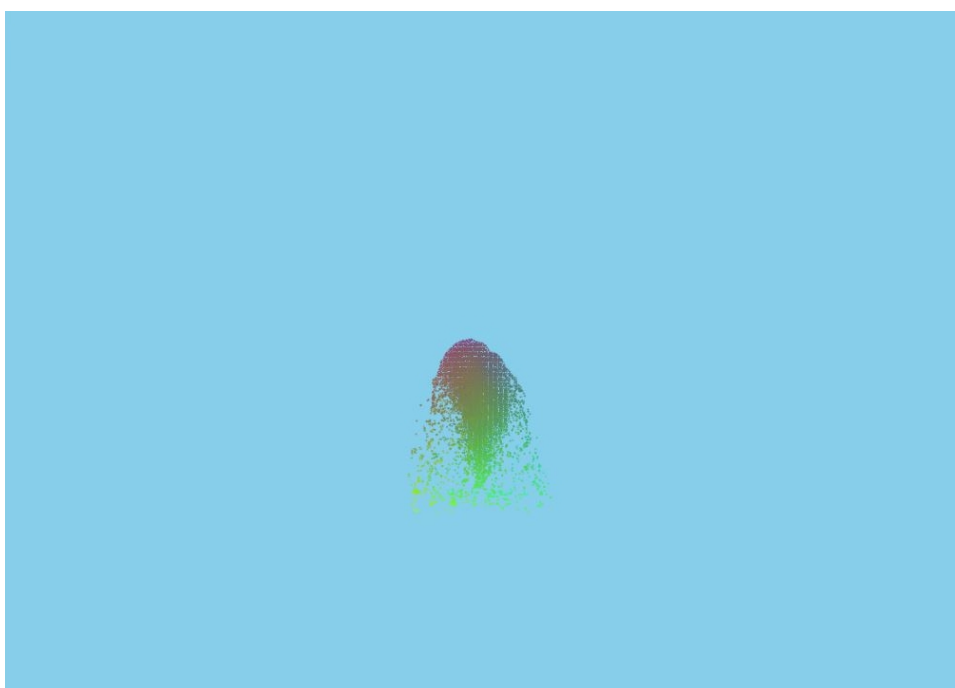
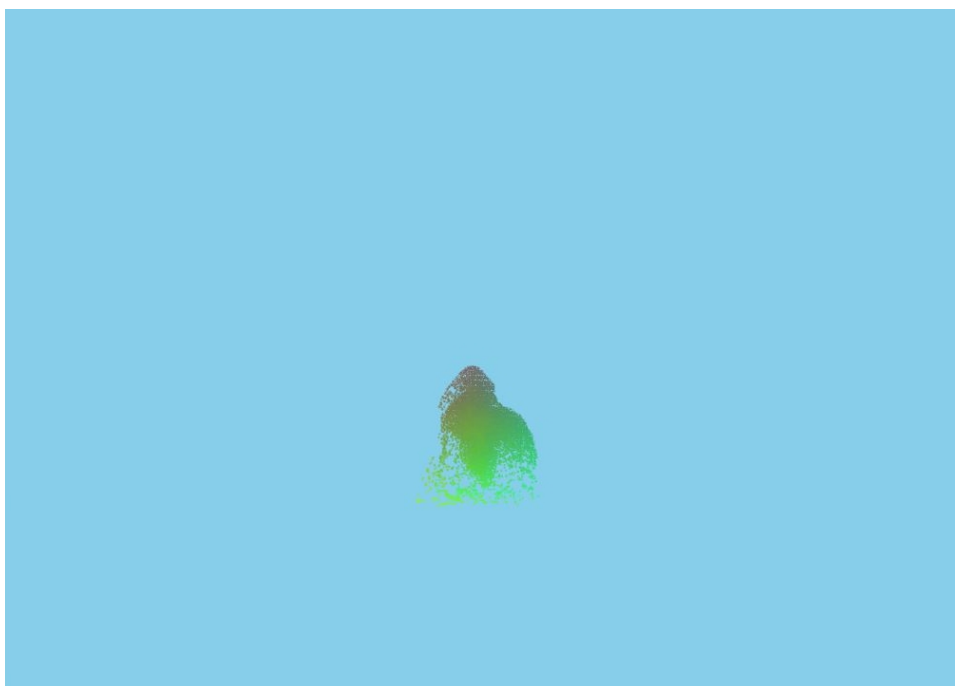
7 Ergebnis

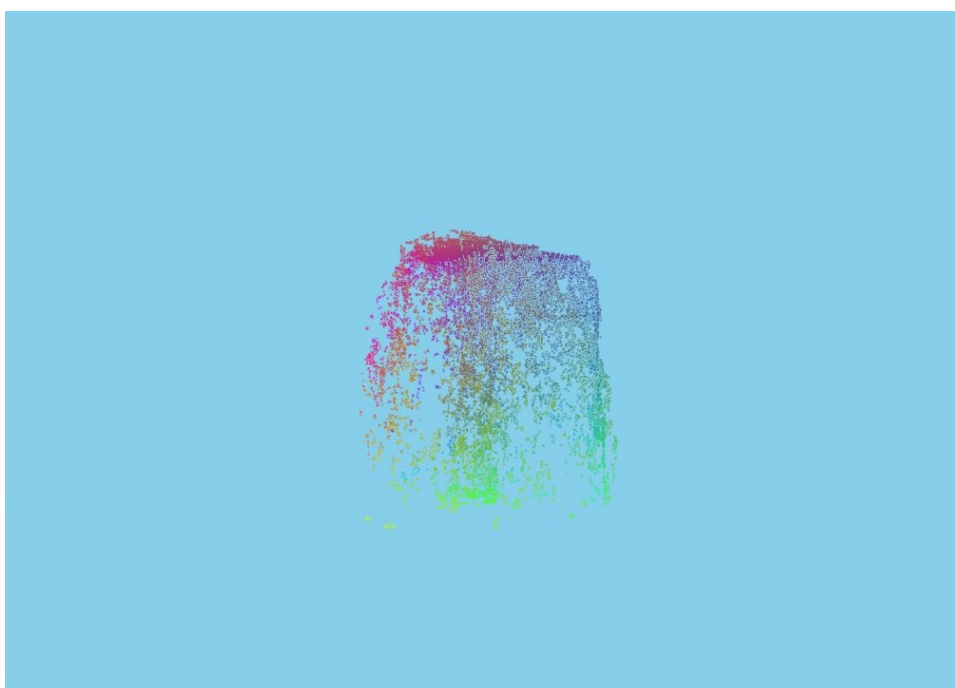
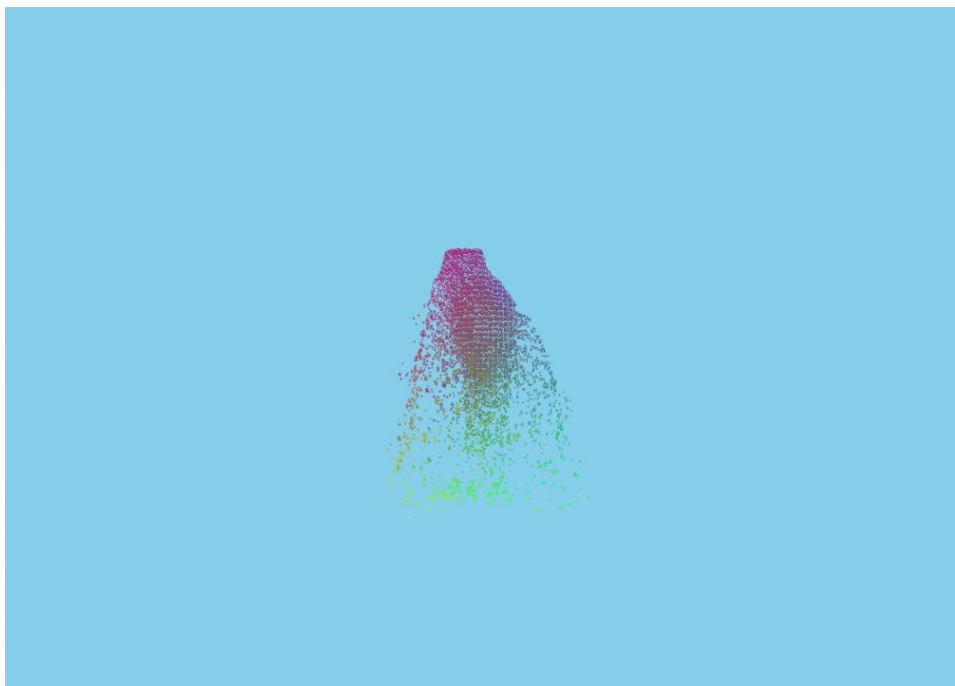
In der Studienarbeit wurde gezeigt, dass eine Realisierung Rauchsimulation in einem Partikelsystem bedingt möglich ist. Hierbei weist das simulierte Fluid einige Eigenschaften von Rauch auf. Der Auftrieb, so wie Druck und Viskosität verhalten sich physikalisch korrekt. Die Wirbelkräfte in einer Rauchsimulation zu realisieren, welche auf einem Partikelsystem basiert, benötigt jedoch eine komplexere Berechnung. Diese lassen sich nicht über simple Partikelinteraktionen simulieren und bedürfen für eine realistische Simulation eine Erweiterung. Eine eigens kreierte Physik zu Realisierung von Wirbelkräften in einem Partikelsystem bedarf einer Einarbeitung und Umsetzung in dieser Thematik, die zu Zeitaufwändig gewesen wäre. Die Simulation kann durch das Anpassen der Variablen ebenfalls für andere Simulationen genutzt werden, wie zum Beispiel Wasser oder andere Flüssigkeiten deren Physik rein auf der Partikelinteraktion in einem Partikelsystem basieren kann.

Die Beschleunigung des Systems war besonders erfolgreich. Die Implementation beider Beschleunigungsverfahren erzielte ihre Vorgabe, sodass das Partikelsystem in Echtzeit simuliert werden konnte. Dabei erwies sich der Countingsort mit den persönlichen Anpassungen als performanter bei einer hohen Partikelanzahl, sowie effizienter im Speicherverbrauch als das Speicherverfahren. Außerdem besitzt der Countingsort eine stabilere Laufzeit als das Speicherverfahren, da dieses durch die scattered reads teilweise sehr starke Schwankungen in der Performance aufweisen.

Das Integrieren der Simulation in eine Engine wäre deshalb mit dem Countingsort möglich und empfehlenswert.

In den folgenden Bildern sieht man die initiale Ausbreitung der Partikel. Diese wurden in Form einer Kugel im unteren Drittel gesetzt. Da die Partikel eine hohe Temperatur besitzen und sich wegen der Dichte zunächst aufheizen, steigen sie auf. Außerdem breiten sie sich auf Grund der Druckkraft aus und bewegen sich wegen der Viskosität in die gleiche Richtung. Nach dem Aufprallen an der Decke sammeln sich die Partikel zunächst an dieser, bevor sie wegen zu geringer Dichte an Temperatur verlieren und mit der Zeit absinken.





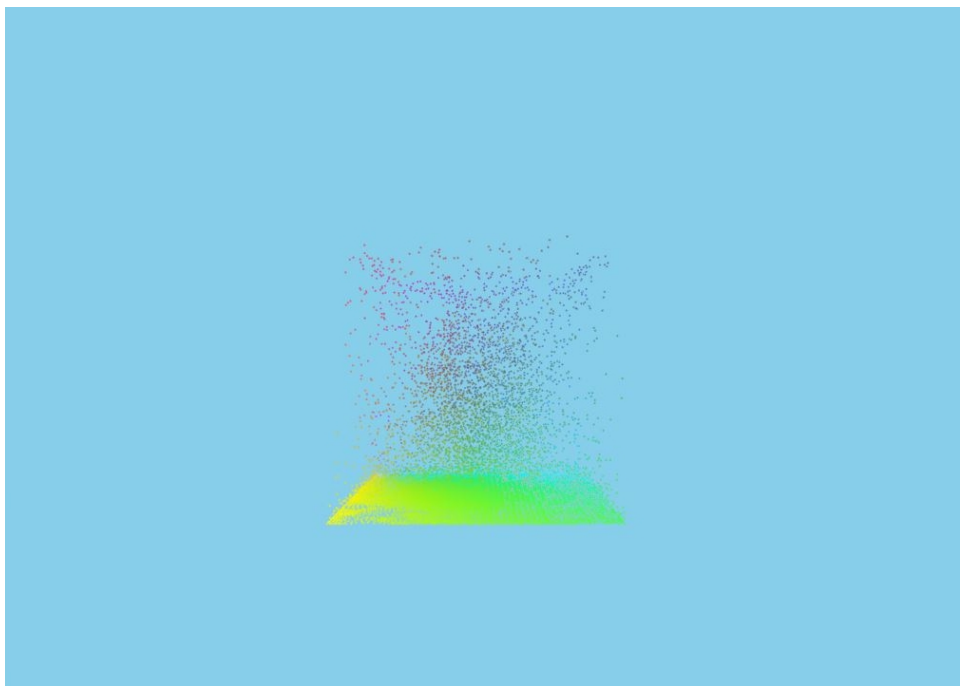
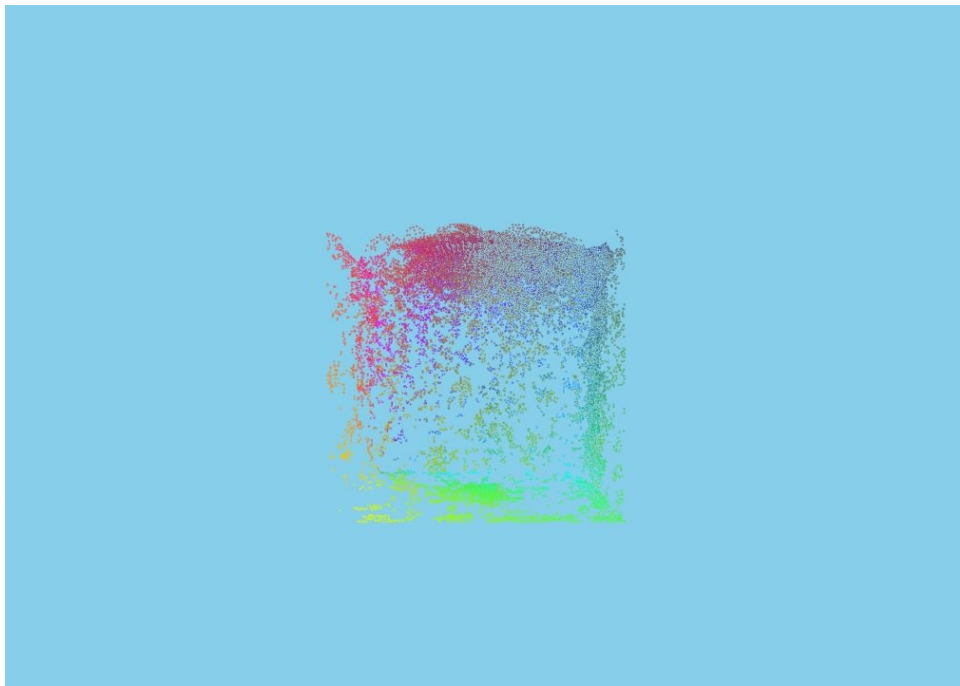


Abbildung 27: Simulationsverlauf des Rauches

8 Ausblick

Die vorliegende Fluidsimulation bietet viele Möglichkeiten zur Erweiterung. Eine **angedachte** wäre die Realisierung einer Technik zur besseren Berechnung der Wirbelkraft, da diese einen großen optischen Einfluss auf die Simulation hat. Diese könnte durch das Einbringen von Vektorfeldern oder einer ähnlichen Technik erfolgen. Vorteilhaft wäre es, wenn diese als Coroutine, unabhängig von der Berechnung der Partikelinteraktionen, berechnet werden könnte. Dadurch müsste diese nicht bei jedem Frame berechnet werden, sondern in einem festen Intervall berechnet werden, um entsprechend **Leitung** zu sparen.

Außerdem wurden bisher nur Point-Sprites gerendert, welche kein aufwendiges Rendering benötigen, aber nicht besonders optisch ansprechend sind. Deshalb würde sich hier ein realistisches Rendern der Partikel anbieten. Dabei könnten die Point-Sprites durch Texturen ersetzt werden, wie es bei Partikelsimulationen in Unity üblich ist. Eine übliche Art des Renderings bei Fluiden wäre ein **marching-cubes**-Algorithmus, welcher ein Mesh erstellen würde. Dies bedarf aber einen hohen Aufwand beim **erstellen** des Meshes. Müller [MSD07] schlägt aber alternativ ein Screen-Space-Fluid-Rendering vor, welches nur die nächstgelegene **Oberfläche, zur Kamera, rendern** würde. Dabei könnte gegenüber dem marching-cubes-Algorithmus wichtige Laufzeit gespart werden.

Um weitere Verbesserungen in der Performance zu erreichen, sollte man noch andere Beschleunigungsverfahren in Betracht ziehen und gegeneinander abwägen. Außerdem sollten weitere Anpassungen beim Counting-sort in Zukunft sich **positiv** auf dessen Beschleunigung des Systems auswirken. **Da dieser keine lineare Laufzeit, in Relation zur Partikelanzahl, hat.** Die genaue Ursache für diese Abweichung könnte untersucht und behoben werden.



Abbildungsverzeichnis

1	Fluidsimulation in Form des Vektorfeldverfahren Quelle: https://thumbs.gfycat.com/CelebratedElasticHartebeest-poster.jpg	4
2	Partikelsimulation von Wasser https://i.ytimg.com/vi/DhNt_A3k4B4/maxresdefault.jpg	5
3	Updateschleife der Physik	6
4	Bedeutung aller Symbole der Berechnungen	7
5	Die Gewichtungsfunktionen W_{poly6} , W_{spiky} , W_{visc} nach [MCG03] mit einem Radius von 1	9
6	Implementation der SPH Kernel	9
7	Implementation der Dichtefunktion	11
8	Anpassbare Variablen per ImGui	15
9	Partikel SSBOs zum lesen und schreiben	16
10	Partikel im Grid mit einem Radius von 1	18
11	Speichern der Partikel-IDs im Grid	19
12	Unsortierter Speicher beim Speicherverfahren	19
13	sortierter Speicher nach Sortieralgorithmus	20
14	Ausgangsarray beim Countingsort	21
15	Anzahl der Werte im Hilfsarray	21
16	Summe der Anzahl der Werte im Hilfsarray	21
17	vollständig sortiertes Array	21
18	Aufbau des Implementierten Countingsort	22
19	SSBO-Struct des Grids	23
20	registrieren der Partikel im Grid mit CubeID-Funktion	23
21	paralleler Prefixsum Scan von Navie nach Harris [HSO07]	24
22	umorganisieren der Partikel im Speicher	25
23	Berechnung der Normalen einen Partikel	25
24	Laufzeit in Relation zur Partikelanzahl	26
25	Laufzeit in Relation zur den Griddimensionen	26
26	Speicherverbrauch in Kb auf der Y-Achse und Griddimensionen auf der X-Achse	27
27	Simulationsverlauf des Rauches	31

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

- [MSD07] MÜLLER, Matthias ; SCHIRM, Simon ; DUTHALER, Stephan: Screen space meshes. In: *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* Eurographics Association, 2007, S. 9–15
- [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
- [TC78] TEMAM, Roger ; CHORIN, A: *Navier Stokes equations: Theory and numerical analysis*. 1978