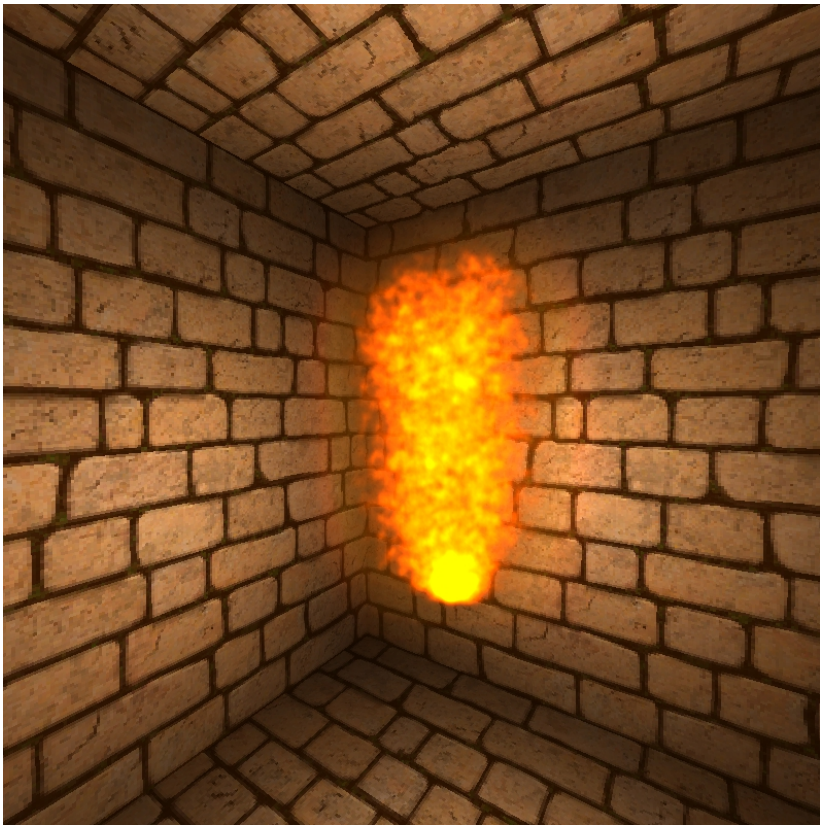
Feuersimulation und -visualisierung

Kristin Schmidt
Sebastian Höffner



Parallele Algorithmen mit OpenCL
M. Sc. Henning Wenke

Universität Osnabrück
September 2013

Inhaltsverzeichnis

1 Einleitung....3
1.1 Schwerpunkte....3

2 Feuer als Partikelsystem....3

3 Zusammenspiel von OpenGL und OpenCL....4

4 Simulation von Partikelbewegungen....4
4.1 Respawn der Partikel....4
4.1.1 Sortieren der Partikel....4
4.1.2 Arrayshift....5
4.1.3 Versetzter Arrayzugriff....5
4.2 Generierung der Werte....5
4.2.1 Anzahl der zu generierenden Partikel....5

5 Bewegung der Partikel....6
5.1 Kräfte als Richtungen....6
5.2 Low Pressure Areas....6

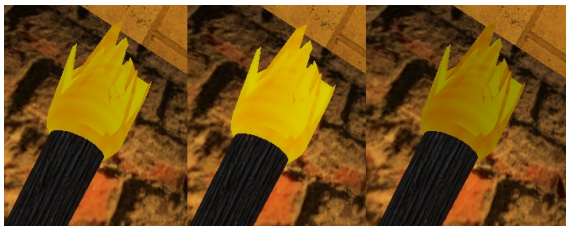
6 Visualisierung durch Deferred Shading....7

7 Post Effects....7
7.1 Noisetexturen....7
7.2 Glow....7
7.3 Hitzeflimmern....7

8 Ergebnis und Evaluation....7

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

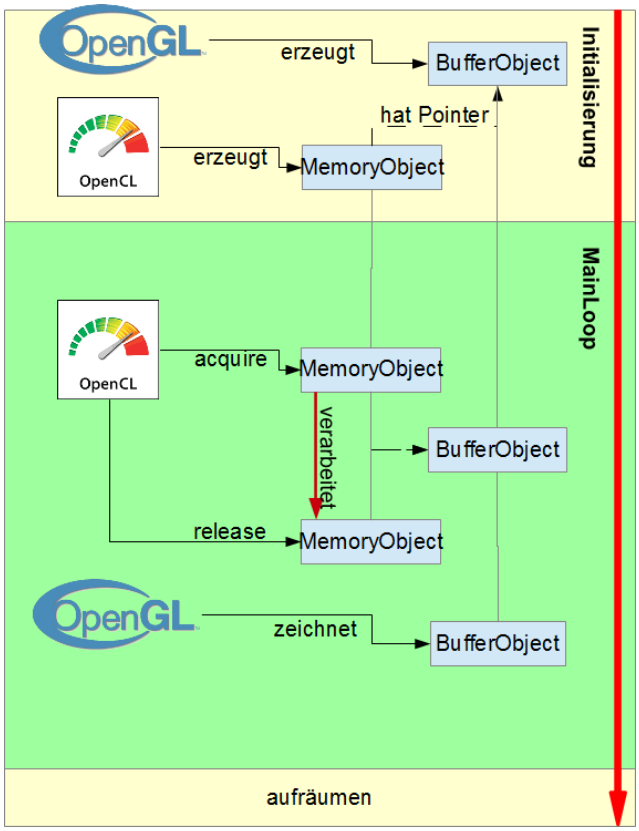
Wir betrachten in unserer Arbeit ein sehr einfaches Modell von Feuer, da physikalische oder chemische Korrektheit für eine schöne und glaubwürdige Visualisierung vernachlässigt werden können. Unser Feuer besteht aus einer festen Anzahl Partikel (im fertigen Programm 65.536 Partikel) die stetig erneuert 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 Hitzeflimmern).

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 respawned

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.



Schematisch: Zusammenspiel von OpenGL und OpenCL

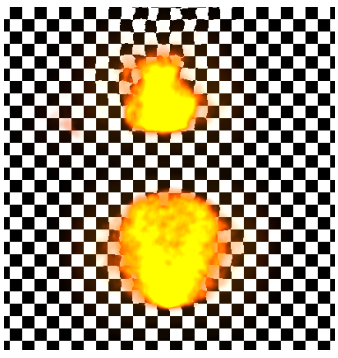
Dadurch, dass wir drei BufferObjects nutzen, ist es allerdings einfacher, die Partikel an einen OpenGL-Shader zu übergeben – hier müssen wir auf keine Strides oder ähnliches achten, was uns Fehlerquellen vermeiden und so effektiver arbeiten lässt.. Insgesamt ist das Programmieren so also einfacher gehalten, 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 Referencen 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 erneut den Zugriff auf die Daten und der Prozess geht von vorn los.

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): Spawn 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 respawned 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 recyclen, 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 FPS[1] 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 1024 sondern 65536 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 ganze 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.

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

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

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 werden, 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 verhinden 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 respawned. 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 Hitzeflimmern (siehe Abschnitt Hitzeflimmern).

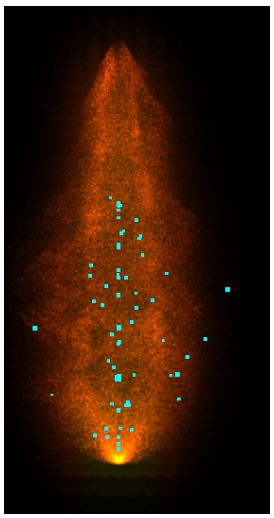
5.1 Kräfte als Richtungen

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 ausgelegte, 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 berechnen wir nun zu einer gewichteten Summe. Nach einigen Iterationen bewegen sich die Partikel so automatisch an den Punkt (0|10|0).

5.2 Low Pressure Areas



Low Pressure Areas (cyan) bestimmen die Form

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.

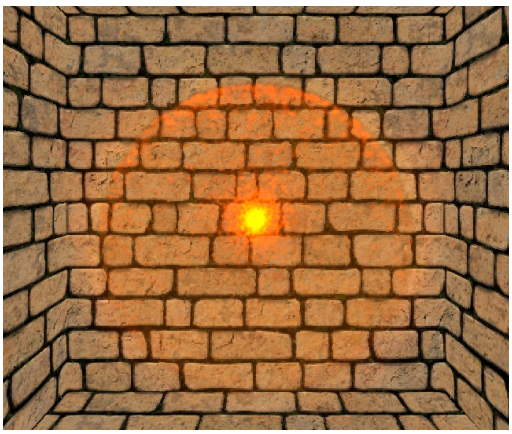
Unsere Low Pressure Areas sind Positionen im Raum, die auf mehreren Ebenen generiert werden und alle 100 ms an neue Positionen gesetzt werden. Wir verwenden in unserer Simulation 64 Low Pressure Areas, die verwendeten acht Ebenen sind mit einem Abstand von jeweils ca. 0.3 Einheiten übereinander gelegt 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.[2] 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 regelten Bahnen verweilen.

2 Der y-Wert der Low Pressure Areas größer der y-Wert der Position des Partikels garantiert eine Bewegung nach oben, da unterhalb liegende Partikel nicht mehr berücksichtigt werden.

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. Neben verschiedenen Gewichtungen (Low Pressure Areas, aktuelle Richtung, Bestreben nach oben) gilt es auch die Generierungsoptionen (Radius der Spawnarea, erste Richtungen, Lebensdauer, Low Pressure Areas, ...) zu berücksichtigen.



Falsche Parameter haben auch schöne Effekte, sind aber nicht gewünscht

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.

6 Visualisierung durch Deferred Shading

Ab hier Kristins Kram, nur grobe Idee für eine mögliche Outline

7 Post Effects

7.1 Noisetexturen

7.2 Glow

7.3 Hitzeflimmern

8 Ergebnis und Evaluation