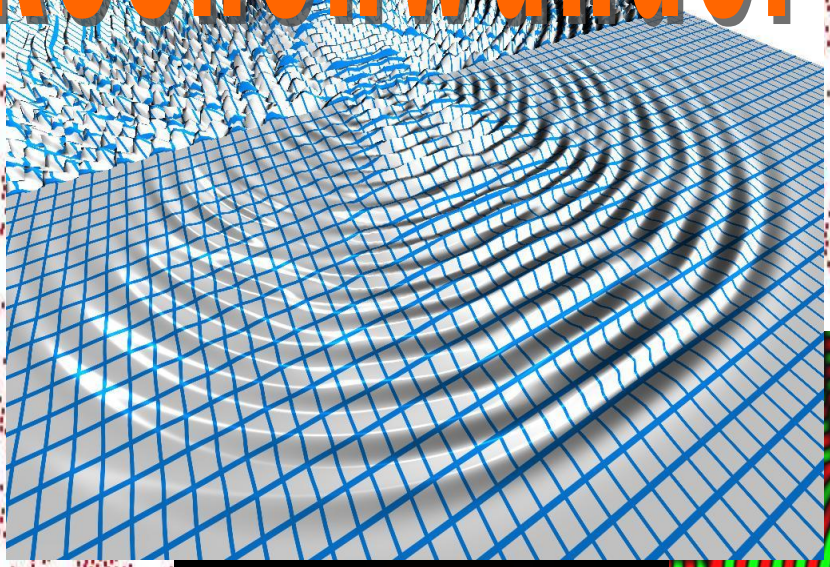
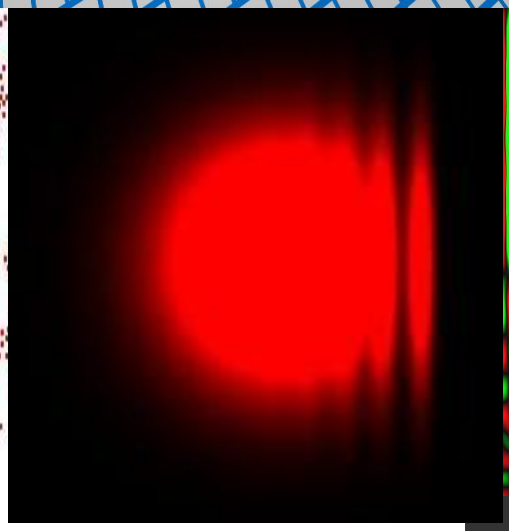
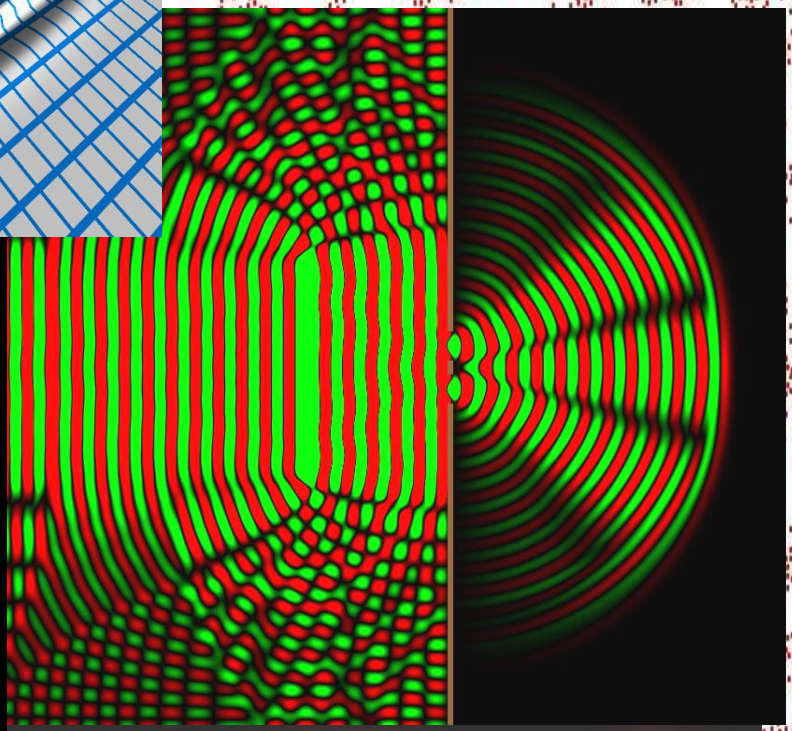
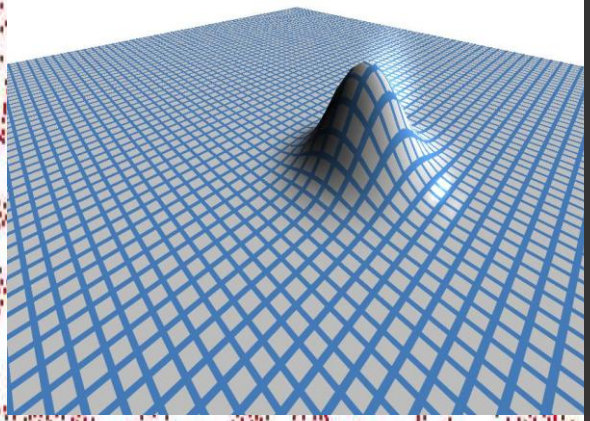
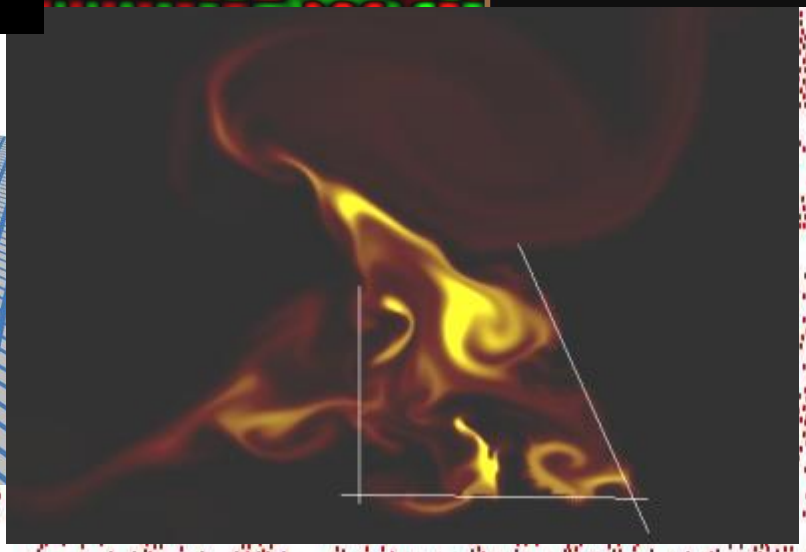
# Rechenwunder Grafikkarte











**Heiko Burau – Jugend forscht 2008**

Inhalt



# 1. Kurzfassung

Wussten Sie, dass die Grafikkarte 100 Mal schneller rechnen kann, als die normale CPU? Wahrscheinlich nicht, denn in der Praxis ist das ja nicht spürbar, weil die Grafikkarte nur Grafik macht und sonst nichts. Das Ziel meines Projektes ist es, ihre enorme Rechenpower für sinnvolle Dinge zu nutzen. Die Grafikkarte hat nämlich das Potenzial, ganz neue Standards für PC-Anwendungen zu setzten. Diese Standards will ich für den Bereich der physikalischen Simulation definieren. Dazu habe ich einige interaktive Systeme entwickelt, die um Größenordnungen schneller und genauer sind, als alles was bisher auf dem PC möglich war. Als erstes entstand ein Wellensimulator für alle Art von Wellen, gefolgt von einer Strömungssimulation. Mit diesen beiden Programmen habe ich mich dann am Forschungszentrum Jülich bei Numerik-Experten vorgestellt, die meine Simulationen kritisch betrachtet haben und sehr beeindruckt waren. Mit Ihnen konnte ich mich frei austauschen und viele neue Ideen sammeln. Danach habe ich mich mit N-Body Systemen beschäftigt und schließlich eine Simulation der Quantenmechanik entwickelt. Zum Schluss habe ich mich mit dem Max-Planck Institut Bonn d.h. dem 100m-Radioteleskop Effelsberg in Verbindung gesetzt und die Magnetosphäre eines Pulsars simuliert, wodurch endlich erklärt werden kann, weshalb diese Sterne strahlen.

# 2. Einleitung

Sicherlich hat jeder schon einmal die eine oder andere Simulation gesehen. Egal ob es die Physik-, Verkehrs- oder Flugsimulation ist, es ist immer wieder faszinierend zu beobachten was in Wirklichkeit passieren würde, wenn dies und jenes geschieht. Zum Beispiel sieht man jetzt oft im Fernsehen diese Klimasimulationen, wo im Zeitraffer gezeigt wird wie sich die Erde in 100 Jahren entwickelt. Da frage ich mich immer wie die das machen. Am besten wäre ja man könnte selbst mal ein paar Sachen ausprobieren und würde dann diesen Zeitraffer sehen. Dann würde man auch das Klima viel besser verstehen.
Aber leider ist es völlig unmöglich so eine Simulation selber zu schreiben. Einerseits ist man kein Meteorologe, andererseits hat man keinen millionenteuren Supercomputer bei sich zu Hause stehen. Aber man kann immerhin, wenn man ein wenig Ahnung vom Programmieren hat, kleinere Experimente für den PC entwickeln. Im Informatikkurs an der Schule lernt man in der Regel Pascal oder Delphi, womit man schon einiges anfangen kann. Man kann z.B. das Sonnensystem simulieren und ansatzweise untersuchen, ob die Planetenbahnen stabil sind (Dreikörperproblem) oder ein gekoppeltes Pendel virtuell schwingen lassen (Chaostheorie). Auch mit Excel kann man viel machen (stehende Wellen, freier Fall, …).
Das Besondere an diesen Dingen ist die Interaktivität. Durch Ausprobieren begreift man einfach viel besser, was hinter den physikalischen Formeln wirklich steckt und wird es nie wieder vergessen. Es gibt auch unzählige fertige Java-Applets im Internet und Programmpakete mit vielen virtuellen Experimenten.
Alle diese Simulation haben jedoch einen gravierenden Nachteil:

Sie stellen niemals die Wirklichkeit dar! Dazu sind sie viel zu idealisiert. Wenn beim Doppelspaltversuch einfach zwei Sinuswellen überlagert werden, dann ist das falsch (wie wir noch sehen werden). Genauso das Quantenteilchen im unendlich tiefen Potenzialtopf. Alles idealisiert. Andere Dinge wie eine Klimasimulation können nicht mal ansatzweise simuliert werden, da schon eine realistische Strömungssimulation, die Rechenleistung einer CPU überstiegt. Oder die Frage: Was ist ein Elektron? Natürlich kein Punktteilchen, sondern eine Materiewelle. Aber wie die genau aussieht und sich verhält, das rechnet man zwar im Physikstudium aus, aber eine intuitive Vorstellung von einem Elektron erhält man dadurch nicht. Auch hier ist eine realistische Simulation mit herkömmlichen Mitteln nicht möglich, weil der PC einfach zu schwach dafür ist.
Die bisherigen Simulationen am PC sind nur qualitativ wertvoll, aber dafür für jeden interaktiv zugänglich. Um quantitativ brauchbare Ergebnisse zu erhalten, mit denen man wirklich neue Dinge erforschen kann, braucht es schon Supercomputer. Allerdings ist es hier mit der Interaktivität vorbei.
Der Schlüssel für ernst zunehmende Simulationen lautet also Rechenleistung. Und genau diese Rechenleistung ist in jedem modernen PC zu finden. Aber nicht in der CPU sondern in der so genannten GPU, also dem Prozessor der Grafikkarte. Dummerweise kann man bei der GPU nicht einfach so „drauf los“ programmieren wie man das gewohnt ist, weil sie eigentlich nur für schnelle 3D-Videospiele konstruiert ist. Zunächst einmal muss man eine ziemlich aufwendige Rahmenanwendung schreiben, um die Karte anzusteuern. Hat man diese Hürde genommen, geht es darum die zahlreichen Register möglichst optimal anzupassen. Das Wichtigste ist aber, dass man Code schreibt, der parallel ausgeführt werden kann. Die GPU besteht nämlich aus vielen kleinen Parallel-Prozessoren, die alle möglichst gleichmäßig ausgelastet werden müssen. Um schließlich die maximale Performance herauszuholen, muss man einige Spezialfähigkeiten kennen, die nur eine GPU hat.
Hat man aber einmal gelernt wie das alles funktioniert, eröffnet sich ein ganz neuer Kosmos. Wo man früher 100 Sterne umeinander kreisen lassen konnte, sind es jetzt mehrere zehntausend. Es ist wirklich unglaublich, was auf einmal alles möglich ist. Endlich realistische Simulationen, die auch eine Aussagekraft haben und zudem noch interaktiv sind. Dieses großartige Potenzial ist aber bisher noch weitgehend ungenutzt. Erst seit einem Jahr werden entsprechende Projekte aktiv vom GPU-Hersteller NVidia gefördert. Das liegt wohl auch daran, dass man erst seit 2-3 Jahren eine GPU überhaupt sinnvoll programmieren kann. Außerdem muss man erst einmal die ganzen Software-technischen Schwierigkeiten überwinden.
**In meiner Jugend-forscht Arbeit möchte ich dies tun, und zeigen dass man echte Physik auf hohem Niveau auch zu Hause am PC betreiben kann**. Außerdem will ich eine Funktions-Bibliothek entwickeln mit der jeder Hobbyprogrammierer mit nur wenigen Befehlen die Grafikkarte für seine Zwecke ausreizen kann.

# 3. Wie programmiert man eine Grafikkarte?

Jede Komponente in einem PC hat eine bestimmte Aufgabe. Da gibt es einmal die Eingabeeinheit, nämlich Tastatur und Maus, dann die Datenverarbeitung, also CPU und Speicher und schließlich die Ausgabeeinheit in Form von Grafikkarte und Soundkarte. Noch vor gut 15 Jahren konnte die Grafikkarte nicht viel mehr als die Daten, die von der CPU kamen, an den Bildschirm weiterzuleiten. Als dann die ersten 3D-Spiele auf dem Markt kamen, änderte sich das. Will man nämlich 3D-Grafik darstellen, explodiert der Rechenaufwand. Es müssen zehntausende Vektoren transformiert und Millionen Pixel berechnet werden, und zwar mindestens 30 Mal pro Sekunde für eine flüssige Bewegung. Das alles war mit der CPU nicht mehr zu machen. Einerseits fehlte es an purer Rechenleistung, andererseits konnte der Datenbus unmöglich so viele Daten in so kurzer Zeit zur Grafikkarte schicken. Also ging man hin und bestückte die Grafikkarte mit ein paar Millionen Transistoren, die nur dazu da waren, Vektoren mit einer Matrix zu transformieren. Sie konnten nur diese eine Sache, aber dafür extrem schnell. Diese spezielle Rechenoperation war fest in der Hardware verschaltet und musste nun nicht mehr in Software auf der CPU implementiert werden. Außerdem wurden die Rohdaten für die Grafikausgabe (3D-Modelle, Texturen, …) in einem besonders schnellen Speicher (Videospeicher) untergebracht, der direkt auf der Grafikkarte saß. Damit war auch der Datenbus der CPU entlastet.

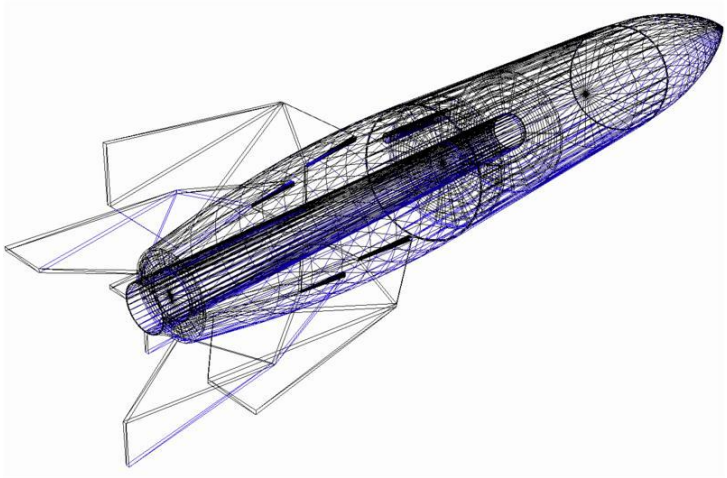


Abb. 3.1: Ein 3D-Modell aus Dreiecken

Aber was hat das alles mit meinem Thema zu tun? Nun, um heraus zu bekommen wie ich die Grafikkarte für meine Zwecke „missbrauchen“ konnte, musste ich mich mit der Hardware vertraut machen und das konnte ich

nur, indem ich mich intensiv mit der 3D-Spieleprogrammierung beschäftigte. Als ich anfing gab es da das Buch „3D-Spieleprogrammierung“ von David Scherfgen, wo Schritt für Schritt erklärt wird wie man ein 3D-Spiel in C++ entwickelt. Das war genau das richtige für mich. So lernte ich nämlich alle Details der Hardware kennen und erfuhr wie man sie mit Microsoft DirectX 9 ansteuert. Aber dazu später mehr. Zunächst einmal: Was macht eine Grafikkarte während eine 3D-Anwendung läuft?

Die virtuelle Welt besteht aus 3D-Körpern, die im Videospeicher liegen. Diese Körper sind aus lauter Dreiecken aufgebaut (Abb. 3.1), denn dies ist die einfachste geometrische Figur mit der man einen Körper aufbauen kann. Die Dreiecke sind definiert durch drei Ortsvektoren, die sogenannten Vertices (Sing. Vertex). Diese Vertices befinden sich im lokalen Koordinatensystem, dessen Ursprung normalerweise in der Mitte des Modells liegt. Die Vertices müssen nun auf eine 2D-Ebene projiziert werden, damit man sie auf dem Bildschirm darstellen kann (denn der ist ja nur 2-dimensional). Das erledigt eine 4x4 Projektionsmatrix. Bevor allerdings projiziert wird, wird noch das lokale Koordinatensystem an die Position des 3D-Modells transformiert, und dann nochmals von der Kamera transformiert. Also haben wir drei 4x4-Matrizen, die man schon im Vorhinein zu einer Matrix kombiniert. Die Grafikkarte transformiert jetzt alle Vertices (also Ortsvektoren) mit dieser Matrix. Jetzt hat man also ein Drahtgittermodell auf dem Bildschirm. Damit es nach einem echten 3D-Körper aussieht, muss man das Modell jetzt nur noch „anstreichen“. Dazu übergibt man der Grafikkarte ein normales 2D-Bild, das ist die Textur, und lässt es wie eine Tapete über das Modell legen (Abb. 3.2).

Die Grafikkarte berechnet jetzt für jeden Pixel auf dem Bildschirm, der in einem texturierten Dreieck liegt, die Position des dazugehörigen Pixels auf der Textur. Die Pixel der Textur nennt man auch „Texel“. Dieser Texel wird dann in den Bildschirm-Pixel kopiert.

Es gibt also einmal Recheneinheiten, welche die Vertices transformieren und dann solche, die die Pixel berechnen. Mit der Zeit erhielten diese zwei Rechenwerke immer mehr Spezialfähigkeiten, wie z.B. Beleuchtung, Multi-Texturen, Bump-Mapping und vieles mehr. Für jede neue Fähigkeit mussten extra ein paar Millionen Transistoren verbaut werden. Und im Quelltext musste man jedesmal, wenn man etwas Zeichnen wollte, etliche Register setzten:

*pDevice->SetRenderState(D3DRS_SPECULARENABLE, TRUE); //Glanzlichtberechnung ein*

Die Grafikprogrammierung ist durch diese Herangehensweise natürlich ziemlich unflexibel, weil man nicht einfach tolle neue Effekte programmieren kann, sondern warten muss bis die dafür nötigen Schaltkreise in der Hardware eingerichtet sind.

Die Revolution kam dann mit der „Geforce 3“ von NVidia. Diese Karte hatte einen Prozessor, die GPU (Graphics Processing Unit), die für jeden Vertex und für jeden Pixel ein kleines Programm, den sogenannten Shader, ausführt. Das Besondere ist, dass man diesen Shader fast beliebig aus einigen Assemblerbefehlen zusammensetzen kann. D.h. man kann jetzt plötzlich ganz frei bestimmen, wie die Farbe des Pixels ist und welche Position der Vertex hat.

Abb.3.2: Ein Würfel mit einer Textur

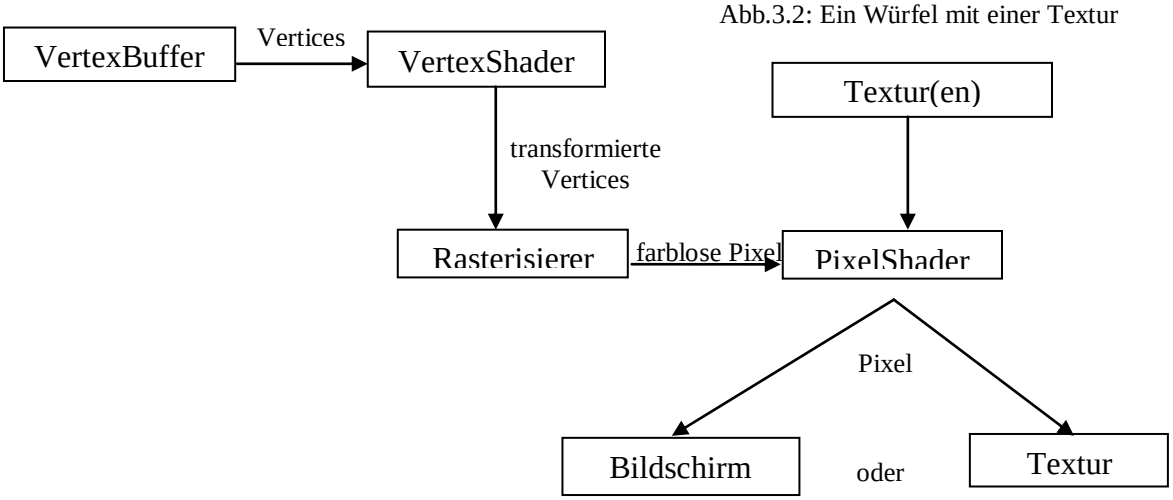



Abb. 3.3: Die vereinfachte Grafikpipeline. Das 3D-Modell liegt im VertexBuffer im Videospeicher vor und wird vom VertexShader transformiert. Die 2-dimensionalen transformierten Dreiecke gelangen zum Rasterisierer, der bestimmt welche Pixel das jeweilige Dreieck füllen. Diese noch farblosen Pixel erhalten vom PixelShader ihre Farbe. Dieser kann dabei auf ein oder mehrere Texturen zurückgreifen. Die fertigen Pixel können jetzt entweder am Bildschirm angezeigt, oder in einer Textur gespeichert werden.

Ab der Shaderversion 2.0 ist es im Prinzip egal, was die beiden Shader machen, Hauptsache es kommt am Ende eine Position bzw. eine Farbe heraus. Jetzt wird es endlich interessant! Meine Idee ist nämlich, dass ich die Simulation komplett im PixelShader ablaufen lasse. Das Ergebnis des Berechnungsschrittes ist dann einfach die Farbe des Pixels. Und alle nötigen Eingangsdaten liegen in Texturen vor, auf die der PixelShader ja Zugriff hat. Der Trick ist dann, die fertigen Pixel nicht am Bildschirm anzuzeigen, sondern in einer neuen Textur zu speichern, die dann wieder im nächsten Schritt vom PixelShader ausgelesen werden kann.
Der VertexShader hat eigentlich fast nichts zu tun. Ihm übergebe ich einfach nur zwei vortransformierte Dreiecke, welche zusammen ein Viereck ergeben, das so groß wie der Bildschirm ist. Diese leitet er dann einfach nur weiter. Wenn die Auflösung also z.B. 1024x768 lautet, dann berechnet der Pixelshader in jeder Iteration 786432 Pixel.
Um die Idee zu testen, habe ich als erstes ein ganz einfaches Beispiel ausprobiert. Das Bild soll sich bei jeder Iteration um einen Pixel nach rechts bewegen und dabei langsam ausgeblendet werden. Die Texturgröße entspricht der Bildschirmgröße.
Im VertexBuffer stehen jeweils Position und Texturkoordinaten der Vertices. Für die Position befindet sich (x,y)

VertexBuffer:
```
Vertex 1:
Pos: -1.0f, 1.0f
Tex: 0.0f, 0.0f
Vertex 2:
Pos: 1.0f, 1.0f
Tex: 1.0f, 0.0f
Vertex 3:
Pos: -1.0f, -
1.0f
Tex: 0.0f, 1.0f
Vertex 4:
Pos: 1.0f, -1.0f
Tex: 1.0f, 1.0f
```

VertexShader:
```
vs.1.1
dcl_position v0
dcl_texcoord v1

mov oPos,   v0
mov oT0,    v1
```

PixelShader:
```
ps.2.0
dcl_2d      s0
dcl         t0.xy

def   c0, -0.000977f, 0.0f, 0.0f,
0.0f
def   c1, 0.999f, 0.999f, 0.999f,
1.0f
add   t0, t0, c0
texld r0, t0, s0
mul   r0, r0, c1
mov   oC0, r0
```

= (-1.0f, 1.0f) in der oberen linken Ecke, und (1.0f, -1.0f) in der unteren rechten Ecke des Bildschirms. Die Texturkoordinate gibt die Position des Texels am Vertex an. Bei Texturen ist (0.0f, 0.0f) oben links und (1.0f, 1.0f) unten rechts. Im Prinzip wird hier bloß festgelegt, dass die Position des n-ten Texels mit der Position des n-ten Pixels übereinstimmt.

Der VertexShader leitet einfach nur die Daten des VertexBuffers weiter. Er wird für jeden Vertex einmal aufgerufen. Die Texturkoordinaten werden automatisch für jeden Pixel innerhalb des Dreieckes interpoliert. Dabei gilt, je näher sich der Pixel am Vertex befindet, desto größer ist der Einfluss des Vertex.
Diese interpolierten Texturkoordinaten erhält der PixelShader über das Register t0. Bevor jedoch die Textur mit dem Befehl „texld“ ausgelesen wird, wird noch mit dem „add“-Befehl die Konstante c0, die oben definiert ist, zu t0 hinzuaddiert. Dadurch zeigt t0 auf *einen* Texel weiter links, denn -0.000977 ist gleich 1/(-1024). 1024 ist die Breite der Textur. Dadurch bewegt sich das Bild nach rechts. Übrigens sind die heutigen GPUs auf 32Bit Fließkommazahlen (floats) optimiert. Das war nicht leider nicht immer so und dieses Problem hat mich mal wochenlang verfolgt…
Na jedenfalls haben wir jetzt den Texel durch texld ausgelesen, der nun im temporären Register r0 wartet. Als nächstes werden rot-, grün-, und blau-Anteil mit 0.999 multipliziert damit das Bild langsam immer dunkler wird. Wie man die Farbanteile interpretiert bleibt natürlich dem Programmierer überlassen. Und das war es auch schon.

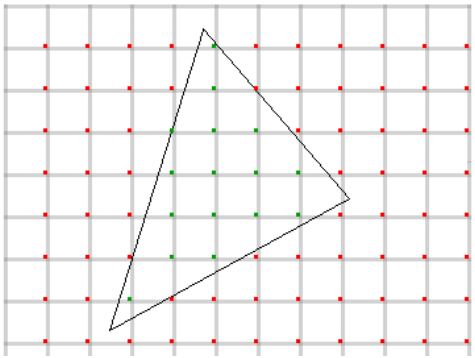


Abb. 3.4: Irgendein Dreieck, das vom VertexShader kommt, und vom Rasterisierer in Pixel aufgelöst wird.

Als ich dieses Programm auf meiner NVidia Geforce 8800GTX gestartet hatte, staunte ich nicht schlecht. Es lief mit über 3300 Iterationen pro Sekunde, bei 786432 Pixeln! Einfach unglaublich. Bestimmt 500 Mal schneller als hochoptimierter CPU-Code! Von diesem Augenblick an war klar: Dieses Thema mache ich zu einer Jugend forscht Arbeit.
Doch wie kommt diese Rechenpower zustande? Dazu vergleiche ich einmal die Konzepte von CPU und GPU. In der CPU läuft nur ein Prozessor. Jedesmal wenn eine Variable aus dem Speicher geholt werden soll, bekommt er sie sofort und kann direkt weiterarbeiten. Dabei ist Arbeitsspeicher heutzutage sehr viel langsamer als ein Prozessor. Der Prozessor muss nur deshalb nicht ewig auf seine Variable warten, weil ein intelligenter Mechanismus in der CPU die Variable schon längst vorsichtshalber in den schnellen Cache-Speicher direkt auf

der CPU geladen hat. Dieses Cache Verfahren ist heute so wichtig, dass die Hälfte der Chipfläche nur für Cache-Speicher verwendet wird.
Bei der GPU ist der Cache nur sehr klein. Dafür arbeiten auch 128 Prozessoren parallel an z.B. einer Million Pixeln. Wenn ein Pixel gerade auf einen Texel warten muss, dann macht der Prozessor erst einmal an einem anderen Pixel weiter, solange bis der Texel bereit steht. Also sollte es ziemlich unwahrscheinlich sein, dass ein Prozessor warten muss. Allerdings darf man nicht vergessen, dass alle Prozessoren den gleichen Shader abarbeiten, und somit auch zur selben Zeit auf die texld Anweisung stoßen. Kurz gesagt sollte man sich jeden Speicherzugriff gut überlegen, um keine Performance zu verschenken. Ich habe schon Situationen erlebt, wo es schneller war, acht arithmetische Anweisungen auszuführen als eine texld Anweisung. Aber das ist nicht immer so. Es hat sich jedenfalls immer gelohnt an den GPU-Programmen zu feilen, solange bis sie mit maximaler Geschwindigkeit liefen.
Angesteuert habe ich die Grafikkarte mit Microsoft DirectX 9, dem Standard für 3D Spiele. Den Quelltext für die Rahmenanwendung hierfür spare ich mir mal, sonst wäre der Bericht hier schon zu Ende.
So, damit wäre das Grundgerüst fertiggestellt. Jetzt kann ich es endlich arbeiten lassen.

# 4. Wellensimulation

Als erste Anwendung habe ich eine Simulation für Wellen entwickelt. Wellen sind nämlich etwas sehr Elementares in der Physik. Egal ob Licht-, Wasser- oder Schallwellen es ist dieses eine Prinzip das für typische Welleneigenschaften wie Beugung und Interferenz zuständig ist, nämlich das Huygenssche Prinzip: „Jeder Punkt einer Wellenfront kann als Ausgangspunkt einer Elementarwelle angesehen werden, die sich mit gleicher Geschwindigkeit und Wellenlänge wie die ursprüngliche Welle ausbreitet“. Wenn sich also auf einer Wellenfront lauter Kreiswellen ausbilden, dann ergeben diese zusammen die nächste Wellenfront. Für mich als Programmierer bedeutet das: ich brauche ein Netz aus gekoppelten Oszillatoren, die alle gleich behandelt werden. Damit ist das Huygenssche Prinzip automatisch erfüllt, denn ich sage der Simulation ja nicht explizit: „Hier ist ein punktförmiger Schwinger also erzeuge eine Kreiswelle“, sondern die Kreiswelle entsteht von sich aus. Das heißt ich verwende keine Idealisierungen wie sie normalerweise üblich sind, vielmehr kann ich es mir dank der Grafikkarte leisten, realistische Physik zu betreiben. Da zeigen sich dann die Details. Zum Beispiel zeigt die Simulation, dass der Laser nicht nur in eine Richtung strahlt, sondern infolge der Objektivbeugung in alle Raumrichtungen. Besonders verblüffend fand ich auch das Phänomen, der Interferenz an der Ecke. Man hat einen Einzelspalt und nimmt eine Seite des Spaltes weg. Die Interferenz bleibt. Das steht in keiner Formelsammlung, weil es nicht so einfach zu beschreiben ist. Auch interessant ist was *vor* dem Doppelspalt passiert. All das ist nur Huygenssches Prinzip und gekoppelte Oszillatoren. Doch wie sieht diese Koppelung aus? Dazu habe ich mir einmal die Wellenmaschine in unserer Schule angesehen:

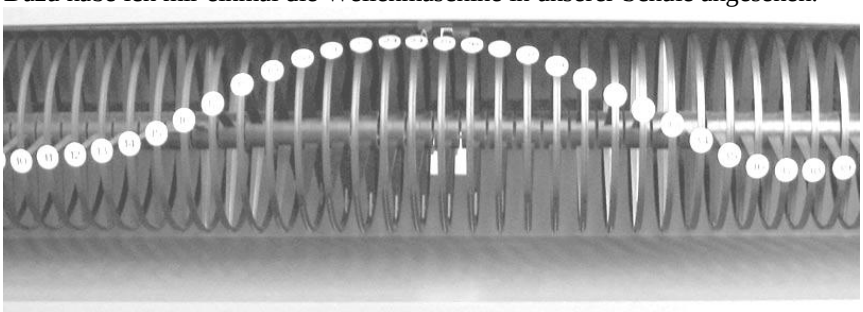
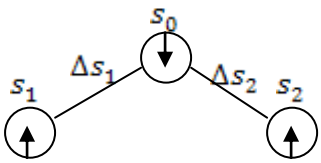


Abb. 4.1: Die Wellenmaschine

Genau das gleiche wollte ich auch machen, nur mit mehr Oszillatoren und in 2D. Zwischen den Oszillatoren der Wellenmaschine sind Federn eingebaut. Das heißt der Ansatz war die Federgleichung:

$$F = -D \cdot \Delta s$$

Dieses Delta ist die Differenz der Auslenkung von zwei benachbarten Oszillatoren. Aber weil es ja zwei Nachbarn gibt, existiert $\Delta s_1$ und $\Delta s_2$.




$$\Delta s_1 = s_0 - s_1$$

$$\Delta s_2 = s_2 - s_0$$

Gesamt Delta s ist somit $\Delta s = \Delta s_2 - \Delta s_1 = (s_2 - s_0) - (s_0 - s_1) = s_2 + s_1 - 2 \cdot s_0$

Ich ziehe die Differenzen voneinander an, weil die beiden Federn in entgegengesetzte Richtungen ziehen. Differenzen voneinander abziehen bedeutet, Steigungen voneinander abziehen, was einer zweifachen Ortsableitung entspricht. $\Delta s$ ist also nichts anderes als die zweitfache Ortsableitung von $s$. So gesehen wird aus dem Delta der Laplace-Operator, der genauso aussieht und für zwei Dimensionen wie folgt definiert ist:

$$\Delta = \nabla^2 = \begin{pmatrix} \frac{d}{dx} \\ \frac{d}{dy} \end{pmatrix}^2 = \frac{d^2}{dx^2} + \frac{d^2}{dy^2}$$

Wobei $\nabla$ der Nabla-Operator ist. Wenn man also das $\Delta$ in der Federgleichung als Laplace-Operator interpretiert, hat man eine Gleichung, die die Kraft auf einen Oszillator in einem 2D-Netz beschreibt. Genau das was ich brauche. Ersetz man $F$ durch $m \cdot \ddot{s}$ erhält man eine partielle Differenzialgleichung:

$$m \cdot \ddot{s} = -D \cdot \Delta s$$

Diese DGL numerisch zu lösen ist die Aufgabe meiner Simulation. Erst einmal umstellen:

$$\ddot{s} = -\frac{D}{m} \cdot \Delta s$$

Jetzt unterteile ich die Zeit in diskrete Zeitschritte, auch Frames genannt. Nach jedem Frame wird die Auslenkung jedes Oszillators anhand seiner Geschwindigkeit und Beschleunigung aktualisiert. Meine Zeit-Integrationsmethode ist folgende:

$$\ddot{s}(t) = -\frac{D}{m} \cdot \Delta s(t-1)$$
$$\dot{s}(t) = \dot{s}(t-1) + \ddot{s}(t) \cdot \Delta t$$
$$s(t) = s(t-1) + \dot{s}(t) \cdot \Delta t$$

Dies ist ein explizites Leap-Frog Verfahren. Es unterscheidet sich vom einfachen Euler-Verfahren, da in der dritten Gleichung schon der Mittelwert der Geschwindigkeit des aktuellen und des vorherigen Frames verwendet wird. Dieses Verfahren ist bis zu einem bestimmten $\Delta t$ stabil und von 2.Ordnung in der Genauigkeit. Im Grunde ist das nichts anderes als eine Taylor-Reihen-Approximation.

Ich muss also immer Geschwindigkeit und Auslenkung der Oszillatoren speichern. Dazu benutze ich eine Textur, die nur rote und grüne Anteile hat. Rot ist die Auslenkung, Grün die Geschwindigkeit. Insgesamt verwende ich aber zwei Texturen. Die eine Textur, *pTextureData*, enthält alle Oszillatoren des letzten Frames. Mit dieser Textur berechnet der PixelShader den aktuellen Frame und speichert ihn in die Textur *pTextureRT*. „RT“ steht für „RenderTarget“, also das Ziel des Zeichenvorgangs. Diese Textur wird dann am Bildschirm angezeigt. Im darauffolgenden Frame werden die Texturen getauscht. Dabei wird nicht wirklich der Inhalt kopiert, es ist viel schneller nur die Zeiger der Texturen zu tauschen. Bevor ich zum PixelShader komme, muss noch geklärt werden wie der 2D Laplace-Operator angewendet wird. In einer Dimension galt: $\Delta s = s_1 + s_2 - 2 \cdot s_0$. Bei zwei Dimensionen ist es ganz analog:

| | | |
|---|---|---|
| 1 | -2 | 1 |

Abb.4.2: 1D-Laplace

| | | |
|---|---|---|
| | 1 | |
| 1 | -4 | 1 |
| | 1 | |

Abb. 4.3: 2D-Laplace. Diagonale Nachbarn sind jedoch null.

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 2 | -12 | 2 |
| 1 | 2 | 1 |

Abb. 4.4: 2D-Laplace mit diagonalen Nachbarn.

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 2 | -12 | 2 |
| 1 | 2 | 1 |

Abb. 4.5: diagonaler 2D-Laplace mit nur fünf Textur-koordinaten.

Zuerst habe ich die naheliegende mittlere Form verwendet (Abb. 4.3). Es zeigte sich jedoch, dass Kreiswellen damit ein klein wenig eckig aussehen. Also habe ich mir die rechte Variante einfallen lassen, die gleichmäßiger verteilt und somit isotroper ist. Jetzt waren die Kreiswellen genau rund. Leider lief die Simulation damit aber nur etwa halb so schnell. Das leuchtet ein, denn ich brauche ja jetzt neun texld Anweisungen, anstelle von fünf. Aber dann ist mir eine Idee gekommen wie ich trotzdem nur fünf texld benötige. Ich nutze nämlich eine Spezialfähigkeit der GPU aus, und zwar die

„bilineare Filterung“. Was passiert wenn die Texturkoordinate, die der PixelShader erhält, nicht genau in die Mitte eines Texels zeigt, sondern irgendwie zwischen die Texel? Dann führt die GPU eine lineare Interpolation mit den vier umliegenden Texeln durch. Je näher die Koordinate an einem Texel ist, desto größer ist dessen Einfluss. Die Idee ist, die Texturkoordinaten wie in Abb. 4.5 zu platzieren.
Jeder Äußerer Abtastungspunkt liefert genau den Durchschnitt aus seinen vier umliegenden Texeln. Nun addiere ich die äußeren Werte und ziehe vier Mal die Mitte ab. Was ich erhalte ist genau der diagonale Laplace-Operator. Dazu musste ich den PixelShader gar nicht mehr umschreiben, ich konnte einfach den für die mittlere Variante benutzen, da sich nur die Texturkoordinaten ändern, und die liegen im VertexBuffer. Das Gute an der bilinearen Filterung ist, dass sie absolut nichts kostet. Ob man sie ein- oder ausschaltet ändert an der Performance nichts. Es ist eben eine Spezialfähigkeit der GPU. Und damit ist die Theorie schon fast abgeschlossen, fehlt nur noch der alles entscheidende PixelShader:

```
texld r0, t0, s0        // mitte
texld r1, t1, s0        // links oben
texld r2, t2, s0        // rechts oben
texld r3, t3, s0        // links unten
texld r4, t4, s0        // rechts unten

def   c2, -4.0f, 0.0f, 0.0f, 0.0f
add   r1.r, r1, r2
add   r1.r, r1, r3
add   r1.r, r1, r4
mad   r2.r, r0, c2, r1  //LP =-4.0f * M + LO + RO + LU + RU

mad   r0.g, r2.r, c1, r0.g    // v = v + LP * d/m * dt
mad   r0.r, r0.g, c0, r0.r    // s = s + v * dt
mov   oC0, r0
```

„mad“ steht für „multiply and add“. Die Argumente zwei und drei werden multipliziert und das Ergebnis mit dem vierten Argument addiert. Zielregister ist immer das erste Argument. cX sind Konstanten-Register, die wie c2 im PixelShader selbst, oder wie c0 und c1 im C++ Programm definiert werden können. oC0 ist die Farbe des Pixels. Alle Register bestehen aus vier float-Zahlen, nämlich rot, grün und blau plus einen Wert für die Deckkraft der Farbe.
Eigentlich nur eine winzige Routine, jedoch mit beeindruckenden Resultaten:
Das Ganze läuft mit 786432 Oszillatoren bei über 2600 Frames pro Sekunde. Neben Kreiswellenemittern kann

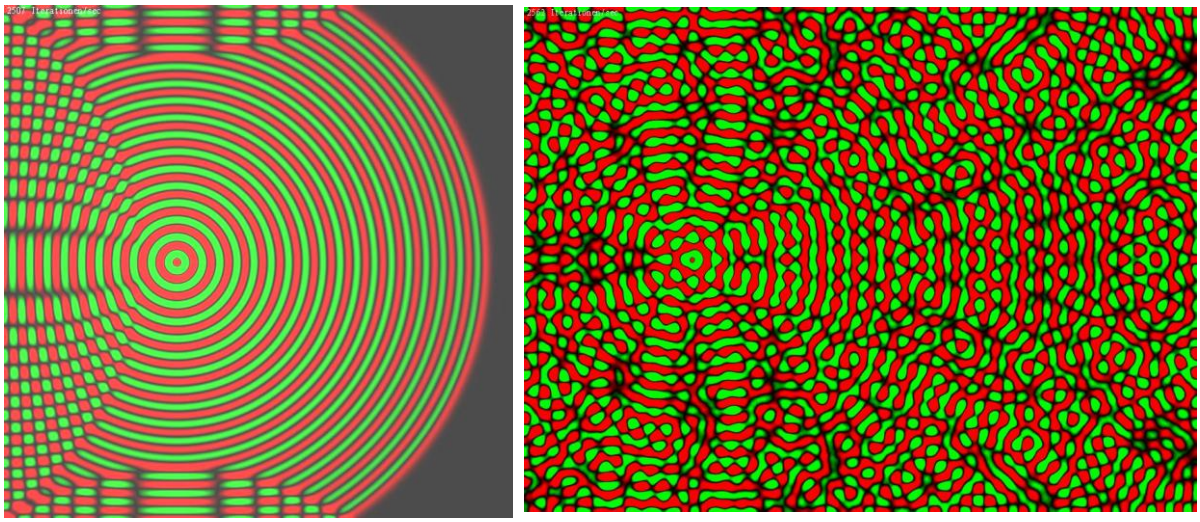


Abb. 4.6: Ein Kreiswellenemitter. Nach einiger Zeit herrscht nur noch Wellenchaos aus stehenden Wellen, das aber doch noch eine interessante Ordnung zeigt. Der Rand reflektiert die Wellen.

der Benutzer auch noch viele andere Objekte hinzufügen und ganz einfach selbst welche erstellen. Mit der grafischen Benutzeroberfläche können beliebige Formen hinzugefügt werden. Der Komplexität des Experimentes sind dabei keine Grenzen gesetzt.

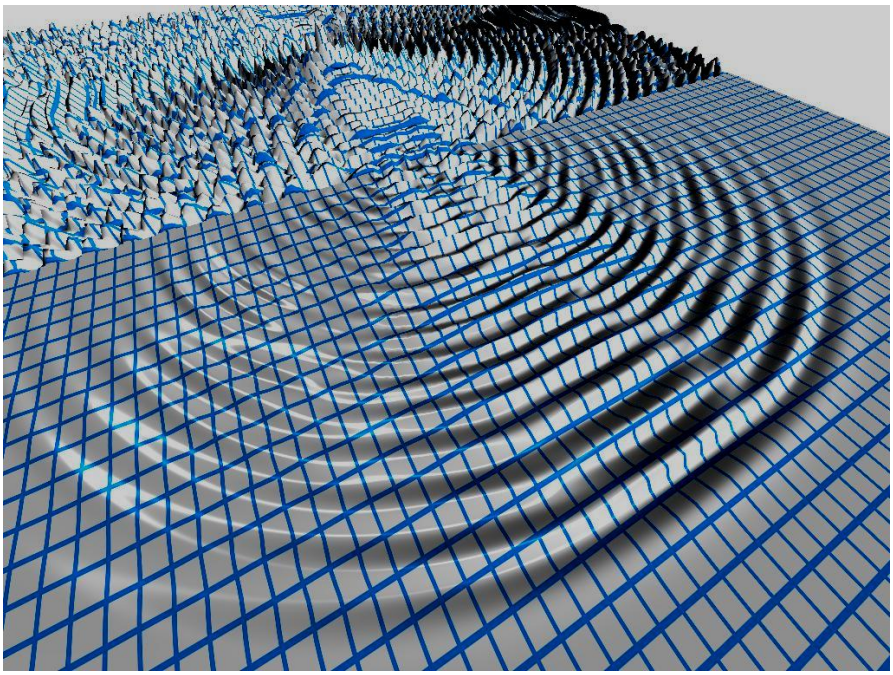


Abb. 4.9: Die Welle kann auch dreidimensional angezeigt werden

Die 3D-Ansicht war etwas aufwendiger zu programmieren, vor allem wegen der Lichteffekte.
Es lassen sich auch Dämpfer an den Rändern einzufügen, damit nicht immer so ein Wellenchaos entsteht. Übrigens kann man die Wellen nicht einfach über den Rand hinweg laufen lassen, weil man dafür eine unendlich große Fläche simulieren müsste. Ich hab die Simulation mal auf der CPU implementiert. Als dann nach 10sek immer noch kein Bild kam, hab ich abgebrochen.
So, das war der Wellensimulator, der zu Hause, in der Schule, im Studium oder sonst irgendwo eingesetzt werden

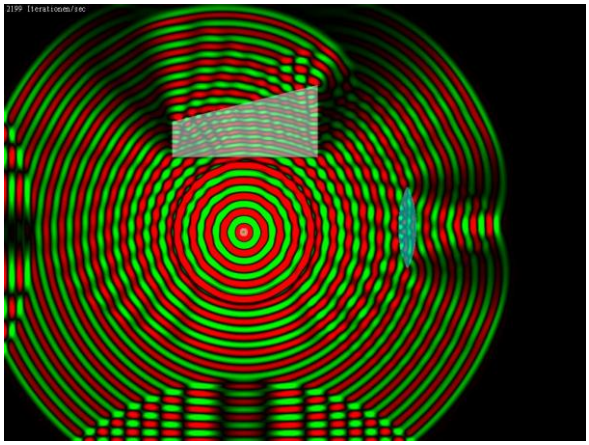


Abb. 4.7: Brechung an Prisma und Linse

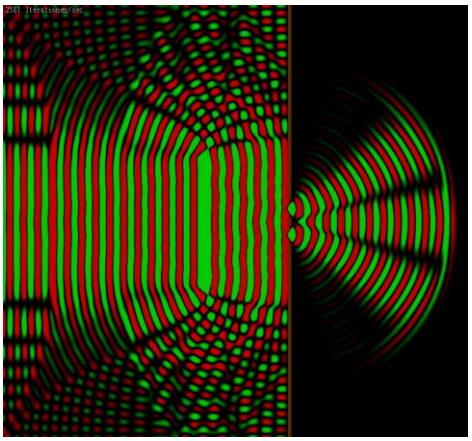


Abb. 4.8: Doppelspaltversuch

kann. Erstaunlich ist, dass sogar physikalische Spezialeffekte wie das optische Nahfeld korrekt simuliert werden. Über diesen Effekt wurde eine ganze Jugend-forscht Arbeit geschrieben (http://stmg.de/ags/jufo/2006/2006.htm). Und das alles nur mit ganz einfacher Mathematik und Huygens.
Bevor ich den Wellensimulator fertig hatte, dachte ich immer ich wäre der einzige, der auf die Idee gekommen ist, Grafikkarten für Simulationen zu verwenden. Der Anstoß war ein Physikreferat. Dann musste ich feststellen, dass seit Anfang 2007, NVidia, AMD und einige Teams in den USA fieberhaft daran arbeiten, ihre Algorithmen auf die GPU umzustellen (Stichwort: folding@home). Das war natürlich sehr ärgerlich. Einen Wellensimulator habe ich aber noch nicht gesehen. Und auch sonst ist die GPU (noch) eine ganz kleine Nische in der Wissenschaft.

# 5. Strömungssimulation

Strömungen sind ein interessantes Forschungsfeld. Sie sind wichtig bei der Wettervorhersage, dem Klimawandel und natürlich der Aerodynamik. Also etwa beim Bau von Autos und Flugzeugen. Überall sind Wirbel, die leider sehr aufwendig zu berechnen sind. Deswegen hat z.B. jedes Formel 1 Team einen Großrechner im Rennstall stehen. Also eine passende Aufgabe für die Grafikkarte, zumal man das Fluid (Sammelbegriff für Gas und Flüssigkeit) sehr gut in einzelne Zellen unterteilen kann. Das werden dann nachher die Texel einer Textur sein. Um Fluide zu beschreiben gibt es eine Standardmethode, die sogenannten „Navier-Stokes Gleichungen“. Da diese recht kompliziert sind, benutzt man unterhalb der Schallgeschwindigkeit immer die inkompressiblen Navier-Stokes Gleichungen:

$$\frac{d\vec{u}}{dt} = -(\vec{u} \cdot \nabla) \cdot \vec{u} + \mu \cdot \Delta\vec{u} + \vec{f}$$

$$\nabla \cdot \vec{u} = 0$$

Die zweite Gleichung verursacht am meisten Rechenzeit, obwohl sie so kurz ist. Sie zwingt das Fluid dazu, überall den gleichen Druck zu haben. Daher auch „inkompressibel“. Der Vektor $\vec{u}$ ist das Geschwindigkeitsfeld des Fluids. Die Änderung nach der Zeit wird durch drei Terme bestimmt. Von rechts nach links: $\vec{f}$ ist eine Quelle, die die Strömung verursacht. Wenn ich mit dem Löffel den Kaffee umrühre, ist $\vec{f}$ der Löffel. $\mu \cdot \Delta\vec{u}$ ist der Diffusionsterm. Die Diffusion wird durch die Wechselwirkungen zwischen den Molekülen des Fluids hervorgerufen: wenn sich in der Mitte das Fluid nach vorne bewegt, reißt es links und rechts die Moleküle mit und wird dabei gebremst. Dieser Effekt ist umso stärker, je viskoser, also zäher das Material ist. μ gibt die Viskosität und damit die Art des Fluids an. Null bedeutet ideales Gas, unendlich heißt Festkörper. Hier haben wir wieder den Laplace-Operator. Und genau wie beim Wellensimulator kann man hier sagen: je mehr sich die Zelle von den Nachbarn unterscheidet, desto stärker wird sie zu den Werten der Nachbarzellen hingezogen. Auf diese Weise verteilt sich die Geschwindigkeit nach und nach im Raum. $-(\vec{u} \cdot \nabla) \cdot \vec{u}$ ist der Advektionsterm. Er sorgt dafür, dass sich das Geschwindigkeitsfeld entlang der eigenen Geschwindigkeitsvektoren bewegt. Wenn der Wind nach Norden zeigt, strömt Luft nach Norden. Aber auch der Wind selbst bewegt sich nach Norden. Das ist Advektion. Genaueres dazu später.

Für die Umsetzung unterteile ich die Zeit wieder in Frames und das Geschwindigkeitsfeld in Zellen. Jede Zelle ist ein Texel und zusammen ergeben die Texel das Geschwindigkeitsfeld. Wenn der Benutzer die Maus bewegt, erzeugt er eine Strömung. Die Mausbewegung ist dann der Vektor $\vec{f}$. Der Diffusionsterm $\mu \cdot \Delta\vec{u}$ ist nicht so einfach so einfach zu berechnen. Ich könnte ihn zwar einfach mit $\Delta t$ multiplizieren und zu $\vec{u}$ hinzuaddieren, aber dann müssten μ und $\Delta t$ sehr klein sein. Ansonsten wird die Simulation instabil. Es kann dann nämlich leicht dazu kommen, dass die Änderung von $\vec{u}$, größer als $\vec{u}$ selbst ist. $\vec{u}$ würde dann negativ werden und das wäre völlig unrealistisch. Ein anderer Ansatz muss her. In der Fachliteratur habe ich dazu gelesen, dass man bei der Diffusion, egal ob Strömungssimulation oder nicht, das implizite Eulerverfahren nimmt. Und das geht so: Man addiert nicht die aktuelle Änderung von $\vec{u}$, sondern die des nächsten Frames.

$$\vec{u}(t+1) = \vec{u}(t) + \frac{d\vec{u}(t+1)}{dt} \cdot \Delta t \Rightarrow \vec{u}(t+1) = \vec{u}(t) + \mu \cdot \Delta\vec{u}(t+1) \cdot \Delta t$$

Nach $\vec{u}(t)$ umgestellt: $\vec{u}(t+1) - \mu \cdot \Delta\vec{u}(t+1) \cdot \Delta t = \vec{u}(t)$

$\vec{u}(t+1)$ ausgeklammert: $(I - \mu \cdot \Delta t \cdot \Delta) \cdot \vec{u}(t+1) = \vec{u}(t)$

I ist die Einheitsmatrix. Was hier steht ist ein lineares Gleichungssystem mit so vielen Gleichungen wie Zellen. Wäre da kein Laplace, wäre die Einheitsmatrix 1 und man bräuchte nur nach $\vec{u}(t+1)$ umstellen. So aber muss man das Gleichungssystem lösen. Wäre es nur Laplace, würde sich folgendes LGS ergeben:

$$\Delta \cdot u = b$$

$$\begin{bmatrix}
4 & -1 & & & -1 & & & & & & & & & & & \\
-1 & 4 & -1 & & & -1 & & & & & & & & & & \\
 & -1 & 4 & -1 & & & -1 & & & & & & & & & \\
 & & -1 & 4 & & & & -1 & & & & & & & & \\
-1 & & & & 4 & -1 & & & -1 & & & & & & & \\
 & -1 & & & -1 & 4 & -1 & & & -1 & & & & & & \\
 & & -1 & & & -1 & 4 & -1 & & & -1 & & & & & \\
 & & & -1 & & & -1 & 4 & & & & -1 & & & & \\
 & & & & -1 & & & & 4 & -1 & & & -1 & & & \\
 & & & & & -1 & & & -1 & 4 & -1 & & & -1 & & \\
 & & & & & & -1 & & & -1 & 4 & -1 & & & -1 & \\
 & & & & & & & -1 & & & & 4 & -1 & & & -1 \\
 & & & & & & & & -1 & & & & 4 & -1 & & \\
 & & & & & & & & & -1 & & & -1 & 4 & -1 & \\
 & & & & & & & & & & -1 & & & -1 & 4 & -1 \\
 & & & & & & & & & & & -1 & & & -1 & 4
\end{bmatrix} * \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{4,1} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \\ u_{4,2} \\ u_{1,3} \\ u_{2,3} \\ u_{3,3} \\ u_{4,3} \\ u_{1,4} \\ u_{2,4} \\ u_{3,4} \\ u_{4,4} \end{bmatrix} = \begin{bmatrix} b_{1,1} \\ b_{2,1} \\ b_{3,1} \\ b_{4,1} \\ b_{1,2} \\ b_{2,2} \\ b_{3,2} \\ b_{4,2} \\ b_{1,3} \\ b_{2,3} \\ b_{3,3} \\ b_{4,3} \\ b_{1,4} \\ b_{2,4} \\ b_{3,4} \\ b_{4,4} \end{bmatrix}$$

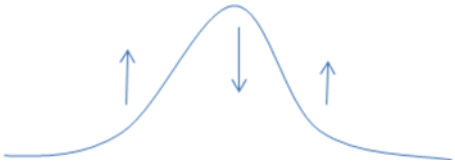
Das kommt bekannt vor. Wir wollen u haben und gegeben ist b. Dieses LGS habe ich auf ganz eigene Weise gelöst. Ausgangspunkt ist das Jakobi-Verfahren, weil die Matrix nur sehr dünn und diagonal besetzt ist. Laut Jakobi-Verfahren haben wir für jedes u folgende Zeile:

$$u_1 = \frac{u_0 - (a_1 + b_1 + c_1 + d_1) \cdot \mu \cdot \Delta t}{1 - 4\mu \cdot \Delta t}$$

Der Frame steht im Index. a-d sind die Nachbarzellen. Laut Jabobi-Verfahren muss diese Iteration mindestens 15 Mal pro Frame wiederholt werden. Mein Trick ist jetzt, dass ich statt $(a_1 + b_1 + c_1 + d_1)$, einfach $(a_0 + b_0 + c_0 + d_0)$ schreibe, also insgesamt:

$$u_1 = \frac{u_0 - (a_0 + b_0 + c_0 + d_0) \cdot \mu \cdot \Delta t}{1 - 4\mu \cdot \Delta t}$$

Jetzt brauche ich keine aufwendigen Iterationen mehr, weil nur bekannte Werte vorkommen. Erst mal habe ich das einfach ausprobiert und festgestellt, dass es ausgezeichnet funktioniert. Und dann habe ich überlegt, warum das so ist. Setzt man probeweise irgendwelche Werte ein, stellt man schnell fest, dass das Ergebnis $u_1$ nahe am Durchschnitt der Nachbarzellen liegt. Und zwar umso näher, je größer das Produkt $\mu \cdot \Delta t$ ist. Ist µ null, tut sich gar nichts, ist es unendlich, legt sich die Zelle genau auf den Durchschnitt der Nachbarzellen. Und somit diffundiert u mit maximaler Geschwindigkeit.



Damit hatte ich ein extrem schnelles, für alle $\Delta t$ stabiles, implizites Zeitintegrationsverfahren gefunden.

Damit ist die Diffusion abgehakt, bleibt noch die Advektion. Auch dabei ist es keine gute Idee, direkt und explizit zu integrieren, weil man sehr schnell in Instabilitäten hineingerät. Hier stammt die Lösung nicht von mir, sondern von Jos Stam, der 1997 mit seinem „Stable Fluids“ Algorithmus einen neuen Standard gesetzt hat. Die Idee ist so einfach wie genial: Advektion heißt, dass sich das Feld entlang der Geschwindigkeitsvektoren bewegt. Ich betrachte eine Zelle und schaue wo $\vec{u}$ hinzeigt. Dort wird der Inhalt der Zelle im nächsten Frame sein. Aber was wird hier in der Zelle im nächsten Frame sein? Antwort: Das was jetzt da ist wo das umgedrehte $\vec{u}$ hinzeigt! Das umgedrehte $\vec{u}$ wird aber kaum genau in die Mitte einer Zelle zeigen, sondern immer irgendwo zwischen vier Zellen. Also wieder ein perfekter Job für den biliniearen Filter. Es ist wirklich ganz simpel und kann mit zwei texld Anweisungen erledigt werden. Das erste texld liest $\vec{u}$ aus, dann wird $\vec{u}$ umgekehrt, und dort wo es hinzeigt liest das zweite texld $\vec{u'}$ aus. Dieses $\vec{u'}$ ist dann die Farbe des Pixels und damit das neue $\vec{u}$. Die Grafikkarte scheint wirklich wie gemacht zu sein für Strömungssimulationen.

Die erste Navier-Stokes Gleichung ist nun implementiert. Als ich das bis hier hin alles programmiert hatte, sah das zwar ganz nett aus, aber ich habe eine sehr wichtige Eigenschaft von Strömungen vermisst: die Wirbel. Sie entstehen erst wenn man auch die zweite Gleichung mit einbezieht. Sie sagt, dass das Geschwindigkeitsfeld divergenzfrei sein muss. Mit anderen Worten: In jede Zelle strömt immer so viel Fluid rein wie raus, d.h. der Druck bleibt konstant. Den Ansatz, den ich hier zu gefunden habe, ist die Helmholtz-Hodge-Dekomposition. Danach besteht jedes Vektorfeld aus einem divergenzfreien Feld und dem Gradiendenfeld des Druckes:

$$\vec{u} = \vec{w} + \nabla p$$

p ist das skalare Druckfeld, $\vec{w}$ das divergenzfreie Feld. Der Druck ist hier wörtlich zu verstehen. Die Gleichung mit $\nabla$ multipliziert ergibt:

$$\nabla \cdot \vec{u} = \nabla \cdot \vec{w} + \nabla^2 p \quad \text{da} \quad \nabla \cdot \vec{w} = 0 \quad \Rightarrow \quad \nabla \cdot \vec{u} = \nabla^2 p \quad \text{Poisson-Gleichung}$$

Diese Poission-Gleichung muss ich lösen um p zu erhalten. Dazu muss wieder ein LGS gelöst werden. Ich benutze dafür das Jakobi-Verfahren, mit 15 Iterationen (diesmal gibt es keine Tricks). Selbstverständlich wieder alles auf der GPU. Habe ich dann p, bilde ich den Gradienden und ziehe ihn von $\vec{u}$ ab. Und dann ist $\vec{u}$ divergenzfrei. Insgesamt benötige ich dafür drei PixelShader. Damit hätten wir auch die Wirbel.

Das Ergebnis ist in Abb. 5.1 zu sehen. Die Striche sind Ventilatoren, die der Benutzer mit der Maus setzen kann. Um das Fluid sichtbar zu machen, habe ich Rauch hinzugefügt. Dieser Rauch bewegt sich entlang des Geschwindigkeitsfeldes und diffundiert.

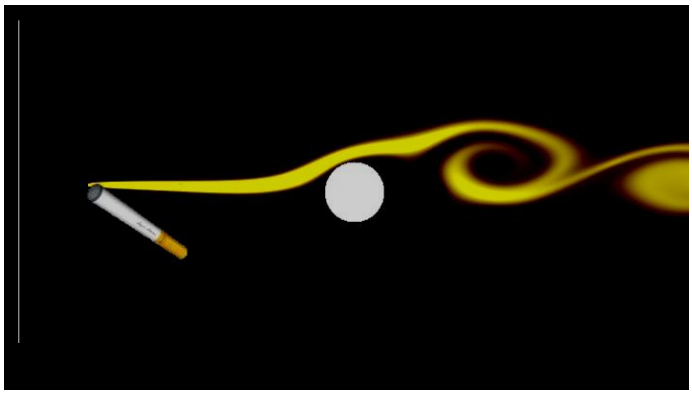


Abb. 5.1 Wirbelbildung hinter einer Kugel

Interessant wird es, wenn man Hindernisse hinzufügt. Dazu muss man die Randbedingungen bei einem Hindernis festlegen:

$\frac{dp}{d\vec{n}} = 0$ sowie $\vec{u}_{Hindernis} = 0$

$\vec{n}$ ist der Normalenvektor der Hindernisoberfläche. D.h. es darf keinen Druckunterschied zwischen innen und außen vom Hindernis geben, damit kein Fluid einströmt. Diese Bedingung muss man in jeder Jakobi-Iteration der Poisson-Gleichung neu durchsetzen. Außerdem ist die Geschwindigkeit im Hindernis immer null.
Die Simulation läuft insgesamt mit gut 500 Frames pro Sekunde. Den Quellcode kann ich aus Platzgründen leider nicht mehr abdrucken.
Mit dem Wellensimulator und der Strömungssache habe ich mich letztes Jahr am Forschungszentrum in Jülich vorgestellt. Dort steht einer der schnellsten Supercomputer der Welt. Da waren genau die richtigen Experten, um meine Simulationen zu beurteilen. Besonders beeindruckt waren sie von der Geschwindigkeit und der Interaktivität. Auch haben sie mir einige mathematische Methoden gezeigt, mit denen ich meine Simulationen noch verbessern kann. Ich konnte mich völlig frei mit den Fachleuten austauschen und habe viele Ideen für die Zukunft gesammelt. An dieser Stelle noch einmal herzlichen Dank an Annika Schiller, Dr. Paul Gibbon, Dr. Jan Meinke, Dr. Bernhard Steffen, Dr. Godehard Sutmann sowie Dr. Rüdiger Esser und meinen Betreuer Walter Stein, die das Treffen möglich gemacht haben.

# 6. N-Body Systeme

N-Body Systeme bestehen aus vielen Teilchen, die miteinander Wechselwirken. Das können Planeten, Sterne, Galaxien sein, oder auch Atome und Moleküle. Beide Extreme habe ich einmal simuliert. Mit den Sternen ging es los, da die Gravitation sehr einfach zu berechnen ist:
$\vec{r}$ ist der Abstand zweier Sterne mit den Massen $m_1$ und $m_2$. Bei mehr als zwei Sternen muss man die Formel

$$\vec{F}_G = f \cdot \frac{m_1 \cdot m_2}{\vec{r}^{\,2}} \cdot \frac{\vec{r}}{|\vec{r}|}$$

mehrmals anwenden, sodass jeder Stern mit jedem anderen Stern einmal wechselwirkt. Nur wie will man das mit Texturen und PixelShader umsetzen? Ich müsste für jeden Texel die Texturkoordinaten der anderen Sterne im Shader berechnen, und eben so viele texld Anweisungen einsetzten. Das ist jedoch völlig undenkbar. Also ist das Projekt gestorben? Nein, aber ich muss mich von DirectX trennen. Zum Glück gibt es seit dem Frühjahr 2007 eine Alternative namens NVidia CUDA. NVidia ist einer der beiden führenden GPU-Hersteller und hat das Potenzial seiner Hardware erkannt. Mit CUDA kann man die GPU in C ansprechen. Es gibt auch keine Shader, Pixel oder Texturen (im klassischen Sinne) mehr, sondern nur noch Routinen, die die GPU abarbeitet. Man kommt viel direkter an die Hardware und Videospeicher heran und hat viel mehr Einfluss und vor allem Optimierungsmöglichkeiten. Zwar sinkt bei CUDA der Aufwand für die Rahmenanwendung, aber es ist auch um einiges komplizierter zu erlernen als DirectX. Deswegen und weil der Platz nicht reicht, kann ich hier nicht näher auf die Software eingehen. Auf dem Wettbewerb hole ich das aber gerne auf Wunsch nach. Allerdings hat CUDA auch Nachteile. Es kann nur sehr eingeschränkt mit DirectX kommunizieren (was für die Bildschirmwiedergabe aber nötig ist) und kann keine Werte in Texturen speichern. Aus diesen und ähnlichen Gründen, würde ich immer noch wenn möglich auf DirectX zurückgreifen.
Aber zurück zur Simulation. Ich simuliere mit CUDA 512 Sterne, die alle direkt miteinander in 2D wechselwirken. Am Anfang hat jeder Stern eine zufällige Masse und Geschwindigkeit. Da der Drehimpuls aber im Mittel null ist, kollabiert das ganze System sehr schnell. Einige Sterne verlieren Impuls und fallen in den Mittelpunkt, während Andere Impuls gewinnen und ins interstellare Medium geschleudert werden. Was sich da in der Mitte bildet

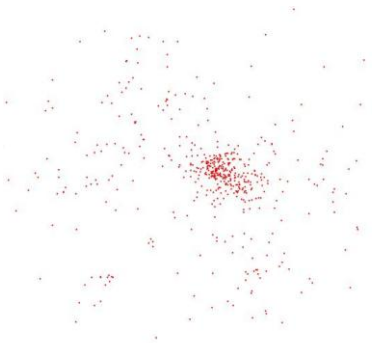
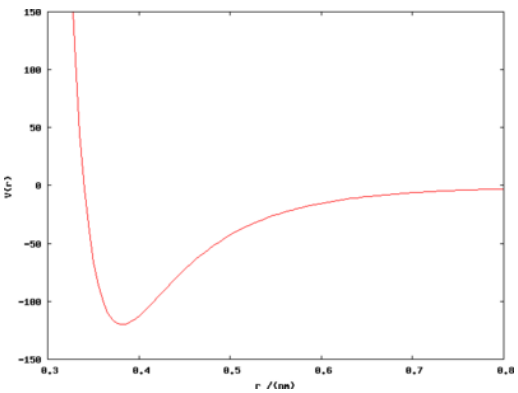


Abb. 6.1: 512 Sterne




Das Lennard-Jones Potenzial

kann als Balkengalaxie interpretiert werden. Balkengalaxien zeichnen sich nämlich dadurch aus, dass sie sich nicht drehen und eher einem Bienenschwarm ähneln. Aber warum nur 512 Sterne? Nun, das sind immerhin ca. 250.000 Wechselwirkungen. Jedoch nutze ich zurzeit nur einen von 16 Multiprozessoren auf der GPU. Und trotzdem läuft das Programm mit 1800 Frames/sek. Bei noch mehr Sternen, müssen mehrere entfernte Massen als ein Objekt betrachtet werden. Das alles zu verwalten, so dass es noch halbwegs schnell ist, ist gar nicht so einfach. An dieser Simulation arbeite ich noch.
Jetzt zu den Molekülen. Oder besser gesagt zu den Kristallen. Ich will nämlich die Struktur und das Verhalten von Kristallen simulieren, insbesondere die der Plasmakristalle. Siehe dazu die Jugend forscht Arbeit „Plasmakristalle“ von Benedikt Lorbach, Binia Neuer und Moritz Plötzing, mit der sie 2003 den zweiten Platz auf dem Bundeswettbewerb gemacht haben (www.stmg.de).
Die Moleküle in einem Kristall sollen elektrisch neutral sein, d.h. es herrscht nur die Van-der-Waalssche Bindung. Diese Wechselwirkung wird gut von einem Lennard-Jones Potenzial beschrieben. Dadurch ziehen sich weit entfernte Moleküle an, sie stoßen sich jedoch stark ab (Pauli-Prinzip), sobald sie sich zu nahe kommen. Durch diesen Mechanismus, ergibt sich ein Grundzustand für Kristalle in der Ebene, bei der Abstand zu allen Nachbarn gleich ist. Daraus resultiert das wohlbekannte Sechseckmuster. Jedoch ist kein Kristall perfekt. Auch nicht in meiner Simulation. So ein Molekül kann auch mal mehr oder weniger Nachbarn haben. Es befindet sich dann in einem höheren Energiezustand und ist instabiler. Das sind dann auch die Stellen wo der Kristall bricht, wenn ich die Temperatur erhöhe.
Im Moment bin ich dabei Wassermoleküle zu simulieren.

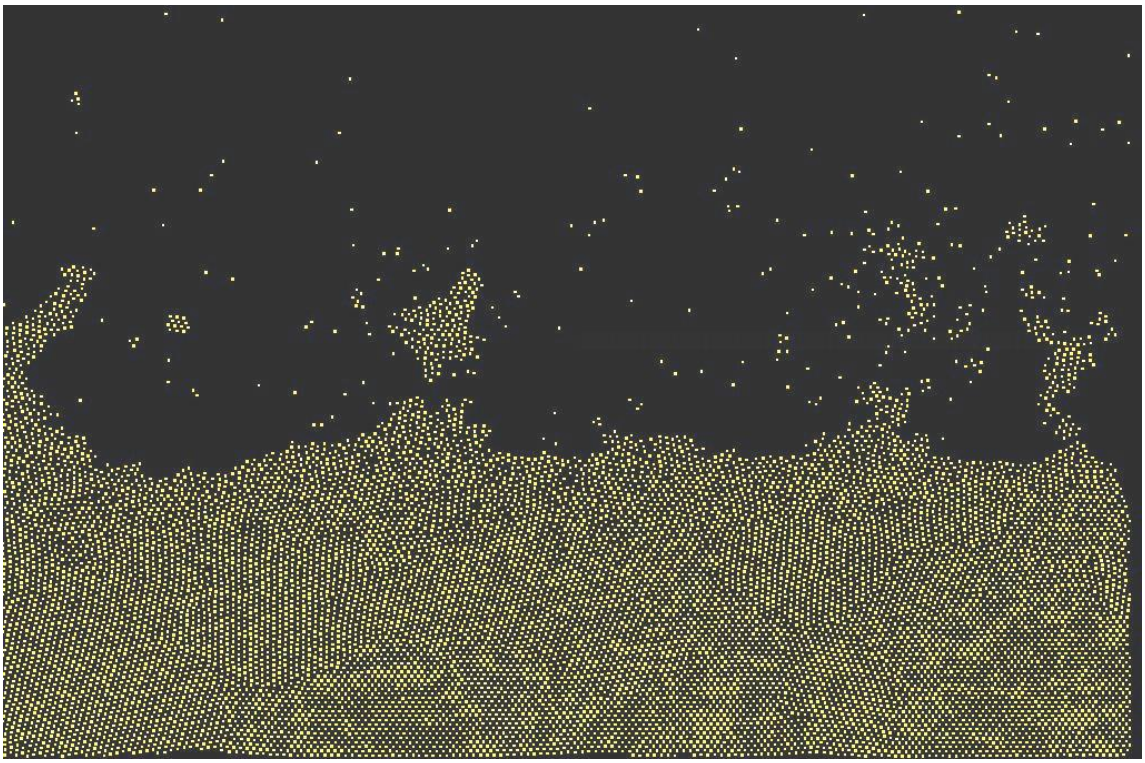


Abb. 6.2: 16000 Teilchen bilden einen Kristall. An der Grenzfläche verdunstet das Material

# 7. Schrödingergleichung

Wenn man noch genauer hinschaut treten Quanteneffekte auf. In diesem Projekt geht es darum ein quantenmechanisches System interaktiv erfassbar zu machen, damit man direkt sieht was hinter dem komplizierten Formalismus steckt.
Aber die Simulation soll noch mehr sein. Man soll auch Forschung am PC damit betreiben können.
Die Grundgleichung der Quantenphysik ist die zeitabhängige Schrödingergleichung:

$$-\frac{\hbar^2}{2m}\cdot\nabla^2\psi(\vec{r},t)+V(\vec{r})\cdot\psi(\vec{r},t)=i\hbar\frac{d}{dt}\psi(\vec{r},t)$$

Diese Gleichung beschreibt Materiewellen. Materiewellen sind Quantenobjekte aller Art: Elektronen, Photonen, Atomkerne, Nanofußbälle… Komischerweise habe ich nirgendwo auch nur den kleinsten Hinweis entdeckt, wie man das Problem der numerischen Lösung der Schrödingergleichung angeht. Also bin ich einfach Schritt für

Schritt vorgegangen (manchmal musste ich auch einige Schritte zurückgehen). Am Ende hat es dann zum Glück funktioniert.
Von vorne: Ich habe wieder eine 2D-Ebene also eine Textur, auf der sich die Materiewelle bewegt. Um überhaupt irgendetwas zu verstehen, habe ich mir mehrere Vorlesungen des dritten Semesters an der Uni Tübingen online angeschaut. Jetzt wusste ich genau Bescheid über Unschärfe, Impuls- und Ortsraum (Fourir-Transformation), komplexe Zahlen und dergleichen mehr. Als erstes hab ich die Schrödingergleichung nach $\frac{d}{dt}\psi(\vec{r},t)$ umgestellt:

$$-i\cdot\frac{\hbar}{2m}\cdot\Delta\psi(\vec{r},t)-i\frac{1}{\hbar}\cdot V(\vec{r})=\frac{d}{dt}\psi(\vec{r},t)$$

Jetzt könnte ich psi ganz normal mit $\Delta t$ integrieren. Aber zunächst brauche ich einen Anfangswert von psi, also $\psi(\vec{r},0)$. Dieser Anfangszustand ist der Zustand bei der letzten Messung. Je größer die Ortsunschärfe desto kleiner die Impulsunschärfe. Ich benutze folgenden Anfangszustand (Herleitung braucht zuviel Platz):

$$\psi(\vec{r},0)=\frac{1}{\sqrt{\sigma\cdot\sqrt{\pi}}}\cdot e^{-\frac{\vec{r}^2}{2\sigma^2}}\cdot e^{i(\vec{r}\cdot\vec{k})}$$

Sigma ist die Orts- unschärfe und $\vec{k}$ der Wellenvektor. Das Wellenpaket bewegt sich in Richtung dieses Wellenvektors. Bei der Gaußglocke ist das Produkt aus Orts- und Impulsunschärfe am geringsten, also am besten. Das Paket ist in Abb. 7.1 dargestellt. Da nicht alle Impulsvektoren, infolge der Impulsunschärfe, in dieselbe Richtung zeigen und die gleiche Länge haben, habe ich erwartet, dass das Wellenpaket zerläuft. Ich wette dafür ist wieder der Laplace-Operator verantwortlich. Übrigens stelle ich nie psi dar, sondern immer |psi|². Das ist nämlich die Wahrscheinlichkeitsdichte. In der Textur ist rot der reelle Anteil und grün der Imaginäre. Als der Startwert implementiert war (siehe Abb. 7.1), habe ich überlegt, was der Laplace-Operator in diesem Fall macht, wo wir uns im komplexen Zahlenraum befinden. Da beschreibt er nämlich keinen einfachen Diffusionsvorgang mehr. Lässt man das Potenzial erst einmal weg, wird die Sache schon überschaubarer:

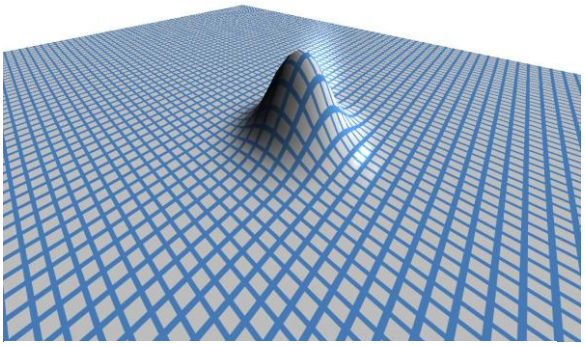
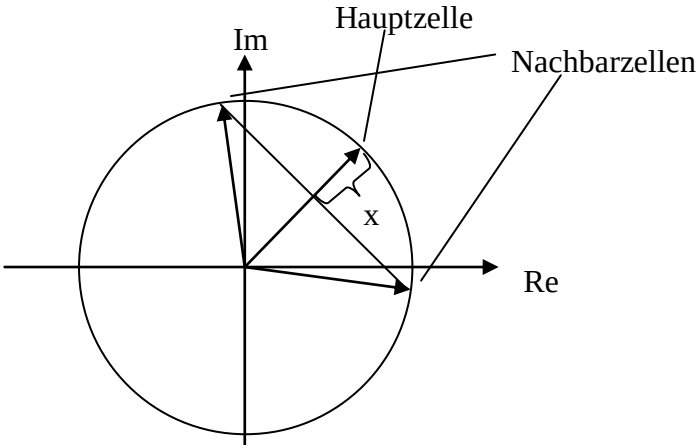


Abb. 7.1: Ein freies Wellenpaket |psi|².

$$\frac{d}{dt}\psi(\vec{r},t)=-i\frac{\hbar}{2m}\Delta\psi(\vec{r},t)$$

Der Faktor *i* macht aus reellen Werte imaginäre Werte und umgekehrt. Nach langem Überlegen und Ausprobieren habe ich dann eine geometrische Lösung gefunden:




Abgebildet sind die Phasenzeiger von psi. Das Ziel ist es die nächste Zeigerposition der Hauptzelle, die von zwei Nachbarzellen umgeben ist, zu berechnen. Die Schrödingergleichung (ohne Potenzial) sagt nun, dass der Winkel, um den sich der Zeiger dreht, proportional zur Länge der Strecke x ist. Und die Länge ist, wie man

leicht sehen kann, umso größer, je größer die Phasendifferenz zu den Nachbarzellen ist. Das macht Sinn, bedeutet es doch, dass die Zeiger umso schneller rotieren, je größer die Frequenz, und damit die Bewegungsenergie des Teilchens ist. Interessanterweise ist die Phase an sich, der Schrödingergleichung völlig egal. Sie hat überhaupt keine Aussage. Es kommt nur auf die Phasendifferenz an. Bei Wasserwellen etwa hat die Phase sehr wohl eine Aussage, denn sie sagt ja wo ein Wellenberg und wo ein Wellental ist. Bei Materiewellen gibt es diese Wellenberge und Wellentäler nur in der mathematischen Beschreibung, in der Realität jedoch haben sie keine Bedeutung. Deswegen „wellt“ im Elektron auch nichts. Die Schrödingergleichung liefert korrekte Resultate, stimmt aber von der Grundidee her nicht mit der Wirklichkeit überein, da Quantenteilchen nun einmal keine Wellen sind.
Meine geometrischen Überlegungen habe ich dann im PixelShader umgesetzt. Der nächste Schritt war die Zeitintegration. Als erstes versuchte ich es mit dem einfachsten expliziten Verfahren:

$$\psi(t+1) = \psi(t) + \frac{d}{dt}\psi(t) \cdot \Delta t$$

Zuerst sah es perfekt aus. Das Wellenpaket zerlief sogar! Aber dann nach etwa zehn Sekunden fing der Hügel an zu zittern und zu wabbern, immer stärker und stärker bis er schließlich spektakulär ins Unendliche explodierte. Offenbar war dieses Verfahren instabil. Von da an hab ich zwei Wochen lang versucht die Ursache für das Wabbern zu finden. Erstaunlich ist, dass selbst am Rand, psi immer noch nicht ganz null ist (obwohl es in der Abb. 7.1 sehr danach aussieht). Ich vermutete, dass sich Interferenzen am Rand bildeten (wie beim Wellensimulator), welche sich immer weiter verstärkten und dann das Wellenpaket erreichten. Dies konnte ich jedoch ausschließen, nachdem ich diese winzig kleinen Wahrscheinlichkeitsamplituden unterdrückt hatte. Ich probierte noch einige Dinge aus. Es half alles nichts. Dann erinnerte ich mich an das implizite Euler-Verfahren, das meine Diffusionsberechnung stabil gemacht hat. Vielleicht würde es hier auch helfen:

$$\psi(t+1) = \psi(t) + \frac{d}{dt}\psi(t+1) \cdot \Delta t \qquad \text{einsetzen}$$

$$\psi(t+1) = \psi(t) + (-i \cdot \frac{\hbar}{2m} \cdot \Delta\psi(t+1) - i\frac{1}{\hbar} \cdot V(\vec{r})) \cdot \Delta t \qquad \text{auf die andere Seite}$$

$$\psi(t+1) + i \cdot \frac{\hbar}{2m} \cdot \Delta\psi(t+1) \cdot \Delta t = \psi(t) - i\frac{1}{\hbar} \cdot V(\vec{r}) \qquad \text{ausklammern}$$

$$\psi(t+1) \cdot (I + i \cdot \frac{\hbar}{2m} \cdot \Delta \cdot \Delta t) = \psi(t) - i\frac{1}{\hbar} \cdot V(\vec{r})$$

Dieses LGS ist etwas länger als das der Strömungssimulation. Und es verkompliziert sich noch erheblich, weil ich mit komplexen Zahlen rechne. Aber als der PixelShader dann endlich fertig war, passierte etwas sehr seltenes: es funktionierte auf Anhieb!

Bevor ich zu den Resultaten komme, ist noch eine Kleinigkeit zu klären. Und zwar die Normierung von $|psi|^2$. Wenn man die Wahrscheinlichkeitsdichte über die Flächen aller Texel summiert, muss 1 herauskommen, weil das Teilchen sich auf jeden Fall irgendwo auf der Ebene befinden *muss*. Wenn man jetzt die Schrödingergleichung laufen lässt und ein von null verschiedenes Potenzial hat, dann steigt oder sinkt die Gesamtwahrscheinlich jedoch die ganze Zeit. Egal, könnte man meinen, wenn $\psi$ eine Lösung ist, dann ist doch auch $\psi' = const \cdot \psi$ eine Lösung. Das stimmt natürlich, aber wenn psi immer größer wird, wird es auch immer ungenauer, weil es eine Fließkommazahl ist. Wenn man auf die Fließkommazahl 5.0e10 eine eins addiert, ändert sich an der Zahl nichts mehr. Aus diesem Grund muss $|psi|^2$ hin und wieder auf 1 normiert werden. Dazu muss die Summe aller Texel bestimmt werden. Um die heraus zu bekommen, nutze ich wieder eine Spezialfähigkeit der GPU aus, nämlich das „Mip-Mapping“. Die GPU kann blitzschnell ein halb so großes Abbild einer Textur erstellen. Wenn sie dann immer weiter halbiert, erhält man am Ende eine 1x1 Textur, welche den Durchschnitt aus allen Texeln enthält. Und damit habe ich auch die Summe, die ich für die Normierung brauche. Das Mip-Mapping kostet erstaunlicherweise fast überhaupt keine Performance. Das Potenzial ist übrigens eine Bitmapdatei, die der Benutzer beliebig mit einem Grafikprogramm festlegen kann. Die Datei wird beim Programmstart gelesen und der Inhalt in eine Textur kopiert.
Mit diesem Simulator kann nun jeder mit seinem PC alle quantenmechanischen Phänomene eines Ein-Teilchen-Systems interaktiv untersuchen. Vom Potenzial bis zum Tunneleffekt und noch viel weiter. Bisher habe ich nur eine kleine 1D-Simulation der Schrödinger-Gleichung als Java-Applet im Internet gefunden. Ansonsten gibt es wohl nichts Vergleichbares. Ein Screenshot befindet sich auf dem Deckblatt.

# 8. Ausblick

Ich hoffe ich konnte zeigen, dass man mit der Grafikkarte einige richtig spannende Sachen machen kann. Einiges was ich entwickelt habe, hätte man auch schon vor zwei Jahren anbieten können, aber damals dachte man noch, die Grafikkarte wäre nur zum Spielen zu gebrauchen. Jetzt ist klar, dass man sie auch perfekt zum Simulieren einsetzten kann, wenn man sich mit der Hardware vertraut macht und sich einige Tricks einfallen lässt (Bilinerares Filtern, Mipmapping, usw.). Denn die Grafikkarte lässt aufgrund der Bauweise keine Standartlösungen für bekannte Probleme zu. Bei jeder Simulation musste ich mir neue, kreative und vor allem schnelle Lösungsalgorithmen einfallen lassen.
Vieles davon lässt sich sicher gut im Unterricht integrieren, um Dinge zu vermitteln, die bisher aufgrund mangelnder Anschauung nicht möglich waren.
Mit der Zeit ist außerdem eine umfangreiche Hilfsbibliothek entstanden, mit der interessierte Programmierer mit vergleichsweise wenig Aufwand die Grafikkarte ansteuern können.
Zwei meiner Projekte konnte ich hier leider aus Platzgründen nicht mehr erwähnen. Da ist zum einen eine Pulsarsimulation, die auch professionellen Anspruch hat, und ein praktisches Projekt, das ich „akustische Kamera“ nenne. Dabei geht es darum, Schall mit Hilfe einer Kopplung mehrerer Mikrofone live sichtbar zu machen.

# 9. Quellen / Anhang


- David Scherfgen, „3D Spieleprogrammierung“, Hanser, 2003
- http://timms.uni-tuebingen.de/Browser/Browser01.aspx
- http://www.dgp.toronto.edu/~stam/reality/Research/pub.html
- http://en.wikipedia.org/wiki/Laplacian
- http://www.tf.uni-kiel.de/matwis/amat/mw1_ge/kap_2/backbone/r2_1_3.html
- http://www.amara.com/papers/nbody.html