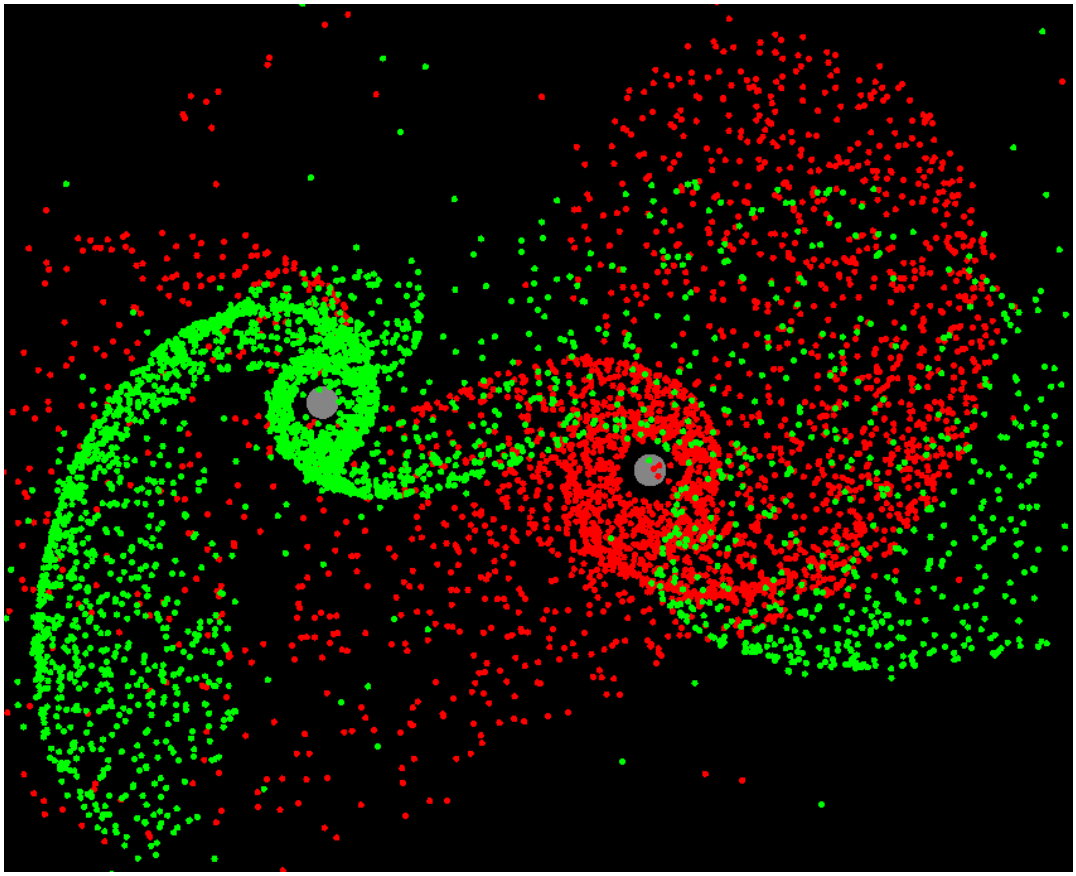
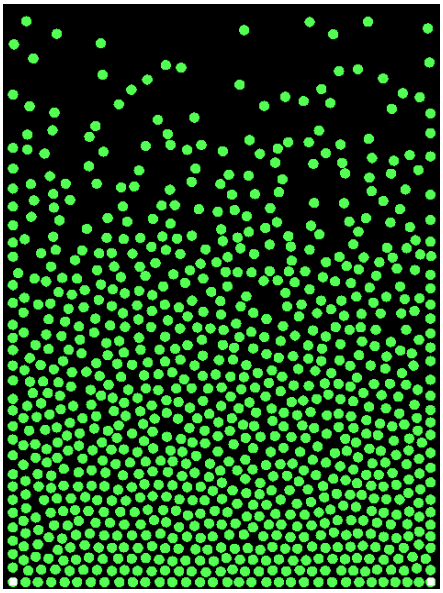
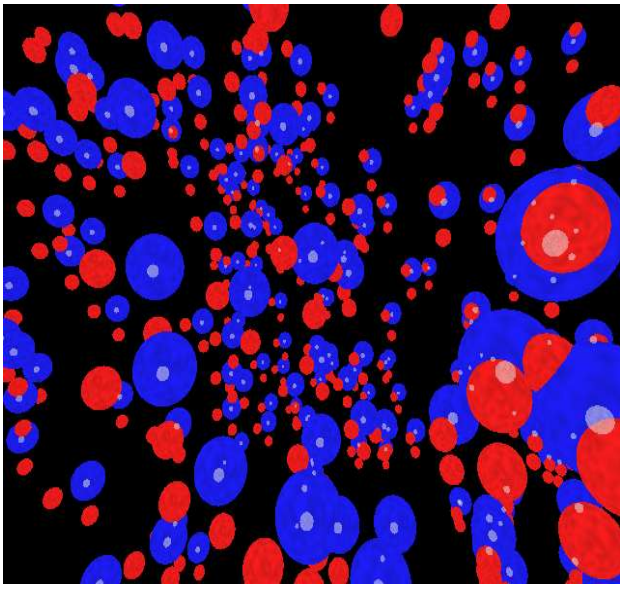
Vielteilchensimulationen mit der Grafikkarte







Jugend forscht Arbeit 2014

Adrian Lenkeit

St. Michael-Gymnasium Bad Münstereifel

Gliederung

1. Zusammenfassung

2. Vorteile von Simulationen

3. Wahl der Hardware

4. Galaxienkollisionen

5. Festkörper

6. Optimierungsverfahren

7. Wassermoleküle

8. Nanofluidik

9. Anhang

10. Linkliste

1. Zusammenfassung

Ziel meiner Jugend-forscht-Arbeit ist es Programme anzufertigen, welche mir Simulationen von Vielteilchensystemen erlauben. Der Vorteil solcher Programme besteht darin, dass Simulationen in der Grundlagenforschung und in der Wirtschaft viel Zeit und Geld sparen können. Ich entschied mich dafür die Grafikkarte als Recheneinheit einzusetzen und schrieb meine Programme in C++. Mein erstes Ziel war es Galaxienkollisionen zu simulieren. Nachdem alle anfängliche Probleme behoben waren, war es mir möglich Galaxienkollisionen im zwei- und dreidimensionalen Raum nachzustellen. Meine Ergebnisse decken sich gut mit den Beobachtungen der Astrophysiker. Weiterhin schrieb ich unter der Verwendung des Lennard-Jones-Potenzials Simulationen für Festkörper, Flüssigkeiten und Gase, an denen ich verschiedene numerische Integratoren austestete. Diese und die Galaxien-Simulationen konnte ich im Rahmen meiner Arbeit zunehmend optimieren. Bei der Simulation von Wassermolekülen stieß ich an die Grenzen meines Simulationsverfahrens mit dem Lennard-Jones-Potential, da dieses keine quantenphysikalischen Aspekte berücksichtigt. Interessante Ergebnisse erzielte ich jedoch im Bereich der Nanofluidik. Hier zeigte sich ein deutlicher Unterschied zwischen der makroskopischen Welt und der Nanowelt. Auch entwickelte ich Bauteile zur Flüssigkeitssteuerung in der Nanowelt, die für die Realisierung von neuen Geräten herangezogen werden können.

2. Vorteile von Simulationen

Die Entwicklung von sicheren und funktionsfähigen Produkten kostet viel Zeit und Geld. Dank der Rechenleistung moderner Computer ist es aber möglich, die Funktionsfähigkeit und Sicherheit von Produkten im Vorfeld zu simulieren. Auch in der Grundlagenforschung gewinnt die Simulation immer mehr an Bedeutung, denn so lassen sich aufwendige Experimente im Vorfeld simulieren, wodurch Zeit und Geld gespart wird. Mit Simulationen ist es aber auch möglich neue Erkenntnisse über physikalische Gesetze zu gewinnen, die dann experimentell überprüft werden können.

3. Wahl der Hardware

Bei der Simulation von Vielteilchensystemen ist zu beachten, dass jedes Teilchen mit jedem anderen Teilchen wechselwirkt. Also gibt es insgesamt $(N(N-1))/2$ Wechselwirkungen. Dies wirkt sich stark auf die benötigte Rechenzeit aus, denn schon 500 Teilchen benötigen ca. 125.000 Wechselwirkungen. Für die meisten Simulationen werden sogar noch viel mehr Teilchen benötigt, wodurch die Zahl der Wechselwirkungen ziemlich schnell in die Höhe schießen kann. Aber einen Vorteil haben Vielteilchensysteme. Jede einzelne Wechselwirkung kann unabhängig von allen anderen berechnet werden. Im besten Fall könnte man also alle Wechselwirkungen gleichzeitig bzw. parallel berechnen. Benutzt man zum Rechnen aber dem im Computer verbauten Hauptprozessor stößt man an ein schwerwiegendes Problem. Der Prozessor kann immer nur schrittweise Berechnungen ausführen, d.h, nur eine Berechnung nach der anderen. Er arbeitet nur seriell. Das große Potential, die Vielteilchensysteme parallel zu rechnen, wird gar nicht ausgenutzt. Der Prozessor eignet sich also sehr schlecht für die Simulation von Vielteilchensystemen. Eine Möglichkeit um dieses Problem zu lösen ist, mehrere Prozessoren zusammenzuschalten, wodurch auch mehrere Rechnungen gleichzeitig ausgeführt werden können. Solche Prozessoren werden auch verkauft, sogenannte Mehrkernprozessoren. Ihre Anzahl an Rechenkernen reicht aber meistens nur bis zu acht Stück, die Preise steigen aber ziemlich schnell an. Der daraus gewonnene Schub an Performance lohnt sich meistens nicht. Eine andere Möglichkeit ist es, mehrere Computer zusammenzuschalten, also der Bau eines großen Computer-Clusters. Mit ihnen kann eine unglaublich hohe Rechenleistung erzielt werden. Der Bau eines solchen Clusters wäre wegen dem hohem Aufwand aber ein eigenes Projekt für sich. Eine Alternative die immer mehr an Bedeutung gewinnt ist die Grafikkarte. Eigentlich ist sie dafür da, dreidimensionale Szenen in ein Bild umzuwandeln. Diese Szenen bestehen meist aus mehreren tausend bis millionen Vertexen, die zusammen Dreiecke bilden. Diese Dreiecke bilden dann Objekte, welche mit Texturen bedeckt werden können und zur Licht- und Schattenberechnung dienen. Sie können mit Spezialeffekten versehen werde, sodass sie wie ein Spiegel wirken oder wie Glas Objekte hinter sich verzerren. Dies erfordert aber eine sehr hohe Rechenleistung. Wie kann die Grafikkarte diese komplexen Szenen errechnen? Wie bei Vielteilchensystemen kann auch die Berechnung von komplexen Szenen parallelisiert werden. Der Vorteil der Grafikkarte ist, das sie aus mehreren hundert bis tausend Prozessoren, den sogenannten Shader-Einheiten, besteht. Einer Shader-Einheit kann der Programmierer selbst sagen, was sie machen soll. Die Grafikkarte würde sich also als Recheneinheit für die Vielteilchensysteme lohnen, da sie sehr kostengünstig, aber auch sehr leistungsstark ist. Aber eine Frage blieb noch offen, wie soll ich die Grafikkarte dazu bringen, anstatt Bilder auszurechnen meine Formel zu berechnen? Die Firma Nvidia, einer der zwei führenden Hersteller von Grafikkarten, bietet ein System namens CUDA an, mit dem sich Grafikkarten zur Berechnung von eigenen Formeln umprogrammieren lassen. Damit stand für mich fest, dass ich die Grafikkarte als Recheneinheit für meine Simulationen benutze.

4. Galaxienkollisionen

Um überhaupt eine Vielteilchensimulation zu programmieren musste ich ein Rahmenprogramm schreiben, welches die Ansteuerung der Grafikkarte ermöglicht und die Ergebnisse der Simulation darstellt. Dies war sehr aufwendig und deswegen würde eine Erklärung des Programms in dieser schriftlichen Arbeit sehr viel Platz verbrauchen. Deshalb werde ich nur die wichtigsten Teile des Programms zeigen, nämlich die Teile des Programms, die auf der Grafikkarte zur Berechnung ausgeführt werden. Auf Nachfrage beim Wettbewerb kann ich natürlich gerne das ganze Programm zeigen. Bevor ich die Simulation aber schreiben konnte, war es zuerst notwendig mich in die

mathematischen Grundkenntnisse einzulesen, die für die Simulation von Gravitation benötigt werden. Notwendig ist die Kraft, die auf einen Körper durch die Anziehungskraft von einem anderen Körper ausgeübt wird. Die Formel, die diese Kraft beschreibt, ist

$$\vec{F}_1 = G \frac{m_1 m_2}{|\vec{r}_{12}|^2} \hat{r}_{12} \quad ,$$

wobei G die Gravitationskonstante ist, m_1 und m_2 die Massen der Körper, $\hat{r}_{12}$ der zum anderen Körper zeigende Einheitsvektor und $|\vec{r}_{12}|$ die Distanz zwischen den Massenmittelpunkten der Körper. $\vec{F}$ ist die Kraft, die auf den Körper ausgeübt wird, zu dem die Masse m_1 gehört. Diese auf den Körper wirkende Kraft bewirkt eine Änderung in seiner Beschleunigung $\vec{a}$, sodass $\vec{a}_{neu} = \vec{a}_{alt} + \vec{F}/m$. Man kann die Gleichung für die Kraft der Gravitation noch umstellen, sodass sie direkt die Beschleunigung zurückgibt, was einiges an Rechenschritten spart und sich so positiv auf die Performance des Programms auswirkt. Die neue Formel würde dann wie folgend lauten.

$$\vec{a}_1 = G \frac{m_2}{|\vec{r}_{12}|^2} \hat{r}_{12}$$

Wenn man diese Formel dann für jedes Paar an Körpern, also für jede mögliche Wechselwirkung, ausrechnet und addiert, dann erhält man auch für jeden Körper seine Gesamtbeschleunigung. Mithilfe der Beschleunigung kann dann auch die neue Position eines Körpers berechnet werden. Dabei gibt es aber ein Problem. Wenn man nur zwei Körper hat, dann kann man für jeden Zeitpunkt genau sagen, wo sie sich befinden. Sobald man aber 3 oder mehr wechselwirkende Körper hat, ist dies nicht mehr möglich. Wie wird dann die neue Position berechnet? Nötig dafür ist ein numerischer Integrator. Eigentlich benutzt man numerische Integratoren zum Lösen von Differentialgleichungen. Da Differentialgleichungen aber auch zum Modellieren von Naturgesetzen benutzt werden [1], kann man mit einem Integrator die Naturgesetze annähernd simulieren. Zu beachten ist aber, das es viele verschiedene numerische Integratoren gibt, die alle eine unterschiedliche Genauigkeit bzw. Ordnung haben. Je höher die Ordnung, desto genauer sind auch die Ergebnisse, dafür steigt aber die benötigte Rechenzeit drastisch an. Die Ordung selber gibt an, wie sich die Genauigkeit verändert, wenn man den Zeitschritt halbiert. Nutzt man einen Integrator von zweiter Ordnung, so verdoppelt sich die Genauigkeit bei einer Halbierung des Zeitschritt. Nimmt man hingehen einen Integrator mit vierter Ordnung, so versechzehntfacht sich die Genauigkeit bei einer Halbierung des Zeitschrittes. Man muss also die richtige Balance zwischen der benötigten Genauigkeit und der vorhandener Rechenkapazität finden. Bei der Simulation von Galaxienkollisionen ist dies aber relativ einfach. Da die Kollisionen zwischen Galaxien über einen relativ langen Zeitraum stattfinden und die Gravitation die schwächste aller Grundkräfte ist, kann man einen Integrator mit einer sehr niedrigen Genauigkeit wählen, man erhält aber trotzdem genaue Ergebnisse. Ich entschied mich für den Leap-Frog-Integrator. Obwohl er nur von zweiter Ordnung ist, hat er einige Vorteile. So kann man den Zeitschritt während der Simulation verändern und die Geschwindigkeit der Körper kann ausgelesen werden, um so z.B. die kinetische Energie zu bestimmen. Um bei dem Leap-Frog-Integrator die neue Position zu bestimmen, muss zuerst die neue Geschwindigkeit $\vec{v}_1$ bestimmt werden mit $\vec{v}_1 = \vec{v}_0 + \vec{a}\, dt$, wobei dt die Zeit bzw. der Zeitschritt ist, der während einer Iteration vergeht. Anhand der neuen Geschwindigkeit kann dann die neue Position $\vec{x}_1$ ermittelt werden mit $\vec{x}_1 = \vec{x}_0 + \vec{v}_1\, dt$. Dies sollte an Grundkenntnissen genügen, um eine funktionierende Simulation der Gravitation zu erstellen. Nötig war jetzt nur noch die Formeln in Code umzuschreiben, der auf der Grafikkarte ausgeführt wird.

```
//Gravitation

int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;

if(i<BODY_COUNT&&j<BODY_COUNT)
{
    Vec2 normal=bodies[j].pos-bodies[i].pos;
    float distSqr=normal.lengthSquared();

    bodies[i].acc+=normal*(G*(bodies[j].mass/distSqr));
}

//Leap-Frog

int i=blockIdx.x*blockDim.x+threadIdx.x;

if(i<BODY_COUNT)
{
    bodies[i].vel+=bodies[i].acc*dt;
    bodies[i].pos+=bodies[i].vel*dt;
    bodies[i].acc=Vec2(0,0);
}
```

Bei der Berechnung der Gravitationskraft geben „i“ und „j“ das wechselwirkende Körperpaar an. „bodies“ ist ein Array, das alle Körper enthält. Da es vorkommen kann, das einer der beiden Werte größer ist als die Anzahl der Körper, hier "BODY_COUNT", prüft eine if-Abfrage ob „i“ und „j“ kleiner als "BODY_COUNT" sind. Zusätzlich muss geprüft werden, ob "i" ungleich "j" ist, denn ohne diese Abfrage würde ein Körper Kraft auf sich selbst ausüben. Wenn dies alles der Fall ist, wird der Richtungsvektor „normal“ errechnet, welcher vorerst aber unnormalisiert bleibt. Dies ist ein Trick um Rechenzeit zu sparen, denn die Länge des Vektors entspricht der Distanz zwischen beiden Körpern, welche dann in "distSqr" gespeichert wird. Danach wird die Beschleunigung, die durch die Gravitation entsteht, berechnet und auf die Beschleunigung des Körpers „i“ addiert. Ähnlich funktioniert auch der numerische Integrator.
Um meinen Code zu testen, war es jetzt nur noch nötig eine schon bekannte Galxienkollision zu simulieren, anhand derer ich meine Ergebnisse vergleichen kann. Die Hubblesequenz [2], ein Diagramm, dass die "Evolution" der Galaxien zeigt, besagt, dass sich zwei Scheibengalaxien bei einer Kollision in zwei Spiralgalaxien umformen sollten. Um dies zu testen generierte ich zwei Scheibengalaxien (Abb. 4.1 oben) die sich aufeinander zubewegten. Jede Galaxie bestand aus 3200 Sternen. Also gab es insgesamt 6400 Körper die miteinander wechselwirken, wodurch schon ca. 20 Millionen Wechselwirkungen entstehen! Wenn mein Programm richtig funktioniert, dann sollten sich zumindest während der Kollision Spiralarme ausbilden. Doch dies war leider nicht der Fall. Bereits bevor die Kollision stattfand, flogen die Galaxien auseinander (Abb. 4.1 unten), so als gäbe es kein schwarzes Loch in der Mitte der Galaxie welches die Sterne im Orbit hält.
Meine erste Vermutung war, das der Code auf der Grafikkarte nicht richtig aufgerufen wird, bzw., dass keine Kräfte berechnet werden. Nachdem ich aber die Geschwindigkeit einzelner Körper genauer betrachtete, fiel mir auf, dass sich die Bewegungsgeschwindigkeiten der Körper änderten. Dies ist aber nur möglich wenn Kräfte auf die Körper wirken. Nur die Änderung der Geschwindigkeit war sehr klein, so als wäre die Gravitation sehr schwach. Warum dies der Fall war

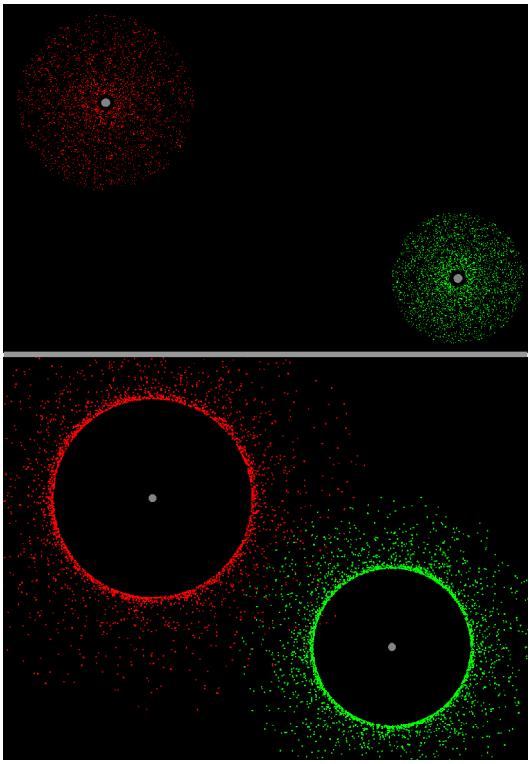


Abb. 4.1: Oben Galaxien am Start, unten während der (fehlerhaften) Simulation

wusste ich erst, nachdem ich mir CUDAs Aufrufmodell genauer anschaute. Wenn man Code auf der Grafikkarte ausführt, ist dieser in verschiedene Kernel eingeteilt. Im meinem Fall gibt es einen Kernel zur Berechnung der Gravitationskraft und einen anderen Kernel um den numerischen Integrator anzuwenden. Ruft man einen Kernel auf, werden auch Prozesse aufgeteilt in Blöcken aufgerufen. Jeder dieser Prozesse führt dann den selben Code aus, abgesehen von der Speicheradresse mit der er arbeiten soll, also seiner zu bearbeitenden Wechselwirkung. Da die Prozesse alle parallel arbeiten, kann es vorkommen, dass mehrere Prozesse gleichzeitig versuchen, die Beschleunigung eines Körpers zu berechnen. Alle Prozesse lesen dann den alten Wert der Beschleunigung ein und addieren auf diesen Wert ihre errechnete Beschleunigung. Danach versuchen alle Prozesse, ihren neuen Wert abzuspeichern. Da das Abspeichern aber gleichzeitig passiert, überschreiben die Prozesse ihren Wert gegenseitig. Welcher Wert dann wirklich gesetzt wird, ist purer Zufall. Dadurch fehlen die Kräfte vieler Wechselwirkungen, da sie von anderen Prozessen überschrieben werden, sodass der Endwert nur sehr klein bleibt. Um dieses Problem zu lösen war es notwendig meine Berechnungsmethoden umzustellen. Anstatt das jeder Prozess eine Wechselwirkung berechnet, berechnet jeder Prozess alle Wechselwirkungen für einen Körper. Die Anzahl der Prozesse entspricht dann ungefähr der Anzahl der Körper. Dies verhindert das mehrere Prozesse den selben Körper gleichzeitig bearbeiten, dies wirkt sich aber auch stark auf die Performance aus. Die große Parallelisierbarkeit von Vielteilchensystemen wird dann aber nicht vollständig ausgenutzt, weshalb die Performance etwas sinkt. Der Code zur Berechnung der Gravitationskraft musste jedoch nur leicht abgeändert werden.

```
//Gravitation - Neu

int i=blockIdx.x*blockDim.x+threadIdx.x;

if(i<BODY_COUNT)
{
    for(int j=0;j<BODY_COUNT;j++)
    {
        if(i==j)
            continue;

        Vec2 normal=bodies[j].pos-bodies[i].pos;
        float distSqr=normal.lengthSquared();

        bodies[i].acc+=normal*(G*(bodies[j].mass/distSqr));
    }
}
```

Der neue Code ist ähnlich dem alten. Es wird aber nicht das zu bearbeitende Körperpaar angegeben sondern nur der zu bearbeitende Körper. Danach wird mithilfe einer for-Schleife über alle anderen Körper iteriert und die Kräfte werden wie bisher berechnet. Die Formel, auf die der neue Code basiert, ist

$$\vec{a}_i = \sum_{j=0}^{n} G \frac{m_j}{|\vec{r}_{ij}|^2} \hat{r}_{12} \quad \text{, wenn} \quad i \neq j \quad .$$

Um den neuen Code zu testen, wollte ich wieder die zwei Scheibengalaxien miteinander kollidieren lassen. Kurz nach dem Start der Simulation war, im Gegensatz zum vorherigen Versuch, sichtbar, dass die Galaxien erhalten blieben. Die Wechselwirkungen wurden also anscheinend richtig berechnet. Während der Kollisionsphase der Galaxien (Abb. 4.2 oben) ist klar erkennbar, dass sich die Galaxien gegenseitig anziehen, was zu einer Deformation der Galaxien führt. Diese Deformation wird dann als Spiralarm sichtbar, den die Galaxien mit sich ziehen (Abb. 4.2 unten). Meine Ergebnisse wollte ich mit der echten Welt vergleichen, um die Korrektheit meiner Simulation zu überprüfen. Dazu suchte ich im Internet nach Aufnahmen von Galaxienkollisionen. In Abb. 4.3 ist eine Galaxienkollision sichtbar, die sich mit meiner simulierten Kollision gut deckt. Dies bestätigte mich darin, das meine Simulationen richtig waren.

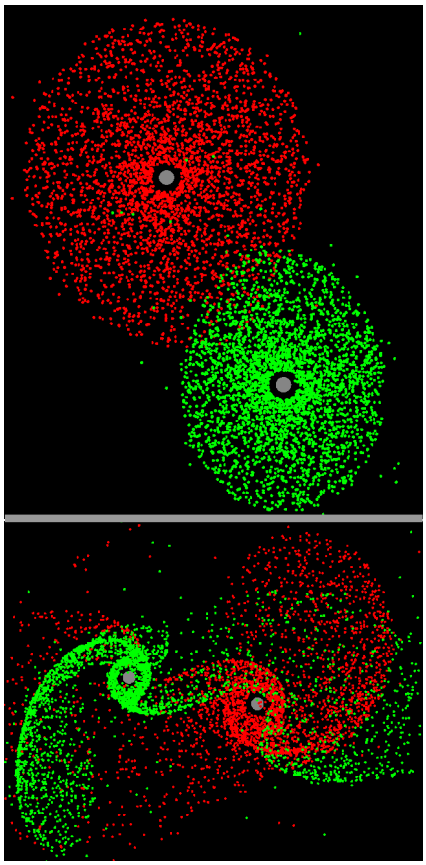


Abb. 4.2: Oben Deformation, unten Bildung eines Spiralarms

Die Galaxienkollision lieferte die zu erwartenden Resultate, funktionierte also schon sehr gut. Auch die Performance war mit ca. 30 ms Rechenzeit pro Iteration gut. Der einzige Nachteil war, das die Simulation momentan noch in zwei Dimensionen dargestellt wurde, was die Simulation von einigen Galaxienkollision verhindert. Viele Kollisionen, die interessante Formationen ergeben, resultieren aus einer Kollision von zwei Galaxien die gewinkelt zu einander stehen. Mit zwei Dimensionen ist dieser Winkel aber nicht darstellbar.

Als nächstes Ziel setzte ich es mir deswegen, die Simulation für drei Dimensionen umzuschreiben. Weil der Großteil des Rahmenprogramms unabhängig von der Anzahl der simulierten Dimensionen war, konnte ich das Rahmenprogramm auch relativ schnell umschreiben. Nötig dazu war nur, die Vektorrechnung an drei Dimensionen anzupassen sowie die Darstellung auf drei Dimensionen umzuändern. Der Code auf der Grafikkarte blieb fast unberührt. Nur ein Vec2, die Klasse für einen zweidimensionalen Vektor, wird zu einem Vec3.

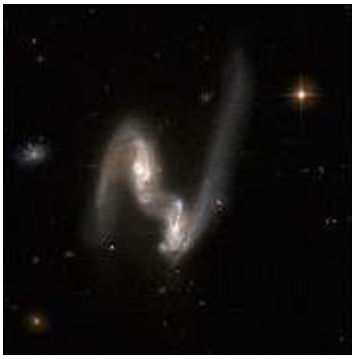


Abb. 4.3: Echte Galaxienkollision [8]

Auch in dieser 3D-Simulation wollte ich zwei Scheibengalaxien miteinander kollidieren lassen, welche aber im rechten Winkel zueinander stehen. Dabei können sich die Galaxien anders verformen. Beide Galaxien sollten aber wieder zu Spiralgalaxien werden. Um dies zu testen generierte ich die beiden Scheibengalaxien (Abb. 4.4), angewinkelt im rechten Winkel, die sich aufeinander zubewegten. Insgesamt gab es wieder 6400 Körper. Obwohl aber eine Dimension mehr zu der Simulation hinzugekommen ist, stieg die benötigte

Rechenzeit nur um 10 ms an, also von 30 ms auf 40 ms. Während der Kollisionsphase waren erst keine großen Änderungen in den Galaxien sichtbar, aber schon kurz nach der Kollision bildeten beide Galaxien Spiralarme aus. Nach längerer Zeit verdichteten sich die Spiralarme der beider Galaxien (Abb. 4.5, Abb. 4.6)
Als letztes wollte ich noch wissen, wie schnell eine Implementierung von meiner Simulation auf dem Hauptprozessor laufen würde. Auf der Grafikkarte brauchte eine Iteration der Galaxienkollision ca. 40 ms, auf dem Prozessor hingegen 1500 ms bis 2000 ms. Die Grafikkarte als Recheneinheit bringt also einen sehr hohen Performanceboost.

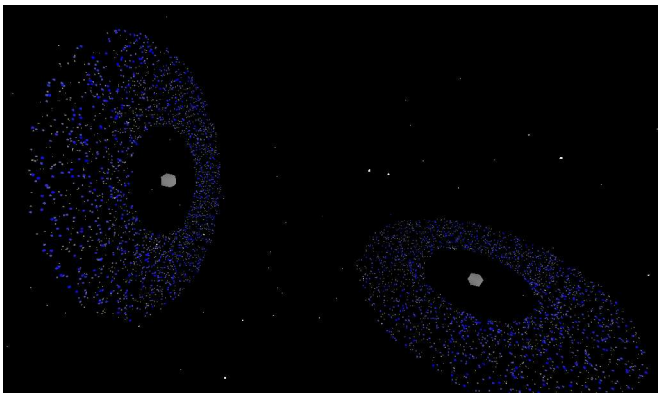


Abb. 4.4: Galaxien kurz vor der Kollision, links vertikale Galaxie, rechts horizontale

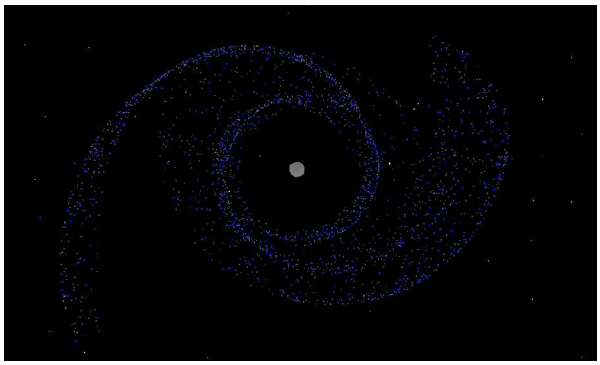


Abb. 4.5: Horizontale Galaxie nach der Kollision

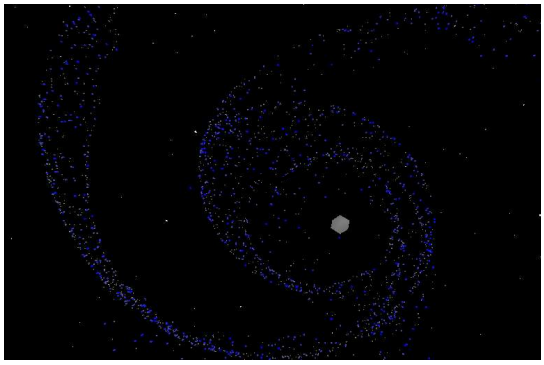


Abb. 4.6: Vertikale Galaxie nach der Kollision

5. Festkörper

Nachdem die Galaxienkollision schon sehr gut funktionierte, wollte ich mich der Simulation von Festkörpern, Flüssigkeiten und Gasen zuwenden. Dazu benötigte ich die Kraft, die zwischen zwei Atomen wirkt. Aufgrund der Ladungsverteilung innerhalb zwei neutral geladenen Atomen wirkt bei "größeren" Abständen eine schwache anziehende Kraft. Auf kurze Distanz hingegen wirkt eine sehr starke abstoßende Kraft. Beschrieben werden diese beiden Kräfte mit dem Lennard-Jones Potential [4].

$$V = 4\epsilon\{(\sigma/r)^{12} - (\sigma/r)^{6}\}$$

ϵ gibt die Tiefe der Potentialmulde an, bzw. die Stärke des Potentials, σ gibt den Abstand an,

bei dem das Potential null ergibt und r ist der Abstand des Atompaars für welches das Potential berechnet wird. Das Potential musste ich jetzt aber noch ableitet, sodass die Gleichung die dazugehörige Kraft pro Masse und damit die Beschleunigung zurückgibt.

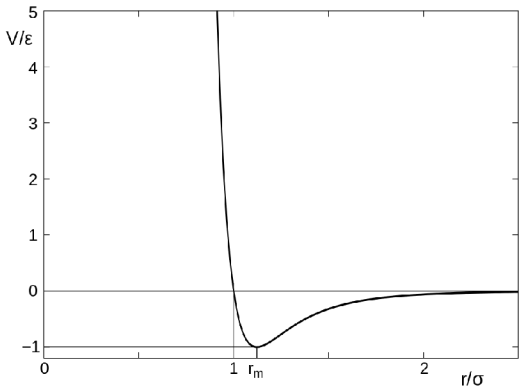


Abb. 5.1: Kurve des Lennard-Jones-Potentials [9]

$$a=-24\epsilon\{2(\frac{\sigma^{12}}{r^{13}})-(\frac{\sigma^{6}}{r^{7}})\}$$

bzw. in Vektorform

$$\vec{a}_1=-24\epsilon\{2(\frac{\sigma^{12}}{|r_{12}|^{13}})-(\frac{\sigma^{6}}{|r_{12}|^{7}})\}\hat{r}_{12}$$

Die Vektorform des Lennard-Jones-Potentials musste ich jetzt noch in Code umschreiben

```
//Lennard-Jones

int i=blockIdx.x*blockDim.x+threadIdx.x;

if(i<BODY_COUNT)
{
    for(int j=0;j<BODY_COUNT;j++)
    {
        if(i==j)
            continue;

        Vec2 norm=bodies[j].pos-bodies[i].pos;
        float distance=norm.length();
        float d2=distance*distance;
        float d6=d2*d2*d2;
        float d7=d6*distance;
        float d12=d6*d6;
        float d13=d12*distance;
        float leftTerm=2*(LENNARD_JONES_ZERO_12/d13);
        float rightTerm=LENNARD_JONES_ZERO_6/d7;
        Vec2 force=norm*(-24*LENNARD_JONES_DEPTH*(leftTerm-rightTerm));
        bodies[i].acc+=force;
    }
}
```

„LENNARD_JONES_ZERO_12“ entspricht σ^{12} und „LENNARD_JONES_ZERO_6“ entspricht σ^{6}. Anstatt r^{12} und r^{6} durch eine Reihe von „distance*distance*...“ zu errechnen, lohnt es sich mehrere Zwischenschritte einzulegen. So wird erst r^{2} errechnet und daraus dann r^{6}, was weniger Rechenoperationen verbraucht. Den Simulationsbereich grenzte ich ein, damit die Atome sich nicht zu weit von einander entfernen können. Die Gravitationskraft, die eigentlich noch zwischen den Atomen wirkt, musste ich bei der Simulation von Festkörpern nicht mehr berechnen. Da die Gravitation viel schwächer als das Lennard-Jones-Potential ist, hat sie fast keine Auswirkungen auf die Simulation und würde nur mehr Rechenzeit verbrauchen.

Als ersten Test erstellte ich am oberen Rand des Simulationsbereich 1000 Atome. Zu erwarten war, dass die Atome auf den Boden des Simulationsbereich treffen und sich dort ausbreiten. Direkt nach dem Start der Simulation geriet diese jedoch schon außer Kontrolle. Fast alle Atome wurden sehr schnell durch den Simulationsbereich geschleudert. Nach genauerer Beobachtung fiel mir auf, dass dies nach der Kollision zweier Atome passierte. Auf eines der Atome bei einer Kollision wirkt eine sehr starke Kraft, die das Teilchen förmlich wegschleudert. Trifft dieses Atome dann wieder ein anderes Atome geschieht derselbe Vorgang erneut. Dies führt nach einiger Zeit dazu, dass alle Atome nur noch wild durch den Raum fliegen. Der Grund für dieses Verhalten liegt an fehlender Genauigkeit der Simulation. Wenn der Zeitschritt zu groß ist, kann es vorkommen das zwei Atome während einer Kollision fast aufeinander liegen. Wegen dem Lennard-Jones-Potential werden dann beide Atome mit enormer Kraft wieder auseinander geschoben. Eine mögliche Lösung wäre, den Zeitschritt zu verkleinern, wodurch sich die Genauigkeit erhöht. Wenn man aber die kinetische Energie aller Atome in einer Kurve aufzeichnet (Abb. 5.2), wird sichtbar, dass die kinetische Energie aller Atome mit der Zeit zunimmt. Der Zeitschritt entscheidet nur, wie schnell der Anstieg der kinetischen Energie verläuft. Um dies zu verhindern kann man einen anderen numerischen Integrator verwenden, da manche Integratoren einen niedrigeren Anstieg der kinetischen Energie aufweisen. Zuerst wollte ich einen neuen Integrator in die Simulation einbauen. Nach intensiver Nachforschung im Internet stellte sich heraus, dass sich der Verlet-Integrator besser zum simulieren von Atomen eignet als der Leap-Frog-Integrator. Der Vorteil vom Verlet-Integrator ist, das er eine bessere Energieerhaltung hat, wodurch die Simulationen genauer werden, setzt dafür aber einen konstanten Zeitschritt voraus. Wie der Leap-Frog-Integrator ist auch der Verlet-Integrator von zweiter Ordnung. Zusätzlich dazu, das der Zeitschritt konstant sein muss, fehlt die Geschwindigkeit als Eigenschaft der Atome. Stattdessen wird die alte Position des Atoms zur Berechnung benutzt. Die neue Position lässt sich dann berechnen mit $\vec{x}_{neu}=2\vec{x}_{jetzt}-\vec{x}_{alt}+\vec{a}\,dt^2$. Um die Atome zusätzlich noch zu verlangsamen bzw. zu kühlen muss $\vec{x}_{alt}$ näher an $\vec{x}_{jetzt}$ geschoben werden. Dies ist möglich mit der Formel $\vec{x}_{jetzt}=(\vec{x}_{jetzt}-\vec{x}_{alt})(-c)+\vec{x}_{jetzt}$, wobei c der Kühlfaktor ist. Das Kühlen ist notwendig, um in meiner Simulation dem System Energie in Form von Bewegung der Atome zu entziehen, damit es den Festkörperzustand einnehmen kann. Auch Heizen ist möglich. Dabei füge ich dem System Energie in Form von Bewegung der Atome hinzu und kann so einen Festkörper flüssig machen. Dies musste ich dann noch in Code für die Grafikkarte umschreiben.

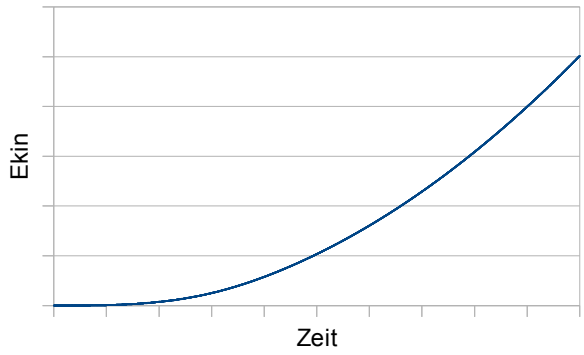


Abb. 5.2: Verlauf der kinetischen Energie der Atome mit dem Leap-Frog-Integrator

```
int i=blockIdx.x*blockDim.x+threadIdx.x;

if(i<BODY_COUNT)
{
    //Kühlen
    bodies[i].oldPos=(bodies[i].pos-bodies[i].oldPos)*-c+bodies[i].pos;
    //Verlet
    Vec2 oldPos=bodies[i].pos;
    bodies[i].pos=bodies[i].pos*2-bodies[i].oldPos+bodies[i].acc*(dt*dt);
    bodies[i].oldPos=oldPos;
    bodies[i].acc=Vec2(0,0);
}
```

Um die verbesserte Simulation zu testen, setze ich wieder 1000 Atome an den oberen Simulationsbereich. Vorerst stellte ich aber noch ein, dass ein Atom nach einer Kollision mit den Wänden 5% seiner Bewegungsgeschwindigkeit verliert, wodurch sich die Simulation mit der Zeit abkühlen sollte und die Atome sich am unteren Rand absetzen sollten. Nach dem Start der Simulation verhielten sich die Atome normal, ohne das die Simulation „explodiert“. Mit der Zeit setzten sich auch immer mehr Atome am Boden ab, bis das ganze System abgekühlt war (Abb. 5.3). Daraus entsteht dann ein solider Festkörper. Jedes Atom ist in dem Festkörper von 6 anderen Atomen umgeben. In einem Festkörper ist dies der Zustand mit der kleinstmöglichen Energie. Würde der Festkörper flüssig oder gasförmig werden, so hätte jedes Atom weniger Nachbarn. Um dies zu testen, baute ich in die Simulation die Möglichkeit ein, die Atome während der Simulation zu erhitzen oder zu kühlen. Zuerst wartete ich bis sich erneut ein Festkörper bildete. Danach erhitzte ich den Festkörper. Beobachten ließ sich, das der Festkörper an der oberen Grenzfläche verdampfte, darunter wurde der Festkörper flüssig (Abb. 5.4). Die Anzahl der Nachbarn der Atome nimmt dabei von unten nach oben ab, hängt also vom Aggregatzustand ab.

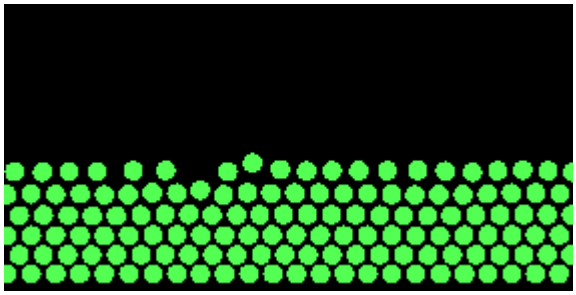


Abb. 5.3: Die Simulation wird gekühlt, wodurch sich die Atome am Boden absetzen

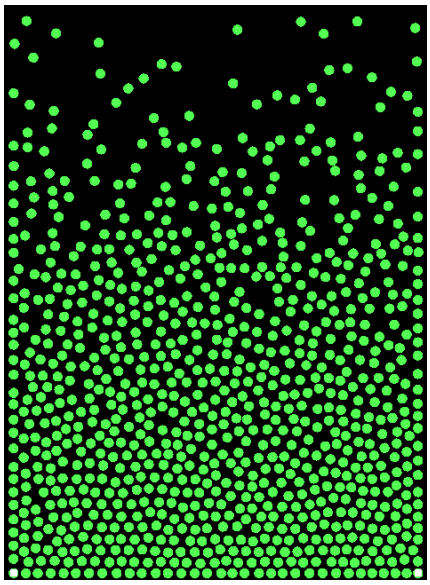


Abb. 5.4: Oben verdampft der Festkörper, unten bleibt er noch fest

6. Optimierungsverfahren

Nachdem die Galaxienkollision und die Simulation von Festkörpern schon gut funktionierte, wollte ich einige Optimierungsverfahren anwenden um die benötigte Rechenzeit zu verringern. Ein Verfahren um die Rechenzeit des Lennard-Jones-Potential zu verkürzen ist, die Berechnungen nur bis zu einer bestimmten Distanz der Atome auszuführen. Ist die Distanz zwischen zwei Atomen größer als 3σ, werden die Kräfte zwischen ihnen nicht mehr berechnet. Möglich ist dies, da das Lennard-Jones-Potential sehr schnell an Stärke verliert. Bei 3σ ist das Potential schon so schwach, das es fast keine Auswirkungen mehr hat [5]. Durch dieses Optimierungsverfahren kann sehr viel Rechenzeit gespart werden ohne die Simulationsergebnisse stark zu verfälschen. Eine Iteration einer Simulation von 2500 Atomen brauchte ohne die Optimierung 8 ms. Schaltete man das Optimierungsverfahren jedoch ein, so sank die benötigte Rechenzeit einer Iteration auf 2 ms. Erst als ich 10.000 Atome simulierte, stieg die benötigte Rechenzeit wieder auf 8 ms an. Durch eine Optimierung des Rahmenprogramms konnte ich die Rechenzeit sogar noch weiter senken. Damit die Grafikkarte Daten zum Rechnen hat, müssen diese erst auf die Grafikkarte geladen werden. Nach den Berechnungen werden die Daten dann wieder auf den Computer geladen. Die Funktion um Daten zu verschieben benötigt aber sehr viel Zeit. Um die Atome darzustellen brauche ich aber die Positionsdaten der Atome. Die Positionsdaten werden dann auf dem Prozessor verarbeitet und dargestellt. Während dem Darstellen werden die Daten jedoch schon wieder auf die Grafikkarte geladen. Insgesamt werden also ziemlich viele Daten zwischen der Grafikkarte und dem Prozessor hin und her geschoben. Besser ist es, die Daten dauerhaft auf der Grafikkarte zu lassen ohne sie immer wieder neu auf die Grafikkarte zu laden. Damit war es mir möglich die benötigte Zeit für eine Iteration ziemlich tief zu senken. Bei 10.000 Atomen konnte ich die benötigte Zeit auf ca. 2 ms senken. Um beide Optimierungen zusammen zu testen wollte ich mit 10.000 Atomen eine Flüssigkeit mit dem Lennard-Jones-Potential simulieren (Abb. 6.1). Dies funktionierte erstaunlich gut mit einer Rechenzeit von 0,5 ms.

Die Simulation von Festkörper konnte ich schon sehr gut optimieren, nun wollte ich aber auch die Galaxienkollision optimieren. Dort ist es aber im Gegenteil zum Lennard-Jones-Potential nicht möglich, die Wechselwirkungen ab einer bestimmten Distanz abzuschneiden. Eine Möglichkeit ist es, auf nahe Distanz die Gravitation wie normal zu berechnen. Sind mehrere Körper jedoch weit entfernt, werden diese zu einem einzigen Körper zusammengefasst, für den dann die Gravitation berechnet wird. Die einzige Ausnahme ist, das die schwarzen Löcher der Galaxien die Gravitation auf normalen Wege berechnen müssen, da sonst die Orbitale der Sterne leicht falsch sind. Weil ich momentan noch an dieser Methode der Optimierung arbeite, kann ich keine genauen Ergebnisse liefern, die benötigte Rechenzeit lässt sich so aber trotzdem schon stark senken.

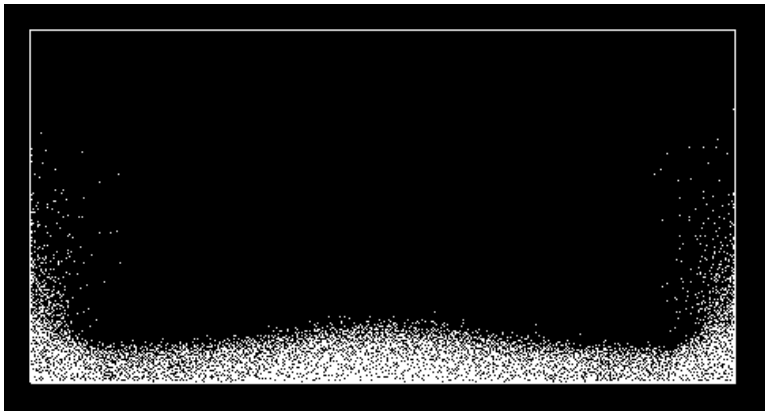


Abb. 6.1: Flüssigkeit mit 10.000 Atomen

7. Wassermoleküle

Als nächstes wollte ich mich der Simulation von Wassermolekülen zuwenden, um die Schneeflockenbildung zu untersuchen. Doch Moleküle im Allgemeinen verhalten sich anders als einzelne Atome. Als wichtigsten Unterschied zu Atomen können Moleküle Schwingungs- und Rotationsenergie besitzen. Würde ich die Moleküle korrekt simulieren müsste ich jedem Atom Elektronenorbitale geben und die Elektronenpaarbindung simulieren. Dies erfordert aber einen extrem hohen Rechenaufwand, da hier schon quantenphysikalische Gesetze simuliert werden müssen. Mein vereinfachter Ansatz war es nun, die Moleküle als starre Körper zu simulieren, um so herauszufinden, ob dieser vereinfachte Ansatz zu brauchbaren Ergebnissen führt. Die Gesetze zu starren Körpern [6] sind auch gut erklärt und leichter zu verstehen als quantenphysikalische Gesetze. Jedes Molekül hat die selben Eigenschaften zur Bewegung wie die Atome, zusätzlich dazu kommt aber noch die Rotation q als Quaternion, die Winkelbeschleunigung $\vec{\alpha}$ und der Massenmittelpunkten $\vec{m}_z$ als Vektoren. Zusätzlich wird noch die Gesamtmasse m_g des Moleküls benötigt. Außerdem muss der Trägheitstensor I als Matrix, der die Trägheit der Rotation angibt, für das Molekül berechnet werden. Wirkt jetzt eine Kraft auf eines der Atome des Moleküls das nicht auf dem Massenmittelpunkt liegt, löst dies eine Beschleunigung des Drehmoments vom Moleküls aus. Diese Winkelbeschleunigung $\vec{\alpha}$ lässt sich über das Drehmoment $\vec{M}=\vec{x}_a \times \vec{F}_a$ berechnen, wobei $\vec{x}_a$ die Position des Atoms bezogen zum Mittelpunkt des Moleküls ist und $\vec{F}_a$ die Kraft die auf das Atom wirkt. Dies muss für alle Atome eines Moleküls berechnet werden. Die neue Position des Moleküls lässt sich genau so wie die neue Position eines Atoms berechnen. Die neue Rotation hingegen wird anders berechnet, da die Zusammenhänge bei der Rotation nicht dieselben wie bei der Position sind. Zuerst muss die Winkelbeschleunigung mit dem Zeitschritt multipliziert werden $\vec{\alpha}=\vec{\alpha}\,dt$. Danach muss die Winkelbeschleunigung in das lokale Koordinatensystem des Moleküls gedreht werden $\vec{\alpha_{loc}}=q^{-1}\vec{\alpha}$ und daraus dann die lokale Winkelgeschwindigkeit $\vec{v_{loc}}$ mit $\vec{v_{loc}}=I^{-1}\vec{\alpha_{loc}}$ berechnet werden. Der Vorteil davon in das lokale Koordinatensystem zu drehen ist, das der Trägheitstensor I nicht für jede Iteration neu berechnet werden muss und dadurch Rechenzeit gespart wird. Die Winkelgeschwindigkeit muss dann wieder zurück in das globale Koordinatensystem gedreht werden $\vec{v_{glo}}=q\,\vec{v_{loc}}$. Als letztes wird die neue Rotation des Moleküls errechnet mit $q=(\vec{v_{glo}}\,dt)\,q$.

```
//Moleküle

int i=blockIdx.x*blockDim.x+threadIdx.x;

if(i<MOLECULE_COUNT)
{
    Vec3 rotForce=Vec3();

    //Kräfte berechnen
    for(int j=0;j<MOLECULE_SIZE;j++){
        molecules[i].acc+=bodies[i*MOLECULE_SIZE+j].force/molecules[i].mass;
        rotForce+=bodies[i*MOLECULE_SIZE+j].localPos.cross(bodies[i*MOLECULE_SIZE+j].force);
        bodies[i*MOLECULE_SIZE+j].acc=Vec3();
    }

    //Neue Position mit Verlet berechnen
    Vec3 oldPos=molecules[i].pos;
    molecules[i].pos=molecules[i].pos*2-molecules[i].oldPos+molecules[i].acc*(dt*dt);
    molecules[i].oldPos=oldPos;
```

```
    //Neue Rotation berechnen
    molecules[i].rotMom+=rotForce*dt;
    Vec3 localRotMom=molecules[i].rot.conjugateStatic()*molecules[i].rotMom;
    Vec3 localRotVel=molecules[i].invInertiaTensor.multiplyEach(localRotMom);
    Vec3 rotVel=molecules[i].rot*localRotVel;
    Quat qRot=Quat::fromRotationVector(rotVel*dt);
    molecules[i].rot=qRot*molecules[i].rot;

    //Neue Position der Atome berechnen
    for(int j=0;j<MOLECULE_SIZE;j++){
        bodies[i*MOLECULE_SIZE+j].pos=molecules[i].rot*molecules[i].bodiesLocalLocalPos[j];
        bodies[i*MOLECULE_SIZE+j].pos+=molecules[i].pos;
    }
}
```

Da ich Wassermoleküle simulieren wollte und dort die einzelnen Atome partiell elektrisch geladen sind, war es noch nötig die Coulomb-Kraft zu simulieren. Diese beschreibt die Wechselwirkung zwischen zwei geladenen Körpern, lässt sich aber auch auf partiell geladene Atome anwenden. Dazu bekommt jedes Atom noch die Ladung q als Eigenschaft zugewiesen. Mit den Ladungen lässt sich dann mit der Formel

$$\vec{F}_1 = \frac{q_1 q_2}{4\pi\epsilon_0} \frac{\hat{r}_{12}}{|r_{12}|^2}$$

die Kraft für das Atom mit der dazugehörigen Ladung q_1 berechnen. In Code umgesetzt wäre dies

```
//Coulomb

int i=blockIdx.x*blockDim.x+threadIdx.x;

if(i<BODY_COUNT)
{
    for(int j=0;j<BODY_COUNT;j++)
    {
        if(i==j)
            continue;

        Vec3 normal=bodies[i].pos-bodies[j].pos;
        float distSqr=normal.lengthSquared();
        normal.normalize();
        float chargeProduct=bodies[i].charge*bodies[j].charge;
        bodies[i].acc+=(normal/distSqr)*(chargeProduct/(4*PI*ELECTRIC_CONST));
    }
}
```

Um die Simulation von Molekülen zu testen, stellte ich zwei Wassermoleküle nebeneinander (Abb. 7.1) um zu sehen wie sie sich verhalten. Zusätzlich zur Coulombkraft aktivierte ich das Lennard-Jones-Potential um den Dipolmoment des Moleküls zu simulieren. Am besten wäre es, wenn man direkt erkennen könnte, dass die Moleküle eine Wasserstoffbrückenbindung eingehen. Dabei sollte sich ein Wasserstoffatom auf das Sauerstoffatom des anderen Moleküls ausrichten. Nach dem Start

der Simulation war sichtbar das im Nahbereich die Wasserstoffatome von den Sauerstoffatomen angezogen wurden. Auch das Lennard-Jones-Potential bewirkte, das die Moleküle sich nicht überschnitten. Trotzdem wurden sie voneinander weggestoßen. Ich versuchte, den Abstand zwischen den Molekülen zu verändern, jedoch ohne Erfolg. Auch Einstellungen an den Stärken der Kräfte brachten wenig Erfolg. Die Moleküle schienen einfach nicht stabil genug zu bleiben um eine Wasserstoffbrücke zu bilden. Nach etwas Nachforschung im Internet erfuhr ich dann, das Wasserstoffbrückenbindungen normalerweise erst bei mehreren Molekülen entstehen. Deswegen ließ ich 200 Wassermoleküle generieren (Abb. 7.2) die zufällig in einem Würfel verteilt waren. Dies sollte, wenn Wasserstoffbrückenbindungen entstehen, diese stabilisieren. Leider kam es trotzdem zu keiner Bildung von Wasserstoffbrücken. Deswegen recherchierte ich erneut im Internet und fand heraus, das es viele verschiedene Modelle zur Simulation von Wasser gibt. Sie unterscheiden sich in der Komplexität und damit in der Genauigkeit und benötigten Rechenzeit. Wassermoleküle richtig zu simulieren erfordert, wie oben schon erwähnt quantenphysikalische Berechnungen. Somit bin ich mit meiner klassischen Vielteilchensimulation hier an Grenzen gestoßen.

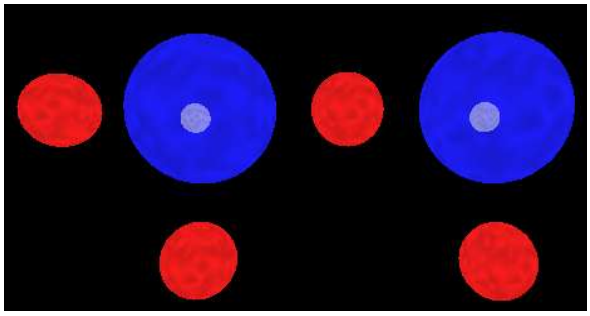


Abb. 7.1: Zwei Wassermoleküle am Start der Simulation. In Rot Wasserstoff, in Blau Sauerstoff

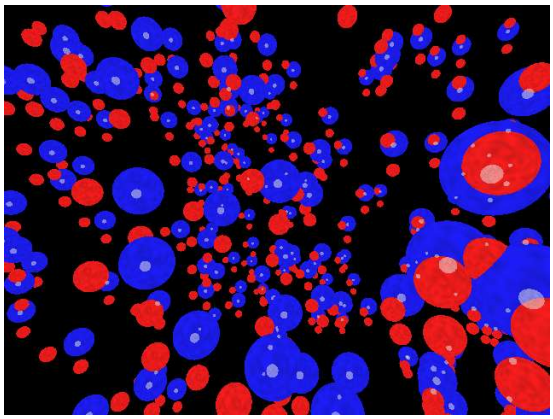


Abb. 7.2: 200 Wassermoleküle im dreidimensionalen Raum

8. Nanofluidik

Bisher konnte ich zeigen, dass meine Simulationen genaue Ergebnisse im Bereich der Galaxienkollisionen und im Bereich der Simulationen von Festkörpern, Flüssigkeiten und Gasen liefern. Jedoch ist es mir nicht möglich, die Quantenwelt zu simulieren. Deswegen beschloss ich den Maßstab etwas zu vergrößern und in die Nanowelt zu gehen.
Ein noch weitgehend unerforschtes Thema ist das Strömungsverhalten von Flüssigkeiten in der Nanowelt. Da im Bereich der Nanofluidik hauptsächlich elektrische Kräfte wirken, musste ich überprüfen, wie weit die Gesetze von Bernoulli und Venturi (Abb. 8.1) hier noch gelten. Nach ihnen verhält sich die Fließgeschwindigkeit einer Flüssigkeit, die durch ein Rohr fließt, antiproportional zum Querschnitt und der Druck proportional zum Querschnitt. Um dies zu testen, baute ich das Rohr aus Abb. 8.1 nach. Es besaß einen Durchmesser von nur 4 nm und an der Engstelle einen Durchmesser von 2 nm. Der Flüssigkeit gab ich einen Impuls nach rechts,

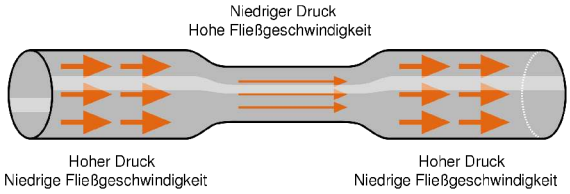


Abb. 8.1: Strömung nach Bernoulli und Venturi [7]

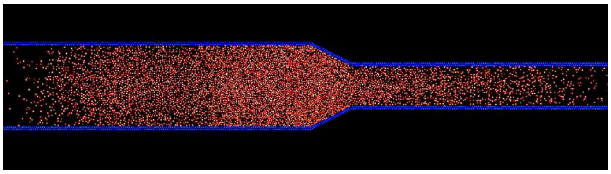


Abb. 8.2: Verstopftes Rohr

sodass sie zur Engstelle fließt. Jedoch staute sich die Flüssigkeit am Übergang zur Engstelle auf und verstopfte so das Rohr (Abb. 8.2). Zusätzlich wurde der Teil der Flüssigkeit, der in die Engstelle geflossen ist, gasförmig. Damit konnte ich zeigen, dass sich die bekannten Gesetze aus der makroskopischen Welt nicht auf die Nanowelt übertragen lassen.

Das Verständnis des Strömungsverhaltens in der Nanowelt wird jedoch immer wichtiger, um z.B. Geräte wie ein Lab-on-a-Chip zu bauen. Ein Lab-on-a-Chip wird verwendet um mit extrem kleinen Flüssigkeitsmengen zu arbeiten. Momentan werden diese Geräte im Mikrometerbereich gebaut. Man plant jedoch diese auch im Nanometerbereich zu bauen. Hier stößt man jedoch auf die oben genannten Probleme. Deshalb habe ich es mir zur Aufgabe gestellt, mit Hilfe von Simulationen Pläne für Bauteile zu entwickeln, die die Realisierung eines Lab-on-a-Chip auch in der Nanowelt erlauben.

Möchte man eine Flüssigkeit durch ein Nanorohr pumpen, so kann man keine Kolben verwenden, da diese eine extrem hohe Reibung mit der Wand aufweisen. Dieses Problem löste ich, indem ich eine Pumpe konstruierte und simulierte, die komplett auf mechanische Komponenten verzichtet. Meine Idee war es, die Flüssigkeit positiv aufzuladen (Abb. 8.3). Wird dann zusätzlich ein Stück der Wand positiv aufgeladen, wird die Flüssigkeit, da sie auch positiv geladen ist, abgestoßen (Abb. 8.4). Vergrößert man dann mit der Zeit den positiv geladenen Bereich der Wand, kann man die Flüssigkeit gezielt in eine Richtung pumpen (Abb. 8.5).

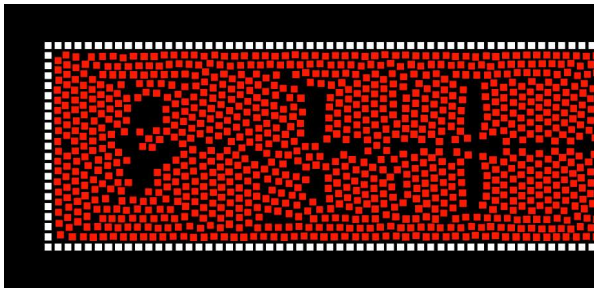


Abb. 8.3: Pumpe am Anfang

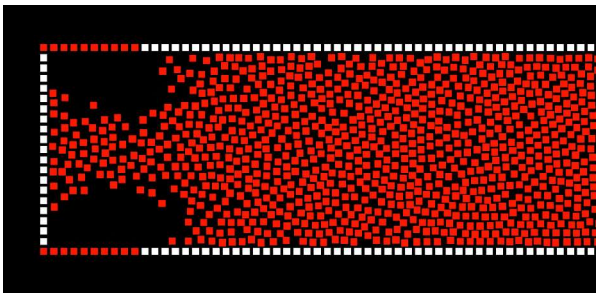


Abb. 8.4: Pumpe nach Start

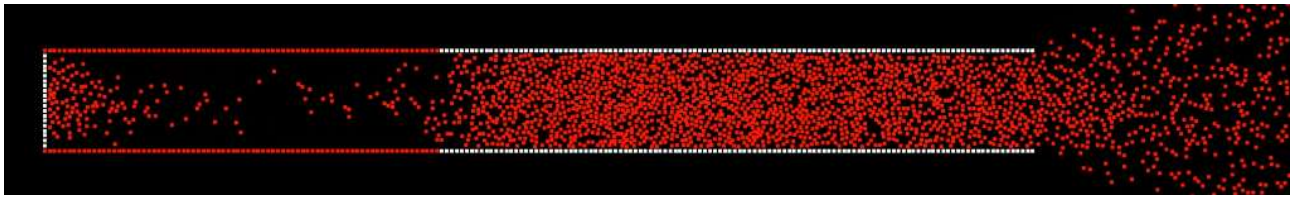


Abb. 8.5: Die Pumpe bewegt die Flüssigkeit mit einem nur sehr kleinen Verlust nach rechts

Nun konnte ich eine Flüssigkeit bewegen, jedoch noch nicht den Flüssigkeitsstrom schalten. Dies ist aber notwendig, wenn man z.B. einen Flüssigkeitsbehälter benutzt und den Flüssigkeitsaustritt gezielt schalten möchte. Dazu musste ich einen Transistor für Flüssigkeiten entwickeln. Dies gelang mir, indem ich die Ladung eines Rohrbereiches gezielt verändern kann. Ist ein Stück des Rohres positiv geladen, kann dort keine Flüssigkeit durchfließen. Der Transistor sperrt (Abb. 8.6). Wird der Transistor jedoch neutral geladen, kann die Flüssigkeit fließen (Abb. 8.7).

Das letzte Bauteil was jetzt noch fehlt, war ein Schalter, der den Flüssigkeitsstrom verzweigen kann. Dies ist von Vorteil, wenn z.B. eine Flüssigkeit zu zwei unterschiedlichen Sensoren geleitet werden soll. Ein Schalter löst dieses Problem. Meine Idee für seine Umsetzung in der Nanowelt war

es, ein T-förmiges Rohr zu konstruieren (Abb. 8.8). An Rohr B und C habe ich einen Transistor angebaut. Durch das Öffnen oder Schließen der Transistoren kann man dann steuern, zu welchem Sensor die Flüssigkeit fließt.
Indem ich neue Bauteile in der Nanowelt entwickelt habe, konnte ich zeigen, dass Vielteilchensimulationen dazu benutzt werden können Neues zu erforschen. Ich hoffe, das meine Ergebnisse den Bau von Geräten in der Nanowelt erleichtern werden.

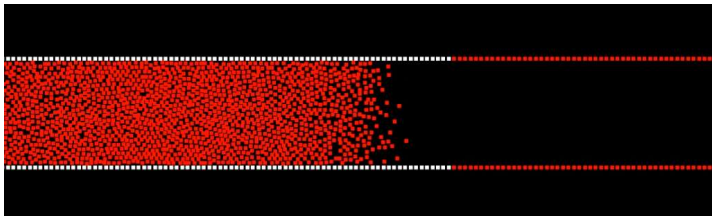


Abb. 8.6: Flüssigkeitsschaltender Transistor sperrt

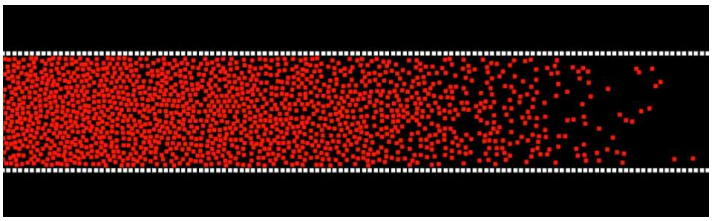


Abb. 8.7: Geöffneter Transistor

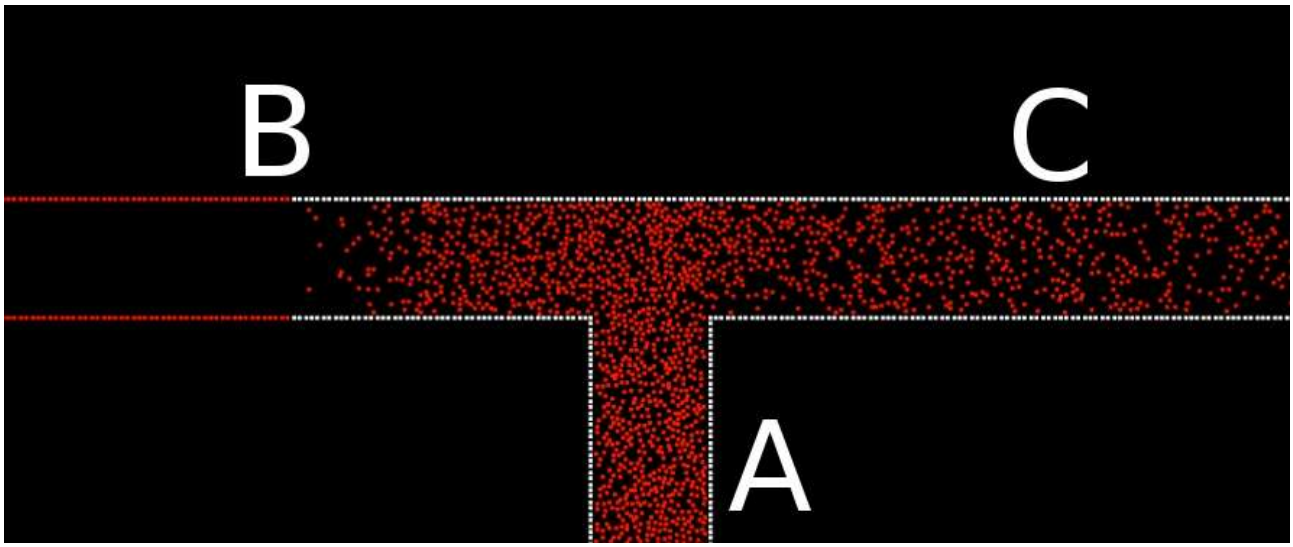


Abb. 8.8: Schalter für Flüssigkeiten (C geöffnet)

9. Anhang

Im Zeitraum vom Hochladen der Arbeit bis zum Bundeswettbewerb konnte ich noch einige Neuerungen erzielen. Nachdem ich im zweidimensionalen Bereich schon Simulationen in der Nanowelt berechnet hatte, wollte ich meine Simulationen auch in der dritten Dimension tätigen. Zusätzlich wollte ich in meinen Simulationen Wassermoleküle benutzen. Wie in Kapitel 7 beschrieben, war mir dies bis jetzt jedoch noch nicht gelungen. Nach längerem Suchen stieß ich dann aber doch auf einen Ansatz, der die Wassermoleküle sogar quantenmechanisch korrekt

simulieren soll. Bei diesem Ansatz werden die Atome eines Moleküls untereinander mit Federn verbunden. Diese Federn simulieren die Wechselwirkung zwischen den Elektronenorbitalen. Gibt man diesen Federn die richtige Länge und Stärke, so erhält man sehr genaue Ergebnisse.
Dieses Verfahren implementierte ich in mein Programm und prüfte es an verschiedenen Simulationen. Dabei fiel mir ein großer Nachteil dieses Verfahrens auf. Die Größe des Zeitschrittes der Simulation muss stark gesenkt werden, um mit dieser Methode zu arbeiten. So vergrößert sich die Zeit, bis man ein brauchbares Ergebnis aus der Simulation bekommt. Jedoch ging es mir auch darum, meine Simulation möglichst schnell zu berechnen, ohne viel Genauigkeit zu verlieren. Deshalb entwickelte ich mein eigenes Verfahren zur Simulation von Wassermolekülen auf Basis von vorherigen Simulationen mit dem Federmodell.
Mir fiel auf, dass Wassermoleküle sich an einem elektrischen Feld ausrichten und so immer von diesem Feld angezogen werden. Auch bilden einzelne Wassermoleküle untereinander Cluster bzw. Tropfen. Beides liegt daran, dass das Wassermolekül einen Dipol darstellt. Meine Idee war es dann, in meinen Simulationen das Wasser als nur ein einzelnes Teilchen darzustellen und diesem Teilchen die effektive Ladung des Wassermoleküls zu geben. Da sich ein Wassermolekül an elektrischen Feldern ausrichtet und deshalb von inhomogenen elektrischen Feldern auch nur angezogen wird, sage ich meinem Programm, dass das Wasserteilchen von elektrischen Feldern nur angezogen werden sollen. Die Clusterbildung von Wasserteilchen untereinander resultiert dann automatisch daraus, dass jedes Wasserteilchen eine Ladung hat und sie sich so auch untereinander anziehen.
Mit meinem Verfahren stelle ich das für Wassermoleküle typische Verhalten dar und kann meine Simulationen 50 bis 100 mal schneller laufen lassen ohne viel Genauigkeit zu verlieren.
Mit meinem neuem Wassermodell wollte ich dann einen Schalter simulieren. Dazu nahm ich ein Y-Nanorohr und füllte dies mit einer hydrophoben Flüssigkeit, z.B. Öl. Gibt man dann in diese Flüssigkeit kleine Wassertröpfen, kann man die Wassertröpfen als Transportbehälter für z.B. Proteine oder DNA benutzen. Um zu schalten, in welches Rohr das Wasser fließt, benutzte ich zwei Elektroden. Da Wasser nach meinem Modell von elektrischen Feldern angezogen wird, sollte es auch von der Elektrode angezogen werden. Durch welches Rohr das Wasser fließt kann gesteuert werden, indem man die Elektrode an der jeweiligen Röhre einschaltet (Abb. 9.1, Abb. 9.2).

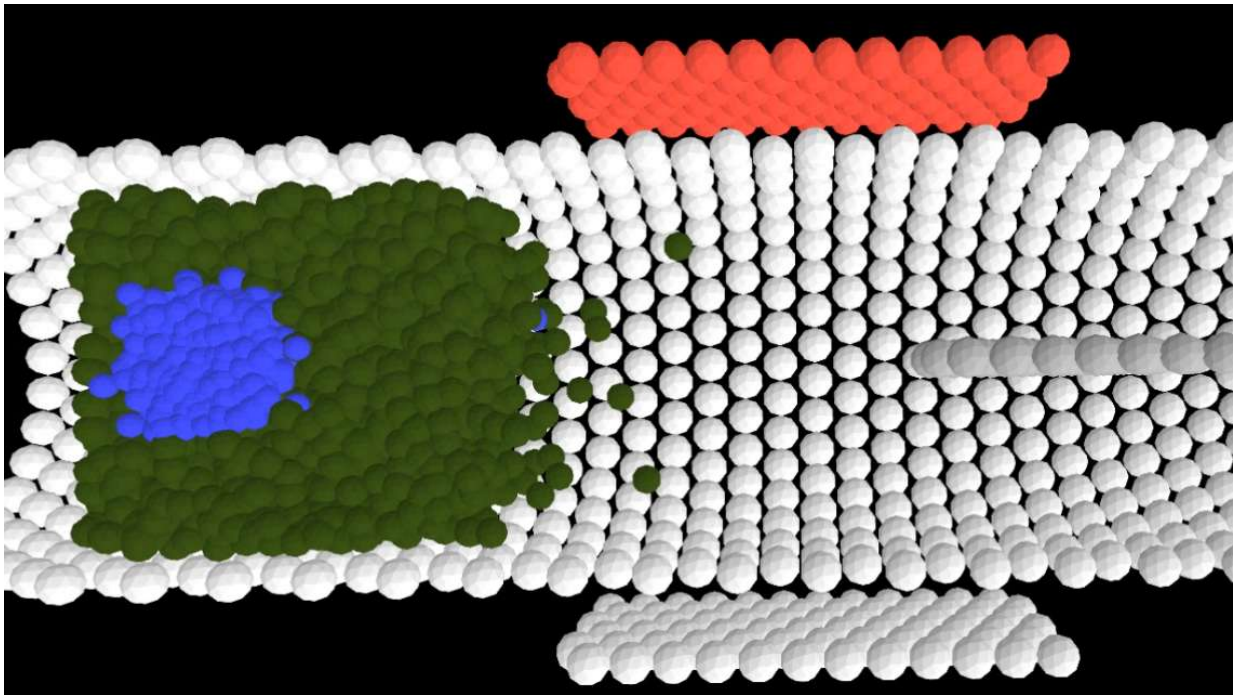


Abb. 9.1: Y-Nanorohr, obere Elektrode eingeschaltet

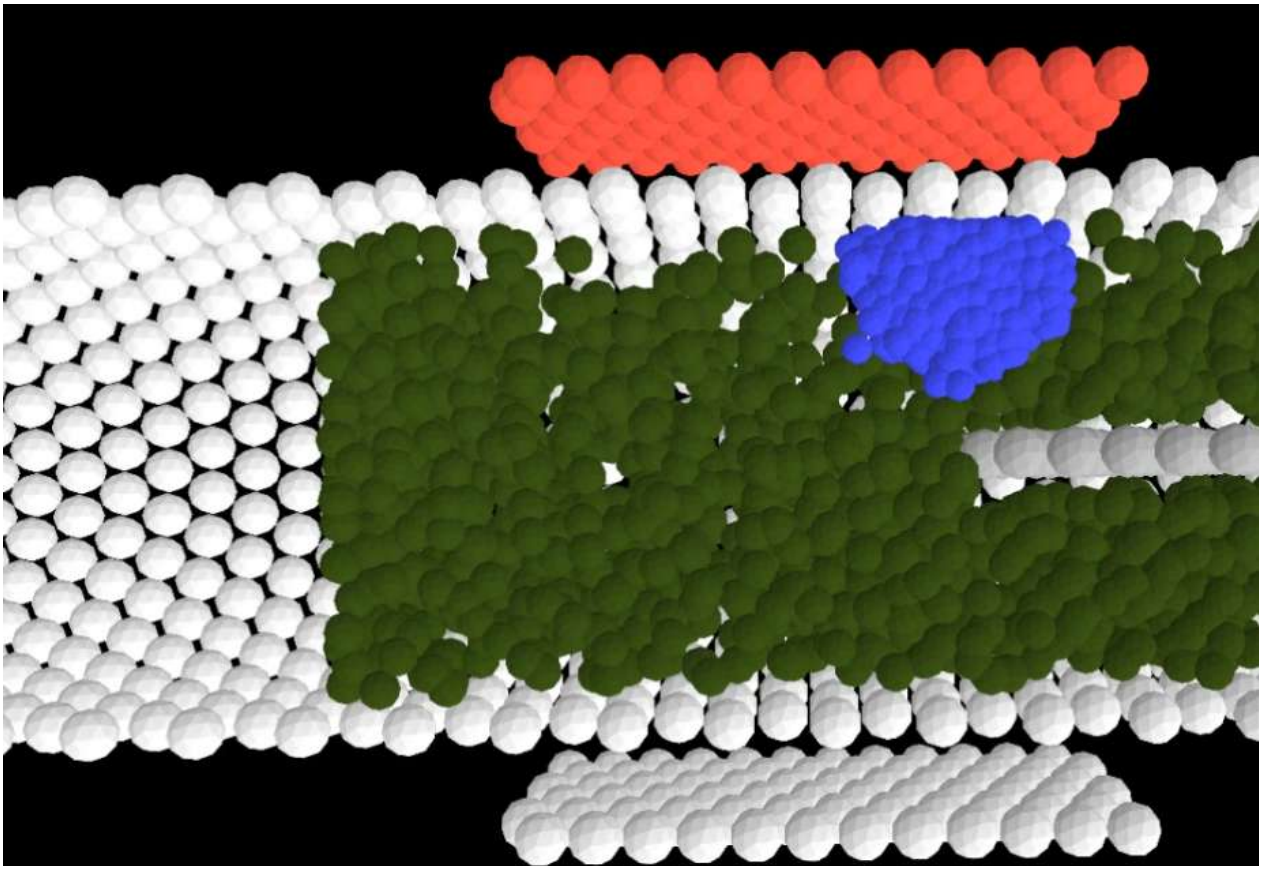


Abb. 9.2: Der Wassertropfen (blau) wird von der Elektrode angezogen und fließt nur durch das obere Rohr

10. Linkliste

[1] - http://de.wikipedia.org/wiki/Differentialgleichung
[2] - http://abyss.uoregon.edu/~js/images/hubblefork.gif
[3] - http://upload.wikimedia.org/wikipedia/de/1/11/Spiralgalaxie_Aufbau.png
[4] - http://chemwiki.ucdavis.edu/Physical_Chemistry/Quantum_Mechanics/Atomic_Theory/Intermolecular_Forces/Lennard-Jones_Potential
[5] - https://en.wikipedia.org/wiki/Lennard-Jones_potential#Truncated_Lennard-Jones_potential
[6] - https://developer.nvidia.com/content/gpu-gems-3-chapter-29-real-time-rigid-body-simulation-gpus
[7] - https://de.wikipedia.org/w/index.php?title=Datei:Schema_eines_Rohres_mit_Engstelle-Bernoulli.svg&filetimestamp=20101113200215&
[8] - http://commons.wikimedia.org/wiki/File:Interacting_Galaxies_Hubble.jpg
[9] - http://commons.wikimedia.org/wiki/File:12-6-Lennard-Jones-Potential.svg
[10] - http://www.nanodaten.de/uploads/downloads/6a1b43df90173439514e7d1c6168d4ac.PDF