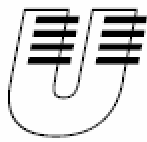


UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Simulation von Feuer mittels eines Partikelsystems

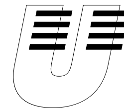
Studienarbeit
Im Studiengang Computervisualistik

Vorgelegt von
Carsten Meffert
56077 Koblenz- Arzheim Am Teebaum 4
cmeffert@uni-koblenz.de

Betreuer: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, Arbeitsgruppe Computergraphik)

Koblenz im März 2007

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de



UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Studienarbeit
des Herrn Carsten Meffert
(Matr.-Nr. 203110025)

Thema: **Simulation von Feuer mittels eines Partikelsystems**

In keinem Bereich der Informatik hat sich in den letzten Jahren die Hardware so rasant entwickelt, wie im Bereich der Computergraphik. So kann man heute mit vergleichsweise kostengünstigen Graphikkarten virtuelle Welten in einer Komplexität und Qualität darstellen, für die bis vor wenigen Jahren noch Investitionen in Millionenhöhe nötig waren. Die Herausforderung besteht nach wie vor in der Entwicklung von innovativen Applikationen und Anwendungsmöglichkeiten.
Ziel dieser Arbeit ist die Entwicklung eines Partikelsystems, das zur Darstellung bzw. Simulation von Feuer genutzt werden soll. Das Partikelsystem wird dabei von der Grafikkarte unterstützt und echtzeitfähig sein. Zunächst wird eine auf Billboards basierende Simulation erstellt, (die dann gegebenenfalls durch eine optisch aufwendigere ersetzt wird).

Schwerpunkte dieser Arbeit sind:

1. Einarbeitung in die Grundlagen von Partikelsystemen und Shaderprogrammierung
2. Entwurf des Simulators.
3. Implementierung des Programms.
4. Demonstration der Ergebnisse in ansprechender Qualität.
5. Schriftliche Ausarbeitung.

Koblenz, den 16. August 2006

- Prof. Dr. Stefan Müller-

Erklärung

	Ja	Nein
Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

Ort, Datum — Unterschrift

Inhaltsverzeichnis:

1 Einleitung 1

2 Was ist ein Partikelsystem 2

2.1 Algorithmen zur Bewegungsberechnung 2

2.2Partikelsystemvarianten 3

2.2.1 CPU basiert 4

2.2.2 Hybridsysteme 4

2.2.3 GPU basiert 4

2.2.4 Systeme mit gegenseitiger Beeinflussung 5

2.2.5 Zustandsorientiert-los 5

2.3 Billboards, Funken, animierte Texturen 6

3 Techniken 6

3.1 Das Framebufferobjekt 6

3.2 Multiple Rendertargets 8

3.3 Vertexbuffer Objekte 8

3.4 Pingpong Rendering 8

3.5 Screenfilling Polygon 9

3.6 Multiple Renderpasses 10

3.7 Die verwendeten Extensions 11

3.7.1 GL_ARB_draw_buffers 11

3.7.2 GL_ARB_texture_rectangle 12

4 Implementierung 12

4.1 Framework 12

4.2 Klassenaufbau 14

4.3 Reihenfolge der Aufrufe 15

4.4 Die Hauptschleife 16

4.5 Shader 18

4.5.1 Fragmentshader Nr. 1 19

4.5.2 Vertexshader Nr.2 21

5 Fazit 21

5.1 Leistungstest. 22

5.2 Ausblick. 24

Literaturverzeichnis..................... 26

Codelistings:

Listing 1: Die Hauptschleife bzw. die Funktion renderPasses aus der Partikelsystemklasse 18
Listing 2: Fragmentshader aus dem ersten Pass……………………………………………….. 20
Listing 3: Vertexshader des zweiten Pass ………………………………………………… . 21

Abbildungsverzeichnis:

Abbildung 1: Positionsberechnung mit Euler Methode …………………………………….. 3
Abbildung 2: Vereinfachtes Modell von Pingpong Rendering …………………………….. 9
Abbildung 3: Beispiel für Multiple Renderpasses…………………………………………... 11
Abbildung 4: Datenflussdiagramm ………………………………………………………. 14
Abbildung 5: Klassendiagramm ………………………………………………………….. 15
Abbildung 6: Feuer mit 1 Mio. Partikel …………………………………………………. 22
Abbildung 7: Frameabhängige Darstellung mit 65k, 1 Mio. und 4 Mio. Partikeln…………. 23
Abbildung 8: Performancetabelle ………………………………………………………… 24
Abbildung 9: Artefakte mit fehlenden oder schlechten Zufallswerten……………………… 24
Abbildung 10: Versuch einer geschlossenen Flamme………………………………………. 25

1. Einleitung

Partikelsysteme sind in der heutigen Zeit kaum mehr wegzudenken, da durch sie eine Vielfalt an visuellen Effekten in 2D Anwendungen realisierbar sind. Fliegende Objekte, wie ein Raumschiff, Bälle, Projektile oder die gewalttätigeren Explosionen und sogar Bluteffekte lassen sich damit in 2D aber natürlich auch in 3D darstellen. Die Geschichte der Partikelsysteme beginnt schon in den 60er Jahren, als kleine Raumschiffexplosionen mit Punkten dargestellt worden sind. Aber das wohl am meisten genannte Beispiel ist „Star Treck 2: Der Fluch des Kahn“ in dem die Genesis Demonstration mit einem Partikelsystem gerendert wurde. Schon damals kam ein hierarchisches System mit über einer Millionen Partikeln zum Einsatz.[WR83] Auch wenn bei diesem Umfang damals von Echtzeit nur geträumt wurde, hat sich in der Zwischenzeit doch einiges getan. Es wurde viel an Partikelsystemen entwickelt und mit neuer Hardware haben sich auch neue Möglichkeiten aufgetan. Die wohl wichtigste Neuerung in den 90er Jahren ist die Grafikkarte. Aber erst seitdem Teile der Fixed Funktion Pipeline programmierbar sind, kann die Grafikkarte auch für die Unterstützung von Partikelsystemen herangezogen werden. In den letzten Jahren haben sich verschiedene High Level Programmiersprachen für die Shaderprogrammierung entwickelt. Die anfängliche Assemblerprogrammierung war zu umständlich und unflexibel in der Wartung. Deshalb sind GLSL, HSLS und CG alle an C/C++ angelehnt, sodass die Einarbeitung leichter fällt. Mit ihnen können schnell und relativ simpel neue Shader geschrieben werden. Viele Partikelsysteme arbeiten immer noch ausschließlich auf der CPU, aber mit der Zeit haben die Shadersprachen ihre alten Schwächen abgelegt und neue Fähigkeiten dazu gewonnen, sodass heute auch Partikelsysteme mit GPU Unterstützung arbeiten können. Die Fähigkeit parallel Berechnungen durchzuführen und der schnellere Umgang mit Floatberechnungen sind hier Hauptargument für die Grafikkarte. Heute ist es möglich Partikelsysteme vollständig auf der GPU berechnen zu lassen.

Im Rahmen dieser Arbeit sollte ein Partikelsystem entstehen das zur Simulation von Feuer genutzt werden kann. Zudem sollte es vollständig auf der GPU berechnet werden und nicht nur eine Mischlösung sein. Die CPU übernimmt dann nur einige initiale Berechnungen und die Steuerung der Kamera. Dabei waren meine Hauptlernziele, die Shaderprogrammierung und die Funktionsweise von Partikelsystemen. Aufgrund besserer Kompatibilität habe ich mich für die Shadersprache GLSL entschieden, um gleichermaßen NVidia als auch ATI Karten bedienen zu können. Mangels einer testfähigen ATI Karte konnte die Funktionalität dahingehend allerdings nicht getestet werden. Das Framework ist in Visual Studio und C++ geschrieben um weitere Einarbeitungen zu sparen. Als Betriebssystem dient Windows XP. Weitere notwendige Vorraussetzungen, die sich im Nachhinein ergeben haben, sind die Bibliotheken glew und glut, sowie eine Shadermodell 3.0 fähige Grafikkarte die die Extensions „GL_ARB_draw_buffers“ und „GL_Rectangle_ARB unterstüzt“. Die Hohe Version wird für den Texturelookup im Vertexshader benötigt. Sie funktioniert mit aktuellen Treibern zwar auch auf älteren Karten, wird dann aber nur

emuliert und erreicht nicht die Performance wie mit richtiger Unterstützung. Empfohlen ist deshalb eine GForce 8800, Tests mit einer älteren Karte verliefen ergebnislos.

Des Weiteren werden einige Techniken gebraucht, die in Kapitel 3 näher beschrieben werden. Zuvor wird im Folgenden Kapitel zunächst einmal generell auf die Eigenschaften von Partikelsystemen eingegangen. In Kapitel 4. wird dann das entstandene Programm beschrieben und erklärt. Abschließend wird das Ergebnis betrachtet und ein kurzer Ausblick in die Zukunft gegeben.

2 Was ist ein Partikelsystem

Heute werden in der Computergrafik oft Partikelsysteme gebraucht, um komplexe Objekte darzustellen die sich über die Zeit verändern können. Polygone sind hier ungeeignet. So genannte „Fuzzy Objects“[WR83], zu Deutsch „weiche Objekte“, wie Wolken, Feuer oder Rauch sind mit Polygonen nicht adäquat wiederzugeben, der Aufwand wäre zu groß und die Anzahl der Polygone würde die Performance einbrechen lassen. Vielmehr stellen diese Objekte eine Art Volumen dar, das sich aus vielen kleinen Partikeln zusammensetzt. Wasser bei Wolken, Ruß bei Rauch etc. Ein Partikelsystem verwaltet all diese Elemente, lässt sie entstehen, bewegt sie gegebenenfalls und lässt sie auch wieder verschwinden. Jedes Partikel hat bestimmte Eigenschaften, wie Lebensdauer, Farbe, Position, Richtung, evt. Masse oder auch Größe. Alle diese Werte müssen durch das Partikelsystem für jeden Frame und für jedes Partikel einzeln berechnet werden. Es terminiert wenn alle Partikel verschwunden sind oder die Anwendung manuell beendet wird. Um die zufällige Erscheinung zu verstärken oder um mehr Kontrollmöglichkeiten zu haben, können Partikelsysteme auch in verschiedenen Hierarchiestufen organisiert sein. In diesem Fall sendet die Quelle der ersten Ebene keine Partikel aus sondern weitere Partikelquellen, die dann die zweite Ebene bilden. Diese Technik wurde in der Genesis Demonstration angewandt, um die Quellen der 2. Ebene zufällig auf der Planetenoberfläche zu verteilen. Mit Hierarchischen Systemen kann schnell eine große Zahl gleichförmiger Objekte verwaltet werden, ob in einem großen Objekt wie einer Quellwolke oder in vielen kleinen Objekten z.B. Schäfchenwolken.

2.1 Algorithmen zur Bewegungsberechnung

Ein Problem das sich nicht geändert hat ist die Wahl des richtigen Algorithmus. Bei innovativen Anwendungen müssen oft ganz neue Algorithmen geschrieben werden, die speziell an die Anforderungen angepasst sind. Etwa zur Kontrolle von Hierarchieebenen oder vielleicht der gegenseitigen Abstoßung von Partikeln. In „traditionellen“ Anwendungsbereichen gibt es dagegen teilweise mehrere Berechnungsformeln zur Auswahl. Allen Partikelsystemen gemein ist die Notwendigkeit die Partikel zu bewegen. Da sie meist eine Kurve im Raum beschreiben und sich nicht im Zickzack bewegen, müsste eigentlich mit Integralen gearbeitet werden, um

Geschwindigkeit und Beschleunigung zu berechnen. Bei so vielen Punkten ist das nicht praktikabel, also müssen Annäherungen gefunden werden. Die Euler Methode berechnet die neue Position nur anhand der alten Geschwindigkeit und einem Zeitintervall. Ebenso wird die Geschwindigkeit nur mit der alten Geschwindigkeit, der Summe aller Kräfte und dem Zeitintervall berechnet. Das Ergebnis ist abhängig von der Größe des Zeitintervalls. Nur kleine Werte liefern ein gutes Ergebnis, sodass die Framerate hoch sein muss. Dadurch wird das System schnell instabil wenn Performanceschwankungen auftreten. Eine alternative Berechnungsmethode nach Runge Kutta arbeitet mit mehreren unterstützenden Gradienten, die aufsummiert und dann gemittelt werden. [SM06] Weitere Ansätze sind Leapfrog, Verlet und Heun. Aufgrund ihres minimal aufwändigen Algorithmus fand im Rahmen dieser Arbeit die Euler Methode Anwendung.

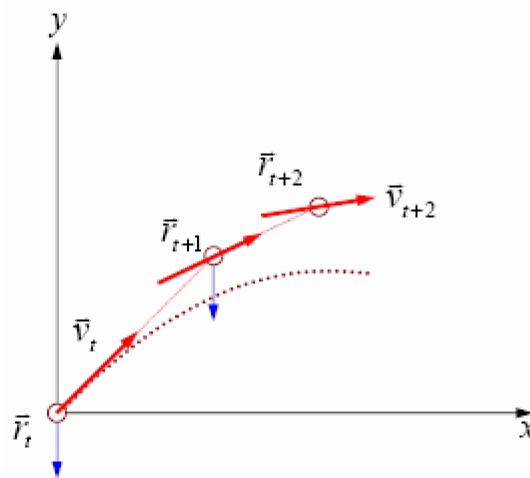


Abbildung 1: Positionsberechnung mit Euler Methode [SM06]

2.2 Partikelsystemvarianten

Durch Verlagerung von Teilen des Systems auf die GPU, ergeben sich verschieden Möglichkeiten Partikelsysteme umzusetzen. Jede hat ihre eigenen Vor- und Nachteile. Aber nicht nur darüber, auf welchem Prozessor die Partikelsysteme laufen, lassen sie sich unterscheiden. Ein davon unabhängige aber grundlegende Unterscheidung liegt in der Art der Berechnung der Physik. „Zustandslose“ bzw. „Zustandsorientiert“ beschreibt dieses Verhalten. Des Weiteren bestimmt auch der Verwendungszweck maßgebend das Physikalische Modell. Partikel können je nach Aufgabe Einfluss aufeinander ausüben. In den folgenden Kapiteln werden verschieden Arten von Systemen eingeführt.

2.2.1 CPU basiert

Wie schon erwähnt ist das naheliegenste Vorgehen ein Partikelsystem auf der CPU zu implementieren. Dafür muss eine eigene Datenstruktur erstellt werden, sowie Algorithmen die Berechnungen darauf ausführen. Zusätzlich zu den Speicherzugriffen muss die Datenstruktur noch sortiert werden, falls ein flexibles System angestrebt ist, damit bei Bedarf auch nur ein Teil der Partikel angesprochen werden kann. So kann in größerer Entfernung die Anzahl der Partikel reduziert werden, oder wenn verschiedene Effekte mit einem System genutzt werden sollen, können kleine Effekte ausgeführt werden ohne das gesamte System zu beanspruchen. Temporäre Effekte können in einer Schleife aufgerufen werden. Was die CPU angeht, sind die zur Positions- und Richtungsberechnung nötigen Floatwerte eine zusätzlich Belastung für das System, neben den obligatorischen caching Operationen und dem Steuern des Betriebsystems, etc.

2.2.2 Hybridsysteme

Sie sind ein Kompromiss aus CPU und GPU. Hier wird die Datenstruktur wieder im Hauptsystem erstellt, nur die Berechnung wird auf den Vertexshader verlagert. Der Nachteil an dieser Variante ist der ständige Datentransfer, weil die im Hauptspeicher gespeicherten Daten in jedem Frame aufs Neue an die GPU übergeben werden müssen. Will man das reduzieren und übergibt die Daten nur einmal, nimmt man damit in Kauf, die Partikel nicht mehr dynamisch bewegen zu können. Jede Bewegung basiert dann auf den einmal am Anfang gespeicherten Daten, was bei zunehmender Laufzeit eine wachsende Abweichung vom Realismusgrad zur Folge hat. Aber schon die Hybridlösung liefert einen beträchtlichen Performancezuwachs gegenüber der CPU.

2.2.3 GPU basiert

Will man auch auf der GPU die Partikel dynamisch berechnen ohne ständig Daten auszutauschen muss fast das gesamte Partikelsystem auf die Grafikkarte ausgelagert werden. Die Daten speichert man dann in Texturen und übergibt sie einmal zu Anfang. Die Vektorkoordinaten XYZ entsprechen dann den Farbwerten RGB. Überschreitet die Anwendung das Limit der Grafikkarte für Texturen im Speicher müssen die Texturen trotzdem pro Frame geladen werden. In einer sehr detaillierten Szene mit vielen verschiedenen Objekten kann das schnell geschehen. Sind die nötigen Daten geladen werden die Berechnungen mit dem so genannten Pingpong Rendering durchgeführt. Mit dieser Technik werden die Ergebnisse der Berechnung auch gleichzeitig gespeichert, so dass sie als Grundlage für den nächsten Frame dienen können. Nach der Berechnung wird nur ein weiterer Renderpass notwendig um die Ergebnisse darzustellen. Möglich wird das erst durch Texturzugriffe im Vertexshader, also Shadermodell 3.0 und Multiple Rendertargets, da es zu umständlich wäre alle Partikeldaten in einer Textur unterzubringen.

2.2.4 Systeme mit gegenseitiger Beeinflussung

Mit dem Begriff des Fuzzy Objects ist noch eine andere Möglichkeit der Partikelsysteme verbunden. Versieht man die Partikel mit Masse und verbindet sie mit Federn untereinander, können so die verschiedensten Stoffe visualisiert werden. Aber Federn allein reichen nicht. Damit eine Fahne nicht in den Fahnenmast hineinfällt oder in sich selbst, muss dem Partikelsystem ein Kollisionserkennung hinzugefügt werden. Eine einfache Kollision mit der Umgebung reicht nicht, wie es z.B. bei Rauch der Fall wäre. Die Partikel müssen untereinander getestet werden. Basierend auf dem Prinzip der Molekulardynamik haben die Partikel auf sehr nahe Distanz eine abstoßende Wirkung und auf lange Distanz eine Anziehende. Überzieht man ein Objekt mit einem Netz aus Partikeln kann so, sehr realistisch, die Kollision von komplexen Objekten berechnet werden. Der zusätzliche Aufwand der Kraftberechnung für die Federn und die verhältnismäßig komplexere Physikberechnung verlangt diesen Systemen noch mal mehr Rechenleistung ab. Zudem sind die mathematischen Formeln sehr viel komplexer, da die Partikel sich auch untereinander abstoßen können müssen, was bei Wolken oder Feuer ja nicht der Fall ist. Kollisionserkennung für mehrere Tausend Partikel ist wohl das größte Problem in diesem Bereich von Partikelsystemen und ein ganzes Thema für sich.

2.2.5 Zustandsorientiert-los

Das physikalische Modell eines Partikelsystems kann auf zwei grundlegenden Arten arbeiten. Die einfachere ist die zustandlose Berechnung. Dabei gibt es einen Startzustand von dem ausgehend alle weiteren Zustände berechnet werden. Aber da sich der Richtungsvektor eines Partikels über die Zeit ändern kann, bei einem Zustandslosen System aber immer der Selbe genommen wird, ergeben sich schnell Abweichungen vom realistischen Verhalten. Der Vorteil ist eine einfachere Programmstruktur. Die Partikel werden nur mit den initialen Werten und dem Zeitintervall berechnet, das heißt das ein Großteil der Speicheroperationen wegfällt und die Partikel so direkt gerendert werden können.

Will man hingegen korrekte Berechnungen, sodass Partikel auch mal an Hindernissen abprallen können oder unterwegs zufällig ihre Richtung ändern, muss man für jeden Frame die berechneten Ergebnisse zwischenspeichern. Damit ist die Größe des Partikelsystems bei der Erstellung des Speichers limitiert. Dadurch dass bei einem zustandslosen System die Berechnung so einfach ist, können andererseits auch nur sehr einfache Effekte erzeugt werden. Prädestiniert dafür ist eine Explosion, Funkensprühen oder ein einfacher Wasserfall, wie aus einem Hahn, oder ähnliches. Wolken, springende Bälle oder gar Strömungssimulationen benötigen auf jeden Fall ein Zustandsorientiertes System.

2.3 Billboards, Funken, animierte Texturen

Billboards sind eine alte Technik, um mit einfachen Mitteln Objekte schöner aussehen zu lassen. Anstatt ein Objekt mit Polygonen zu modellieren wird es mit einer einzigen Textur dargestellt. Diese richtet sich immer nach der Kamera aus sodass das Bild immer von vorne zu sehen ist. Auffällig wird das, wenn sich das Billboard im Augenwinkel bzw. am Bildschirmrand bewegt und natürlich wenn man darauf achtet. Eine fortschrittlichere Art von Billboards sind Sprites. Sie funktionieren im Prinzip genau wie Billboards nur das anstatt 4 Vertices und 2 Polygonen nur ein Vertice gebraucht wird, der das Zentrum der Textur einnimmt, um es darzustellen. Das ist vor allem bei Partikelsystemen mit hoher Partikeldichte ein enormer Vorteil, bei anderen Anwendungen fallen ein paar Polygone mehr nicht so sehr ins Gewicht.

Feuer wird in aktuellen Grafikanwendungen meist mit Hilfe von animierten Texturen dargestellt. So eine Folge von Bildern erzielt ein schon recht ansehnliches Ergebnis und kostet nicht viel Rechen oder Programmieraufwand. Dafür sieht das Feuer aber auch immer gleich aus, die Sequenz wiederholt sich nach wenigen Sekunden. Partikelsysteme werden dann nur eingesetzt um Funken dem Feuer hinzuzufügen. Funken sind technisch nur eine Interpolation von der alten zur neuen Position. Das Gleiche funktioniert auch für Explosionen.

3 Techniken

In diesem Kapitel wird etwas näher auf die verwendeten Techniken eingegangen, die nötig sind, um das im Rahmen dieser Arbeit entstandene Programm umzusetzen. Dabei werden nur Techniken behandelt, die in direktem Zusammenhang mit dem Partikelsystem stehen. Dinge wie Kamerasteuerung, Vektorberechnung oder Texturen zu laden wird hier vernachlässigt.

3.1 Das Framebufferobjekt

Etwas älter aber in ihrer Funktionsweise sehr ähnlich sind sozusagen die Pixelbufferobjekte die Grundlage für Framebufferobjekte. PBO’s sind eine Extension, die genau wie das normale OpenGL auf den Buffern Depth, Color, Stencil und Accumulation des Windowsrendering arbeiten, mit dem Unterschied, dass ihr Output nicht unmittelbar sichtbar ist. Das Ergebnis wird nicht in den Framebuffer geschrieben sondern in einen der genannten Buffer. Das Pixelformat muss aus einer Liste mit Konstanten ausgewählt werden, weshalb jedes Objekt einen eigenen Kontext braucht um Konflikte untereinander zu vermeiden. Um in eine Textur rendern zu können ist auch hier eine weitere Extension nötig. Neben den Windows spezifischen Buffern sind zusätzliche auxilliary Buffer verfügbar.

Das Framebuffer Objekt ist eine wesentlich bessere Alternative als das konventionelle Pixelbuffer Objekt. Neben den bekannten Puffern wie Stencil, Depth und Colorbuffer wird bald wohl auch die Unterstützung für Accumulationbuffer hinzugefügt. Wichtiger im Zusammenhang mit dieser Arbeit, und auch vorrangig für die Bedeutung der FBO'S, ist aber die Fähigkeit direkt in eine Textur zu rendern, bzw. multiple rendertargets angeben zu können. Zuvor musste, um das zu erreichen, mit dem Befehl „glCopyTexImage()“ das Bild beziehungsweise das Rechenergebnis aus dem Framebuffer in eine Textur kopiert werden. Das ist umständlich und kostet unnötig Rechenleistung. Benutzt man Multiple Rendertargets müssen die Partikeldaten nicht alle in einer einzigen Textur abgelegt werden, sondern können über Texturkoordinaten in der jeweiligen Textur abgerufen werden und dann auch wieder in eine entsprechende Textur hineingeschrieben werden. Dadurch dass keine Daten mehr auf dem Hauptspeicher zwischengelagert werden, entfallen überflüssige Bustransfers und die Performance verbessert sich. Ohne die FBO's müssten die Daten auf anderem Weg aktualisiert werden, zum Beispiel mit mehr Renderpasses. Das würde bedeuten, dass pro Textur ein weiterer Renderpass anfallen würde, also insgesamt 3. Und das bezieht sich nur auf die aktuelle Datenstruktur. Mehr Daten pro Partikel, etwa ein Massewert und vielleicht noch ein Spin oder ähnliches, würden mehr Texturen zum Speichern benötigen und damit in mehr Renderpasses und weniger Performance resultieren. Nicht so mit den FBO's, ohne weiteres könnten mehr Texturen den Rendertargets hinzugefügt werden, natürlich hat auch das ein Limit, aber für die meisten Anwendungen sollte es reichen.

Ein weiterer Vorteil ist die einfache Handhabung. Es ist kein Kontext wie bei den Pixelbufferobjekten mehr nötig. Somit auch kein Wechsel zwischen diesen, wenn mit mehreren Objekten gearbeitet wird. Für die Erstellung sind im Normalfall nur wenige Befehle nötig, was die Handhabung zusätzlich erleichtert. Rendertargets werden einfach als Attachment dem FBO hinzugefügt. Wichtig zu erwähnen ist, das FBO's erst durch eine Extension

"GL_EXT_framebuffer_object"

zugänglich gemacht werden müssen, die aber in der glew Bibliothek mit enthalten ist. Eine weitere Einschränkung ist das Texturformat. Jede verwendete Textur in einem FBO muss dieselben Maße haben. Werden Texturen als Attachments genommen müssen diese vorher schon erstellt worden sein, das kann auch noch geschehen während das FBO schon aktiv ist, Hauptsache es geschieht vor dem „attach“ Befehl. Vorteil an dieser Stelle ist das das Format nicht extra definiert werden muss, es wird einfach durch die verwendete Textur festgelegt. FBO's rendern nur in 2D Bilder, 3D rendertargets sind nicht erlaubt. Anwendung finden FBO's zum Beispiel bei dynamischen Texturen, Reflektionen, Anti Aliasing, Motion Blur, Image Processing und natürlich GPGPU.

3.2 Multiple render Targets

Im normalen Ablauf der Fixed Function Pipeline wird am Ende ein 2D Bild in den Framebuffer gerendert. Auch Fragmentshader schreiben normalerweise nur in einen 2D Ausgabearray genannt „FragColor[]“. Manche Anwendungen verlangen aber das Schreiben in mehrere Bilder bzw. Texturen. In dem Fall müssen entweder mehrere Renderpasses gerechnet werden, oder man arbeitet mit Bufferobjekten. Im ersten Fall wird die Fixed Function Pipeline X-mal durchlaufen und erst am Ende alle Ergebnisse verwendet. Mit der entsprechenden Extension schreibt man gleichzeitig in mehrere Texturen und die Fixed Function Pipeline wird nur einmal durchlaufen. Das ist je nach Betriebsystem oder verwendeter Programmierumgebung verschieden. Zusätzlich limitiert die Hardware die maximale Anzahl zulässiger Rendertargets. In meinem Programm werden zwei Rendertargets gebraucht weil jedes Partikel acht Byte Daten speichert, also zwei RGBA Texturen. Für abstrakte Berechnungen im Sinne der GPGPU sind Multiple Render Targets sehr gut geeignet, finden aber auch Anwendung im grafischen Bereich.

3.3 Vertexbuffer Objekte

Obwohl sie ähnlich klingen wie die oben erwähnten Framebuffer Objekte haben sie nicht viel gemeinsam. Sogar ihre Konzepte, bzw. die Handhabung unterscheiden sich, sodass eine Verbindung nicht unbedingt erwartungskonform ist. VBO’s sind seit openGL1.5 ein Core- Feature von diesem. Sie sind eine Art Erweiterung der Vertexarrays und funktionieren ähnlich wie Displaylisten. Ihre Daten werden im VRAM der Grafikkarte abgelegt, so müssen sie nicht im Hauptspeicher zwischengespeichert werden. Damit verbinden VBO’s die Flexibilität der Vertexarrays mit der Geschwindigkeit von Displaylisten. Die Daten in einem VBO können auf viele verschiedene Weisen angeordnet sein. Eine Möglichkeit ist z.B. diese in einem Record anzuordnen, der pro Vertice mit Daten belegt werden muss[SW06-2], oder man nimmt einfach ein 1D Array und übergibt dieses als Input an das VBO. Dadurch das man sich seine eigene Struktur überlegen kann, die ein Vertexarray speichert, ist es sehr vielseitig einsetzbar. Im Vertexshader muss dann nur entsprechend auf die Daten zugegriffen werden. Die Verantwortung liegt also beim Programmierer, genau wie beim FBO, bei dem man selber darauf achten muss, dass die Formate stimmen.

3.4 Pingpong Rendering

Ein großes Problem bei GPGPU Algorithmen und vielen Bild bearbeitenden Algorithmen ist die Unfähigkeit gleichzeitig aus einer Quelle zu lesen und darauf zu schreiben. In Shadern muss Input und Output streng getrennt sein, aufgrund der Read- und Writeonly Zugriffe. Um diese Problematik zu umgehen bzw. elegant zu lösen, muss ein Trick angewendet werden, den man Pingpong Rendering nennt. Um in einem Rechenschritt die berechneten Daten aus dem vorhergehenden

Schritt nutzen zu können, müssen Input und Output jedes Mal getauscht werden. Man könnte natürlich in jedem Frame die Shader wechseln oder If- Abfragen im Shader implementieren, was allerdings die Codemenge verdoppeln würde und die Anwendung ausbremst. Die bessere alternative ist, man übergibt einfach bei jedem Frame abwechselnd die benötigten Uniformparameter als Input. Um jetzt auch den Output dynamisch wechseln zu können ohne den Shader zu wechseln, nimmt man zwei statt einem FBO und wechselt diese ebenfalls pro Frame ab. So genügt ein Shader, um, im Bezug auf Lesen und Schreiben, 2 „verschiedene" Dinge zu tun. Konkret heißt das: im ersten Frame wird Input1 an den Shader übergeben, der dann in FBO2 rendert. Im zweiten Frame wird dann jedoch Input2 an den Shader übergeben, der wiederum in FBO1 rendert. Da die FBO's ohne großen Aufwand und sehr performant gewechselt werden können, sind sie prädestiniert für diese Aufgabe. Weil der Shader automatisch in das aktive FBO rendert, muss diesem nur einmal während der Erzeugung das Rendertarget mitgeteilt werden. Gleiches gilt für multiple Rendertargets.

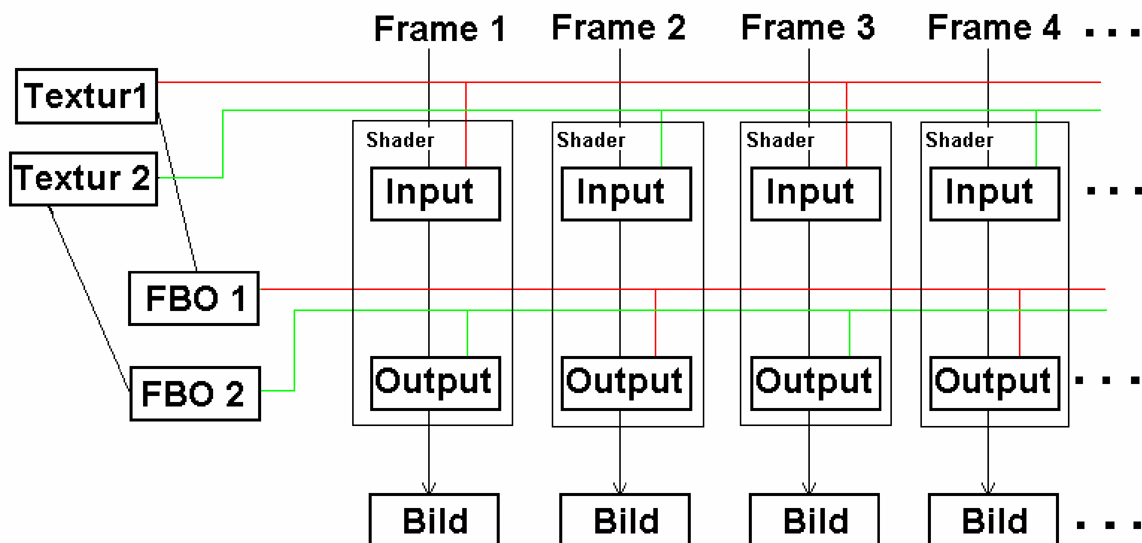


Abbildung 2: Vereinfachtes Modell von Pingpong Rendering

3.5 Screenfilling Polygon

Eigentlich eine sehr einfache Maßnahme, aber dennoch essentiell, ist die Verwendung eines Screenfilling Polygon bei jeder Art von Offscreenrendering oder Visualisierungen deren Input nicht direkt aus der Rendering-Pipeline kommt. Im Prinzip alle GPGPU Anwendungen benötigen eines. Das Problem mit Fragmentshadern ist, dass nicht auf beliebige Objekte oder Daten zugegriffen werden kann, sondern man muss sich an ihre feste Arbeitsweise anpassen. Fragmentshader werden nach der Rasterisierung pro Bildschirmpixel der Anwendung aufgerufen. Um zu gewährleisten, dass

die Texturkoordinaten eindeutig definiert sind und sich nicht ändern, benötigt man ein Screenfilling Polygon. Dabei muss noch nicht einmal etwas auf dem Polygon dargestellt werden geschweige denn darauf gerechnet. Einzig die Texturkoordinaten sind wichtig. Jeder Pixel muss einer Position in der Textur entsprechen mit deren Daten gerechnet werden soll, und das jeden Frame aufs Neue. Um Abweichungen zu vermeiden sind deshalb Rotationen und Translationen auf das Polygon nicht anzuwenden. Seine Position muss unveränderbar sein. Indem man vorher die Modelviewmatrix und die Projektionsmatrix mit der Einheitsmatrix gleichsetzt und den Projektionsmodus auf orthografisch wechselt, gewährleistet man eine optimale Nutzung. Danach können alle Zustände wieder zurückgesetzt werden, z.B. für den nächsten Pass.

3.6 Multiple Renderpasses

Die Bezeichnung für einen Durchlauf der Rendering Pipeline ist Renderpass, egal ob dabei Shader benutzt werden oder nicht. Aber nicht notwendigerweise muss nach einem solchen Durchlauf auch ein Bild auf dem Bildschirm erscheinen. Um z.B. dynamische Texturen zu erzeugen kann man eine Szene in den Framebuffer rendern, diese in eine Textur kopieren und dann, in einem zweiten Pass, diese Textur in einer anderen Szene, etwa auf einem virtuellen Bildschirm innerhalb der Szene, verwenden. Auf dem virtuellen Bildschirm ist dann immer zu sehen was gerade in der anderen Szene geschieht. Wie ein Rückspiegel in einem Autorennen oder Bilder einer Überwachungskamera. Aber auch die GPGPU Gemeinschaft verwendet multiple Renderpasses. In einem Pass, der nicht zu sehen ist, können dann Berechnungen angestellt werden, die im zweiten Pass dann angezeigt werden können. Z.B. die Anwendung von BV-Filtern oder in meinem Beispiel Vektorberechnungen. Wenn der Shader nur für Matrizzenmultiplikation eines Mathemathik- programms benutzt wird, kann auch gar nichts von den Berechnungen auf dem Bildschirm landen. Pro Frame sind beliebig viele Passes möglich, nur die Leistungsfähigkeit der Hardware setzt hier ein Limit. Zwischen jedem Pass sollte der Color- sowie der Depthbuffer gelöscht werden, sonst überschreibt man die alte Szene mit der neuen und beides ist zu sehen. Um Performance zu sparen wird oft im Offscreenrenderpass die Größe des Viewport verkleinert, da später in der eigentlichen Szene die vorher berechnete Textur nur sehr klein dargestellt wird. Bei abstrakten Berechnungen hängt die Größe des Viewport von der Aufgabe ab, die der Pass durchzuführen hat.

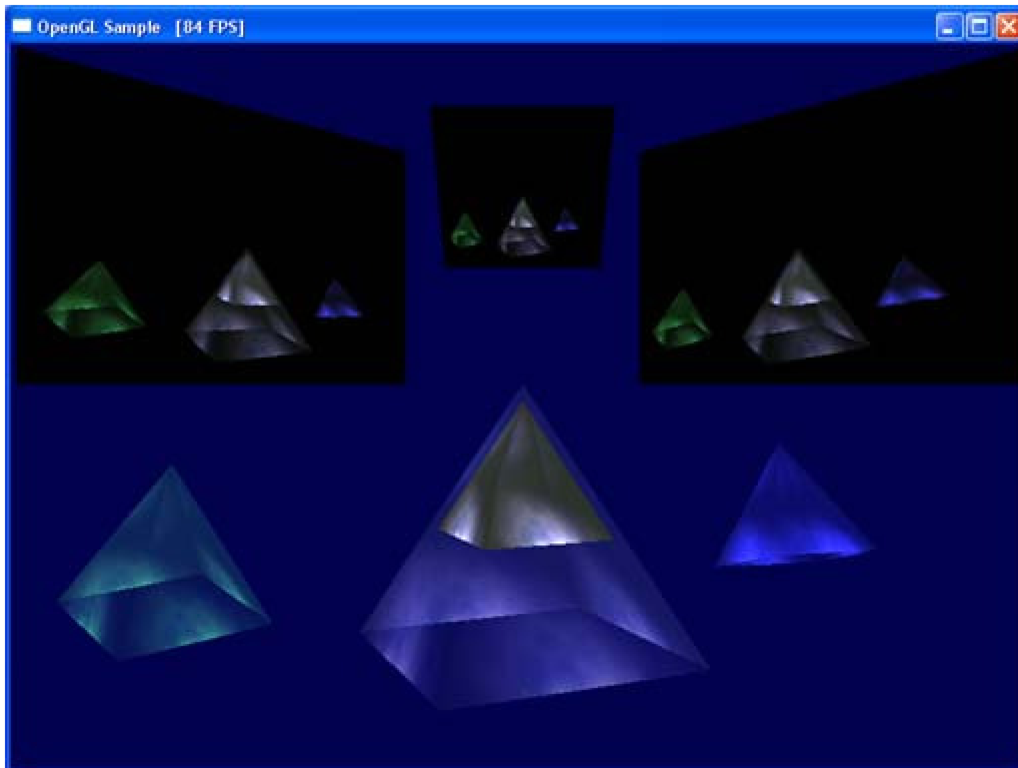


Abbildung 3: Beispiel für Multiple Renderpasses [FS06]

3.7 Die verwendeten Extensions

Durch Einbindung der openGL Extension Wrangler Library, kurz glew, und der Verwendung von openGl 2.0 sind schon die meisten Extensions abgedeckt. FBO's und VBO's gehören z.B. dazu. Aber es wurden auch 2 Extensions verwendet die separat eingebunden werden müssen.

3.7.1 GL_ARB_draw_buffers

Diese Extension ist zwar seit openGL 1.5 mit dabei, muss aber im Shadercode des Fragmentshaders noch mal extra eingefügt werden. Sie ermöglicht das Schreiben in mehr als einen Colorbuffer, bzw. das Schreiben in verschiedene Texturen. Das Rendertarget wechselt von GL_FragColor zu GL_FragData[i]. Der alte Colorbuffer befindet sich dabei an Stelle i = 0, wenn er nicht anders definiert wurde. Mit dieser Extension kann auch in die oben eingeführten FBO's gerendert werden. Welches Target hinter welchem i steckt wird im FBO definiert.

Im Shader muss die Extension mit dem Befehl

```
#extension GL_ARB_draw_buffers : enable
```

noch vor der Mainmethode und vor den Variablendeklarationen includiert werden.

3.7.2 GL_ARB_texture_rectangle

Auch diese Extension muss im Shadercode neben „GL_ARB_draw_buffers“ includiert werden. Normale Texturen werden ausnamslos in den Wertebereich [0,1] geclampt. Das würde die Berechnungen mit Vektoren in Weltkoordinaten extrem erschweren. Innerhalb des Shaders müssten die Daten in einen geeigneten Wertebereich skaliert werden, etwa [0,1000]. Damit verbunden sind allerdings Multiplikationen und Divisionen nach dem Einlesen und vor dem Ausgeben. Trotz allem wäre der Aktionsradius wesentlich mehr beschränkt als bei normalen Floatwerten. Bei größeren Dezimalverschiebungen werden die Berechnungen zunehmend ungenau, weshalb dieses Vorgehen nicht zu empfehlen ist. Um Texturen verwenden zu können die nicht geclampt werden gibt es diese Extension. Nicht nur, dass Float und Double in verschieden Bittiefen verfügbar sind, auch die Ausmaße der Texturen ist weniger Grenzen unterworfen. So sind Texturen dieser Extension nicht an Zweierpotenzen gebunden (NPOT), im Gegensatz zu den normalen Texturen. Für mein Programm bedeutet das, dass ich fast jede beliebige Anzahl an Partikeln innerhalb des Wertebereichs darstellen kann, und nicht nur eine feste Auswahl an Mengen, die bei großen Werten keinen Spielraum lässt. Das Maximum bei einer GForce 8X00 liegt bei über 16 Mio. Partikeln.

4 Implementierung

Da nun die wichtigen Grundprinzipien die in dieser Arbeit Anwendung gefunden haben erklärt wurden, kann dazu übergegangen werden die eigentliche Funktionsweise näher zu erläutern. Auch hier wird wieder eine Auswahl getroffen und nur die Kernstücke des Programms ausführlich dargestellt. Im Folgenden wird nach einem Gesamtüberblick über das Framework, die Klassenstruktur und den Programmablauf die Funktion der Hauptschleife und der Shader näher erklärt, die die Hauptlast des Programms tragen. Andere Teile des Partikelsystems dienen der Vorbereitung und der Zuarbeit zu diesen beiden Programmteilen.

4.1 Das Framework

Erst im Verlauf der Recherchen verdeutlichte sich das genaue Konzept der Arbeit. Aufgrund des Artikels GLSL Partikel, der das Prinzip erklärte aber nicht sehr vollständig war, sollte das Partikelsystem komplett auf der GPU untergebracht werden [DGLP]. Es wäre zwar auch ein zustandsloses System möglich gewesen, aber sinnvoller wäre das in einer gemischten Variante umgesetzt worden. Aus diesem Grund wurde dies nur als Ausweichoption gesehen, falls die gesetzten Ziele nicht erreicht werden könnten. Die Komponenten für beide sind quasi dieselben, nur die verwendeten Algorithmen zur Berechnung der Physik unterscheiden sich mitunter sehr. In einem zustandslosen System, das die neuen Werte nicht speichert, kann auch die Lebensdauer der Partikel nicht mitgezählt werden. Also muss die an der Laufzeit des Programms abgelesen werden. Dann ist allerdings unklar wann ein Zyklus vorbei ist,

die Laufzeit steigt ja konstant an. Wird das initiale Alter mit dem erwünschten Zeitintervall verglichen, können Partikel, deren festes Alter vom Zeitintervall überholt wird, mit einer kurzen Abfrage als abgelaufen betrachtet werden. Wird das Zeitintervall regelmäßig zurückgesetzt, können zwar neue Partikel entstehen, aber durch das ständige zurücksetzen des Zeitindex ist auch die Kräfteberechnung nicht kontinuierlich. Also viel die Entscheidung auf das mächtigere System. Trotz der ähnlichen Komponenten reicht eine einfache Erweiterung aber nicht aus. Bei einem zustandserhaltenden System ist es nötig die Daten auf einer Grafikkarte gleichzeitig lesen und schreiben zu können. Da dies nicht ohne weiteres möglich ist, muss mit einem kleinen Trick gearbeitet werden. Dabei hilft das so genante Pingpong- Rendering, der wichtigste Bestandteil des Programms. Damit werden die Berechnungen des Partikelsystems koordiniert. D.h. die Shader bekommen ihre Uniformvariablen übergeben und das entsprechende FBO wird jeden Frame neu aktiviert. Direkt im Anschluss werden im zweiten Renderpass anhand des VBO und einem zweiten Satz Shader die Punkte gezeichnet und an die vorher berechneten Positionen verschoben.

Die Entscheidung, vollständig auf der GPU zu arbeiten, hatte leider zur Folge, dass die ursprünglich beabsichtigten Pointsprites nicht mehr ins Konzept passten. Bei der erreichten Masse an Partikeln sind Punkte von einem Pixel Größe schon zu groß. Im Nachhinein stellte sich auch der Aufwand, vor allem was die anderen Komponenten betrifft, als höher heraus als erwartet. So konnten, auch aus Zeitmangel, Pointsprites in dieser Arbeit leider nicht umgesetzt werden.

Damit das System überhaupt starten kann sind zwei Texturen mit initialen Daten gefüllt, die durch die Random-Funktion verrauscht werden. Erstellt werden diese während der Laufzeit im Hauptprogramm, nachdem die Shader geladen wurden. Aus Mangel an einem geeigneten Zufallsgenerator innerhalb der Shader ist das auch bis jetzt die einzige Möglichkeit. Zufallsvariablen könnten von außen nur pro Frame übergeben werden und sind damit für alle Partikel die in diesem Frame starten dieselben, was zu sehr auffälligen Artefakten führt. Um das Ergebnis besser beurteilen zu können wurde dem Programm eine Kamerasteuerung hinzugefügt, die um die Partikelquelle rotiert, sowie ein einfacher FPS Zähler.

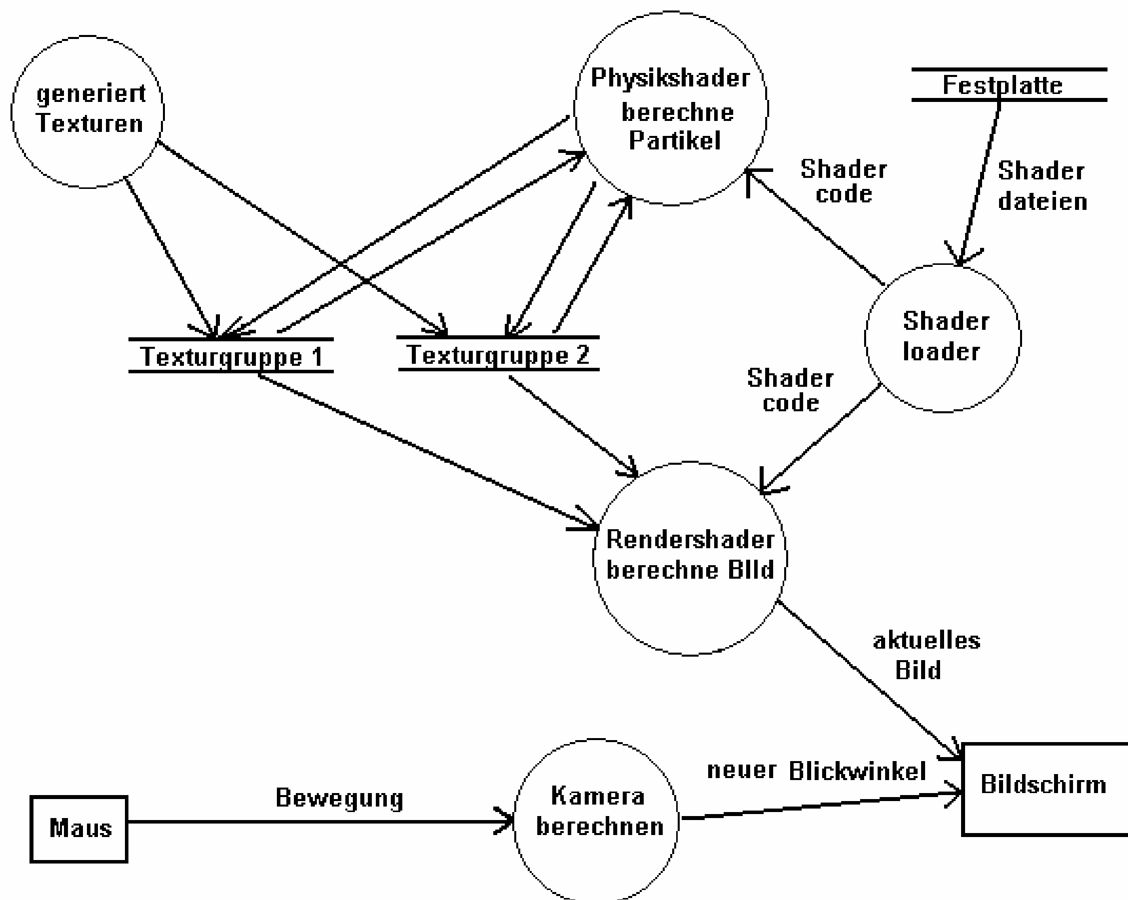


Abbildung 4: Datenflussdiagramm

Vom Prinzip ist das Partikelsystem eine GPGPU Anwendung. Der Grafikprozessor wird nur nebenbei noch zur Darstellung gebraucht. Der Großteil der Rechenleistung wird für abstrakte Vektorberechnungen benötigt, die nicht sofort zu visualisieren sind. GPGPU ist theoretisch seit 2000 möglich, aber erst durch einen Kurs auf der Siggraph 2004 und einem Kapitel in GPU Gems2 wurde es wirklich bekannt. Natürlich ergaben sich erst durch die fortschreitende Hardwaretechnik weit reichende Möglichkeiten bei der Implementierung.

4.2 Klassenaufbau

Der Klassenaufbau des Partikelsystems ist an und für sich relativ einfach. Es gibt im Wesentlichen nur 3 Klassen: Shader, Textur und Partikelsystem, von denen jeweils mindestens ein Objekt in der Hauptklasse angelegt werden muss. Shaderobjekte werden pro Partikelsystem 2 Mal benötigt. Die Klasse Partikelsystem ist das Herzstück des Programms, hier laufen alle Stricke zusammen bevor sie an die „main“ übergeben werden. Für die Handhabung der FBO's und des VBO wurden keine extra Klassen geschrieben, obwohl das möglich gewesen wäre. Der Nutzungsumfang beschränkt sich hier auf eine Methode in der die Objekte initialisiert werden und eine handvoll Befehle zur

Ausführung in der Hauptschleife. Die Kamerasteuerung wurde in eine eigene Klasse ausgelagert, um die Main so einfach wie möglich zu halten. Es ist möglich um den Ursprung des Weltkoordinatensystems zu rotieren sowie die Distanz zu verändern. Es existiert auch ein Klasse die Vektorberechnungen steuert. Sie wird aber bis jetzt nur für die Kamera gebraucht, da die Vectorberechnungen der Partikel auf der GPU durchgeführt werden und GLSL die nötigen Methoden bereits beherrscht. Zur Vorberechnung globaler Kräfte und zur Beeinflussung der Partikelquelle kann die Klasse aber noch herangezogen werden, sowie zur Berechnung der Ausrichtung von Billboards.

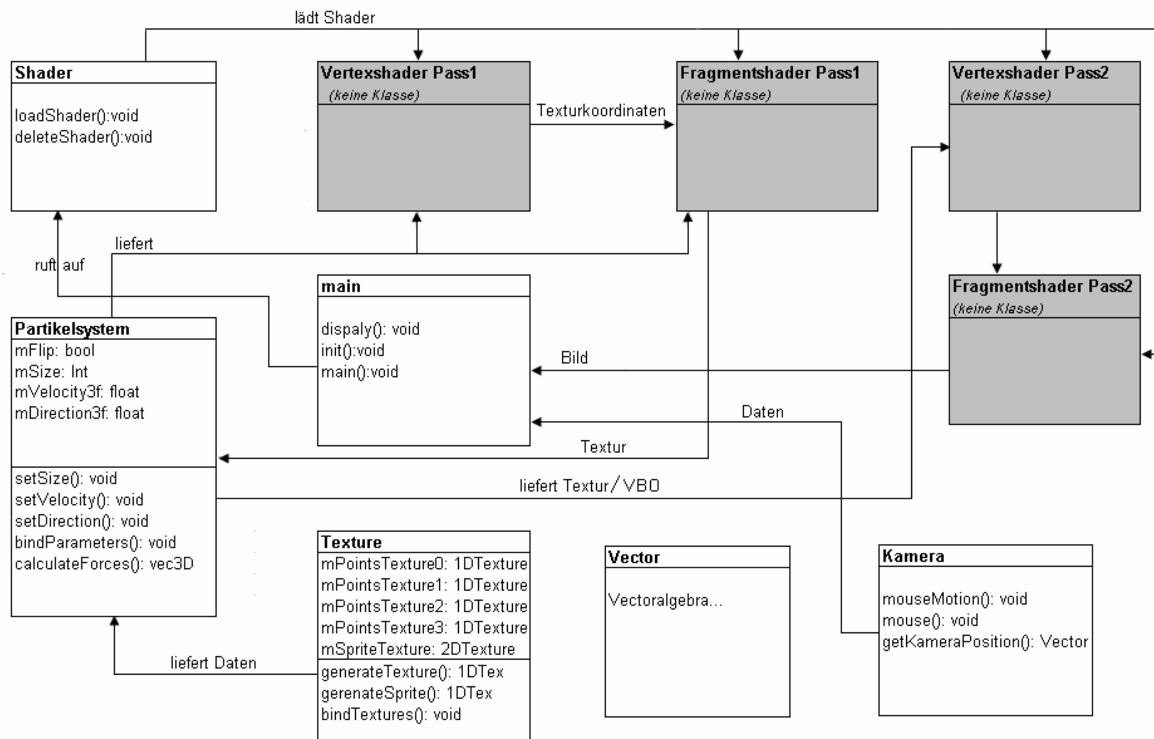


Abbildung 5: Klassendiagramm

4.3 Reihenfolge der Aufrufe

Nach dem Programmstart werden als erstes einige Variablen deklariert, darunter die Klassenobjekte sowie Höhe und Breite, mit der, neben der Fenstergröße, auch die Partikelanzahl definiert wird. Die Fenstergröße hat aber keinen rückwirkenden Einfluss auf die Partikelanzahl. In der Main wird dann als ersts Glut initialisiert, sowie glew und die eigene Initfunktion aufgerufen. Darin werden als erstes die Klassenobjekte Textur, 2 Mal Shader, Kamera und Partikelsystem initialisiert. Da das Partikelsystem 2 Shaderobjekte und ein Texturobjekt übergeben bekommt, müssen diese natürlich vor dem Partikelsystem initialisiert werden. Nachdem die Perspektive und der Matrixmode gesetzt wurden, generiert die Funktion durch das Texturobjekt die initalen Texturen, die Bufferobjekte

durch das Partiklsystemobjekt und lädt die Shader durch die beiden Shaderobjekte. Danach werden nur noch die Inputtexturen der Shader als Uniformparameter übergeben, da sie nicht jeden Frame wechseln.

In der Displayfunktion befindet sich nicht direkt die Hauptschleife des Partikelsystems, sondern nur die üblichen openGL Befehle. Zwischen dem Aktualisieren der Kameraposition und dem swapbuffers Befehl, wird die Methode renderPasses aus dem Partikelsystemobjekt aufgerufen, in der alle wichtigen Operationen durchgeführt werden. Dazu mehr im nächsten Kapitel. Des Weiteren werden in der Display als erstes Color und Depthbuffer gelöscht .

4.4 Die Hauptschleife

Wie schon erwähnt ist die Methode „renderPasses“ das Herz des Programms, hier werden jeden Frame die Daten für die Berechnungen „durchgepumpt“. Die Methode wird in zwei Renderpasses unterteilt. Dazwischen wir nur einmal der Color- und der Depthbuffer gelöscht.
Direkt zu Anfang wird das Shaderprogramm für die Physik aktiviert. Dazu gehört ein Vertexshader der nichts macht außer die Texturkoordinaten zu übergeben, und ein Fragmantshader der Positionen und Ausrichtung der Partikel berechnet. Zuvor muss aber noch eine Uniformvariable übergeben werden, die das Zeitintervall zwischen den letzten beiden Frames beschreibt. Das macht die Funktion „bindDelta“ mit Hilfe der Funktion glutGet(GLUT_ELAPSED_TIME). Jetzt kann das FBO aktiviert werden um die Rendertargets für die Shader anzugeben. Beide FBO wurden in einem Array angelegt, sodass jetzt ohne „if“ Klausel das richtige FBO ausgewählt werden kann. Auch die Datentexturen wurden in einem Array angelegt. So können alle Objekte mit Hilfe der Variable „mFlip“ dynamisch ausgewählt werden. Auf Grund der Tatsache dass der Shadercode für den ersten Pass nicht wechselt muss durch das FBO die Ausgabe definiert werden und gleich danach durch Binden der beiden Texturen auch die Ausgabe des Shaders. Nur durch die Variable mFlip wechselt dann jeden Frame Input und Output die Positionen. Darum wird die Variable auch nach diesen Aufrufen „umgedreht“.

Jetzt ist fast alles vorbereitet für die Berechnung der Partikel. Um den Fragmentshader jetzt pro Partikel aufrufen zu können wird das in Kapitel 3 Technik beschriebene Screenfilling Polygon gezeichnet. Dazu wird die Projektionsmatrix und die Modelviewmatrix gespeichert und dann der Einheitsmatrix gleichgesetzt. Damit das Screenfilling Polygon immer die Größe der Textur hat wird jetzt der Viewport auf die Größe der im Hauptprogramm definierten Variablen Höhe und Breite gesetzt, mit denen auf die Texturen erstellt wurden. Auch für den neuen orthografischen Projektionsmodus, der hier gesetzt werden muss, werden die beiden Variablen gebraucht. Gleichzeitig mit den vier Eckpunkten des Polygons werden auch die Texturkoordinaten angegeben.

Auch wieder mit den Variablen für Höhe und Breite. Sonst wäre das Polygon nicht fensterfüllend. Normalerweise würden die Texturkoordinaten mit 0 und 1 angegeben, wenn keine Wiederholung der Textur gewünscht ist. Aber da in diesem Programm eine Extension für Texturen benutzt wird, muss auch hier mit der Höhe und Breite gearbeitet werden. Stünde hier nur 0 und 1 würde nur der eine Pixel in der linken oberen Ecke behandelt und der ganze Rest der Textur würde außer Acht gelassen. Durch „Zeichnen" des Screenfilling Polygon sind jetzt alle Berechnungen ausgeführt und auch schon gespeichert, sie müssen nurnoch dargestellt werden. Da der Shader in das FBO gerendert hat und dorthin auch nur die abstrakten Ergebnisse der Berechnungen, wurde das Screenfilling Polygon nicht wirklich gezeichnet. Es kamen nie Daten im Colorbuffer an. Trotzdem wird nach dem Wiederherstellen der ursprünglichen Matrizen der Color und Depthbuffer gelöscht, der Vollständigkeit halber. Um den ersten Pass abzuschließen muss nur noch FBO und Shaderprogramm deaktiviert werden.

Der zweite Pass wird mit aktivieren des zweiten Shaderprogramms eingeläutet. Hier übernimmt der Vertexshader den wichtigen Part. Deshalb wird auch die im letzten Pass berechnete Positionstextur als neuen Input an den Shader gebunden. Texturzugriff im Vertexshader ist ein neues Feature des Shadermodell 3.0. Ohne diese Möglichkeit müsste der Vertexshader die Positionsdaten aus dem Hauptspeicher lesen, was große Leistungseinbußen bedeutet, da die Daten erst aus dem Texturformat für das Hauptsystem zugänglich gemacht werden müssten. Das entspräche aber nicht dem Prinzip dieser Arbeit das Partikelsystem vollständig auf der GPU zu implementieren. Nachdem die Textur mit den Positionsdaten übergeben wurde kommt auch das zusammen mit den FBO's erzeugte VBO an die Reihe. Zuerst werden VBO's generell aktiviert und dann das bereits bestehende an die Grafikkarte mit dem Bind-Befehl übergeben. „glVertexPointer" gibt an wie das Format der Daten in dem VBO ist, und wo die Daten abgelegt sind. Der Pointer ist Null damit der vorherige Buffer verworfen wird, dabei entsteht kein Konflikt. Dann werden mit glDrawArrays eine Anzahl Punkte entsprechend der schon bekannten Höhe und Breite gezeichnet. Jetzt wird pro Punkt ein Vertexshader aufgerufen, der dank der übergebenen Textur und des VBO weiß welcher Partikel wohin verschoben werden soll. Der Fragmentshader sorgt nur noch für einen simplen Farbverlauf der Partikelfarbe abhängig von ihrer Lebensspanne. Sprites ersetzen diesen Part, so dass der Fragmentshader im 2 Shaderprogramm weggelassen werden kann. Zuletzt werden noch VBO's und das Shaderprogramm deaktiviert.

```
void Partikelsystem::RenderPasses(int width, int height)
{
    ////-------------Pass 1-------------\\\\
    //--------Shader Stufe eins---------\\
        glUseProgram(phyShader->getProgram());
        bindDeltaT();
        //Texturen an TextureUnit des aktiven Shaderprograms binden
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, mfbo[mFlip]);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_RECTANGLE_ARB, mTexture->getmPointsTexture(mFlip *2));
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_RECTANGLE_ARB, mTexture->getmPointsTexture(mFlip *2 +1));
        mFlip =(mFlip+1)%2;
        // rendert scene zwischen den FBO-befehlen ins FBO-objekt
        glMatrixMode(GL_PROJECTION);
        glPushMatrix();
        glLoadIdentity();
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        glLoadIdentity();
        gluOrtho2D(0,width,0,height);

        glClearColor(0.0, 0.0, 0.2, 0.0);  // Dunkelblau
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glPushAttrib(GL_VIEWPORT_BIT);
        glViewport(0, 0, width, height);   // Größe entsprechend der verwendeten Textur

        glBegin(GL_QUADS);                // Screenfilling Polygon
          glTexCoord2i(      0,       0); glVertex2f(      0,       0); //links unten
          glTexCoord2i(width-1,       0); glVertex2f(width-1,       0); //links oben
          glTexCoord2i(width-1,height-1); glVertex2f(width-1,height-1); //rechts oben
          glTexCoord2i(0      ,height-1); glVertex2f(      0,height-1); //rechts unten
        glEnd();

        glPopAttrib();                              //viewport wieder herstellen
        glMatrixMode(GL_PROJECTION);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
        glPopMatrix();

        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);//Framebufferobjekt wieder ausschalten
        glUseProgram(0);                     //Shader ausschalten

        ////-------------Pass 2-------------\\\\
        glClearColor(0.0, 0.2, 0.2, 1.0); //Kontrollfarbe für Hauptbildschirm
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

      //----------Shader Stufe 2------------\\
      glUseProgram(renShader->getProgram());

        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_RECTANGLE_ARB, mTexture->getmPointsTexture(mFlip *2));
        glEnableClientState(GL_VERTEX_ARRAY);
        glBindBuffer(GL_ARRAY_BUFFER, vertex);
        glVertexPointer(2, GL_FLOAT, 0, (char*)NULL);
        glPointSize(1);
        glDrawArrays(GL_POINTS, 0, width*height);
        glDisableClientState(GL_VERTEX_ARRAY);
        glUseProgram(0);
}
```

Listing 1: Die Hauptschleife bzw. die Funktion renderPasses aus der Partikelsystemklasse

4. 5 Shader

Unter den vier verwendeten Shadern sind nur zwei wirklich wichtig. Beide werden in diesem Kapitel ausführlich behandelt. Sie ergänzen die Hauptschleife um die nötigen Berechnungen und sorgen dafür, dass die Partikel auch an ihre neuen Positionen gelangen. Im Gegensatz dazu sind die anderen beiden nur Beiwerk und könnten wahrscheinlich wegrationalisiert werden.

4.5.1 Fragmentshader Nr.1

Bevor mit dem eigentlichen Code begonnen werden kann müssen zuerst zwei Extensions für den Shader aktiviert werden. Obwohl beide in „glew“ enthalten sind, muss das geschehen, weil Shader erst zur Laufzeit kompiliert werden, und damit auch zu diesem Zeitpunkt ihre Extensions dem Compiler mitgeteilt werden müssen. Da ist zunächst die Texture Rectangle Extension, die dafür sorgt, dass die übergebenen Texturen überhaupt gelesen werden können. Sowie die Draw Buffers Extension die es ermöglicht in mehrere Rendertargets zu rendern. Danach geht es ganz normal mit der Variablendeklaration weiter. Aus dem Programm übergeben wurden zwei Input Texturen vom Typ Texture Rectangle und ein float der das Zeitintervall der Frames angibt. In der Main werden dann noch je ein Vierer-Vektor für Position, Velocity und einen Randomwert angelegt, sowie ein Dreier-Vektor für die Gravitation und ein Floatwert für das echte Alter. Der Gravitationswert fällt sehr viel kleiner als der reale aus, sonst würde er in der Kalkulation viel zu schnell ins Gewicht fallen. Zudem wird er negativ mit eingerechnet um den Wärmeauftrieb zu simulieren. Die beiden Vektoren Position und Velocity werden direkt aus der jeweiligen Inputtextur mittels Texturelookup ausgelesen. Dadurch, dass für beide dieselbe Texturkoordinate gilt, ist sichergestellt, dass immer die Daten von einem Punkt und nicht von verschiedenen gelesen werden. Der Randomvektor ist einfach das Ergebnis der buildin noisefunc mit der Velocity als Input.

Der Floatwert t errechnet sich aus der Differenz des Maximalalters mit dem aktuellen Alter, da der gespeicherte Alterswert pro Frame dekrementiert wird und nicht inkrementiert. Es gibt einen Maximalwert für das Alter, der heruntergezählt wird, damit bei einer zufälligen Altersneudefinition bei der Geburt, durch Schwankungen der Zufallswerte, nicht ungewollte Lücken im Partikeloutput entstehen. Bei der Geburt wird dann zufällig eine Lebensspanne bestimmt und im Alphawert der Positionstextur abgelegt. An Hand diesem Wert wird gleich darauf die verbleibende Zeit bis zum Maximalwert errechnet und im Alphawert der Velocitytextur abgelegt. Dadurch dass nach Ablauf der Lebenspanne der verbleibende Zeitraum bis zum Maximalalter abgewartet wird bevor das Partikel neu startet, wird garantiert dass immer gleich viele Partikel pro Frame von der Quelle ausgestoßen werden können. Um Speicherplatz zu sparen und da jedes Partikel auch pro Frame individuell gestaltet sein soll, wird die Lebenszeit bei der Geburt bestimmt und dann heruntergezählt. Zurzeit ist mangels eines einfachen Zufallsgenerators für Shader die Lebensspanne noch hardgecoded. Ein zufälliger Faktor ist wichtig damit die Partikel einer Generation nicht alle gleichzeitig sterben. Die ältesten sind alle an den äußersten Positionen der Simulation zu finden. Ein gleichzeitiges Verschwinden ist somit sehr auffällig. Durch eine gute zufällige Verteilung der initialen Velocityvektoren und auslassen einer regelmäßigen Normierung dieser, wird dieser Effekt stark abgeschwächt. Trotzdem sind aus bestimmten Blickwinkeln noch Artefakte zu beobachten.

Nachdem die Lebenszeit abgelaufen ist, wird das Partikel zwar noch gezeichnet, aber es verbleibt mit allen anderen inaktiven Partikeln im Ursprung und bildet eine kaum sichtbare Singularität. Ohne Sprites ist sie nur einen Pixel groß, mit Sprites allerdings wäre sie wahrscheinlich unübersehbar. In diesem Zustand wird der bei der Geburt berechnete Restwert heruntergezählt. Sind beide Lebenswerte auf Null wird das Partikel neu gestartet und neue zufällige Werte eingesetzt bzw. berechnet. Diese 3 Fälle werden mit If Klauseln im Shader abgefragt. Zuerst wird unterschieden ob noch Lebenszeit übrig ist, ist das der Fall, wird die Velocity berechnet und in Rendertarget 1 geschrieben. Sobald die Lebenszeit abgelaufen ist, wird mit einer zweiten Fallunterscheidung das Partikel entweder in der Singularität gehalten oder, wenn auch der zweite Alterswert Null ist, neu initialisiert.

```
#extension GL_ARB_draw_buffers : enable
#extension GL_ARB_texture_rectangle : enable

uniform sampler2DRect dataPoints0;
uniform sampler2DRect dataPoints1;
uniform float deltaT;

void main(void)
{
    vec4  posOld;
    vec4  oldVelocity;
    vec3  gravity;
    vec4  random;
    float t;

    gravity      = vec3(0.0, -0.0000001, 0.0);
    posOld       = vec4(texture2DRect(dataPoints0, gl_TexCoord[0].st));
    oldVelocity  = vec4(texture2DRect(dataPoints1, gl_TexCoord[0].st));
    random       = noise4(oldVelocity);
    t  = (1000.0 -posOld.a); //Performanceabhängig => wenige Punkte viel Grav und umgekehrt

    if (posOld.a > 0.0 )             //solange Lebensenergie übrig ist
    {
         vec4 position = vec4(vec3(vec3(posOld.r, posOld.g, posOld.b)+ vec3(oldVelocity.r,
         oldVelocity.g, oldVelocity.b) *deltaT *0.01 -0.5 *gravity *t*t), posOld.a -1.0);
         gl_FragData[0] =  position;
         gl_FragData[1] = vec4(vec4(oldVelocity.r, oldVelocity.g, oldVelocity.b,
          oldVelocity.a) + random * vec4(gravity,1.0) * deltaT );
       }
       else if (oldVelocity.a > 0.0)    // Partikel in Singularität halten
       {
          gl_FragData[0] = vec4(0.0,0.0,0.0,0.0);
          gl_FragData[1] = vec4(oldVelocity.r, oldVelocity.g , oldVelocity.b,
            oldVelocity.a - 1.0);
       }
       else                              // neu initialisieren
       {
          gl_FragData[0] = vec4(0.0,0.0,0.0,999.0);
          gl_FragData[1] = vec4(oldVelocity.r, oldVelocity.g, oldVelocity.b,
             oldVelocity.a - 1.0);
       }
```

Listing 2: Fragmentshader aus dem ersten Pass

4.5.2 Vertexshader Nr.2

Jetzt da die neuen Positionen berechnet sind, müssen die Partikel nur noch gezeichnet und an eben diese Position gebracht werden. Dafür ist der Vertexshader des zweiten Passes zuständig. Er erhält die Positionsdaten aus der Positionstextur, in die eben noch der Fragmentshader hineingerendert hat. Dazu muss auch hier als erstes die texture Rectangle eingebunden werden. Danach wird nur ein Vierervektor „look" deklariert, der zum Zwischenspeichern des Texturelookups dient. Das ist auch die erste Aktion in der Main. Mit dem Unterschied, dass nicht mit Texturkoordinaten auf die Textur zugegriffen wird, sondern mit den Werten die aus dem VBO ausgelesen werden. Bei der Erstellung des VBO hatte jeder Punkt seine eigenen Koordinaten zugewiesen bekommen, so dass jetzt kein Punkt ausgelassen oder doppelt angesprochen wird. Im Alphawert befindet sich noch der Lebenswert. Er ist im Wertebereich von [0, Maxage] und so nicht für einen Alphawert geeignet. Deshalb wird er mit einem einheitlichen Wert überschrieben bevor die Modelviewmatrix mit dem ermittelten Wert multipliziert und dann mit der Position gleichgesetzt wird.

```
#extension GL_ARB_texture_rectangle : enable

uniform sampler2DRect renderPoints0;
vec4 look;

void main(void)
{
    //Texturelookup mit VBO-Daten
    look = texture2DRect(renderPoints0, gl_Vertex.xy);
    look.w = 1.0;   // kann hier ruhig überschrieben werden, gelangt ja nicht in die Textur
    gl_Position     = gl_ModelViewProjectionMatrix * look;
    gl_TexCoord[0]  = gl_MultiTexCoord0;
}
```

Listing 3: Vertexshader des 2. Pass

5 Fazit

Es ist nicht schwer ein einfaches Partikelsystem auf der GPU umzusetzen. Die Einschränkungen durch die Hardware nehmen mehr und mehr ab. Im Vergleich zur CPU müssen zwar mehr Dinge berücksichtigt werden, aber dafür erhält man eine wesentlich höhere Performance. Der Zweite Vorteil, dass die CPU entlastet wird, ist mit Vorsicht zu genießen. Kleine Programme und CPU lastige Programme können durch die GPU durchaus entlastet werden. Heutige Grafikanwendungen beanspruchen die Grafikkarte aber sehr stark und machen eine Mehrbelastung schwierig. Im Hinblick auf die aktuelle Entwicklung des CPU-Marktes in Richtung Multicoreprozessoren liegt eine Verlagerung auf die CPU nahe. Die Verwendung eines reinen GPU Partikelsystems kann schon sinnvoll sein, wenn man vorher darauf achtet die GPU nicht zu überbelasten. Die verfügbaren Ressourcen sollten immer im Blick behalten werden. Als gute Alternative sorgt die Technik damit für mehr Spielraum bei der Planung von Programmen. Im Allgemeinen kann man sagen, dass Partikelsysteme mächtige Werkzeuge sind um teilweise komplexe Effekte realitätsnah wiederzugeben. Zum Vorteil gereicht ihnen dabei ihre Fähigkeit der Selbstorganisation. Nach dem

Start muss der Benutzer nicht mehr eingreifen, das System übernimmt sämtliche Aufgaben zur Berechnung, Koordination und Verwaltung selbst. Gleichzeitig macht es die elementare Herangehensweise sehr flexibel und anpassungsfähig. Damit verbunden ist allerdings ein großer Implementierungsaufwand. Denn für jedes Problem muss ein neues spezielles Partikelsystem geschrieben werden. Außerdem sind die hohen Systemanforderungen ein Problem wenn der Effekt authentisch sein soll.

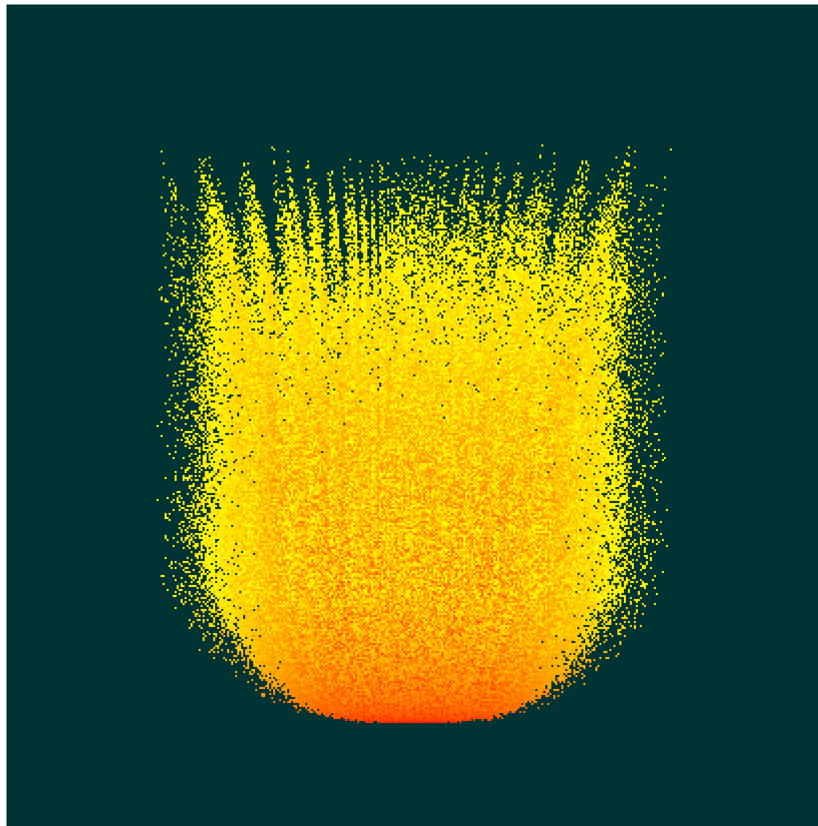


Abbildung 6: Feuer mit 1 Mio. Partikel

5.1 Leistungstest

Das entstandene Partikelsystem ist in der Lage mehrere Millionen Partikel flüssig darzustellen. Durch die objektorientierte Implementierung ist es auch möglich mehrere Systeme zu erstellen und separat zu verwalten.

Das Darstellungsergebnis ist derzeit sehr Leistungsabhängig. Zwar kann so die Leistung anhand der Frames pro Sekunde gemessen werden, aber das hat zur Folge das kleine Partikelmengen eine wesentlich höhere Framezahl aufweisen als größere Systeme. Dadurch ist auch die zeitliche Framedifferenz niedriger, die jedoch direkt in die Berechnung mit eingeht. Kleine Mengen unterliegen damit stärker dem Auftrieb und bewegen sich auch schneller, während große Mengen eher träge sind und nur langsam dem Auftrieb folgen. Um der Instabilität der Eulerberechnung

entgegen zu wirken muss das Intervall von der Framerate unabhängig gemacht werden. Denkbar wäre eine Art Synkronisation auf eine feste Framezahl mittels eines festen Wertes für das Intervall oder indem man überflüssige Frames verwirft. Es kann auch auf eine andere Berechnungsmethode zurückgegriffen werden, aber an die Einfachheit und damit Schnelligkeit der Eulermethode kommt kaum eine andere heran.

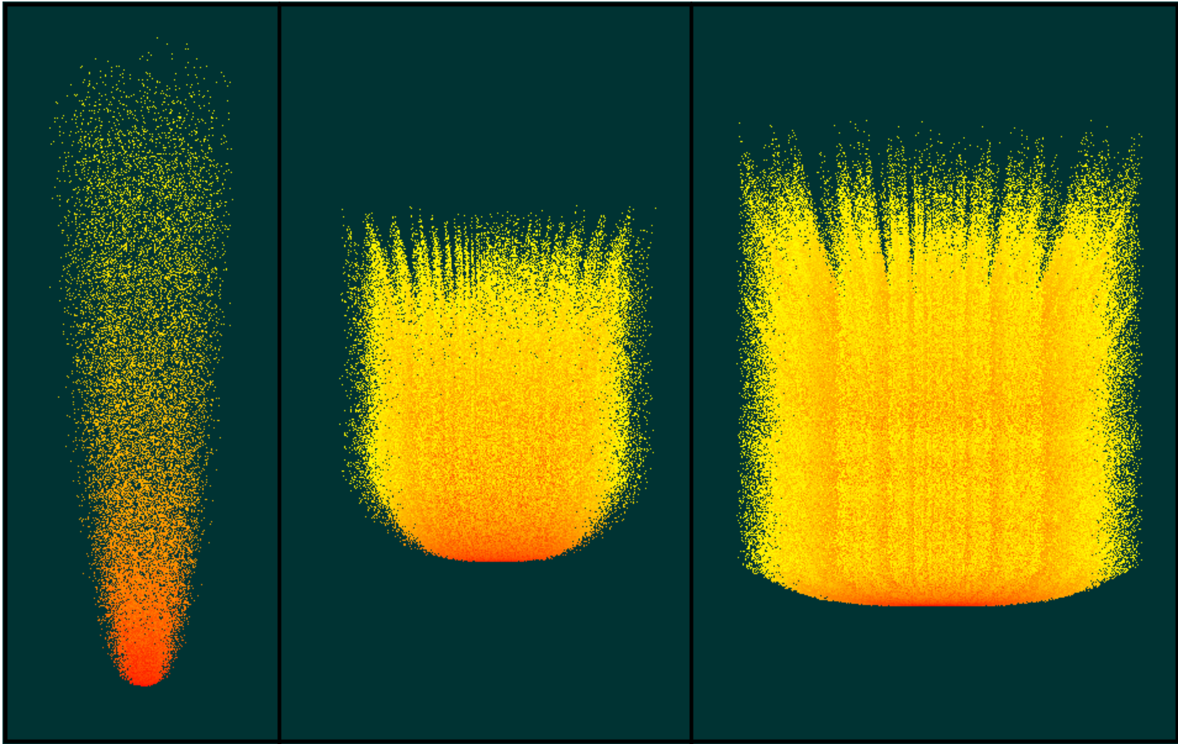


Abbildung 7: Frameabhängige Darstellung mit 65k, 1 Mio. und 4 Mio. Partikeln

Da das Partikelsystem derzeit nur auf einer GForce 8800 lauffähig ist, wurden auch keine Vergleiche mit anderen Grafikkarten durchgeführt. Bei Leistungstest ist aufgrund der speziellen Architektur auch nur die Grafikkarte der maßgebende Faktor für dieses Programm. Die CPU kann vernachlässigt werden. Gemessen wurden verschiedene Partikelmengen von 512 bis 16 Mio. Partikel. Obwohl die GForce 8800 Texturen von der Größe 8182x8182 unterstützt, konnten keine Texturen die größer als 4096x4096 sind dargestellt werden. Das System gibt dann die Fehlermeldung „out of Memory". Die folgende Abbildung zeigt eine Tabelle mit verschiedenen Texturauflösungen und den dazugehörigen erreichten Framezahlen.

Textur Auflösung	Frames / sek
64x64	ca. 5300
128x128	ca. 5300
256x256	ca. 2500
256x512	ca. 1300
512x512	ca. 700
750x750	ca. 315
1024x1024	ca. 190
1500x1500	ca. 85
2048x2048	ca. 46
3000x3000	ca. 22
4000x4000	ca. 12
8182x8182	out of Memory

Abbildung 8: Performancetabelle

5.2 Ausblick

Der Code könnte auf Performance hin noch optimiert werden. Im Hauptprogramm werden hauptsächlich nur einmalige Berechnungen durchgeführt. Die Hauptlast tragen damit natürlich die Shader. Deshalb sollte dort nach Möglichkeit deren Code optimiert werden. Vor allem die If Konstruktionen und eventuell überflüssigen Multiplikationen können in der Masse merklichen Einfluss auf die Leistung ausüben. Aber auch ohne Leistungssteigerung hat das Programm noch genug Reserven um es zu erweitern. Nach kurzer Einarbeitung ist das mit einfachen Mitteln schnell möglich. Der Code für die Eingabeparameter ist vorhanden und muss bei Bedarf nur vervielfältigt werden. Danach können einfach neue Shader an Stelle der alten eingefügt werden. Die Physikberechnung findet vollständig innerhalb der Shader statt, so dass verschiedene Shader für die unterschiedlichsten Effekte genutzt werden können. Die verwendete GForce 8800 schafft bis zu 8 Drawbuffers. Damit sind 32 Byte Daten pro Partikel möglich. Wenn nicht alle Daten jeden Frame aktualisiert werden müssen sogar mehr. Wichtig für ein gutes Partikelsystem auf der GPU ist auch ein solider Zufallsgenerator im Shadercode, der noch nicht vorhanden ist.

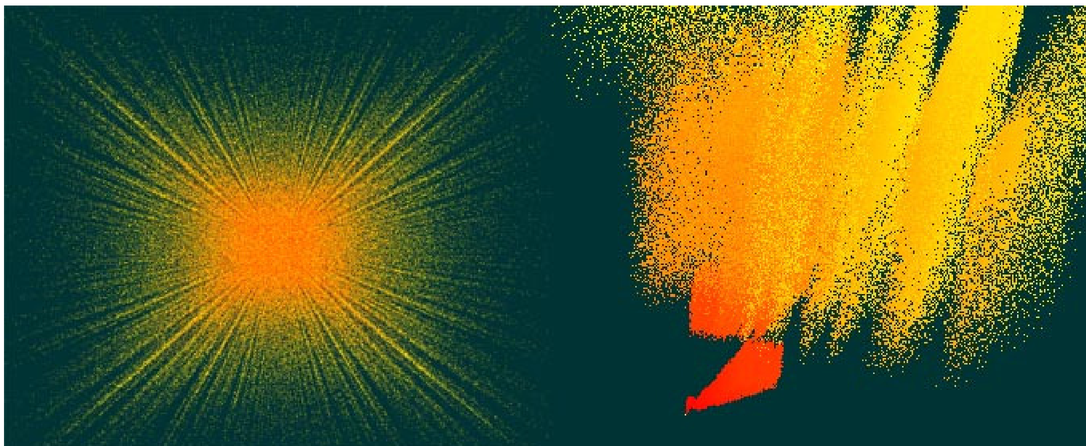


Abbildung 9: Artefakte mit fehlenden oder schlechten Zufallswerten

Aber jede etwas komplexere Anwendung kommt ohne eigentlich nicht aus. Eine Möglichkeit wäre die weit verbreitete Perlin Noise. Allerdings sind dazu mehrere Texturen als Input nötig. Um die Anzahl der Partikel hoch zu halten wäre ein weniger anspruchsvoller Algorithmus wünschenswert. Bei großen Mengen an Partikeln fällt ein nicht ganz perfekter Algorithmus auch weniger auf.

Der erste Schritt auf dem Weg ein flackerndes Feuer zu gestalten ist eine einzige geschlossene Flamme, die dann zum flackern gebracht wird. Abbildung 10 zeigt ein Zwischenergebnis. Mit einer Cosinusfunktion wurden die Partikel dazu gebracht wieder zu konvergieren. Aber ohne geeignete Zufallswerte entsteht ein Bild wie in Abbildung 9 (rechts). Um den Eindruck eines flackernden Lagerfeuers weiter zu verstärken, könnte man den Emitter erweitern. Dabei würde nicht nur die flächige Abstrahlung der Partikel helfen. Bei der Geburt müssten diese auch mit einem vorherrschenden Richtungsvektor ausgestattet werden, der vom Ausgangspunk abhängt. Damit aus dem Feuer dann kein Igel wird, sollte der Vektor und der dazugehörige Geschwindigkeitsfaktor über die Zeit so variieren, dass ein flackerndes Feuer entsteht. Eine Alternative wäre mehrere kleine Systeme zu erstellen und diese entsprechend anzuordnen oder eine zweite Hierarchiestufe zu implementieren.

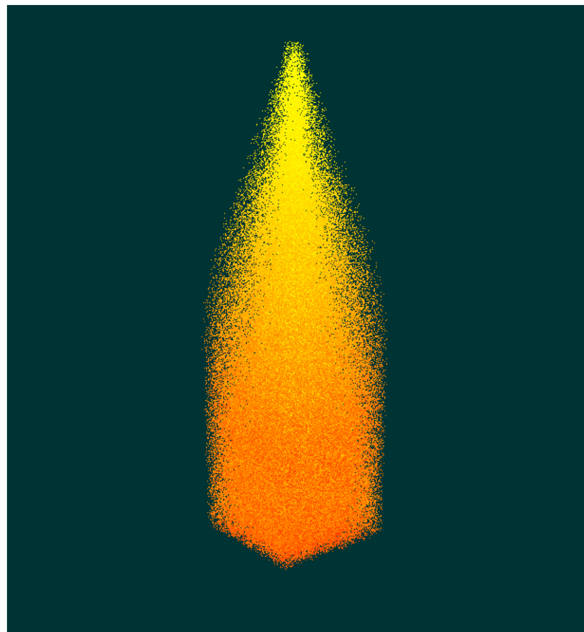


Abbildung 10: Versuch einer geschlossenen Flamme

Obwohl anfangs geplant, ist das System, wie sich im Laufe der Arbeit herausstellte, nicht auf ATI Hardware lauffähig. Auch die neusten ATI- Karten unterstützen zurzeit kein Texturelookup in Vertexshadern. In näherer Zukunft scheint ATI dahingehend auch nicht nachzubessern. Stattdessen werden sogenannten Überbuffer angestrebt. Damit könnte dann mit Fragmentshadern in Vertexbuffer

gerendert werden, die dann als Input für Vertexshader dienten. Also anstatt in fragmentverwandte Texturen zu rendern und den Vertexshadern dann dieses unpassende Format aufzuzwängen, geht ATI dazu über mit dem Fragmentshader in ein ungewohntes Format zu rendern, das dann aber besser zum Vertexshader passt. Damit wäre auch auf ATI Karten ein reines GPU berechnetes Partikelsystem möglich. Sobald die Extension herausgegeben wird, ist sie auf allen Shadermodell 3.0 fähigen ATI-Karten verfügbar. In Zukunft wird natürlich die Hardware weiter an Leistung gewinnen. Mehr Partikel können berechnet werden und Systeme die heute noch hinken können dann in Echtzeit laufen. Aber mit wachsender Leistung steigt auch die Erwartung der Benutzer, sodass eine einfache Erhöhung der Anzahl nicht ausreicht. Immer neue Algorithmen, Tricks zur Berechnung und innovative Ansätze werden gebraucht um die gleichzeitig wachsenden Erwartungen zu erfüllen. Denkbar ist eine Kombination aus General Purpose, Daten Parallelen Algorithmen und traditioneller Grafikprogrammierung, wie sie Aron Lefohn in seiner Dissertation "Glift: Generic Data Structures for Graphics Hardware" anspricht [AL06].

Literaturverzeichnis:

[RR06] Rost, Randi J. (2006) : OpenGL Shading Language, Second Edition. Addison-Wesley, Upper Saddle River, NJ. ISBN 0-3213-3489-2

[FA05] Lighthouse 3D - A Resource For 3D Programmers (2005) Fernandes, António Ramires: GLSL Tutorial. http://www.lighthouse3d.com/opengl/glsl/; Abruf: März 2007

[DGLP] DGL Wiki - Freies OpanGL Wissen für alle! (2006), Autor unbekannt, GLSL Partikel http://wiki.delphigl.com/index.php/GLSL_Partikel; Abruf: März 2007

[SW06] DGL Wiki - Freies OpenGL Wissen für alle! (2006) Sascha Willems, Tutorial GLSL. http://wiki.delphigl.com/index.php/Tutorial_glsl; Abruf: September 2006

[JB06] DGL Wiki - Freies OpenGL Wissen für alle! (2006) Jakob Breu , Tutorial GLSL2 http://wiki.delphigl.com/index.php/Tutorial_glsl2; Abruf: September 2006

[FS06] DGL Wiki - Freies OpenGL Wissen für alle! (2006) Florian Sievert: Tutorial Renderpass http://wiki.delphigl.com/index.php/Tutorial_Renderpass; Abruf: November 2006

[SW06-2]DGL Wiki - Freies OpenGL Wissen für alle! (2006) Sascha Willems: Tutorial Vertexbufferobject, http://wiki.delphigl.com/index.php/Tutorial_Vertexbufferobject; Abruf: Februar 2007

[SM06] AniSim Foliensatz (2006) Prof. Dr. Stefan Müller: Dynamics of Masspoints;

[PP05] Philipp Pätzold (2005), Entwicklung eines Partikelsystems auf Basis moderner 3D-Grafikhardware, Studienarbeit, Universität Koblenz-Landau

[YP06] Yvo Pesek (2006), Simulation von Feuer mit Hilfe eines Partikelsystems, Studienarbeit, Universität Koblenz-Landau

[WH07] Ian Williams und Evan Heart, Efficient rendering of geometric data using OpenGL VBOs in SPECviewperf, http://www.spec.org/gpc/opc.static/vbo_whitepaper.html; Abruf: Februar 2007

[DG06] Dominik Göddeke, GPGPU::Basic Math Tutorial (2006), http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html; Abruf: Februar 2007

[DB05] Dave Baumann, Vertex Shader, Artikel, (2005) http://www.beyond3d.com/reviews/ati/r520/index.php?p=02; Abruf: März 2007

[JH02] Johannes Hein, Partikelsysteme, Ausarbeitung (2002), http://medien.informatik.uni-ulm.de/lehre/courses/ss02/Computergrafik/JohannesHein.pdf; Universität Ulm,

[SG05] Simon Green, The OpenGl Framebuffer Object extension, Präsentation (2005), http://http.download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf;

[WR83] William T Reeves, Particle Systems A Technique for Modeling a Class of Fuzzy Objects, Paper (1983), Lucasfilm Ltd

[JB00] John van der Burg, Building an Advanced Partiklesystem, Artikel (März 2000), Game Developer Magazin

[AJ03] Almar Joling, A Simple Point Sprite Based Particle Engine, Artikel (2003), http://www.gamedev.net/reference/articles/article2002.asp; Abruf: September 2006

[RJ06] Rob Jones, OpenGL Frame Buffer Object 201, Artikel (2006) http://www.gamedev.net/reference/articles/article2333.asp; Abruf: Januar 2007

[EP07] Emil Persson, Framebuffer Objects, Tutorial http://www.cs.sunysb.edu/~mueller/teaching/cse564/labs/FramebufferObjects.pdf; Abruf: Januar 2007

[DG05] Dominik Göddeke, Playing Ping Pong with Render-To-Texture, Tutorial (2005) http://www.mathematik.uni-dortmund.de/~goeddeke; Universität Dortmund

[AL06] Aron Lefohn, Glift: Generic Data Structures for Graphics Hardware (PhD Thesis), Dissertation (2006), http://graphics.idav.ucdavis.edu/publications/print_pub?pub_id=900;