UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Molekulardynamik: Simulation der Bewegung von Molekülen

Bachelorarbeit

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

vorgelegt von
Lukas Baulig

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

Koblenz, im September 2018

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)

Zusammenfassung

In dieser Bachelorarbeit wird ein System zur Simulation der Bewegung von Molekülen entworfen. Die Berechnungen der Kräfte zwischen chemisch gebundenen Atomen sowie zwischenmolekularer Kräfte werden fast vollständig auf der GPU durchgeführt. Die Visualisation der Simulation findet in einer interaktiven Bildwiederholrate statt. Um eine Darstellung in Echtzeit auf den meisten handelsüblichen Grafikkarten zur ermöglichen, sind geschickte Optimierungen und leichte Abstraktionen der physikalischen Modelle notwendig. Zu jeder Zeit kann die Ausführungsgeschwindigkeit der Simulation verändert oder vollständig gestoppt werden. Außerdem lassen sich einige Parameter der zugrundeliegenden physikalischen Modelle der Simulation zur Laufzeit verändern. Mit den richtigen Einstellung der Parametern lassen sich bestimmte Phänomene der Molekulardynamik, wie zum Beispiel die räumliche Struktur der Moleküle, beobachten.

Abstract

In this bachelor thesis a system for the simulation of the movements of molecules is developed. The calculation of the forces between chemically bonded atoms as well as intermolecular forces is done almost entirely on the GPU. The visualization of the simulation happens at an interactive framerate. To achieve rendering in realtime on off-the-shelf graphics cards, apt optimizations and slight abstractions of the underlying physical models are needed. One can control the execution speed or completely stop the simulation at any given moment. Some of the parameters of the underlying physical models of the simulation can be modified at runtime. With the right settings for the parameters, some phenomena of molecular dynamics can be observed, for example the spacial structure of the molecules.

Inhaltsverzeichnis

1 Einleitung

Auf Grund der stetig steigenden Rechenleistung von Grafikarten und dem Aufkommen von GPGPU (General Purpose Computation on Graphics Processing Unit) ist es uns heute möglich, mit Hilfe von paralleler Programmierung Berechnungen auf einer großen Anzahl von Recheneinheiten auszuführen. Dies ermöglicht uns, viele Berechnungen gleichzeitig durchzuführen und so sogar physikalische Systeme mit einer großen Anzahl an Elementen zu simulieren. Eine Anwendung, welche von der Verfügbarkeit von GPGPU und der Nutzung paralleler Algorithmen auf modernen Grafikkarten profitiert, sind Molekulardynamik (MD) - Simulationen.

Eines der schnellsten und bekanntesten Softwarepakete in dieser Kategorie ist *GROningen MAchine for Chemical Simulations* (GROMACS) [HKvdSL08]. GROMACS basiert auf *GROningen MOlecular Simulation* (GROMOS), welches schon seit 1978 entwickelt wird [vG18]. Die (Weiter-)Entwicklung von MD-Simulation ist aber noch heute ein interessantes, forschungsrelevantes Thema. Derartige Softwarepakete sind ein bewährtes Werkzeug zur Entwicklung neuartiger Medikamente (Wirkstoffdesign) [SLT09], aber auch zur Analyse von Proteinfaltungsprozessen, die zu einem besseren Verständnis von Krankheiten wie zum Beispiel Alzheimer [CL08] eingesetzt werden.

Im Zuge dieser Arbeit soll eine System zur Molekulardynamik-Simulation erstellt werden, welches die Möglichkeiten moderner GPU und Grafik-APIs wie OpenGL 4.3 *Compute Shader* zur parallelen Berechnung einer solchen Simulation aber auch zur Visualisierung in Echtzeit ausnutzt.

In Kapitel 2 sollen zu diesem Zweck zuerst die physikalische Grundlagen zur Entwicklung einer solchen MD-Simulation behandelt werden. Dazu muss die Aufbau von Molekülen, sowie die zwischen den einzelnen Atomen und Molekülen wirkende Kräfte und deren Berechnung verstanden werden. Es folgt in Kapitel 3 die Konzeption des Systems, dort werden die Ziele der Simulation definiert sowie die Modellannahmen, Randbedingungen und verwendete Techniken besprochen. Die Implementation des Systems wird in Kapitel 4 erläutert. Unter anderem werden die verschiedenen *Compute Shader*-Stufen zur Simulation der Molekülbewegung auf der GPU vorgestellt. Anschließend folgt in Kapitel 5 eine Bewertung der Performance und der erreichten Ziele. Zum Abschluss werden in Kapitel 6 Möglichkeiten zur Verbesserung und Erweiterung der Simulation dargelegt.

2 Physikalische Grundlagen

Im folgenden Kapitel werden die physikalischen Grundlagen einer MD-Simulation erläutert. Zunächst soll der atomare Aufbau eines Moleküls und anschließend die auf die einzelnen Atome der Moleküle wirkenden Kräfte anschaulich gemacht werden.

2.1 Atome

Atome sind die kleinste Einheit im Kontext dieser Arbeit. Sie werden als kugelförmige Teilchen angenommen und besitzen eine(n) spezifische(n) Radius und Masse. Da ein absoluter Atomradius aufgrund der Quantenmechanischen Natur der Atome und damit einhergehender statistischer Verteilung der Elektronen um den Atomkern nicht angegeben werden kann, ist im weiteren mit Atomradius stets der *Kovalenzradius* der Atome gemeint. Der *Kovalenzradius* kann durch die Analyse der Kristallstruktur von Molekülen, deren Atome kovalent gebunden sind, ermittelt werden[CGPP$^+$08]. Zur Berechnung der intermolekularen *Van-der-Waals-Kräfte* (siehe Abschnitt 2.4) wird ein weiterer, vom Kovalenzradius abweichender Wert genutzt, der *Van-der-Waals-Radius*. Auch dieser ist experimentell bestimmbar.

2.2 Moleküle

Moleküle sind aus zwei oder mehreren Atomen aufgebaute Teilchen. Die Atome eines Moleküls werden durch chemischen Bindungen zusammengehalten. Die Atombindungen eines Moleküls haben spezifische Bindungslängen und Bindungswinkel, welche sowohl von dem Element des Atoms sowie von den an den Atombindungen beteiligten Elektronen abhängig und für jede Atombindung charakteristisch sind. Die Bindungslängen und -winkel können wie die Kovalenzradien mittels Kristallstrukturanalyse ermittelt werden.

2.3 Intramolekulare Kräfte

Intramolekulare Kräfte verbinden Atome zu Molekülen. In dieser Simulation wird als solche intramolekulare Kraft nur die kovalente Bindung von Atomen betrachtet. Eine kovalente Bindung kommt zustande, wenn die Elektronen zweier Atome mindestens ein Elektronenpaar bilden. Diese Elektronenpaarbildung führt insgesamt zu einem stabileren, niedrigeren Energiezustand beider Atome. Es muss also Energie zugeführt werden, um die Bindung wieder zu lösen. Die Energie, welche nötig ist um die Bindung zu lösen, wird Bindungsenergie genannt, und ist gleichzeitig ein Maß für die stärke der Bindung. Die unterschiedlichen Bindungsenergien führen zu den Längenunterschieden der chemischen Bindungen.

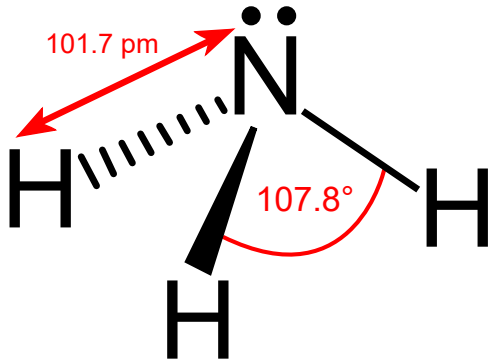

Abbildung 1: Bindungswinkel und -längen des Ammoniak-Moleküls. Zu sehen ist außerdem ein mit zwei Punkten angedeutetes freies Elektronenpaar. Diese beide Elektronen besetzen ihr zugehöriges Orbital im Stickstoffatom vollständig und sind nicht an weiteren Atombindungen beteiligt, stoßen die übrigenen Elektronen aber ab und führen so zu den angegebenen Bindungswinkeln.
`https://commons.wikimedia.org/wiki/File:Ammonia_2D_dimensions.svg`

2.4 Intermolekulare Kräfte

Als interamolekulare oder zwischenmolekulare Kräfte bezeichnet man diejenigen Kräfte, welche zwischen verschiedenen chemisch nicht gebundenen Molekülen wirken. In dieser Simulation wird die Anziehung zwischen Molekülen im speziellen durch die *London-Kraft* (auch *Londonsche Dispersionswechselwirkung*) simuliert. Als Teil der *Van-der-Waals-Kräfte* wird die *London-Kraft* auch *anziehenden Van-der-Waals-Kraft* oder *Van-der-Waals-Kraft im engeren Sinne* genannt. Diese Bezeichnungen werden hier synonym verwendet. Ursächlich für die Anziehungskraft zwischen nicht miteinander chemisch gebundenen Molekülen sind die aus quantenmechanischer Sicht unbestimmten Aufenthaltsorte der Elektronen eines Atoms, und der daraus resultierenden ständig wechselnden Ladungsverteilung. Falls der Schwerpunkt der elektronischen Ladungsverteilung innerhalb eines Atoms nicht mit dem Schwerpunkt des Atomkerns zusammenfällt (der die gesamte positive Ladung trägt), entsteht ein temporärer Dipol, der so mit anderen (temporären) Dipolen wechselwirken kann. Die *London-Kraft* ist allerdings im Vergleich zu den intramolekularen Kräften eher schwach, und deren Wechselwirkungsenergie nimmt mit etwa der sechsten Potenz des Abstandes der Atome ab. Sie ist also proportional zu $\frac{1}{r^6}$.

2.5 Kraftfeld

Als Kraftfelder (engl. *force field*) werden in der computergestützten Physik und Chemie Computersimulationen zur Berechnung der Bewegung von Molekülen auf Basis der klassischen (Newtonschen) Mechanik bezeichnet.

Meistens ist mit „Kraftfeld“ sowohl dessen funktionale Form als auch dessen Parameterset gemeint. Viele Molekulardynamik-Simulation sind „Kraftfelder“. Ein bekanntes solches Kraftfeld ist das AMBER (Assisted Model Building with Energy Refinement) Kraftfeld [CCB+95], welches als Inspiration für diese Arbeit dient. Das AMBER Kraftfeld nutzt zur Berechnung der potentiellen Energie die folgenden Terme:

$$E_{bonds} = K_r \cdot (r - r_{eq})^2 \tag{1}$$

E_{bonds} ist die Energie, die beim Dehnen oder Stauchen der chemischen Bindung zwischen zwei Atomen auftritt (engl. *stretching energy*). r ist gleich dem Abstand der beiden Atome, den diese derzeitig in der Simulation haben. r_{eg} ist die Bindungslänge der chemischen Bindung im Ruhezustand (engl. *equilibrium bond length*). $(r - r_{eq})^2$ ist somit die quadrierte Abweichung des Abstandes von der Bindungslänge im Ruhezustand. K_r ist ein Skalierungsfaktor, der die Energie und Abweichung in Relation zueinander setzen.

$$E_{angles} = K_\theta \cdot (\theta - \theta_{eq})^2 \tag{2}$$

Term (2) zur Berechnung der Biegeenergie (engl. *bending energy*) ist bis auf die Parameter identisch zu Term (1), der Dehnungsenergie. θ ist in diesem Fall der gemessene Winkel zwischen zwei Atomen. θ_{eq} ist der Winkel im Ruhezustand. $(\theta - \theta_{eq})^2$ ist gleich der quadrierte Abweichung des Winkels vom Ruhezustand. K_θ ist wieder ein Skalierungsfaktor.

$$E_{non-bonded} = \left[\frac{A_{ij}}{{R_{ij}}^{12}} - \frac{B_{ij}}{{R_{ij}}^{6}} + \frac{q_i q_j}{\epsilon R_{ij}} \right] \tag{3}$$

Dieser Term berechnet die Energie zwischen ungebundenen Atomen. Der vordere Teil, $\frac{A_{ij}}{{R_{ij}}^{12}} - \frac{B_{ij}}{{R_{ij}}^{6}}$, ist eine Form des sog. *Lennard-Jones-Potential* [LJ24]. Der hintere Teil $\frac{q_i q_j}{\epsilon R_{ij}}$ beschreibt die elektrostatische Anziehungskraft, diese wird in der Implementation allerdings nicht verwendet.

Das *Lennard-Jones-Potential* nähert die Wechselwirkungen zwischen nicht chemisch aneinander gebundenen Atomen an. Wie in Abschnitt 2.4 beschrieben, nimmt die *London-Kraft* mit etwa der sechsten Potenz des Abstandes der Atome ab. Dies entspricht dem Ausdruck $\frac{B_{ij}}{{R_{ij}}^6}$. Der erste Teil, $\frac{A_{ij}}{{R_{ij}}^{12}}$, ist der abstoßende Anteil des Potentials. Dieser abstoßende Anteil kommt zustande, da die Elektronen zweier Atome bei Annäherung nach dem Pauli-Prinzip nicht das selbe Orbital besetzen können und sich deshalb stark abstoßen.

In vielen Implementationen ist das *Lennard-Jones-Potential* in der Form

$$V(r) = \varepsilon \left[\left(\frac{r_m}{r} \right)^{12} - 2 \left(\frac{r_m}{r} \right)^{6} \right] \tag{4}$$

definiert. Der abstoßende Anteil $\left(\frac{r_m}{r}\right)^{12}$ wurde so gewählt, da sich der Term folgendermaßen umformen lässt:

$$V(r) = \varepsilon \cdot \left(\frac{r_m}{r} \right)^{6} \cdot \left[\left(\frac{r_m}{r} \right)^{6} - 2 \right] \tag{5}$$

In dieser Form wird das *Lennard-Jones-Potential* auch in der Implementierung verwendet.

3 Konzeption

3.1 Ziele

Auf Basis der in Kapitel 2 erarbeiteten Grundlagen soll im folgenden Kapitel der Programmcode für einer interaktive Molekulardynamik-Simulation konzipiert werden, welche auf Computern mit handelsüblichen Grafikkarten lauffähig sein soll. Die Simulation der Bewegung soll darüber hinaus ansprechend visualisiert werden. „Interaktiv“ bedeutet in diesem Kontext, die Simulation der Bewegung sowie die Visualisierung der Moleküle in akzeptabler Bildwiederholrate zu bewerkstelligen. Außerdem soll der Nutzer während der Laufzeit noch gewisse Parameter der Simulation mittels Maus und Tastatur verändert können. Dabei steht die Interaktivität im Fokus, sodass sich die Simulation den gegebenen Möglichkeiten und den Wünsche des Nutzers anpassen können soll.

3.2 Modellannahmen

Um die Interaktivität der Simulation zu gewährleisten, müssen Modellannahmen getroffen und die physikalischen Kräfte möglichst effizient berechnet oder gegebenenfalls abstrahiert werden. Zu diesem Zweck beschränkt sich die Simulation auf die Berechnung von kovalenten Atombindungen. Außerdem werden nur vordefinierte Moleküle (Wasser, Ammoniak, Methan) mit Einfachbindungen zwischen den Atomen unterstützt. Außer den in Abschnitt 2 genannten inter- und intramolekularen Kräften werden keine weiteren Einflüsse wie z.B. Gravitation, Temperatur oder Druck simuliert.

3.3 Anfangs- und Randbedingungen

Zu Beginn der Simulation sollen alle Moleküle gleichmäßig im Raum verteilt sein. Da alle Moleküle in der Simulation aus mindestens 3 Atomen bestehen, soll jeweils das zentrale Atom des Moleküls an einer zufällig Position generiert, die übrigen Atomen in Abhängigkeit ihrer Bindungslängen und -winkel um das zentralen Atom verteilt werden. Damit sich die Bindungslängen und -winkel der Atome nicht sofort im Gleichgewichtszustand befinden sollen diesen etwas normalverteiltes Rauschen hinzugefügt werden, da sich die Atome eines Moleküle ansonsten untereinander kaum bewegen. Die Geschwindigkeit der Moleküle soll auf einen Wert gesetzt werden, der einer Normalverteilung um einen vordefinierten, im Bezug zur Simulation realistischen Wert entspricht.

3.4 Verwendete Techniken

3.4.1 Partikel

Die Moleküle sollen durch einfache Partikel abstrahiert werden. Dazu müssen alle relevanten Moleküleigenschaften in Partikelattribute abgebildet werden, die die Bewegung der Moleküle und die Interaktion untereinander sinnvoll beschreiben.

3.4.2 Sortierung

Um die zwischen den einzelnen Molekülen bzw. Atomen wirkende Kräfte effizient berechnen zu können, sollen die Partikel in Abhängigkeit ihrer Entfernung zueinander in Gruppen eingeteilt werden. Dadurch muss jedes Partikel nur noch diejenigen Kräfte berechnen, die durch andere Partikel in dessen unmittelbarer Nachbarschaft bewirkt werden. Bis zur welchen Entfernung die Kräfte berechnet werden, soll dabei einstellbar bleiben.

3.4.3 Kräfte

Die Kräfte sollen Anhand der in Kapitel 2 beschriebenen physikalischen Gesetzte und Zusammenhänge simuliert werden. Falls notwendig, soll deren Berechnung insofern vereinfacht oder abstrahiert werden, dass die Simulation die in Abschnitt 3.1 definierten Ziele soweit wie möglich erfüllt.

3.4.4 Integration

Um aus den auf die Partikel wirkenden Kräften die neuen Positionen und Geschwindigkeiten der Partikel zu berechnen, muss ein geeignetes Verfahren ausgewählt werden, dass die numerische Integration der Bewegungsgleichung löst. Die Berechnung sollte einerseits möglichst effizient sein, andererseits gleichzeitig akzeptable Ergebnisse liefern. Auch muss bei der Integration sichergestellt werden, das die Geschwindigkeiten und Positionen der Partikel festgelegte Grenzwerte nicht überschreiten.

3.4.5 Visualisierung

Die Moleküle sollen ansprechend visualisiert werden. Die einzelnen Partikel, die jeweils ein Atom repräsentieren, sollen als Kugeln im Raum an deren jeweiliger Position dargestellt werden. Um die Atome, aus denen ein Molekül aufgebaut ist, unterscheidbar zu machen, sollen diese je nach ihrem chemischen Element eingefärbt und in Abhängigkeit ihrer Größe (Kovalenzradius) skaliert werden.

3.5 Zusammenfassung des Simulationsalgorithmus

Der Simulationsalgorithmus soll also Iterativ die folgenden Schritte Abarbeiten:

0. Generiere und Initialisiere die Partikel, welche die Atome repräsentieren.
1. Sortiere jedes Partikel nach Abhängigkeit dessen Position im Simulationsraum in die Gruppe des entsprechenden Gitterabschnitts ein
2. Berechne für jedes Partikel die auf dieses wirkenden Kräfte unter zuhilfenahme der in Schritt 1 generierten Suchstruktur.
3. Berechne die neue Position und Geschwindigkeit eines jeden Partikels per numerische Integration.
4. Stelle die Moleküle da
5. Solange die Simulation läuft, wiederhole die Schritte 1 - 4.

4 Implementation

mit einen normalverteilten Faktor multipliziert werden. Die Geschwindigkeit der Moleküle wird auf einen Wert gesetzt, der einer Normalverteilung um einen vordefinierten Wert entspricht. Diese Wert ist an die mittlere Geschwindigkeit eines Teilchens in einem idealen Gas (siehe Maxwell-Boltzmann-Verteilung) angelehnt.

4.1 Aufbau

Die Implementation nutzt OpenGL 4.3 zur Berechnung und Visualisierung der Molekulardynamik-Simulation. Mit *Compute-Shadern* steht dem Entwickler seit OpenGL4.3 ein einfaches Mittel zur Verfügung, beliebige Berechnungen parallele auf der Grafikkarte durchzuführen. Diese frei programmierbaren Shader werden in dieser Arbeit umfangreich genutzt, so finden jegliche Schritte der Simulation, von der Berechnung des Gitters bis zur Visualisierung der Moleküle, vollständig auf der Grafikkarte statt.

Hierzu werden die einzelnen Berechnungsschritte in mehreren Stufen in verschiedenen OpenGL *Compute-Shadern* nacheinander abgearbeitet. Die CPU verarbeitet die Eingaben des Benutzers, gibt diese an die *Compute-Shadern* weiter und steuert den Aufruf und Ablauf der verschiedenen *Compute-Shadern*-Stufen. Zur Verwaltung der Kommunikation zwischen CPU und GPU wurde ein objektorientiertes Framework geschaffen, das die Kommunikation vereinfacht. Das Framework basiert dabei zu Teilen auf bestehendem Code, der vom Institut für Computervisualistik der Universität Koblenz erstellt und gepflegt wird. Außerdem werden die Bibliotheken

GLFW, zur Handhabung von Nutzereingaben, zur Erzeugung eines OpenGL Kontexts und des Fensters;

gl3w zum dynamischen Laden des OpenGL *Core Profiles*; sowie

imgui zur Bereitstellung der Grafischen Benutzeroberfläche genutzt.

Diese Bibliotheken sind frei zugänglich und quelloffen.

4.2 Datenstrukturen

Die Simulation nutzt eine Vielzahl an Datenstrukturen, die im folgenden kurz erläutert werden. Die Daten müssen allesamt in sog. *Shader Storage Buffer Objects* (SSBO) gespeichert werden, um von den *Compute-Shadern* verarbeitet werden zu können. Auf die einzelnen Elemente der SSBOs kann parallel zugegriffen werden, es sind aber auch atomare Operationen auf die Werte in SSBOs möglich, die über alle Instanzen der *Compute-Shader* synchron ablaufen.

Die Atom- bzw. Partikeleigenschaften, auf die von verschiedenen *Compute-Shadern* häufig gemeinsam zugegriffen werden, werden auch als Gruppe zusammengefasst in einem SSBO abgespeichert. Attribute, die weniger häufig

vorkommen bzw. nur von einzelnen Shadern benötigt werden, sind über separate SSBOs zugänglich. Außerdem wurde auf das Padding der im SSBO abgespeicherten Daten geachtet. Dies ist notwendig, da OpenGL ein bestimmtes Layout der im SSBO befindlichen Daten erfordert (*layout std430*), welches in bestimmten Fällen Padding-Bytes verursacht.

4.2.1 Partikel

Ein Partikel, welches in der Simulation ein einzelnes Atom repräsentiert (Atom und Partikel werden im folgenden synonym verwendet), besitzt einen dreidimensionalen Positionsvektor, einen eindimensionalen Kugelradius des Partikels (die Größe ist äquivalent zum Kovalenzradius des Atoms); sowie einen dreidimensionale Geschwindigkeitsvektor und eine eindimensionale Masse. Diese Partikelattribute werden zusammengefasst in folgender Struktur im SSBO abgespeichert:

```
struct Particle
{
vec4 position_and_radius;
vec4 velocity_and_mass;
};

layout(std430, binding = 0) buffer particles_ssbo
{
Particle particles[];
};
```

Da jedes Moleküle in der Simulation aus mehreren Atomen besteht, wird in einem gesonderten SSBO die Atombindungen der Atome gespeichert. Jedes Atom in der Simulation kann Teil von maximal vier Bindungen gleichzeitig sein. Für jedes Partikel werden deshalb die Indizes der vier möglichen Bindungspartner in einem *ivec4* gespeichert. Falls das jeweilige Partikel weniger als vier Bindungspartner hat, wird an den restlichen Indices des Wert -1 gespeichert, um dies zu kennzeichnen. Das SSBO der Atombindungen (im Folgenden *bonds_ssbo*) hat somit folgende Form:

```
layout(std430, binding = 8) buffer bonds_ssbo
{
ivec4 bonds[];
};
```

4.2.2 Kräfte

Die Kräfte, welche auf ein Partikel wirken, werden in einem vierdimensionalen Vektor (*vec4*) gespeichert. Die x, y, und z Komponenten geben den Richtungsvektor der Kraft an. Die vierte Koordinate hat für die Simulation keine Bedeutung, ist aber notwendig, um das geforderte Speicherlayout (*layout std430*) zu erfüllen, da dieses ein 4, 8 oder 16 Byte alignment der Daten im SSBO erfordert (*vec4* hat die Größe 4 * 4 Byte = 16 Byte , wohingegen *vec3* nur 12 Byte groß ist, weshalb 4 Byte padding nötig sind). Der in dieser Simulation genutzte Integrator (siehe 4.5) benötigt zu Berechnung der Bewegung der einzelnen Partikel sowohl die aktuelle auf die Partikel wirkenden Kräfte sowie auch die Kräfte der vorherigen Iteration. Aus diesem Grund gibt es das dazugehörige SSBO in zweifacher Ausführung:

```
layout(std430, binding = 5) buffer force_old_ssbo
{
vec4 forces_old[];
};

layout(std430, binding = 6) buffer force_new_ssbo
{
vec4 forces_new[];
};
```

4.2.3 Suchdatenstruktur

Zur effizienten Berechnung der Kräfte zwischen den einzelnen Partikeln (Atome der Moleküle) ist es notwendig, die Partikel Anhand ihrer Position im Simulationsraum in eine Gittergruppen einzuteilen. Hierfür wird das in 4.3 vorgestellte verfahren genutzt. Zur Berechnung dieses Verfahrens werden weitere SSBOs benötigt, die dort explizit genannt werden.

4.3 Präfix-Summe und Sortierung

Die Sortierung der Partikel in Gruppen basiert auf dem *Counting Sort*-Algorithmus [CCL+01]. Die Sortierung folgt in 3 Schritten in jeweils einem eigenen *Compute Shader*.

4.3.1 Gruppieren und Zählen

Im ersten Schritt (ersten *Compute Shader*) müssen die Partikel in Gruppen eingeteilt und gezählt werden. Der gesamte Simulationsraum (bzw. die *bounding box*) ist dabei so aufgeteilt, dass jede Gruppe ein Volumen von

mindestens $(4\mathring{A})^3$ ($1\mathring{A}$ (*Ångström*) enspricht 10^{-10} Meter bzw. 100 Pikometer. Atomabstände werden in *Ångström* gemessen.) des Simulationsraumes abdeckt, also die Seiten des enstprechenden Würfels jeweils mindestens $4\mathring{A}$ lang sind. Die Länge von $4\mathring{A}$ wurde gewählt, da ab dieser Länge die zwischenmolekulare Anziehungskraft Aufgrund der *London-Kräfte* vernachlässigbar wird, und Partikel die weiter als $4\mathring{A}$ voneinander entfernt sind nicht mehr miteinander wechselwirken. Jedem Partikel wird Anhand einer Abbildung, welche in Abhängigkeit der Größe des Simulationsraumes sowie der Auflösung des Gitters berücksichtigt, die jeweilige Gruppen-ID zugeordnet. Der Codeabschnitt 1 (*computeGridIndex*) zeigt diese Abbildung. Die Größe des Simulationsraumes (bzw. der *bounding box*) wird dem Algorithmus mittels der ***uniform***-Variablen *bmin* bzw. *bmax* übergeben. Diese entsprechen den beiden diagonal gegenüberliegenden Eckpunkten des Quaders. Die Auflösung des Gitters wird wiederum mit der ***uniform***-Variablen *gridres* übergeben.

Dabei wird im SSBO *counts_ssbo* gezählt, wie viele Partikel zu jeweiligen Gruppe gehören. Um die Anzahl der Elemente in der jeweiligen Gruppe zählen zu können nutzt der *Compute Shader* atomare Addition (*countIndex = atomicAdd(counts[index], 1);*, siehe 1). Dies garantiert, dass die Speicherstelle im SSBO während der Ausführung der Addition von keinem anderen Shader gleichzeitig beschrieben werden kann. Desweitern gibt die *atomicAdd* Funktion den Speicherinhalt **vor** der Addition zurück, diesen Wert *countIndex* speichert sich das jeweilige Partikel im SSBO *count_index_ssbo*, da dieser Wert später für die Sortierung genutzt wird.

4.3.2 Berechnung der Präfix-Summe

Als nächstes muss die *Präfix-Summe* des im vorigen Schritt erstellten *counts_ssbo* Array berechnet werden. Hierfür wird das Verfahren nach Hillis-Steele verwendet [HS86], welches die Berechnung teilweise parallelisiert. Abbildung 2 zeigt den grundsätzlichen Ablauf des Algorithmus. Der vollständige Shader-Code ist in Abschnitt 2 aufgelistet und entspricht weitestgehend der Referenzimplementation. Da allerdings in jedem Schritt das Ergebnis der Summation gespeichert werden muss, ohne dabei den Originalwert zu überschreiben, werden zwei seperate SSBOs genutzt, die abwechselnd als Quelle/Ziel der Berechnung dienen.

4.3.3 Sortierung der Partikel

Im letzten Schritt der Sortierung (und damit auch im letzten *Compute Shader*) werden schlussendlich die aus den in Abschnitt 4.3.1 und Abschnitt 4.3.2 berechneten Werten zusammengefügt und damit eine sortierten Liste aller Partikel erstellt. Sei i der Index einer Gruppe, und j der Index eines Partikels in Gruppe i. Es gilt, das der Wert *sum_counts[i]* gleich der Summe der Anzahl aller Partikeln in den Gruppen mit Index $\leq$ i ist.

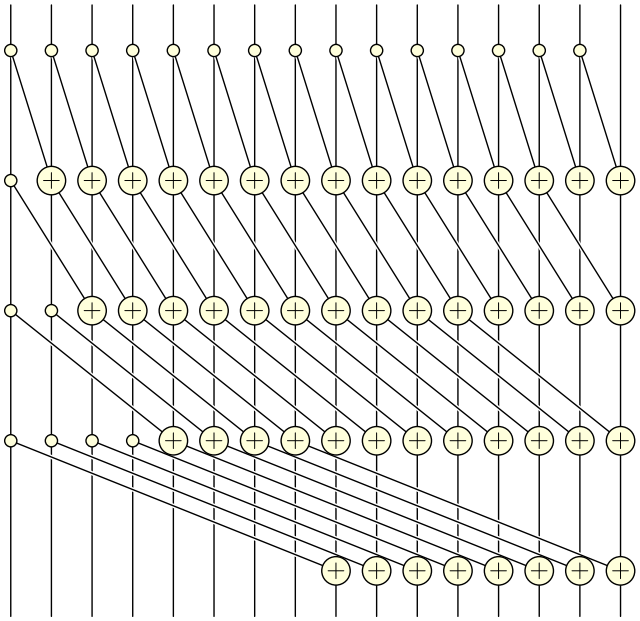

Abbildung 2: Die Berechnung der Präfix-Summe von einer Menge von n Elementen erfolgt parallel, wobei $log(n)$ Iterationen nötig sind. In jedem Schritt wird das auf Element mit Index i das Element mit Index $i-m$ aufaddiert. Der Indexoffset m verdoppelt sich in jedem Schritt.
`https://commons.wikimedia.org/wiki/File:Hillis-Steele_Prefix_Sum.svg`

sum_ counts[i] $= \sum_{n \leq i} |groups[i]|$
Daraus folgt das *sum_ counts[i]* - *sum_ counts[i-1]* genau der Anzahl an Partikeln in Gruppe i entspricht. Dies kann dazu genutzt werden, den Index des ersten Partikels in jeder Gruppe zu bestimmen, so entspricht *sum_ counts[i-1]* genau dem Index des ersten Partikels aus der Gruppe mit Index i. Da wird aber in Abschnitt 4.3.1 im SSBO *count_ index_ ssbo* zudem den Index der Partikel **innerhalb** der Gruppe abgespeichert haben, kann der Index eines jeden Partikels j in der sortierten Liste direkt ausgerechnet werden, und entspricht:

new_ particle_ index = sum_ counts[i-1] + count_ index[j]

In der sortierten Liste wird nun also nur noch dem Element an Index *new_ particle_ index* der Index des Partikels j zugewiesen:

sorted_ index[new_ particle_ index] = j

Damit ist die Sortierung der Partikel abgeschlossen.

4.4 Kräfteberechnung

Zur Berechnung der auf die Partikel wirkenden Kräfte wird für jedes Partikel i eine Instanz des Kräfte-Shaders aufgerufen. Der Kräfte-Shader prüft für

jedes Partikel j in der „Nachbarschaft“ von Partikel i den Abstand der beiden Partikel zueinander. Die „Nachbarschaft“ ist definiert als alle Partikel, die entweder in der selben Gittergruppe wie Partikel i oder in einer der direkt angrenzenden Gittergruppen sind. Da die Partikel nach dem in Abschnitt 4.3.3 beschriebenen Verfahren bereits in Gruppen sortiert vorliegen, ist das Finden der Partikel in der Nachbarschaft trivial. Des weiteren ist durch die Festgelegte Größe der einzelnen Gitterzellen garantiert, das alle Partikel im Umkreis von $4\mathring{A}$ um Partikel i geprüft werden.

Ist ein Partikel j in der Nachbarschaft von Partikel i nun also gefunden worden, werden folgende 3 Fälle geprüft:

Fall 1 Die beiden Partikel sind direkt chemisch miteinander gebunden. Dies kann direkt über das in Abschnitt 4.2.1 beschriebene *bonds_ssbo* der beiden Partikel überprüft werden, da in den Listen der gespeicherten Bindungen der beiden Partikel das jeweils andere vorkommen muss.

Fall 2 Die beiden Partikel sind nicht direkt chemisch miteinander gebunden, sind aber Teil desselben Moleküls. Dies kann ebenso über das *bonds_ssbo* der beiden Partikel getestet werden, da in den Listen der gespeicherten Bindungen der beiden Partikel i und j ein weiteres Partikel k vorkommen muss, das in beiden Listen vorkommt.

Fall 3 Die beiden Partikel sind weder direkt chemisch miteinander gebunden noch Teil des selben Moleküls, dies ist der „Default-Case“ und ergibt sich aus den vorherigen beiden Fällen.

In Abhängigkeit der 3 beschriebenen Fälle werden nun die Kräfte zwischen den Partikeln berechnet. Die Berechnung der Kräfte orientiert sich an den Funktionen des in Kapitel 2 beschrieben *AMBER force field* (siehe 2.5). Da das *AMBER* Kraftfeld allerdings die potentielle Energie der Atome und nur indirekt die Kräfte zwischen einzelnen Atomen berechnet, werden die Terme im folgenden abstrahiert, um Kräftevektoren zu erhalten, die direkt zur Lösung der Bewegungsgleichung der Atome verwendet werden können.

4.4.1 Kraft zwischen chemische gebundenen Atomen

Zur Berechnung der *bond stretching force* $\overrightarrow{F_{bond}}$ zwischen zwei chemisch gebundenen Atomen (siehe Fall 1) wird folgender Term verwendet:

$$\overrightarrow{F_{bond}} = \frac{\vec{r}}{\|\vec{r}\|} \cdot (\|\vec{r}\| - r_{eq}) \cdot k_{bond} \cdot 0.5 \tag{6}$$

Dabei ist $\vec{r}$ der Vektor zwischen den beiden Partikeln. r_{eq} ist die Bindungslänge der chemischen Bindung zwischen den beiden Atomen und k_{bond} ist ein zur Laufzeit anpassbarer Proportionalitätsfaktor. Diese Form entspricht einer klassischen Federkraft $(\|\vec{r}\| - r_{eq}) \cdot k_{bond} \cdot 0.5$, die in Richtung

des normierten Vektors $\vec{r}$ wirkt (vgl. AMBER Term 1). k_{bond} kann deshalb auch als Federkonstante verstanden werden.

4.4.2 Winkelkraft

Zur Berechnung der *angle bending force* $\overrightarrow{F_{angle}}$ zwischen zwei Atomen i und j, die Teil desselben Moleküls mit dem zentralen Atom k sind (siehe Fall 2), wird folgender Term verwendet:

$$\begin{aligned} \vec{a} &= \frac{pos(i) - pos(k)}{\|pos(i) - pos(k)\|} \\ \vec{b} &= \frac{pos(i) - pos(k)}{\|pos(j) - pos(k)\|} \\ \theta &= acos(\vec{a} \cdot \vec{b}) \\ \vec{t} &= \vec{a} \times (\vec{b} \times \vec{a}) \\ \overrightarrow{F_{angle}} &= \vec{t} \cdot (\theta - \theta_{eq}) \cdot k_{angle} \cdot 0.5 \end{aligned} \tag{7}$$

Die Vektoren $\vec{a}$ und $\vec{b}$ entsprechen den normierten Richtungsvektoren von dem zentralen Atom k des Moleküls zu den beiden äußeren Atomen i und j. θ ist der Winkel zwischen $\vec{a}$ und $\vec{b}$. Das Kreuzprodukt $(\vec{b} \times \vec{a})$ ergibt den Drehvektor, um den sich $\vec{a}$ um den Winkel θ drehen müsste, um zu $\vec{b}$ zu gelangen. t ist ein Vektor, der tangential zu dieser Drehung liegt. Bei θ_{eq} handelt es sich um den Bindungswinkel der beiden Atome i und j, k_{angle} ist wieder ein Proportionalitätsfaktor bzw. die Federkonstante. Prinzipiell handelt es sich auch bei Term 7 um eine Federkraft, diesmal in Richtung der Tangente. Dieser Term entspricht Formel 2 im AMBER Kraftfeld.

4.4.3 Zwischenmolekulare Kraft

Zuletzt wird zur Berechnung der zwischenmolekularen Kräfte (siehe Fall 3) analog zu Term 3 im AMBER Kraftfeld eine auf dem *Lennard-Jones-Potential* basierende Formel verwendet. Diese hat eine ähnliche Struktur wie der Term zur Berechnung der Kräfte zwischen chemisch gebundenen Atomen (siehe 6) mit dem Unterschied, das nicht eine lineare Federkraft zur Skalierung des Kraftvektors sondern ein Faktor basierend auf dem *Lennard-Jones-Potential* genutzt wird:

$$\begin{aligned} r_{eq,vdW} &= r_{i,vdW} + r_{j,vdW} \\ a_{ij} &= r_{eq,vdW} / \|\vec{r}\| \\ \overrightarrow{F_{vdW}} &= \frac{\vec{r}}{\|\vec{r}\|} \cdot -[{a_{ij}}^6 - 2] \cdot {a_{ij}}^6 \cdot k_{vdW} \end{aligned} \tag{8}$$

$\vec{r}$ ist wieder der Vektor zwischen den beiden Atomen, $r_{i,vdW}$ und $r_{j,vdW}$ sind die *Van-der-Waals-Radien* der beiden Atome, und mit k_{vdW} lässt sich

der Kraftvektor wieder skalieren.

4.4.4 Zusammenfassung

Nachdem für jedes Partikel alle auf dieses wirkenden Kräfte berechnet worden, werden diese summiert und im SSBO *force_new_ssbo* gespeichert. Zuvor wird der alte Wert aber noch im SSBO *force_old_ssbo* gespeichert, da dieser noch von Belang für die Integration ist.

4.5 Integration

Zur Lösung der Bewegungsgleichungen der Partikel wird das Leapfrog-Verfahren in der *„kick-drift-kick“*-Form verwendet:

$$\begin{aligned} v_{i+1/2} &= v_i + a_i \frac{\Delta t}{2} \\ x_{i+1} &= x_i + v_{i+1/2} \Delta t \\ v_{i+1} &= v_{i+1/2} + a_{i+1} \frac{\Delta t}{2} \end{aligned} \tag{9}$$

Die Beschleunigungen a_i und a_{i+1} ergeben sich aus dem alten und neuen Kraftvektoren und der Masse des Partikels. Der *Integration-Shader* sorgt außerdem dafür, das Partikel den vordefinierten Simulationsraum (*bounding box*) nicht verlassen können. Außerdem können die Geschwindigkeiten sowohl gedämpft als auch beschleunigt werden.

4.6 Visualisierung

Zur Visualisierung der Moleküle werden Kugeln an den Positionen der Atome gerendert. Dabei werden die selben Vertexdaten (die eine Kugel darstellen) für jedes Atoms mittles *glDrawElementsInstanced* erneut in die OpenGL-Pipeline gegeben. Der *Vertex-Shader* bekommt von der Pipeline für jeder neue Instanz eine *gl_InstanceID* übergeben. Die *gl_InstanceID* entspricht dem Index eines Partikels. So kann der *Vertex-Shader* jeder Instanz der Kugel mittels einer Translationsmatrix an die Position des jeweiligen Atoms verschieben. Außerdem wird die Kugel auf den Atomradius des jeweiligen Partikels skaliert und in Abhängigkeit des Elements des Atoms eingefärbt. Der *Fragment-Shader* fügt außerdem noch Phong-Shading hinzu.

5 Evaluation

5.1 Performance

Es wurde getestet, inwiefern sich die Simulationsgeschwindigkeit bei verschieden großen Mengen an Partikeln ändert. Außerdem wurde zwei verschiedenen Implementationen miteinander verglichen. Einmal wurde bei der Simulation nur die Kräfte zwischen den Partikel in direkter Nachbarschaft (mittels der im Abschnitt 4.3) untereinander berechnet; während bei der zweiten, naiven Implementation die Kräfte zwischen allen Partikeln berechnet wurden.

Performance der naiven Implementation:

Anzahl Partikel	FPS	Simulationen pro Frame
1000	60	15
1000	30	27
1000	10	75
2000	60	7
2000	30	10
2000	10	22
3000	60	5
3000	30	7
3000	10	12

Performance nach vorheriger Sortierung:

Anzahl Partikel	FPS	Simulationen pro Frame
1000	60	17
1000	30	30
1000	10	81
2000	60	7
2000	30	12
2000	10	27
3000	60	5
3000	30	9
3000	10	14

Die Implementation mit vorheriger Sortierung der Partikel in Gruppen hat bei allen getestet Fällen zu einer höheren Simulationsgeschwindigkeit geführt. Zumal hat die Berechnung der Präfix-Summe fast keinen Einfluss auf die Zeit, welche ein Simulationsschritt benötigt. Der Größte Faktor für die Performance ist allerdings nicht die Simulation selbst, sondern die Visualisierung der Molekülen, die mit einer höheren Anzahl an darzustellender Atome immer aufwendiger und dementsprechend langwieriger wird.

5.2 Ergebnisse

Die Moleküle nehmen in den Simulation nach einiger Zeit ihrer natürliche, räumliche Struktur an 4, außerdem stoßen sich die Moleküle bei Kollision

gegenseitig ab. Des weiteren Schwingen die Moleküle um ihre Ruhelage, was sie in der Realität auch tun.

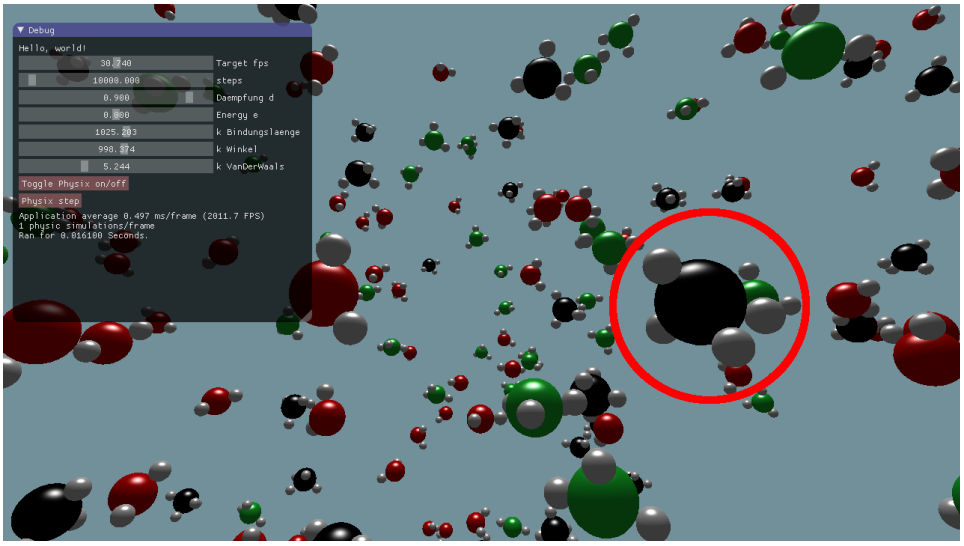

Abbildung 3: Das die weißen Wasserstoffatome des markierte Methan-Moleküls sind noch zufällig um das schwarze Kohlenstoffatom angeordnet, dies entsrpicht nicht ihrer natürlichen Form.

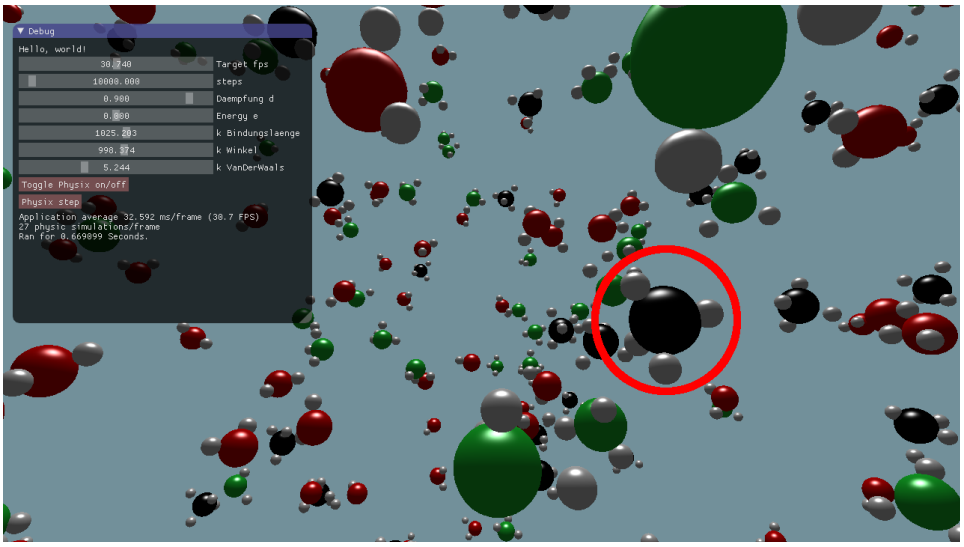

Abbildung 4: Die Wasserstoffatome des Methan-Moleküls haben sich ihrer trigonal-pyramidalen Form angenähert.

6 Ausblick

Die Simulation hat vielen Stellen, die ausgebaut oder verbessert werden könnten. Um auch andere Elemente simulieren zu können, muss für jedes neue Element die Parameter eingestellt werden. Auch die Berechnung der Kräfte ist noch sehr einfach gehalten. Es könnten außerdem weitere Terme zur Simulation von Elektrostatische Anziehungskraft, Wasserstoffbrückenbindung, Torsionskräfte und vieles weitere hinzugefügt werden. Speziell die Visualisierung ist noch sehr rudimentär und bietet Raum für Verbesserungen. Ein konkrete Verbesserungsmöglichkeit stellt die Präfixsumme da. In einer Präsentation von NVIDIA [Har07] wurde eine Implementation zur parallelen Berechnung der Präfixsumme vorgestellt, die potenziell weitaus performanter als die hier implementierte Lösung ist.

Anhang

Source Code 1: Berechnung der Gittergruppe und Zählen der Aufkommen von Partikeln

```
#version 430 core

layout(local_size_x = 1, local_size_y = 1, local_size_z=1) in ;

struct Particle
{
vec4 position_and_radius;
vec4 velocity_and_mass;
};

layout(std430, binding = 0) buffer particles_ssbo
{
Particle particles[];
};

layout(std430, binding = 1) buffer counts_ssbo
{
uint counts[];
};

layout(std430, binding = 2) buffer count_index_ssbo
{
uint count_index[];
};

uniform vec3 bmax;
uniform vec3 bmin;
uniform ivec3 gridres;


ivec3 computeGridCoordFromPos (vec3 p)
{
vec3 delta = bmax-bmin;
vec3 prel = (p-bmin)/delta;

vec3 gridres_temp = vec3(gridres);
gridres_temp = gridres_temp * prel;
ivec3 pgrid = ivec3(gridres_temp);
pgrid = clamp(pgrid, ivec3(0), gridres - ivec3(1,1,1));
```

```
return pgrid;
}

uint computeGridIndex(ivec3 coord)
{
return coord.z * gridres.y * gridres.x
+ coord.y * gridres.x
+ coord.x;
}

uint index3d(uvec3 c)
{
return c.z * gl_NumWorkGroups.x * gl_NumWorkGroups.y +
c.y * gl_NumWorkGroups.x +
c.x;
}

void main()
{
uvec3 coord = gl_GlobalInvocationID;
uint g_id = index3d(coord);

vec3 p = particles[g_id].position_and_radius.xyz;
ivec3 gridCoord = computeGridCoordFromPos(p);
uint index = computeGridIndex(gridCoord);

uint countIndex = 0;
countIndex = atomicAdd(counts[index], 1);
count_index[g_id] = countIndex;
}
```

Source Code 2: Berechnung der Präfix-Summe

```
#version 430 core

layout(local_size_x = 1, local_size_y = 1, local_size_z=1) in ;

layout(std430, binding = 1) buffer counts_ssbo
{
uint counts[];
};

layout(std430, binding = 3) buffer sum_counts_ssbo
```

```
{
uint sum_counts[];
};

layout(std430, binding = 7) buffer sum_counts_ssbo_new
{
uint sum_counts_new[];
};


uniform ivec3 gridres;
uniform uint stride;
uniform bool swapBuffer;

uint index3d(uvec3 c)
{
return c.z * gl_NumWorkGroups.x * gl_NumWorkGroups.y +
c.y * gl_NumWorkGroups.x +
c.x;
//return c.y * w * d + c.z * w + c.x;
}

void main()
{
uvec3 coord = gl_GlobalInvocationID;
uint g_id = index3d(coord);
uint offset = stride / 2;
g_id += offset;

if(swapBuffer)
{
if(g_id < stride)
{
uint x = sum_counts[g_id];
sum_counts_new[g_id] = x;
}
else
{
uint x = sum_counts[g_id];
uint x_minus_stride = sum_counts[g_id - stride];
sum_counts_new[g_id] = x + x_minus_stride;
}
}
else
```

```
{
if(g_id < stride)
{
uint x = sum_counts_new[g_id];
sum_counts[g_id] = x;
}
else
{
uint x = sum_counts_new[g_id];
uint x_minus_stride = sum_counts_new[g_id - stride];
sum_counts[g_id] = x + x_minus_stride;
}
}

}
```

Source Code 3: Berechnung der sortierten Liste

```
#version 430 core

layout(local_size_x = 1, local_size_y = 1, local_size_z=1) in ;

struct Particle
{
vec4 position_and_radius;
vec4 velocity_and_mass;
};

layout(std430, binding = 0) buffer particles_ssbo
{
Particle particles[];
};

layout(std430, binding = 1) buffer counts_ssbo
{
uint counts[];
};

layout(std430, binding = 2) buffer count_index_ssbo
{
uint count_index[];
};

layout(std430, binding = 3) buffer sum_counts_ssbo
```

```
{
uint sum_counts[];
};

layout(std430, binding = 4) buffer sorted_particles_index_ssbo
{
uint sorted_index[];
};

uniform vec3 bmax;
uniform vec3 bmin;
uniform ivec3 gridres;


ivec3 computeGridCoordFromPos (vec3 p)
{
vec3 delta = bmax-bmin;
vec3 prel = (p-bmin)/delta;

vec3 gridres_temp = vec3(gridres);
gridres_temp = gridres_temp * prel;
ivec3 pgrid = ivec3(gridres_temp);
pgrid = clamp(pgrid, ivec3(0), gridres-ivec3(1,1,1));

return pgrid;
}


uint computeGridIndex(ivec3 coord)
{
return coord.z * gridres.y * gridres.x
+ coord.y * gridres.x
+ coord.x;
}

uint index3d(uvec3 c)
{
return c.z * gl_NumWorkGroups.x * gl_NumWorkGroups.y +
c.y * gl_NumWorkGroups.x +
c.x;
}

void main()
{
```

```
uvec3 coord = gl_GlobalInvocationID;
uint g_id = index3d(coord);

vec3 p = particles[g_id].position_and_radius.xyz;
ivec3 gridCoord = computeGridCoordFromPos(p);
uint index = computeGridIndex(gridCoord);

// the grid index is the sum of all particles in all grids before this one,
// therefore we need to right shift the array

uint gridIndex = 0;
if(index>0)
{
gridIndex = sum_counts[index-1];
}

uint new_index = gridIndex + count_index[g_id];

sorted_index[new_index] = g_id;
}
```

Source Code 4: Berechnung der Kräfte

```
#version 430 core

layout(local_size_x = 1, local_size_y = 1, local_size_z=1) in ;

struct Particle
{
vec4 position_and_radius;
vec4 velocity_and_mass;
};

layout(std430, binding = 0) buffer particles_ssbo
{
Particle particles[];
};

layout(std430, binding = 1) buffer counts_ssbo
{
uint counts[];
};

layout(std430, binding = 2) buffer count_index_ssbo
```

```
{
uint count_index[];
};

layout(std430, binding = 3) buffer sum_counts_ssbo
{
uint sum_counts[];
};

layout(std430, binding = 4) buffer sorted_particles_index_ssbo
{
uint sorted_index[];
};

layout(std430, binding = 5) buffer force_old_ssbo
{
vec4 forces_old[];
};

layout(std430, binding = 6) buffer force_new_ssbo
{
vec4 forces_new[];
};

layout(std430, binding = 8) buffer bonds_ssbo
{
ivec4 bonds[];
};


uniform vec3 bmax;
uniform vec3 bmin;
uniform ivec3 gridres;

uniform float k_bond;
uniform float k_angle;
uniform float k_vanDerWaals;

ivec3 computeGridCoordFromPos (vec3 p)
{
vec3 delta = bmax-bmin;
vec3 prel = (p-bmin)/delta;

vec3 gridres_temp = vec3(gridres);
```

```
gridres_temp = gridres_temp * prel;
ivec3 pgrid = ivec3(gridres_temp);
pgrid = clamp(pgrid, ivec3(0), gridres-ivec3(1,1,1));

return pgrid;
}


uint computeGridIndex(ivec3 coord)
{
return coord.z * gridres.y * gridres.x
+ coord.y * gridres.x
+ coord.x;
}


int index (ivec2 c, int w, int h)
{
return c.y*w + c.x;
}

uint index3d(uvec3 c)
{
return c.z * gl_NumWorkGroups.x * gl_NumWorkGroups.y +
c.y * gl_NumWorkGroups.x +
c.x;
}

bool checkindex(uint min, uint max, uint current)
{
if (current < min || current >= max)
return false;
return true;
}



vec4 quat_from_axis_angle(vec3 axis, float angle)
{
vec4 qr;
float half_angle = (angle * 0.5) * 3.14159 / 180.0;
qr.x = axis.x * sin(half_angle);
qr.y = axis.y * sin(half_angle);
qr.z = axis.z * sin(half_angle);
```

```
qr.w = cos(half_angle);
return qr;
}

vec4 quat_conj(vec4 q)
{
return vec4(-q.x, -q.y, -q.z, q.w);
}

vec4 quat_mult(vec4 q1, vec4 q2)
{
vec4 qr;
qr.x = (q1.w * q2.x) + (q1.x * q2.w) + (q1.y * q2.z) - (q1.z * q2.y);
qr.y = (q1.w * q2.y) - (q1.x * q2.z) + (q1.y * q2.w) + (q1.z * q2.x);
qr.z = (q1.w * q2.z) + (q1.x * q2.y) - (q1.y * q2.x) + (q1.z * q2.w);
qr.w = (q1.w * q2.w) - (q1.x * q2.x) - (q1.y * q2.y) - (q1.z * q2.z);
return qr;
}

vec3 rotate_vertex_position(vec3 position, vec3 axis, float angle)
{
vec4 qr = quat_from_axis_angle(axis, angle);
vec4 qr_conj = quat_conj(qr);
vec4 q_pos = vec4(position.x, position.y, position.z, 0);

vec4 q_tmp = quat_mult(qr, q_pos);
qr = quat_mult(q_tmp, qr_conj);

return vec3(qr.x, qr.y, qr.z);
}

void main()
{
uvec3 coord = gl_GlobalInvocationID;

uint max_particles = gl_NumWorkGroups.x * gl_NumWorkGroups.y * gl_NumWorkGrou

uint g_id = index3d(coord);

vec3 pos = particles[g_id].position_and_radius.xyz;
vec3 vel = particles[g_id].velocity_and_mass.xyz;
float radius = particles[g_id].position_and_radius.w;

ivec4 own_bonds = bonds[g_id];
```

```
ivec3 gridCoord = computeGridCoordFromPos(pos);
uint grid_index = computeGridIndex(gridCoord);
int grid_size = gridres.x * gridres.y * gridres.z;
int grid_size_xy = gridres.x * gridres.y;

vec3 f = vec3(0);
vec3 n = vec3(0);
vec3 other = vec3(0);
float length_n = 0.0f;
vec3 norm_n = vec3(0);

for (int z = -1; z <= 1; z++)
{
for (int y = -1; y <= 1; y++)
{
for(int x = -1; x <= 1; x++)
{
ivec3 otherGridCoord = gridCoord + ivec3(x, y, z);
if(otherGridCoord.x < 0 || otherGridCoord.x >= gridres.x || otherGridCoord.y
|| otherGridCoord.y >= gridres.y || otherGridCoord.z < 0|| otherGridCoord.z >
continue;

uint otherGridIndex = computeGridIndex(otherGridCoord);

if(!checkindex(0, grid_size, otherGridIndex)) //check if index is still in gr
continue;

uint numOfNeighboursInGroup = counts[otherGridIndex];
uint otherParticleGroupIndex = 0;
if(otherGridIndex > 0)
{
otherParticleGroupIndex = sum_counts[otherGridIndex-1];
}

for(int i = 0; i < numOfNeighboursInGroup; i++)
{
uint particleIndex = sorted_index[otherParticleGroupIndex + i];


if (g_id == particleIndex) //same particle
continue;

Particle other_particle = particles[particleIndex];
```

```
other = other_particle.position_and_radius.xyz;
float otherRadius = other_particle.position_and_radius.w;
ivec4 other_bonds = bonds[particleIndex];

n = other - pos;
length_n = length(n);


bool bonded = false;
bool isHydrogen = radius < 0.4;
if(isHydrogen)
{
bonded = (own_bonds.x == particleIndex);
}
else
{
bonded = (other_bonds.x == g_id);
}

if(bonded)
{
// bond stretching energy
float d0 = 1.0f;

if(otherRadius < 0.4f)
{
if(radius < 0.7f)
{
d0 = 0.9584;
}
else if(radius < 0.75f)
{
d0 = 1.017;
}
else
{
d0 = 1.087;
}
}
else if(otherRadius < 0.7f)
{
d0 = 0.9584;
}
else if(otherRadius < 0.75f)
```

```
{
d0 = 1.017;
}
else
{
d0 = 1.087;
}

float factor = (length_n - d0) * k_bond * 0.5;
f +=  n * factor;
}
else
{
if(isHydrogen)
{
uint origin_ID = own_bonds.x;

if(other_bonds.x == origin_ID)
{

Particle origin_particle = particles[origin_ID];

float originRadius = origin_particle.position_and_radius.w;
vec3 origin_pos = origin_particle.position_and_radius.xyz;

vec3 ownVec = normalize(pos - origin_pos);
vec3 otherVec = normalize(other - origin_pos);

vec3 rotVec = cross(otherVec, ownVec);
vec3 tangente = cross(ownVec, rotVec);

float skalarProdukt = dot(otherVec, ownVec);
float angle = acos(skalarProdukt);
float theta0 = 0.0;

if(originRadius < 0.7f)
{
theta0 = 1.8229964;
}
else if(originRadius < 0.75f)
{
theta0 = 1.8814649;
}
else // molekuel ist CH4
```

```
{
theta0 = 1.9111355;
}

float factor = (angle - theta0) * k_angle * 0.5;
f += tangente * factor;
}
}
else
{
if(length_n > 4.0)
continue;

float ownVdWRadius = 0.0;
if(radius < 0.4f)
{
ownVdWRadius = 1.1;
}
else if(radius < 0.7f)
{
ownVdWRadius = 1.52;
}
else if(radius < 0.75f)
{
ownVdWRadius = 1.55;
}
else
{
ownVdWRadius = 1.7;
}

float otherVdWRadius = 0.0;
if(otherRadius < 0.4f)
{
otherVdWRadius = 1.1;
}
else if(otherRadius < 0.7f)
{
otherVdWRadius = 1.52;
}
else if(otherRadius < 0.75f)
{
otherVdWRadius = 1.55;
}
```

```
        else
        {
        otherVdWRadius = 1.7;
        }
        //van der Waals energy
        float equilibrium = (ownVdWRadius + otherVdWRadius);
        float aij = equilibrium / length_n;
        float power = pow(aij, 6.0);
        float vanDerWaals = power - 2.0;
        vanDerWaals *= power;

        float factor = -1.0 * vanDerWaals * k_vanDerWaals;
        vec3 n_norm = normalize(n);
        f += n * factor;
        }
        }
        }
        }
        }
        }

        forces_old[g_id] = forces_new[g_id];
        forces_new[g_id] = vec4(f, 0.0f);
        }
```

Source Code 5: Integration-Shader

```
#version 430 core

layout(local_size_x = 1, local_size_y = 1, local_size_z=1) in ;


struct Particle
{
        vec4 position_and_radius;
        vec4 velocity_and_mass;
};

layout(std430, binding = 0) buffer particles_ssbo
{
        Particle particles[];
};

layout(std430, binding = 5) buffer force_old_ssbo
```

```
{
        vec4 forces_old[];
};

layout(std430, binding = 6) buffer force_new_ssbo
{
        vec4 forces_new[];
};


uniform vec3 bmax;
uniform vec3 bmin;
uniform ivec3 gridres;

uniform float damping;
uniform float energy;
//uniform float gravity;
uniform float dt;
uniform float time;

uint index3d(uvec3 c)
{
        return c.z * gl_NumWorkGroups.x * gl_NumWorkGroups.y +
        c.y * gl_NumWorkGroups.x +
        c.x;
        //return c.y * w * d + c.z * w + c.x;
}

bool checkindex(uint min, uint max, uint current)
{
        if (current < min || current >= max)
        return false;
        return true;
}


void main()
{
        uvec3 coord = gl_GlobalInvocationID;

        //int g_id = index3d(coord, w, h, d);
        uint g_id = index3d(coord);

        vec3 pos_old = particles[g_id].position_and_radius.xyz;
```

```
vec3 vel_old = particles[g_id].velocity_and_mass.xyz;
float length_vel = length(vel_old);
float mass = particles[g_id].velocity_and_mass.w;

vel_old *= damping;
vec3 force_old = forces_old[g_id].xyz;
vec3 force_new = forces_new[g_id].xyz;

vec3 pos_new = pos_old;
vec3 vel_new = vel_old;

float dt_half = dt * 0.5f;
vec3 accel_old = force_old / mass;
vec3 accel_new = force_new / mass;


//add bounding box force
if(pos_new.x < bmin.x)
{
        pos_new.x = bmin.x;
        vel_new.x = 0.0f;
}
else if (pos_new.x > bmax.x)
{
        pos_new.x = bmax.x;
        vel_new.x =0.0f;
}

if(pos_new.y < bmin.y)
{
        pos_new.y = bmin.y;
        vel_new.y = 0.0f;
}
else if (pos_new.y > bmax.y)
{
        pos_new.y = bmax.y;
        vel_new.y =0.0f;
}

if(pos_new.z < bmin.z)
{
        pos_new.z = bmin.z;
        vel_new.z = 0.0f;
}
```

```
        else if (pos_new.z > bmax.z)
        {
                pos_new.z = bmax.z;
                vel_new.z =0.0f;
        }

        //leap frog
        vec3 accel_2 = (accel_old + accel_new);
        accel_2 *= 0.5f;

        pos_new += vel_old * dt;
        pos_new += accel_old * dt * dt_half;
        vel_new += accel_2 * dt;

        vel_new *= 1.0 + energy;

        particles[g_id].position_and_radius.xyz = pos_new;
        particles[g_id].velocity_and_mass.xyz = vel_new;
}
```

Literatur

[Bon64] A. Bondi. van der waals volumes and radii. *The Journal of Physical Chemistry*, 68(3):441–451, 1964.

[CCB+95] Wendy D. Cornell, Piotr Cieplak, Christopher I. Bayly, Ian R. Gould, Kenneth M. Merz, David M. Ferguson, David C. Spellmeyer, Thomas Fox, James W. Caldwell, and Peter A. Kollman. A second generation force field for the simulation of proteins, nucleic acids, and organic molecules. *Journal of the American Chemical Society*, 117(19):5179–5197, 1995.

[CCL+01] Thomas H.. Cormen, Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction To Algorithms*. MIT Press, Cambridge, 2nd edition, 2001.

[CGPP+08] Beatriz Cordero, Verónica Gómez, Ana E. Platero-Prats, Marc Revés, Jorge Echeverría, Eduard Cremades, Flavia Barragán, and Santiago Alvarez. Covalent radii revisited. *Dalton Trans.*, pages 2832–2838, 2008.

[CL08] Soto C and Estrada LD. Protein misfolding and neurodegeneration. *Archives of Neurology*, 65(2):184–189, 2008.

[Har07] Mark Harris. Parallel prefix sum (scan) with cuda. `https://www.mimuw.edu.pl/~ps209291/kgkp/slides/scan.pdf`, April 2007. [Online; accessed 27-September-2018].

[HKvdSL08] Berk Hess, Carsten Kutzner, David van der Spoel, and Erik Lindahl. Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of Chemical Theory and Computation*, 4(3):435–447, 2008. PMID: 26620784.

[HS86] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.

[LJ24] J.E. Lennard-Jones. On the determination of molecular fields. —ii. from the equation of state of a gas. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 106(738):463–477, 1924.

[SLT09] Chun Meng Song, Shen Jean Lim, and Joo Chuan Tong. Recent advances in computer-aided drug design. *Briefings in Bioinformatics*, 10(5):579–591, 2009.

[vG18] Prof. Dr. W. F. van Gunsteren. About the GROMOS software for biomolecular simulation. `http://www.gromos.net//`, 2018. [Online; accessed 27-September-2018].