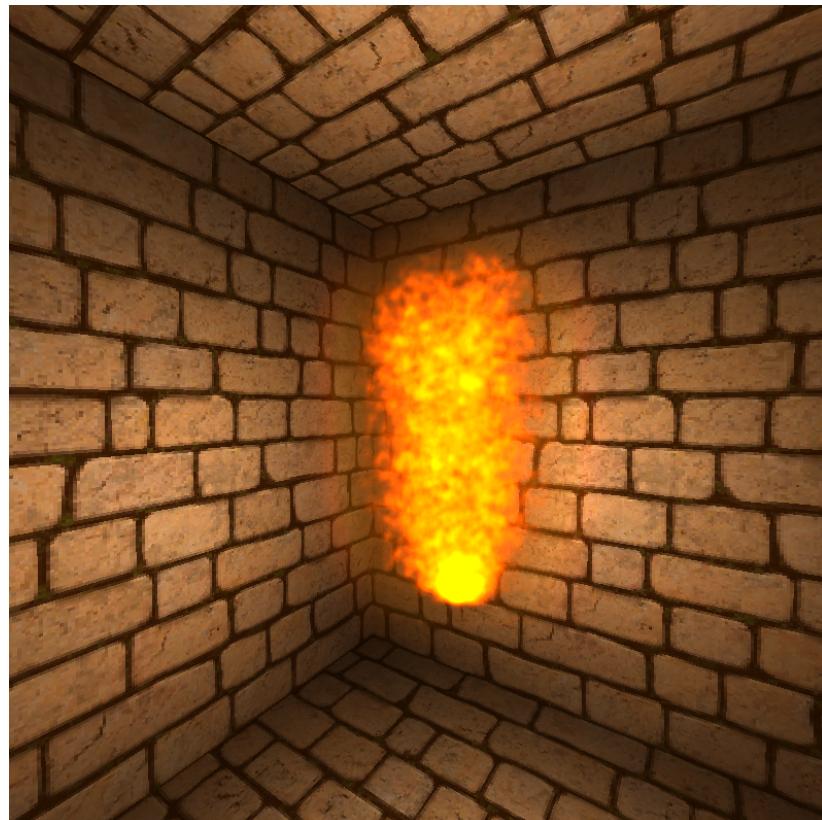


Feuersimulation und -visualisierung

Kristin Schmidt



Parallele Algorithmen mit OpenCL
M. Sc. Henning Wenke

Universität Osnabrück
September 2013

Inhaltsverzeichnis

1 Einleitung.....	1
1.1 Schwerpunkte.....	1
2 Feuer als Partikelsystem.....	1
3 Zusammenspiel von OpenGL und OpenCL.....	2
4 Simulation von Partikelbewegungen.....	3
4.1 Respawn der Partikel.....	3
4.1.1 Sortieren der Partikel.....	3
4.1.2 Arrayshift.....	3
4.1.3 Versetzter Arrayzugriff.....	4
4.2 Generierung der Werte.....	4
4.2.1 Anzahl der zu generierenden Partikel.....	4
5 Bewegung der Partikel.....	5
5.1 Kräfte als Richtungen.....	5
5.2 Low Pressure Areas.....	5
6 Visualisierung.....	6
6.1 Deferred Shading.....	6
6.2 Thickness.....	7
7 Post Effects.....	8
7.1 Noise.....	8
7.2 Glow.....	8
7.3 Hitzeflimmern.....	9
7.4 Umgebung.....	10
7.5 Gesamtbild.....	11
8 Verwendete Literatur.....	III

1 Einleitung

Im Abschlussprojekt zum Kurs "Aufbau interaktiver 3D-Engines" entwarfen wir ein kleines, an das Brettspiel "Das Verrückte Labyrinth" angelehntes, Computerspiel und setzten dieses mit einer uns zur Verfügung gestellten 3D-Engine um. Da der Fokus des Kurses natürlich auf der Logik und nicht

auf der Darstellung des Spiels lag, bauten wir nur wenige graphische Effekte ein. Einer dieser Effekte war ein sich verändernder, flackernder Fackelschein.



Fackelschein durch wechselnde Farben

Doch einen Fackelschein nur durch Farbwechsel mit einem statischen Model darzustellen war uns nicht genug. Deshalb überlegten wir uns, nun ein wenig mit graphischen Effekten zu arbeiten. Der Kurs "Parallele Algorithmen mit OpenCL" bietet dafür

eine ideale Plattform, da wir eine Fackel bzw. deren Flamme als Partikelsystem beschreiben und so schönere Effekte erzeugen können. Das Partikelsystem ist deshalb interessant, weil es ideal ist, um parallele Algorithmen anzuwenden.

Da wir natürlich nicht nur rote Punkte im Raum oder gar einige Zahlen in der Konsole sehen wollen, sondern ein hübsches Feuer betrachten möchten, müssen wir neben dem Partikelsystem noch eine Visualisierung implementieren. Für die Visualisierung benutzen wir deferred shading mit mehreren Renderpaths, um verschiedene Effekte anzuwenden und die Visualisierung zu verbessern – Es ist also im Hinterkopf zu behalten, dass es in diesem Projekt nicht um physikalische oder chemische Korrektheit von Feuer geht, sondern darum, ein möglichst schönes Feuer auf dem Bildschirm anzeigen zu können.

1.1 Schwerpunkte

Thema unserer Arbeit ist nicht nur die Simulation, sondern auch die Visualisierung. Für beide Teile müssen wir unterschiedliche Werkzeuge benutzen.

Für die Simulation bietet sich OpenCL an. Die vielen einzelnen Partikel können unabhängig voneinander betrachtet werden (zumindest in unserem Fall), was eine parallele Verarbeitung begünstigt.

Für die Visualisierung verwenden wir OpenGL. Nicht nur, weil OpenGL und OpenCL beide in der LWJGL implementiert sind, sondern auch, weil sie auf den selben Speicher zugreifen können. Genauer können wir ein OpenCL MemoryObject erzeugen, welches als Ressource ein bereits angelegtes BufferObject des OpenGL Contexts verwendet. Das verringert den Datentransfer zwischen CPU/RAM vom Host (Java-Applikation) und GPU/VRAM vom Client (Graphikkarte), da die Daten nur einmal in den VRAM geladen werden müssen und danach immer auf der Graphikkarte bereit liegen.

2 Feuer als Partikelsystem

Wir betrachten in unserer Arbeit ein sehr einfaches Modell von Feuer, da physikalische oder chemische Korrektheit für eine schöne und glaubwürdige Visualisierung vernachlässigt werden können. Unser Feuer besteht aus einer festen Anzahl Partikel (im fertigen Programm 65.536 Partikel), die stetig erneuert werden, also in einem bestimmten Startbereich neu generiert werden. Von dort bewegen sie sich "nach oben". Um die Partikel zu beschreiben, stellen wir sie als 8-Tupel

dar: 3 Werte für die Position ($x|y|z$), 3 Werte für die aktuelle "Richtung" ($x|y|z$), 1 Wert für die verbleibende Lebenszeit, 1 Wert für den Zustand (Flamme oder Hitzeblämmern).

Diese Partikel, in Java generiert und mit Startwerten versehen, übergeben wir zuerst an einen OpenCL-Kernel, der ihre neue Position und Richtung bestimmt. Danach werden Partikel respawnt und im Anschluss mit OpenGL visualisiert. Diese Dokumentation über unser Projekt soll die verschiedenen Schritte der Simulation und Visualisierung zeigen und erklären.

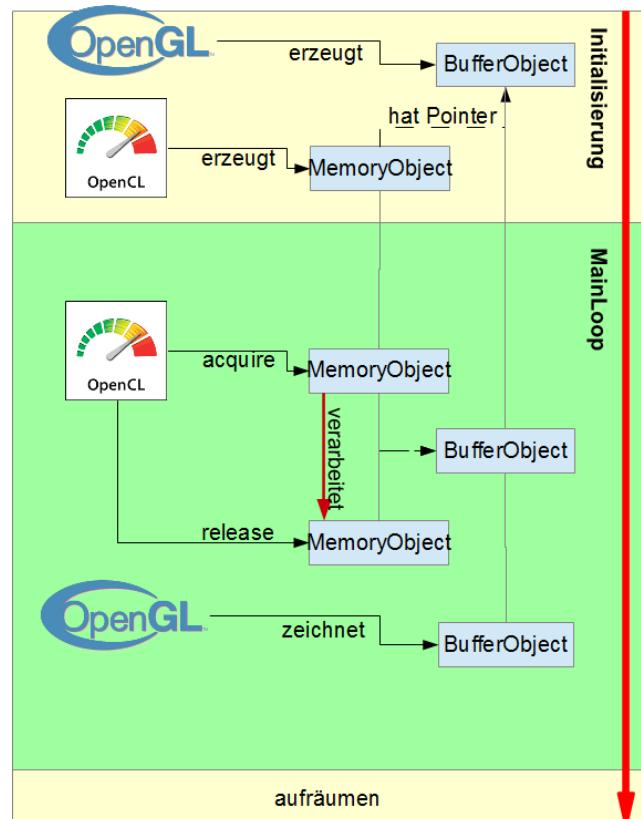
3 Zusammenspiel von OpenGL und OpenCL

Für die Partikel erzeugen wir zuerst einige BufferObjects mit OpenGL. Eines für die Positionen, eines für die Richtungen, eines für die Lebenszeit und den Zustand. Außerdem brauchen wir noch einige weitere BufferObjects, u.a. für Low Pressure Areas (s. Abschnitt "Bewegung der Partikel"). Wir könnten auch alle Daten in einem BufferObject speichern, entscheiden uns aber dagegen, damit wir nicht allzu komplizierte Speicherzugriffe haben und den Code besser durchschauen können. Möglicherweise wäre die Verwendung nur eines BufferObjects deshalb eine Performancesteigerung.

Dadurch, dass wir drei BufferObjects nutzen, ist es allerdings einfacher, die Partikel an einen OpenGL-Shader zu übergeben – hier müssen wir auf keine Strides oder ähnliches achten, was uns Fehlerquellen vermeiden und so effektiver arbeiten lässt.. Insgesamt ist das Programmieren so also einfacher gehalten, was es uns ermöglicht, schnell und viel zu debuggen und Fehler besser zu bearbeiten.

Nachdem die BufferObjects erzeugt sind, generieren wir für jedes BufferObject ein korrespondierendes MemoryObject. Diese MemoryObjects haben keine "eigenen Daten", sie haben lediglich Referenzen bzw. Pointer auf die Daten der BufferObjects. OpenCL kann sich nun einen exklusiven Zugriff auf die Daten verschaffen und mit ihnen arbeiten. Nachdem die Daten wieder freigegeben werden, ist es für den OpenGL-Context möglich, die Daten an die Pipeline zu überreichen und zu zeichnen.

Nach dem Zeichnen sichert sich der OpenCL-Context erneut den Zugriff auf die Daten und der Prozess geht von vorn los.



Schematisch: Zusammenspiel von OpenGL und OpenCL

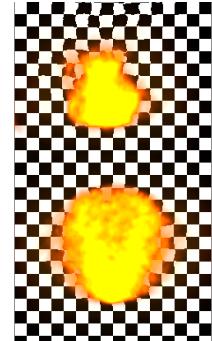
4 Simulation von Partikelbewegungen

4.1 Respawn der Partikel

Wichtig für ein Partikelsystem mit konstanter Anzahl Partikel ist, dass man Partikel nicht dann respawnen muss, wenn sie "zu alt" sind, sondern dann, wenn sie benötigt werden. Sonst erhält man Lücken in der Simulation (s. Bild rechts): Spawnt man alle Partikel auf einmal oder erneuert sie erst, wenn ihre Lebenszeit abgelaufen ist, gibt es Momente, in denen man keine Partikel (re-)spawnen kann.

Um trotzdem zu verhindern, dass zufällig immer die selben Partikel respawnen werden (was einen ähnlichen Effekt zur Folge hätte, da gerade neu gespawnte Partikel so wieder zurückgesetzt würden und sich eine Lücke zwischen ihnen und den alten bildete), versuchen wir möglichst alte Partikel zu recyceln, also solche, deren verbleibende Lebensdauer ziemlich gering ist.

Zur Lösung dieses Problems verfolgen wir drei verschiedene Ansätze.



4.1.1 Sortieren der Partikel

Der erste Ansatz ist das Sortieren der Partikel. Wir verwenden einen Bitonic Sort um die vielen Werte der Partikel (Position, Richtung, ...) nach einem Schlüssel (verbleibende Lebenszeit) zu sortieren.

Dabei stoßen wir jedoch auf zwei Probleme. Das erste Problem ist die Laufzeit: Bitonic Sort ist für diesen Zweck unglaublich schlecht geeignet, unsere FPS¹ sinken drastisch auf z.T. unter 30. Möglicherweise kann man das mit optimierten Versionen von Bitonic Sort verbessern, doch das zweite Problem lässt uns vom Sortiervorhaben zurückschrecken: Unser Sortierverfahren ist stark beschränkt. Lediglich einige wenige Tausend Werte lassen sich sortieren, weil wir durch die maximale Workgroup-Size eingeschränkt sind. Da wir aber nicht nur 1.024 sondern 65.536 Partikel haben möchten, ist Sortieren definitiv der falsche Weg.

4.1.2 Arrayshift

Als zweite Variante überlegen wir uns den Arrayshift. Das heißt, vor jedem Respawn verschieben wir alle Daten im Array um m (= Anzahl der zu respawnenden Partikel) Stellen und respawnen an den so "freigewordenen" Stellen m neue Partikel. Dieser Ansatz ist um einiges schneller, hat nur ein sehr großes Problem: Er ist eindeutig iterativ zu verwenden, es sei denn man will mit viel Speicher arbeiten. Denn wenn man direkt auf dem Array arbeiten möchte, bekommt mit paralleler Arbeitsweise das aus dem Threading bekannte Problem der Dirty Reads und Lost Updates: Während ein Workitem noch die Daten an einer bestimmten Stelle liest, versucht ein weiteres schon zu schreiben. Oder ein Workitem liest ein bereits erneuertes Partikel und dupliziert dieses dadurch, auf Kosten eines anderen Partikels, welches so verloren geht.

Iterativ ist das gar kein Problem, da einfach die nicht mehr benötigten Partikel der Reihe nach überschrieben werden können. Ebenso ist es kein Problem, wenn man den doppelten Speicher nutzen will.

¹ FPS: Frames per second, Bilder/Main-Loop-Durchläufe pro Sekunde

Da wir aber weder das eine, noch das andere möchten, entscheiden wir uns für folgende dritte Lösung.

4.1.3 Versetzter Arrayzugriff

Diese Lösung ist sehr einfach und effizient. In jedem Main-Loop-Durchlauf möchten wir m Partikel respawnen. Dazu müssen wir lediglich in jedem Main-Loop-Durchlauf den Offset für unseren Arrayzugriff ändern und direkt m Partikel ab dort überschreiben. Da wir die Partikel mit ähnlicher Lebensdauer spawnen, werden mit dieser Methode auch immer ungefähr die aktuell ältesten Partikel erneuert.

Die großen Vorteile dieser Lösung sind zum Einen ihre geringe Workgroup-Size von nur m statt n (= Gesamtzahl der Partikel) und zum Anderen ihre dadurch gleichzeitig sehr hohe Geschwindigkeit.

4.2 Generierung der Werte

Doch zu wissen, wo die m neuen Partikel zu spawnen sind, ist nicht alles. Die Partikel müssen wir auch irgendwie generieren. Dazu erzeugen wir Pseudozufallszahlen in der Java-Applikation, die bestimmten Mustern folgen. So werden für die Positionen der Partikel nur Zufallszahlen in einer Art gequetschter Sphere erzeugt, die Richtungen nur auf der oberen Hälfte einer Kugeloberfläche und für die Lebenszeiten nur Zahlen in einem bestimmten Intervall. Nachdem all diese Zahlen erzeugt wurden, werden sie in ein kleineres MemoryObject gepackt und auf die Graphikkarte geladen, die die Daten dann mit dem für den Respawn zuständigen Kernel an die richtigen Stellen in ihrem gespeicherten Partikelarray schreibt.

Die Generierung der Werte könnte man prinzipiell auch auf der Graphikkarte erledigen, das Problem hierbei ist jedoch, dass OpenCL von sich aus keine Funktion zur Generierung von Pseudozufallszahlen besitzt. Und Javas Methode der Klasse `java.util.Random` ist so schnell, dass die Generierung nicht ins Gewicht fällt.

4.2.1 Anzahl der zu generierenden Partikel

Ein großes Problem im Respawn ist die Frage nach der Anzahl der Partikel. Dadurch, dass wir für unsere Testreihen oft ohne eine Display-Synchronisation von 60 FPS arbeiten möchten, ist eine feste Anzahl zu erneuernder Partikel nicht sehr effizient. Schließlich würden wir, spawnten wir 100 Partikel pro Main-Loop-Durchlauf, bei 60 FPS 6.000 Partikel, bei 400 FPS 40.000 Partikel erneuern. Um das zu verhindern berechnen wir in Abhängigkeit der zuletzt gemessenen Framerate die Anzahl der zu erneuernden Partikel. Liegt sie unter einem bestimmten Schwellwert, so werden aber mindestens eben diesem Schwellwert entsprechende Partikel respawnt. Als Referenzwert legen wir für die Berechnung fest, wie viele Partikel pro Sekunde gespawnt werden sollen. Die errechnete Anzahl Partikel wird anschließend wie bereits beschrieben generiert und an die Graphikkarte gesendet.

```
int neuePartikel = 32;

// Div 0 verhindern, 23000 Partikel pro Sekunde auf aktuellen Frame umrechnen
neuePartikel = (int)fps > 0? (int)(23000 / (int)fps) : neuePartikel;

// mindestens 32 Partikel garantieren
neuePartikel = Math.max(neuePartikel, 32);

Vereinfachte Routine zur Berechnung der Anzahl zu generierender Partikel.
```

5 Bewegung der Partikel

Damit die Partikel nach ihrer Erzeugung nicht statisch im Raum ruhen, müssen wir in jedem Zeitschritt ihre Position neu berechnen. Dabei müssen wir den Betrag, um den wir die Position verändern, mit der vergangenen Zeit gewichten – denn sonst würden die Partikel bei einer hohen Framerate schneller durch den Raum bewegt werden als bei einer niedrigen. Das liegt daran, dass ihre Position viel häufiger um den gleichen Faktor verändert würde.

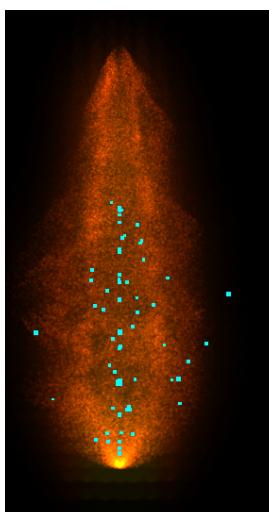
Für die Bewegung müssen wir deshalb unter anderem die Dauer des letztens Frames an den für die Positionsupdates (= Bewegung) zuständigen Kernel übergeben. Diese können wir gleichzeitig auch von der Lebensdauer abziehen. Selbst "tote" Partikel werden jedoch weiterbewegt, sie dienen als Hilfe für unser Hitzeblimmen (siehe Abschnitt Hitzeblimmen).

5.1 Kräfte als Richtungen

Die Bewegung an sich ist eine relativ simple Vektorrechnung. Zuerst werden verschiedene Kräfte, die auf jeden Partikel wirken, gewichtet zusammengerechnet und normalisiert. Dieser normalisierte Vektor entspricht der neuen Richtung. Um nun den Vektor zu verschieben, müssen wir den Richtungsvektor lediglich mit einem Geschwindigkeitsmodifikator und der vergangenen Zeit skalieren und können ihn direkt auf die Position addieren. So erhält der Partikel seine neue Richtung und Position, die im nächsten Zeitschritt erneut als Ausgangswerte verwendet werden können.

Dieses simple System erlaubt es, beliebig viele Kräfte mit beliebiger Stärke auf einen Partikel einwirken zu lassen – lediglich eine weitere Addition ist notwendig. In unserem System sind alle Kräfte ausschließlich Richtungen, und werden der Einfachheit halber auch als solche bezeichnet. Trotzdem ist das System so ausgelegt, dass man schnell verschiedene andere Kräfte einwirken lassen könnte (z.B. Wind verschiedener Richtungen und Stärken, Explosionsdruck, ...).

Die Basis aller Kräfte ist das Bestreben der Partikel, nach oben zu gelangen. Sie bekommen zuerst das Ziel ($0|10|0$), das weit außerhalb ihrer Reichweite liegt. Die Differenz aus ($0|10|0$) und der Partikelposition gibt die Richtung an, in der ($0|10|0$) vom Partikel aus gesehen liegt. Diese Richtung und die aktuelle (zufällig generierte) Richtung des Partikels summieren wir nun gewichtet auf. Nach einigen Iterationen bewegen sich die Partikel so automatisch an den Punkt ($0|10|0$).



Low Pressure Areas (cyan)
bestimmen die Form

5.2 Low Pressure Areas

Damit die Partikel nicht nur nach oben streben, sondern auch eine Flammenform annehmen, führen wir die Low Pressure Areas (zu Deutsch etwa "Gebiete niedrigen Luftdrucks") ein. Sie sind die wichtigste Kraft in der Formgebung unserer Simulation.

Unsere Low Pressure Areas sind Positionen im Raum, die auf mehreren Ebenen generiert werden und alle 100 ms an neue Positionen gesetzt werden. Wir verwenden in unserer Simulation 64 Low Pressure Areas, die verwendeten acht Ebenen sind mit einem Abstand von jeweils ca. 0,3 Einheiten übereinander gelegt und haben verschiedene Radii: nachdem sie sich von unten nach oben erst vergrößern, nehmen die Radii bald wieder stark ab. So bekommen wir eine gewölbte Flammenform. Damit die Flamme sich noch glaubhafter verhält, werden die Low Pressure Areas

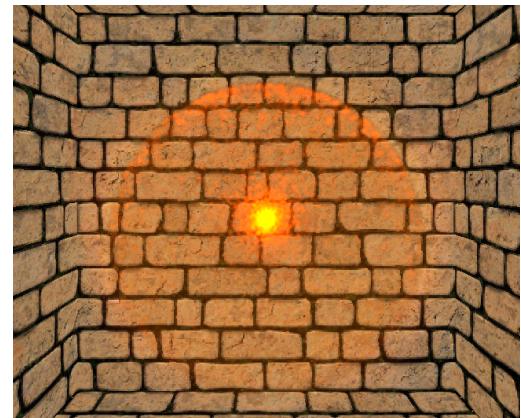
stetig verändert und bekommen ein wenig zufällige Radii. So wirkt die Flamme weniger statisch und lebhafter.

Die Partikel navigieren nun durch die Low Pressure Areas. Dazu berechnen wir für jeden Partikel den Abstand zu allen Low Pressure Areas, die einen y-Wert größer als die Position in y des Partikels haben. So wird eine Bewegung nach oben garantiert, da unterhalb liegende Partikel nicht mehr berücksichtigt werden. Aus den vier Low Pressure Areas mit dem geringsten Abstand wird zufällig (Zufallszahl erneut aus Java) eine ausgewählt. Diese zufällige Auswahl bringt Unruhe in die Partikel und lässt sie nicht auf geregelten Bahnen verweilen.

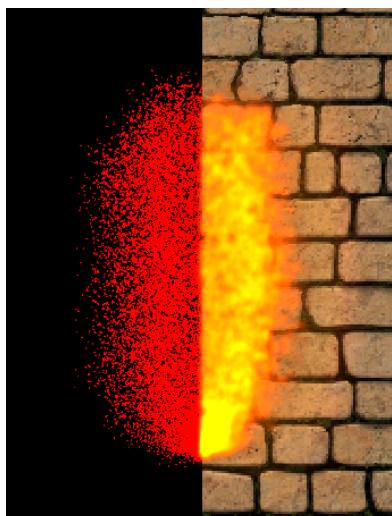
Die Differenz aus der gewählten Low Pressure Area und dem Partikel bestimmt erneut eine vom Partikel aus gesehene Richtung. Diese Richtung wird nun ebenfalls gewichtet mit der aktuellen Richtung des Partikels verrechnet.

Als letzten Schritt in der Simulation müssen wir die Parameter richtig einstellen. Das ist eine schwierige Sache, denn als Parameter haben wir viele verschiedene Größen, die das Ergebnis unterschiedlich stark beeinflussen (Low Pressure Areas, aktuelle Richtung, Bestreben nach oben) gilt es auch die Generierungsoptionen (Radius der Spawnarea, erste Richtungen, Lebensdauer, Low Pressure Areas, ...) zu berücksichtigen.

Wichtig ist dabei in erster Linie, dass das Ergebnis sich wie eine Flamme verhält – zumindest so aussieht, als verhalte es sich wie eine Flamme. Für die optische Darstellung übergibt OpenCL die nun ausführlich bearbeiteten MemoryObjects wieder dem OpenGL-Context, der als nächstes für die Visualisierung sorgt.



Falsche Parameter haben auch schöne Effekte, sind aber nicht gewünscht



6 Visualisierung

Die Visualisierung haucht der Simulation Leben ein. Sie macht aus langweiligen Punkten im Raum erst das, was wir als Feuer wahrnehmen sollen.

Für die Visualisierung müssen wir uns vom Partikelsystem weg bewegen, hin zur pixelweisen Manipulation des Bildes. Dazu erzeugen wir ein Bild mit den Pixeln, das immer weiter bearbeitet wird. Dieser Vorgang ist bekannt als Deferred Shading.

Vor allem beim Deferred Shading ist die Visualisierung sehr parallel, weil Texturen pixelweise bearbeitet und berechnet werden. Alle Berechnungen im FragmentShader werden parallel für alle Fragments auf der Grafikkarte berechnet. Somit sind bei uns nicht nur Partikel parallel berechenbar, sondern auch Pixel.

6.1 Deferred Shading

Beim Deferred Shading wird das Bild nicht wie sonst üblich direkt auf den Bildschirm gezeichnet, sondern zunächst in eine Textur, die so groß ist wie das Fenster der Anwendung. Diese Textur kann dann in weiteren Renderpaths verwendet, bearbeitet und für die Erzeugung weiterer Texturen genutzt werden. So kann man noch viele Post-Effects auf sein Bild anwenden, muss aber die Geometrien, wie zum Beispiel Partikel, nur einmal zeichnen. Außerdem können Effekte wie unser Glow-Effekt angewandt werden, die ohne Deferred Shading gar nicht möglich wären. Um dann eine Textur zu zeichnen, benötigt man nur ein sogenanntes ScreenQuad, was einfach ein Rechteck ist, das die Koordinaten der Bildschirmmecken bekommt. Die endgültige Textur kann so dann am Ende auf den Bildschirm gezeichnet werden.

6.2 Thickness

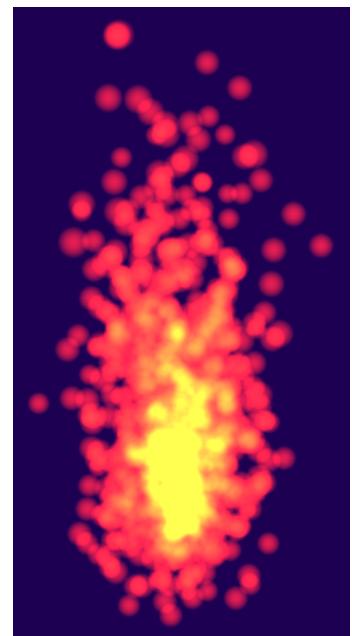
Als erstes müssen die Partikel alle einmal gerendert werden. Wir zeichnen die Partikel als kleine halbdurchsichtige Kreise auf schwarzen Hintergrund.

Um jeden Partikel einzeln zu zeichnen, verwenden wir `GL_POINTS`. Da diese jedoch einfache Quadrate sind, müssen wir sie noch zu Kreisen "zurechtschneiden". Im FragmentShader können wir die Koordinaten des Fragments auf diesem Quadrat bekommen. Die Koordinaten werden so umgerechnet, dass der Nullpunkt in der Mitte des Quadrats liegt. Das Skalarprodukt dieser Koordinaten beschreibt den Abstand zum Mittelpunkt des Quadrats. Nun wird für jedes Fragment, bei dem dieser Wert über 1 liegt, der `discard`-Befehl aufgerufen, wodurch das Fragment nicht gezeichnet wird und nun nur die Fragments innerhalb des Kreises gezeichnet werden.

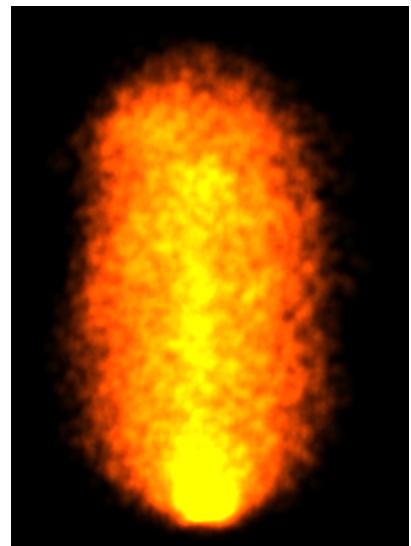
Nun müssen wir noch erreichen, dass dort, wo viele Partikel übereinanderliegen, das Feuer eine hellere, intensivere Farbe bekommt und in den Randbereichen, wo nur wenige Partikel übereinanderliegen, eine durchsichtigere, dunklere Farbe. Deshalb passiert das Zeichnen mit aktiviertem Blending und ohne Tiefentest. Das Blending bewirkt, dass nicht nur das zuletzt gezeichnete Fragment die Farbe bestimmt, sondern die Farben aller übereinanderliegenden Fragments miteinander verrechnet werden. Die verwendete Blending-Funktion sorgt dafür, dass einfach die Werte beider Fragments aufeinander addiert werden.

```
glBlendFunc(GL_ONE, GL_ONE);
 glEnable(GL_BLEND);
 glDisable(GL_DEPTH_TEST);
```

Die richtigen Einstellungen für das Rendering sind wichtig.



früher Screenshot mit weniger Partikeln (zur besseren Sichtbarkeit)



Thickness-Textur unserer endgültigen Version

Als Farbe für die einzelnen Fragments haben wir ein sehr dunkles Orange gewählt. Es hat einen sehr geringen Rot-Wert, einen noch geringeren Grün-Wert und enthält kein Blau. Dadurch ist das Feuer in den Bereichen, wo wenig Partikel sind, dunkelrot, welches mit zunehmender Partikeldichte immer kräftiger wird

und in den mittleren Bereichen, wo sich die meisten Partikel befinden, geht die Farbe in gelb über. Dieser Wechsel von schwarz über kräftiges rot zu gelb funktioniert dadurch, dass sich zunächst nur der Rot-Wert 1 annähert, während der Grün-Wert noch so niedrig ist, dass er kaum ins Gewicht fällt. Bei höherer Partikeldichte kann der Grün-Wert jedoch auch auf 1 oder höher steigen, während der Rot-Wert dann schon einen Wert weit größer als 1 erreicht. Da 1 jedoch der maximale Wert ist, der für eine Farbe dargestellt wird, sind rot und grün dann beide 1 und somit ergibt sich gelb.

Bei der Farbberechnung wird ebenfalls der Radius, also der Abstand zum Kreismittelpunkt miteinbezogen. Je weiter entfernt vom Mittelpunkt ein Fragment ist, umso dunkler wird die Farbe. Das sorgt dafür, dass die Kanten weicher sind und das Feuer so ebenmäßiger erscheint und man vor allem keine einzelnen Kreise erkennen kann.

Mit Ausprobieren vieler verschiedener Werte und Anpassungen an unsere verwendete Partikelanzahl ergab sich dann die folgende Formel für unsere Farbe:

```
PixelColor = vec4(0.08f * (1.0f - radius)^2,
                  0.03f * (1.0f - radius)^3,
                  0.00f,
                  0.05f * (1.0f - radius)^2);
```

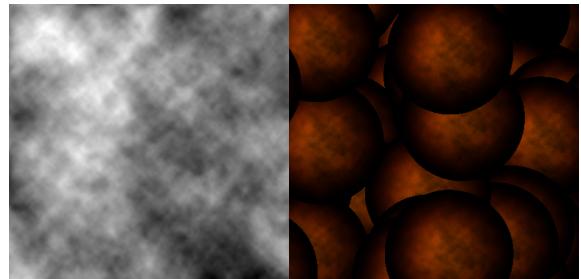
Vereinfachter Pseudocode für die Farbbestimmung.

(Der Alpha-Wert ist an dieser Stelle noch nicht wichtig, er wird allerdings später zur Berechnung des Hitzeblimmsen benötigt. Die Potenzierungen haben wir im Shader zur Verbesserung der Geschwindigkeit durch Multiplikationen ersetzt und teilweise vorberechnet.)

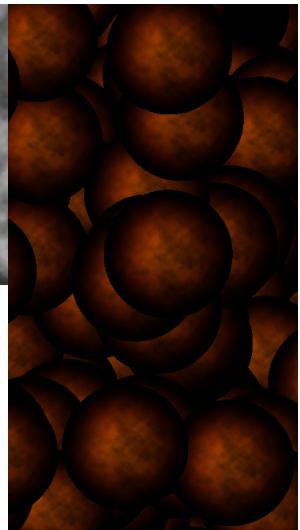
7 Post Effects

7.1 Noise

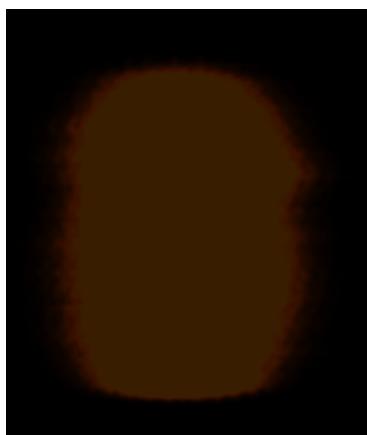
Da es uns wichtig ist, unser Feuer lebendiger und weniger gleichmäßig wirken zu lassen, haben wir zusätzlich Noise eingebaut. Um im Shader diese Art des Zufalls nutzen zu können, übergeben wir eine Noise-Textur an den entsprechenden Fragment-Shader. Diese wird beim Zeichnen von jedem einzelnen Partikel berücksichtigt. Bei jedem Fragment, das für einen Partikel gezeichnet wird, wird anhand der Koordinaten des GL_POINTS eine Farbe aus der Noise-Textur ausgelesen. Nun wird einer der Farbwerte, der immer zwischen 0 und 1 liegt, auf die Fragment-Farbe multipliziert. Dadurch variiert die Intensität der Farbe auf dem Kreis, der für jeden Partikel gezeichnet wird, anhand der Noise-Textur.



Perlin-Noise



Darstellung mit geänderten Shadern um Noise auf den einzelnen Partikeln zu verdeutlichen



glow-Textur

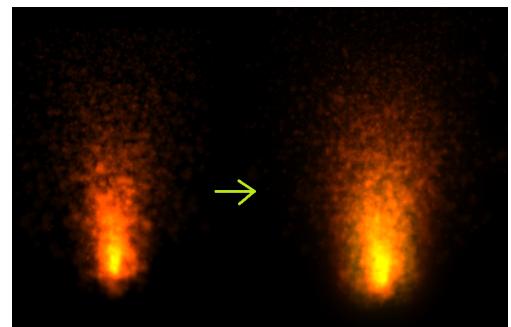
7.2 Glow

Den Glow-Effekt des Feuers haben wir ähnlich wie den typischen Bloom-Effekt, den man aus vielen Computerspielen kennt, mithilfe eines Blur-Filters umgesetzt. Wir nutzen hierfür die Faltung mit einem Gauss-Kernel. Das bedeutet, dass für jeden Pixel ein mit der Gauss-Funktion gewichteter Durchschnitt aus ihm und seinen Nachbarpixeln berechnet wird. Je nachdem welche Kernelgröße man wählt, werden unterschiedlich viele Pixel als Nachbarpixel mitberechnet. Wir haben hier einen Wert von 17 gewählt und somit müssen für jeden Pixel $17 * 17 (= 289)$ Pixel verrechnet werden.

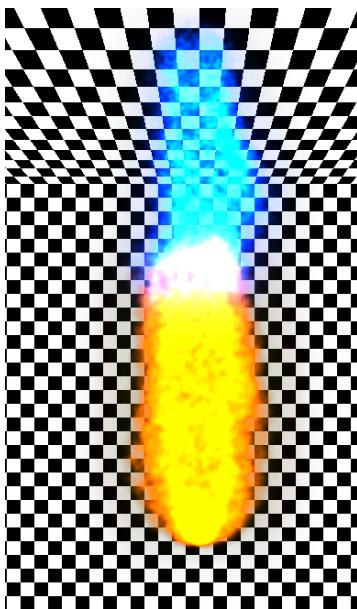
Da die Berechnung der Gauss-Funktion jedoch sehr aufwendig ist, haben wir die vorberechneten Werte für unsere Kernel-Größe in den Shader geschrieben. Um einen zusätzlichen Geschwindigkeits-

gewinn zu erzielen, berechnen wir die Faltung in zwei Schritten, erst in horizontaler Richtung, dann das resultierende Bild in vertikaler Richtung. Dadurch erreichen wir das gleiche Ergebnis wie bei einem Schritt, aber müssen jeweils nur auf die in einer Richtung umliegenden Pixel zugreifen, also in unserem Fall zweimal 17. Dadurch haben wir also statt einer quadratischen Laufzeit nur noch eine lineare.

Die so erhaltene Farbe wird für jeden Pixel auf einen maximalen Rot-Wert von 0,2 und einen maximalen Grün-Wert von 0,1 beschränkt. Dadurch erhält man eine Textur wie auf dem oberen Bild. Diese wird dann auf die Thickness-Textur aufaddiert.



Unterschied ohne und mit glow-Effekt



blau: "tote" Partikel

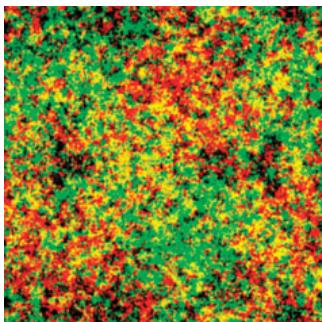
7.3 Hitzeflimmern

Um unser Feuer noch realistischer wirken zu lassen, möchten wir gerne auch das typische Hitzeflimmern einbauen, das man typischerweise über einer heißen Flamme sehen kann.

Da wir keine physikalisch korrekte Simulation haben, die die Temperatur der umliegenden Luft und dadurch entstehende Lichtbrechungen berechnet, müssen wir uns einen Trick überlegen, diesen schönen Effekt trotzdem realistisch wirkend darzustellen.

Hierzu nutzen wir die Partikel, deren Lebenszeit bereits abgelaufen ist, die also nicht mehr gezeichnet werden, aber noch nicht respawnt wurden (zu sehen in der Abbildung links). Und zwar wird schon im ersten Schritt der Darstellung, beim Zeichnen der Partikel, im Alpha-Wert der Textur ein ähnlich wie der Rot-Wert berechneter Wert abgespeichert. Dies wird auch für die "toten" Partikel gemacht, die nun nicht mithilfe des discard-Befehls verworfen werden, sondern einfach schwarz als Farbe bekommen, damit sie eben diesen Alpha-Wert speichern können.

Beim Auslesen aus der Hintergrund-Textur werden nun dort, wo dieser Alpha-Wert größer als 1 ist, die Texturzugriffe verändert.

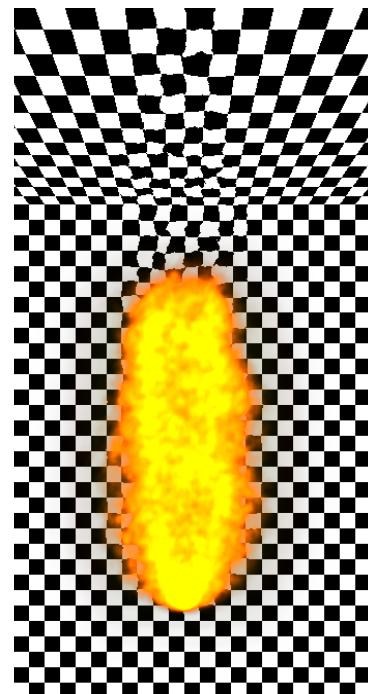


Rot-Grün-Noise

Dies geschieht mithilfe einer Rot-Grün-Noise-Textur. Diese Textur enthält im roten und grünen Farbkanal jeweils unabhängige Zufallswerte. Wenn nun ein Pixel aus der Hintergrund-Textur ausgelesen werden soll, wird zunächst eine Farbe aus der Noise-Textur ausgelesen. Diese Farbe aus dem Intervall $[0; 1]$ wird nun auf das Intervall $[-0,0016...; 0,0016...]$ abgebildet, da der Zugriff in der Hintergrund-Textur nur um wenige Pixel verschoben werden soll.

Anschließend wird dieser Wert noch mit dem Alpha-Wert gewichtet, damit dort, wo mehr "tote" Partikel sind, ein stärkerer Effekt auftritt und er zu den Rändern hin schwächer wird. Nun wird der Rot-Wert auf die x-Koordinate der Textur-Koordinaten addiert und der Grün-Wert auf die y-Koordinate.

Wenn nun mit diesen leicht veränderten Koordinaten aus der Hintergrund-Textur ausgelesen wird, erhält man hauptsächlich über dem Feuer eine verzerrte Version des Hintergrundes, die stark an das typische Hitzeflimmern erinnert und sich auch dynamisch mit der Bewegung der "toten" Partikel bewegt.

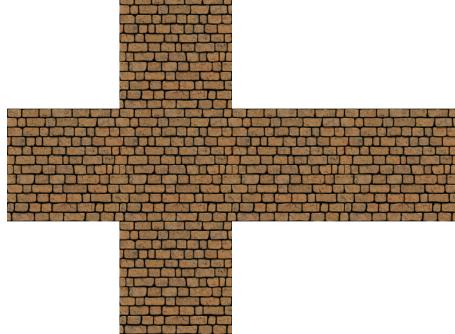
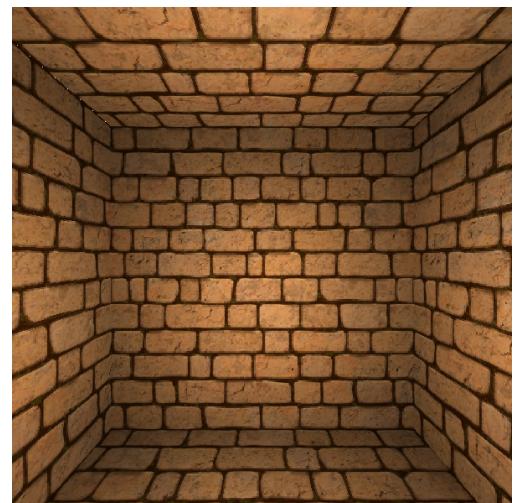


7.4 Umgebung

Um dem ganzen noch eine schönere Atmosphäre zu geben und zusätzlich das Leuchten des Feuers zur Geltung zu bringen, haben wir noch eine Art kleinen Raum um das Feuer eingefügt.

Hierzu brauchten wir zunächst einen Würfel, den wir mit OpenGL zeichnen können. Da wir nicht jede Fläche einzeln zeichnen wollen, brauchen wir eine Textur für den gesamten Würfel. Hierbei erwies sich das Zuweisen der passenden Textur-Koordinaten als das Schwierigste, vor allem weil einige der Vertices für verschiedene Flächen

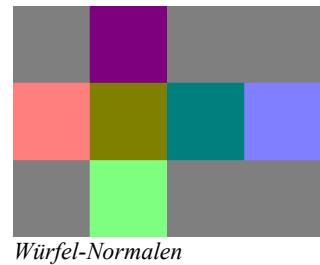
unterschiedliche Textur-Koordinaten benötigen. In diesem Fall



Würfel-Textur

beschreiben die Textur-Koordinaten wo auf der Textur sich der Vertex befindet, damit jede Fläche am Ende den richtigen Teil der Gesamt-Textur bekommt. Durch das Verwenden mehrerer Vertices mit unterschiedlichen Textur-Koordinaten für dieselbe Ecke des Würfels konnten wir dieses Problem jedoch lösen und einen schönen texturierten Würfel zeichnen.

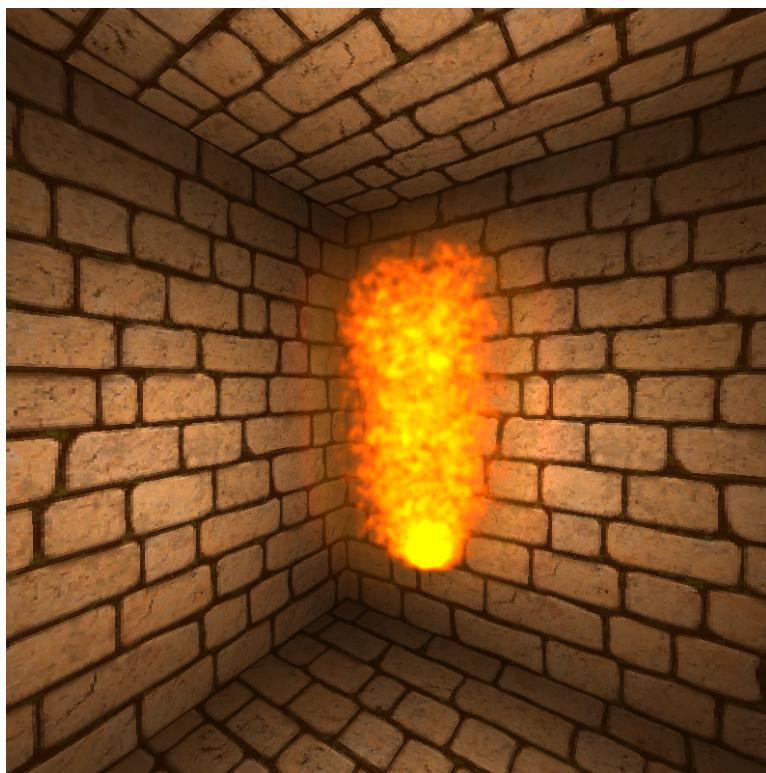
Um diesen Würfel nun auch beleuchten zu können, mussten wir zunächst noch Normalen berechnen. Da die automatisch von OpenGL anhand der Vertex-Normalen berechneten Normalen interpoliert sind und somit die Wände irgendwie gebogen wirken, mussten wir selbst Normalen übergeben. Da jede Wand nur eine Normale braucht, übergeben wir einfach eine 3×4 Pixel große Textur, die für jede Fläche des Würfels die passende Normale als Farbwert gespeichert hat. Da Normalen auch negative Werte annehmen, wir aber nur positive Farbwerte speichern können, bilden wir die Normalen auf das Intervall $[0; 1]$ ab. Beim Auslesen müssen wir deshalb die Farbwerte minus 0,5 rechnen, um wieder positive und negative Werte zu erhalten.



Um unsere Wände jetzt zu beleuchten, verwenden wir drei der Low Pressure Areas als Lichtquellen. Dadurch erhalten wir ein flackerndes und sich leicht bewegendes Licht von der Position des Feuers. Die Beleuchtung erfolgt durch eine abgewandelte Variante diffuser Beleuchtung, wie sie z.B. im Phong-Beleuchtungsmodell verwendet wird. An den hellen Stellen addieren wir zusätzlich noch gewichtet einen orangen Farbton hinzu, um das Licht des Feuers rötlicher wirken zu lassen.

7.5 Gesamtbild

Am Ende müssen nun noch das Feuer und der Hintergrund zusammengefügt werden. Beim Auslesen aus der Hintergrund-Textur wird die im Abschnitt "Hitzeflimmern" beschriebene Modifikation angewandt, um den dort beschriebenen Effekt zu erhalten. Um darzustellen, dass das Feuer vor allem in den Randbereichen leicht durchsichtig ist, wird es nicht einfach nur auf den Hintergrund gezeichnet. Stattdessen wird das Feuer an jedem Pixel mit seinem halben Rot-Wert gewichtet und der Hintergrund mit eins minus dem halben Rot-Wert. Somit ist an Stellen ohne Feuer nur der Hintergrund sichtbar und das Feuer ist auch in der Mitte noch ganz leicht durchscheinend.



Das finale Bild, generiert aus vielen Einzeltexturen.

8 Verwendete Literatur

Somasekaran, S.: *Using Particle Systems to Simulate Real-Time Fire*. University of Western Australia, 2005.

Hong, J.-M. et. al.: *Wrinkled Flames and Cellular Patterns*. Stanford University, 2007.

Green, S. & Horvath, C.: *Flame On: Real-Time Fire Simulation for Video Games*. GPU Technology Conference, 2012.

Horvath, C. & Geiger, W.: *Directable, High-Resolution Simulation of Fire on the GPU*. ACM Transactions on Graphics 28-3, August 2009.

Nguyen, D. Q. et. al.: *Physically Based Modeling and Animation of Fire*. Stanford University, 2002.

de Kruijf, M.: *firestarter – A Real-Time Fire Simulator*. University of Wisconsin.

Nishita, T. & Dobashi, Y.: *Modeling and Rendering of Various Natural Phenomena Consisting of Particles*. Computer Graphics International Conference, 2001.