UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Partikelsimulation

Studienarbeit
im Studiengang Computervisualistik

vorgelegt von
Nina Damasky

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

Koblenz, im Oktober 2008

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

Nina Damasky

(Matr.-Nr. 203 110 137)

Thema: **Partikelsimulation**

Moderne Rechnersysteme ermöglichen die 3D-Darstellung komplexer, virtueller Umgebungen in sehr hoher Qualität. Eine wichtige Herausforderung besteht nun darin, das Verhalten von Objekten z.B. auf Basis physikalischer Gegebenheiten zu simulieren, um diese virtuellen Umgebungen attraktiv bzw. realitätsnah erscheinen zu lassen. Hierzu gibt es eine Reihe von Methoden und Algorithmen, wobei die wichtigsten Vertreter auf der Simulation von Partikeln beruhen.

Ziel dieser Arbeit ist die Implementierung von Verfahren zur Simulation von Partikeln und die Umsetzung dieser Verfahren in einer Beispielapplikation mit ansprechender Qualität.

Schwerpunkte dieser Arbeit sind:

1. Einarbeitung in die Methoden der Partikelsimulation.
2. Festlegung einer zu entwickelnden Beispielapplikation.
3. Implementierung der Verfahren zur Simulation von Partikeln, inklusive der relevanten Datenstrukturen und der Darstellung.
4. Implementierung der Beispielapplikation in ansprechender Qualität.
5. Dokumentation der Ergebnisse.

Koblenz, den 29. 4. 2008

- Prof. Dr. Stefan Müller-

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

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

.....
(Ort, Datum) (Unterschrift)

Zusammenfassung

Diese Studienarbeit liefert einen Einblick in die Methode der Computergraphik, mittels vieler einzelner Partikel Objekte mit stark dynamischer, geometrisch schlecht beschreibbarer Oberfläche, darzustellen.

Zu Beginn wird der Begriff *Partikelsimulation* erläutert. Es wird festgestellt, dass der Begriff ein Verfahren benennt, mit Hilfe von vielen einzelnen Teilchen ein „großes Ganzes“ zu beschreiben, ohne dass der Anwender oder Entwickler das einzelne Partikel explizit steuert oder beschreibt. Folgend wird auf die Geschichte der Partikelsimulation in der Computergraphik eingegangen: die ersten Anwendungen in den 1960er Jahren und die Entwicklung des Partikelsystems von William T. Reeves im Jahr 1983, welches als Grundlage sämtlicher folgender Entwicklungen gesehen werden kann. Es wird ein Versuch unternommen, die zwischenzeitliche Entwicklung in die Bereiche *Spezialeffekte*, *Darstellung von Pflanzen* und *Schwarmverhalten* zu gliedern, um anschließend festzustellen, dass sie im Laufe der Zeit in Partikelsystem-Tools zusammenfließen, auf die im Kapitel *Stand der Technik* näher eingegangen wird. Es wird sich rausstellen, dass heutige Partikelsysteme die Möglichkeit bieten, mit relativ geringem Aufwand eine unzählige Menge an verschiedensten „großes-Ganzes-durch-viele-kleine“-Effekten - von der Simulation von Feuer, über die Erzeugung von Pflanzen bis hin zur Beschreibung von Fischschwärmen - in anschaulicher Qualität zu realisieren.

Im Kapitel *Entwurf von Partikel, Partikel!* wird die Grundstruktur und der Ablauf des eigens entwickelten Programms *Partikel, Partikel!* zur Simulation von Feuer, Schnee und eines Feuerwerks vorgestellt. Das Kapitel *Entwicklungsumgebung* geht auf die programmiertechnischen Rahmenbedingungen ein. Im Anschluss wird die Implementierung des Entwurfes von *Partikel, Patikel!* vorgestellt. Es werden die fünf erstellten Klassen und das Template mit allen zugehörigen Funktionen sowie die Struktur `Particle` erläutert. Die Benutzeroberfläche und das finale Programm mit Bildern der laufenden Simulation werden im Kapitel *Ergebnisse* vorgestellt und in Kapitel *Fazit* nach eigenen Maßstäben kommentiert, Schwachstellen aufgezeigt und mögliche Verbesserungen vorgeschlagen.

Inhaltsverzeichnis

1 Einleitung

****

1.1 Motivation

In der Computergraphik werden Objekte in der Regel durch geometrische Beschreibung ihrer Oberfläche modelliert. Diese Methode kommt bei der Modellierung von Objekten, deren Oberfläche nicht fest definiert ist, wie z.B. bei Fluiden der Fall, schnell an ihre Grenzen. Der Grund liegt hier in der Komplexität der Bewegung der Oberfläche bzw. des Volumens: Zwar lässt sich auch die Obefläche von geometrisch modellierten Gegenständen verändern, aber dies geschieht immer durch Transformation der Primitive und muss für jedes Primitiv explizit angefordert werden - das „Starre“ ist der „Normalfall“ und die „Deformation“ die „Ausnahme“. Bei Fluiden (und vergleichbaren Objekten) hingegen ist die Bewegung des Volumens der Normalfall und das Starre die Ausnahme. Anstelle der geometrischen Beschreibung gelingt die Beschreibung derartiger dynamischer Objekte mittels unzähliger einzelner Partikel, die zusammen das Volumen des Objekts modellieren. Hauptaugenmerk ist das Kreieren von Rahmenbedingungen, innerhalb denen sich die einzelnen Partikel zufällig bewegen oder verändern können. Auf das genaue Verhalten eines konkreten Partikels kann hierbei durch den Modellierer kein Einfluss genommen werden. Mögliche Rahmenbedingungen sind z.B. bestimmte Farben, die ein Partikel annehmen darf oder Regeln, die die Bewegung bestimmen. Einmal das Grundgerüst implementiert, lassen sich mit konkurrenzlos wenig Aufwand unterschiedlichste Simulationstypen realisieren. Ausgereifte Partikelsystem-Implementierungen bieten sogar die Möglichkeit, oberflächenbasierte Objekte als Partikel zu verwenden, um z.B. Fischschwärme zu simulieren. Aufgrund der vielseitigen Einsatzmöglichkeiten und variablen Gestaltung sind die Grenzen zwischen den hier vorgestellten Partikelsystemen, der Simulation von Gruppenverhalten mit Werkzeugen der künstlichen Intelligenz und physikalisch korrekter Simulation von Fluiden, fließend. Die Simulation mittels Partikeln bietet somit ein interessantes und in der Computergraphik konkurrenzlos effektives Verfahren an, um verschiedenste Phänomene zu modellieren, die aus derart vielen ähnlichen kleineren Objekten bestehen, dass eine manuelle Modellierung und Animation aller einzelnen Bestandteile zu aufwändig und uneffektiv wäre.

1.2 Ziel

Ziel dieser Studienarbeit ist die Einarbeitung in die Partikelsimulation und die Implementierung eines eigenen Partikelsystems. Zu Beginn erfolgt eine Auseinandersetzung mit der Geschichte der Partikelsimulation mit Schwerpunkt auf das von William T. Reeves entwickelte Partikelsystem. Nach Erarbeitung der Grundlagen soll ein Konzept für das zu entwickelnde Programm aufgestellt werden: Das Programm soll die Rahmenbedingungen für die Implementierung verschiedener konkreter Simulationstypen bereit stellen und zur Demonstration der Funktionsfähigkeit die Umsetzung mindestens eines Types liefern. Die Datenstruktur soll hierbei derart gestaltet sein, dass das Programm im Nachhinein relativ einfach um weitere Typen erweitert werden kann.

2 Grundlagen

****

2.1 Begriffsklärung *Partikel*

„kleines Teilchen, z.B. mikroskopisch kleines Wassertröpfchen im Nebel(...).“ [Bertelsmann]

2.2 Begriffsklärung *Simulation*

„[lat. Vorspiegelung] in der Informatik die Nachbildung von Vorgängen mithilfe eines Computers. Sie wird zur Untersuchung von Abläufen eingesetzt, die man in der Wirklichkeit aus Zeit-, Kosten-, Gefahren- oder anderen Gründen nicht durchführen kann oder will (...).

Man unterscheidet zwischen der deterministischen und der stochastischen Simulation. Bei der deterministischen Simulation sind alle an dem Modell beteiligten Größen exakt definiert oder aufgrund mathematischer Zusammenhänge berechenbar. Bei der stochastischen Simulation werden in dem Modell auch zufallsabhängige Größen verwendet, z.B. bei der Monte-Carlo-Methode.“ [Brockhaus]

2.3 Begriffsklärung *Partikelsimulation*

Die Partikelsimulation ist ein Verfahren, mittels vieler einzelner Partikel das Volumen eines „Objektes“ anzunähern, das durch die Bewegung, Selbständigkeit und Ähnlichkeit seiner einzelnen Bestandteile definiert ist. Hierzu werden bestimmte Rahmenbedingungen geschaffen, innerhalb denen sich ein Partikel bzw. die Simulation bewegen darf (z.B. die maximale Anzahl an möglichen Partikeln, deren ungefähre Bewegungs-Richtung und Geschwindigkeit oder einen Bereich, in dem sich das Partikel aufhalten kann). Das endgültige Aussehen und Verhalten der Partikel und somit des „großen Ganzen“ bestimmen stochastische Prozesse, auf die der Anwender keinen Einfluss hat - es handelt sich also um eine stochastische Simulation.

Nach eigenem Wissen wird der Begriff „Partikelsimulation“ hauptsächlich in der Computergraphik eingesetzt, wobei er theoretisch in beliebigen Disziplinen verwendet werden kann, in denen Phänomene auf Basis kleiner Partikel simuliert werden. Diese Ausarbeitung befasst sich jedoch lediglich mit der Partikelsimulation in der Computergraphik.

3 Geschichte der Partikelsimulation

****

3.1 Die Anfänge ab ca. 1965

Wie [Reeves] zu entnehmen ist, begann die Geschichte der Partikelsimulation in den 1960er Jahren: Es wurden Raumschiffe in Computerspielen mittels glühenden Punkten dargestellt, es wurde Rauch, der aus einem Schornstein steigt, simuliert [CHPCH]und Sterne und drei-dimensionale Fraktale erzeugt [PBS] bzw. [Norton].

3.2 William T. Reeves Partikel-System

Ein Meilenstein in der Entwicklung der Partikelsimulation stellt das von William T. Reeves im Jahr 1983 entwickelte Partikelsystem dar. Wie schon erwähnt, gab es bereits vorher verschiedene Ideen, mittels Partikeln Objekte zu modellieren. Reeves Ansatz ist jedoch, soweit bekannt, der erste Ansatz zur Partikelsimulation, der derart flexibel und umfassend gestaltet ist, dass er sich als Ausgangspunkt für weitere Entwicklungen nennen lässt.

In der zugehörigen Publikation *Particle Systems - A Technique for Modeling a Class of Fuzzy Objects* [Reeves] beschreibt Reeves zuerst das von ihm entworfene Partikelsystem und im Folgenden dessen Umsetzung im Film *Star Trek II: The Wrath of Kahn*. In diesem Beispiel wird Feuer simuliert - Reeves Partikelsystem legt jedoch die Basis für die Simulation jeglicher vergleichbarer „Objekte“ bereit.

3.2.1 Die Theorie

Ein Partikelsystem besteht aus einem Koordinatensystem, in dem sich eine oder mehrere Emissionsflächen befinden, von der aus einzelne Partikel oder wiederum neue Partikelsysteme generiert werden. Die Emissionsfläche kann verschiedene Formen annehmen (rechteckig, kreis- oder kugelförmig). Partikel bekommen vom zugehörigen Partikelsystem verschiedene Eigenschaften zugewiesen, die ihr Verhalten und Aussehen im Laufe der Simulation bestimmen (z.B. Position, Farbe, Lebenszeit, Bewegungs-Richtung und Geschwindigkeit). Pro Frame werden neue Partikel generiert und die alten Partikel anhand ihrer Attribute neu berechnet oder gelöscht.

Aufbau

Ein Partikelsystem befindet sich in seinem eigenen lokalen Koordinatensystem, dessen Ursprung ein Punkt im 3-dimensionalen Raum bzw. Weltkoordinatensystem ist. Die Orientierung des lokalen Koordinatenystems im Weltkoordinatensystem wird mittels zwei Rotationswinkeln beschrieben. Um den Ursprung des lokalen Sysetems befindet sich eine sogenannte „generation shape“, innerhalb der die Partikel generiert werden können. Sie kann kugelförmig, kreisförmig oder rechteckig sein. Bei einer kugelförmigen Fläche, wie Abbildung 1 auf Seite 8 zu entnehmen, bewegen sich die Partikel vom Ursprung weg. Bei der kreisförmigen und der rechteckigen bewegen sie sich vertikal von der Emissionsfläche weg, wobei ein sogenannter `ejectionAngle` dem Partikel einen gewissen zufälligen Bewegungs-Spielraum lässt (Abbildung 2).

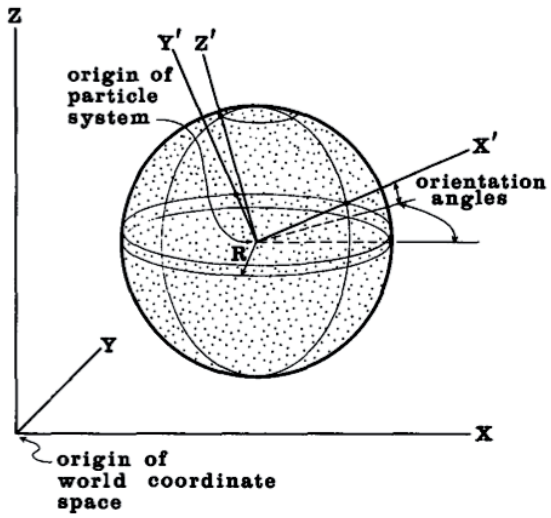

Abbildung 1: 3D-Szene

Programm-Ablauf

1. Generierung neuer Partikel

Es gibt zwei Möglichkeiten, die aktuelle Anzahl der Partikel zu berechnen: Entweder bestimmt der Anwender die durchschnittliche Partikel-Anzahl, zu der das Programm einen zufälligen Wert hinzufügt, oder das Programm bestimmt die Anzahl in Abhängigkeit von der Größe, die das Objekt auf dem Bildschirm einnimmt.

2. Setzen der Partikel-Attribute

Nachdem die Anzahl der Partikel für das aktuelle Frame bestimmt ist, werden die Partikel mit den unten genannten Attributen belegt. Der endgültige Wert wird durch stochastische Prozesse bestimmt.

Initiale Position
Initiale Geschwindigkeit und Richtung
Initiale Farbe
Initiale Grösse
Initiale Transparenz
Form
Lebenszeit

3. Bewegung

Die Geschwindigkeit, mit der sich das Partikel fortbewegt, errechnet sich wie folgt:

```
InitialSpeed = MeanSpeed + Rand()*VarSpeed
```

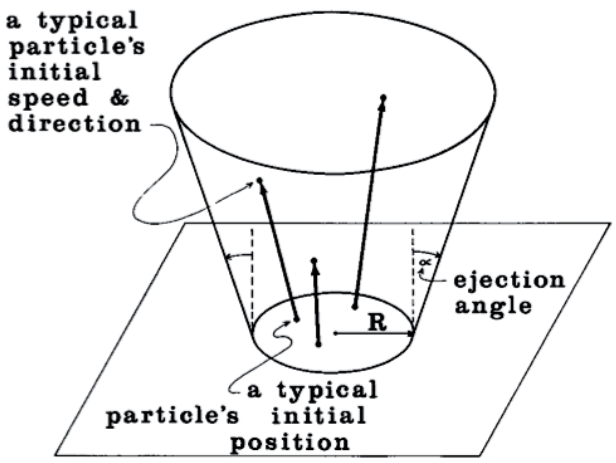

Abbildung 2: Position und Bewegungsrichtung eines Partikels

Jedes Partikel bekommt also die gleiche Start-Geschwindigkeit, zu der gewisser Varianz-Wert hinzu addiert wird. Nach dem gleichen Prinzip errechnen sich die Werte für Farbe und Transparenz jedes Partikels.

Durch Addition des Geschwindigkeits- und Positions-Vektors lässt sich die Bewegung des Partikels zwischen zwei Frames simulieren. Mit diesen Werten allein lässt sich allerdings nur eine lineare Bewegung beschreiben, weshalb noch der *accelaration-factor* eingeführt wurde. Mit Hilfe dieses Faktors lässt sich eine Partikel-Flugbahn berechnen, die z.B. die Auswirkung der Gravitationskraft berücksichtigt.

4. Löschen eines Partikels

Zu Beginn bekommt jedes Partikel eine maximale Lebenszeit. Spätestens nach Ablauf dieser Lebenszeit, die in Frames gemessen wird, wird das Partikel gelöscht. Vor Ablauf der Lebenszeit kann ein Partikel gelöscht werden, wenn seine Intensität (also die Kombination von Farb- und Transparenz-Wert) so gering ist, dass das Partikel kaum noch zu sehen ist oder aber sich das Partikel zu weit vom Ursprung des zugehörigen Partikelsystems entfernt hat.

5. Rendern

Zum Rendern werden die Partikel mittels Punkt-Licht-Quellen realisiert, mit denen sich prima das Glühen von Feuer simulieren lässt (Punkt-Licht-Quellen eignen sich allerdings nicht zur Simulation von z.B. Wasser oder Wolken). Hierbei tragen alle Partikel, die auf das gleiche Pixel projiziert werden, ihren Teil zur endgültigen Intensität des jeweiligen Pixels bei.

Partikel-Hierarchie

Sollen einzelne Partikelsysteme voneinander abhängig sein, kann dies mittels einer hierarchischen Baumstrukur umgesetzt werden. Dabei generiert ein Vorgänger-Partikelsystem keine Partikel, sondern neue Partikelsysteme. Diese Partikelsysteme können dann Partikel oder wieder Partikelsysteme generieren. Partikelsysteme des selben Astes sind dabei voneinander abhängig. Bewegt sich ein Vorgänger-Partikelsystem, bewegen sich alle Nachkommen bis hin zu den endgültigen Partikeln mit. Hiermit lässt sich globale Kontrolle auf das Objekt ausüben.
Anwendungsbeispiel ist z.B. eine Wolken-Decke: um ein Ur-Partikelsystem herum entstehen weitere in sich geschlossene Partikelsysteme, die jedoch alle abhängig von dem Ur-Partikelsystem sind. Eine Windböhe etwa würde das Ur-Partikelsystem bewegen und alle anderen mit sich ziehen.

3.2.2 Anwendungsfall „Genesis-Bombe“ in *Star Trek II: The Wrath of Khan*

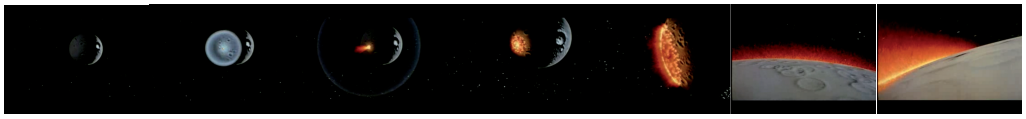

Abbildung 3: Die „Genesis-Bombe“ in *Ster Trek II: The Wrath of Khan*

Zum ersten Mal fand das oben beschriebene Partikelsystem Anwendung im Film *Star Trek II: The Wrath of Khan*. In einer Szene trifft eine Bombe (die „Genesis-Bombe“) einen toten, mondähnlichen Planeten. Die Bombe explodiert und das entstandene Feuer verbreitet sich ringförmig über den ganzen Planeten. Diese sich mit der Zeit ausbreitende Feuerwand, wie Abbildung 3 zu entnehmen, wurde mittels eines in zwei Hierarchien aufgeteilten Partikelsystems realisiert. Das *top-level-System* ist an der Einschlagstelle der Bombe platziert und generiert von dort aus *second-level-Systeme*. Hierzu werden auf der Oberfläche des Planeten Kreise errechnet, um die herum die *second-level-Systeme* zufällig verteilt werden, wobei die Anzahl der Partikelsysteme von dem jeweiligen Kreis- bzw. Planeten-Umfang und einem `density`-Parameter abhängt (siehe Abbildung 4 auf Seite 11). Diese *second-level*-Partikelsysteme wiederum generieren die endgültigen Partikel, die gemeinsam das Feuer darstellen.

Die Orientierung der *second-level*-Systeme ist hierbei so gewählt, dass sich die Partikel von der Oberfläche weg bewegen. Damit die Simulation nicht zu gleichmäßig aussieht, findet der bereits erwähnte `ejectionAngle`, der dem Partikel einen gewissen Bewegungsspielraum lässt, und der Gravitations-Parameter, der zu einer parabolischen Flugbahn des Partikels führt, Anwendung. Die Anzahl der Partikel orientiert sich an der Bildschirmfläche, die das zugehörige *second-level*-Partikelsystem in Anspruch nimmt.

Wie bereits oben erläutert, hängt die Farbe eines Partikels, das als Punkt-Licht-Quelle realisiert wird, von einem festen Durchschnittswert und einem hinzuaddierten zufälligen Varianz-Wert ab. Die Farbe des Pixels wiederum setzt sich aus allen dahinterliegenden Partikel-Farben zusammen: im Zentrum ist Feuer eher blau-weiß, wohingegen es in weiter entfernten Regionen über orange-gelbe Töne einen rötlichen Farbton annimmt. Nun wählt man für die Partikel einen relativ hohen Rot-Wert, einen kleinen Grün-Wert und einen kleinen Blau-Wert. Ein einzenes Partikel hat somit einen deutlichen Rot-Ton. Bei der Summe sehr vieler Partikel gelangt der Rot-Wert jedoch schnell an sein Maximum

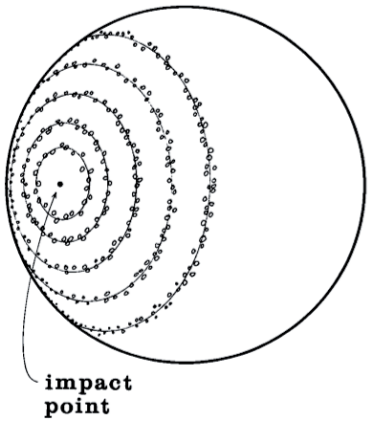

Abbildung 4: Verteilung der Partikelsysteme auf der Planetenoberfläche

und die Grün- und Blau-Werte können sich so weit aufaddieren, bis sie einen großen Teil zur Partikelfarbe beitragen. Resultat ist ein orange-gelb- bis weiß-farbiges Pixel. Nun muss nur noch die Partikel-Dichte in Nähe der Genesis-Bomben-Einschlags-Stelle sehr hoch eingestellt werden und mit wachsender Entfernung des Feuers von diesem Punkt abnehmen.

3.3 Weiterentwicklungen

****

Das von Reeves entwickelte Partikelsystem ist so strukturiert, dass es sich relativ einfach um verschiedenste Anwendungsfälle erweitern lässt. Im Prinzip handelt es sich „nur“ um ein Grungerüst, das es ermöglicht, eine große Anzahl an Partikeln (was auch immer sie gemeinsam darstellen sollen) zu generieren, zu bewegen und darzustellen. An dieser Stelle wird ein Versuch unternommen, die verschiedenen Weiterentwicklungen in drei Bereiche einzuteilen, die wiederum, wie wir sehen werden, in „Partikelsystem-Tools“ aus 3D-Animationsprogrammen oder der Spieleentwicklung zusammenfließen.

3.3.1 Spezialeffekte

Ein Bereich ist die generelle Verbesserung und die konkrete Implementierung von weiteren Spezialeffekten, die sich mit dem ersten Anwendungsfall (der „Genesis-Bombe“) direkt vergleichen lassen. Veröffentlichungen zu diesem Bereich sind etwa *Particle Animation and Rendering Using Data Parallels Computation* von Karl Sims aus dem Jahr 1990 [Sims], *Solid spaces and inverse particle systems for controlling the animation of gases and fluids* von Ebert, Carlson und Parent aus dem Jahr 1994 [ECP] oder die freie Particle System API von David K. McAllister [McAllister].

3.3.2 Schwarmverhalten

Ein weiterer Bereich, in dem das Partikelsystem Einzug fand, ist die Simulation von Schwarmverhalten, wie es bei manchen Vögeln oder Fischen zu beobachten ist. Hierzu veröffentlichte Craig Reynolds 1987 seine Forschungen in *Flocks, Herds and Schools: A Distributed Behavioral Model*. Anstelle der Partikel werden nun *Boids*, wie Reynolds sie getauft hat, generiert. Diese *Boids* haben, im Gegensatz zu Partikeln, eine Orientierung und ihr Verhalten wird zusätzlich durch verschiedene Regeln bestimmt, die die eigene Bewegung in Abhängigkeit von den Nachbarn bestimmt. So soll jedes Boid so nah am Nachbarn bleiben wie möglich ohne jedoch mit ihm zu kollidieren und sich seiner Geschwindigkeit anpassen [Reynolds].
Unter *http://www.red3d.com/cwr/boids/* ist eine Demo-Sequenz zu finden (Stand 12. Oktober 2008).

3.3.3 Darstellung von Pflanzen

Reeves selber veröffentlichte im Jahr 1985 gemeinsam mit Ricki Blau *Approximate an Probabilistic Algorithm for Shading and Rendering Structured Particle Systems* [Reeves-Blau]. Aufgabestellung war die Erzeugung eines Waldes, bestehend aus unzähligen verschieden Bäumen und Gräsern.

Hierzu wird eine gewünschte Anzahl an Pflanzen zufällig auf der Wald-Fläche verteilt. Bei der Generierung eines Baumes werden wiederum sich in einem vorher festgelegten Rahmen befindende Zufallswerte für z.B. Höhe, Breite oder Farbe erzeugt. Wesentlicher Unterschied zwischen dem hier vorgestellten Verfahren und dem oben beschriebenen ist die Tatsache, dass das hier beschriebene strukturierte Partikelsystem Abhängigkeiten zwischen einzelnen Partikelsystemen und Partikeln (und somit der einzelnen Bestandteile eines Baumes) berücksichtigt. Lässt sich also ein Feuer-Partikel lediglich einem Partikelsystem zuordnen, kann ein Ast dem vorangegangenen Ast, aus dem er entstanden ist und letztlich dem zugehörigen Baum zugeordnet werden und somit Eigenschaften (etwa eines

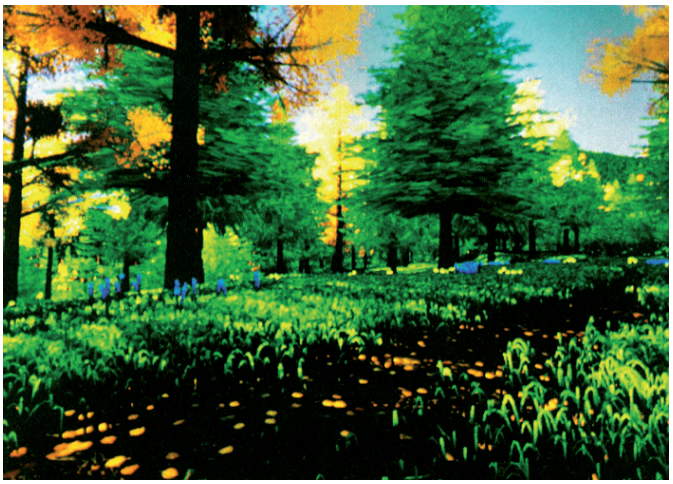

Abbildung 5: Szene aus *The Adventures of Andre and Wally B.*

bestimmten Baum-Typs) erben und vererben.
Anwendung fand das Strukturierte Partikelsystem in dem Film *The Adventures of Andre and Wally B.*.

3.4 Verwandte Disziplin

****

Eine verwandte Disziplin ist die Simulation von Gruppenverhalten. Wie auch bei der Partikelsimulation wird das Verhalten jedes einzelnen Gruppenmitgliedes durch global gesetzte Rahmenbedingungen gesteuert. Es kommt auch auf ein „großes Ganzes durch viele kleine“ an. Unterschied ist jedoch die Gewichtung zwischen Autonomie des Einzelnen und Einfluss des Ganzen. Einem Partikel kann man so gut wie keine Autonomie andichten und es ist ohne seine Nachbarn nicht wirklich existenzwürdig. Ein Gruppenmitglied hingegen muss, noch mehr als beim Schwarmverhalten, eine gewisse Autonomie zu Tage bringen, um auf Ereignisse, die während der Simulation geschehen, reagieren zu können. Ein Krieger etwa in einer Kampfszene muss sich, genauso wie all die anderen Krieger, dem Gesamtbild des Schlachtfeldes fügen aber gleichzeitig bei einer Nahaufnahme und der damit erfolgenden Aufmerksamkeit auf das eigene Verhalten eine gute Figur machen. Um derartiges zu simulieren, bieten sich die Werkzeuge der künstlichen Intelligenz wesentlich besser an.

4 Stand der Technik

****

Heutige Einsatzgebiete sind z.B. Tools in 3D-Animationsprogrammen wie Cinema 4D und Maya sowie die Spieleentwicklung. Somit dienen die hier vorgestellten Verfahren zur Partikelsimulation hauptsächlich der Unterhaltungsindustrie oder dem Designer und nicht etwa der physikalisch korrekten Bestimmung der Phänomene, wie sie etwa für naturwissenschaftliche Beobachtungen benötigt werden. Die Grenze zwischen Weiterentwicklungen der hier vorgestellten Ansätze und physikalisch korrekten Simulationen ist nach eigenem Wissensstand jedoch fließend. Um einen Eindruck zu bekommen, welche Effekte in der Computergraphik mittels Partikeln derzeit erreicht werden können, sind im Folgenden einige Einstzgebiete vorgestellt:

4.1 Das Partikelsystem-Tool aus Cinema 4D

Cinema4D von MAXON Computer GmbH ist ein 3D-Grafik-Programm, mit dem sich 3D-Szenen modellieren und animieren lassen. Im Partikelsystem-Tool von Cinema4D finden sich alle vorgestellten Einsatzmöglichkeiten (und noch viele mehr) zusammen. Es gibt einen Emitter, der beliebig viele Partikel verwaltet. Die Modellierung der Partikelsysteme geschieht mittels verschiedenster, vom Anwender veränderbarer Werte wie Emissionsrate, Lebensdauer oder Geschwindigkeit der Partikel. Ein Partikel kann hierbei alle möglichen Formen und Farben annehmen. Durch die vielen Variablen lassen sich sämtliche nur erdenkliche Simulationstypen wie etwa Rauchwolken, Feuerflammen, Fischschwärme oder Flüssigkeiten erzeugen. Die im Folgenden gezeigten Bilder sind aus dem Referenzhandbuch von Cinema4D Release 9.52 von MAXON [Maxon] entnommen:

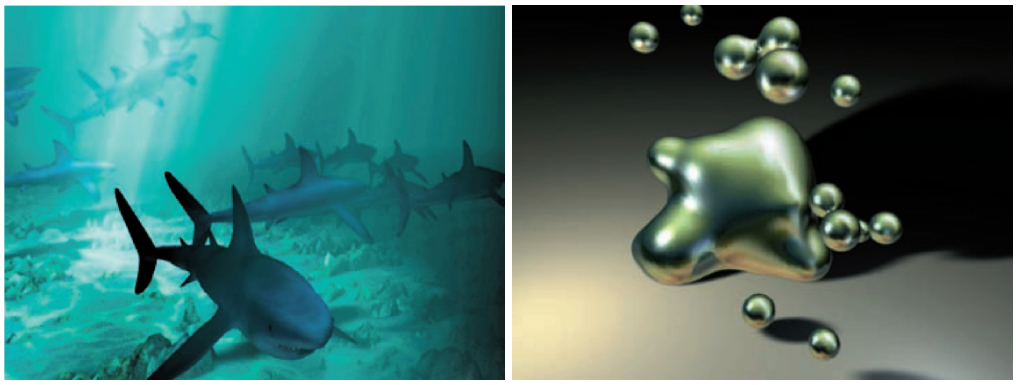

Abbildung 6: Fischschwarm und Blubberblasen mit Cinema 4D modelliert

4.2 Eine freie Partikelsystem-API von David K. McAllister

Diese freie Partikelsystem-API ist z.B für Programmierer von Computerspielen interessant. Sie bietet Quellcode für viele verschiedene Effekte und ist beliebig nach eigenen Wünschen erweiterbar. Die im Folgenden gezeigten Bilder sind der Internetseite *http://www.particlesystems.org* (Stand 28. August 2008) von McAllister entnommen [McAllister]:

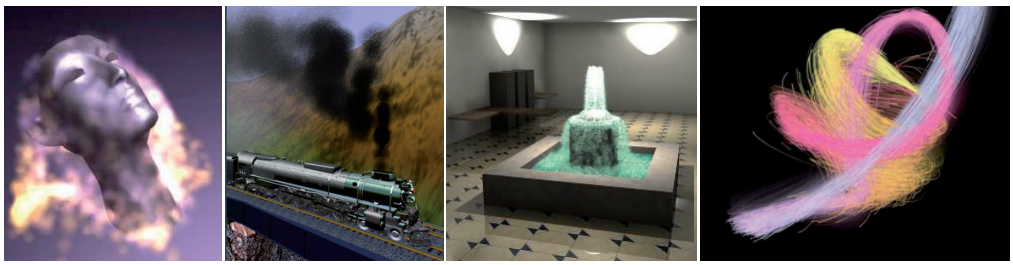

Abbildung 7: Beispiele von McAllisters Particle-System-API

4.3 GPU Gems: *Fire in the „Vulcan“ Demo*

Das in dem Kapitel *Fire in the „Vulcan“ Demo* vorgestellte Monster (Abb. 8) wurde vom NVIDIA-Demo-Team für die Einführung der Grafikkarte GeForce FX 5900 Ultra entwickelt. Das um das Monster herum lodernde Feuer besteht aus Emitter-Flächen, die auf seinem Körper platziert wurden und von dort aus Partikel in den Äther schießen. Besonders an diesem Beispiel ist der Einsatz einer Video-Textur, die der Simulation einen eindrucksvollen Effekt verleiht. Zudem ist durch den Einsatz dieser Textur eine viel geringere Anzahl an Partikeln nötig, was den Rechenaufwand verringert.
Die im Folgenden gezeigten Bilder sind dem Kapitel *Fire in the „Vulcan“ Demo* der Online-Version von GPU Gems entnommen [GPU Gems]. Auf der linken Seite der Abbildung ist das von Feuer-umgebene Monster zu sehen, rechts einige Sequenzen einer Video-Textur, mit der jedes Partikel bekleidet wurde:

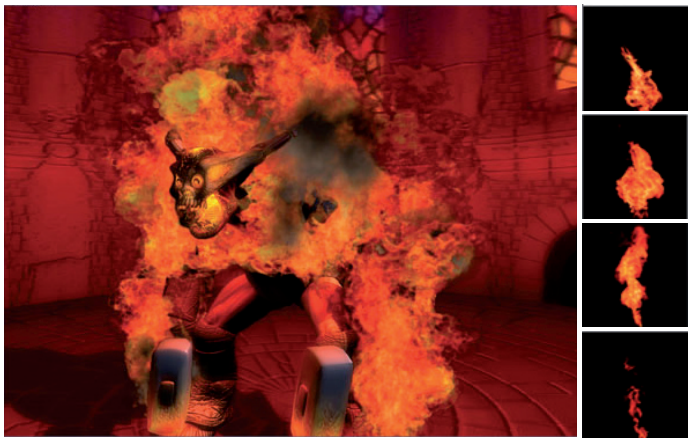

Abbildung 8: Das Monster *Vulcan* und Sequenzen einer animierten Textur

5 Entwurf von *Partikel, Partikel!*

****

5.1 Aufbau

Das Programm besteht aus den fünf Klassen `SimulationDesktopInterface`, `ParticleSystem` mit den abgeleiteten Klassen `Schnee`, `Feuer` und `Boeller`, dem Template `ParticleSystemInspector`, der Struktur `Particle` und drei globalen Funktionen. Eine aktive Simulation besteht aus genau einem `ParticleSystemInspector` und mindestens einem `ParticleSystem`. Abbildung 9 stellt eine mögliche 3D-Szene vereinfacht dar, auf Abbildung 10 auf der nächsten Seite ist die zugehörige interne Struktur zu sehen.

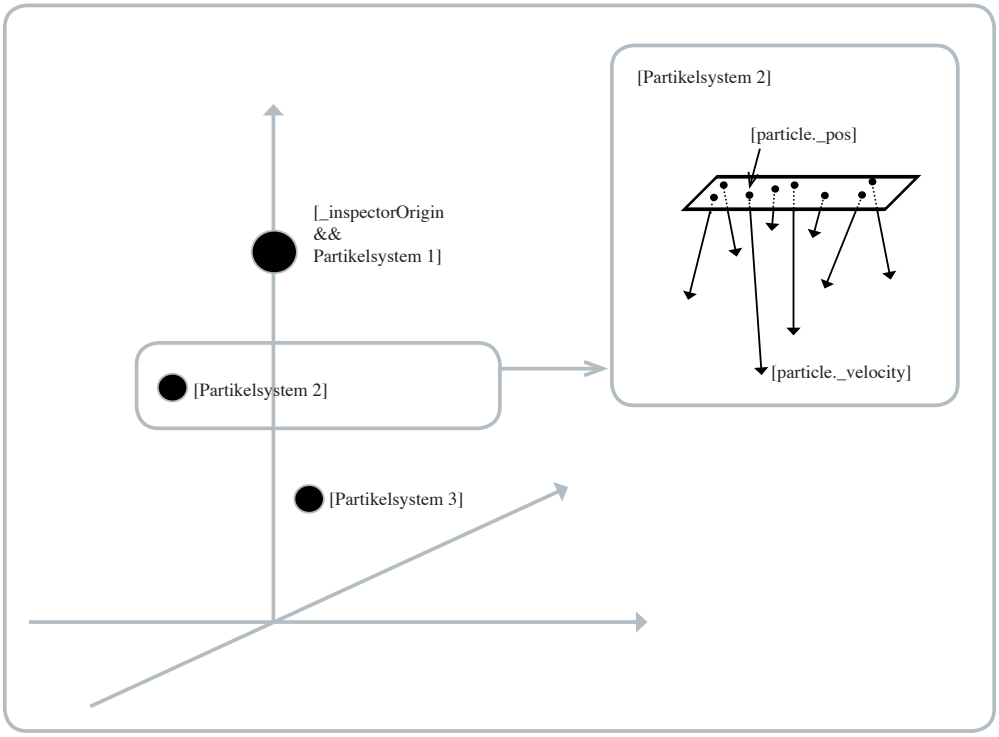

Abbildung 9: 3D-Szene

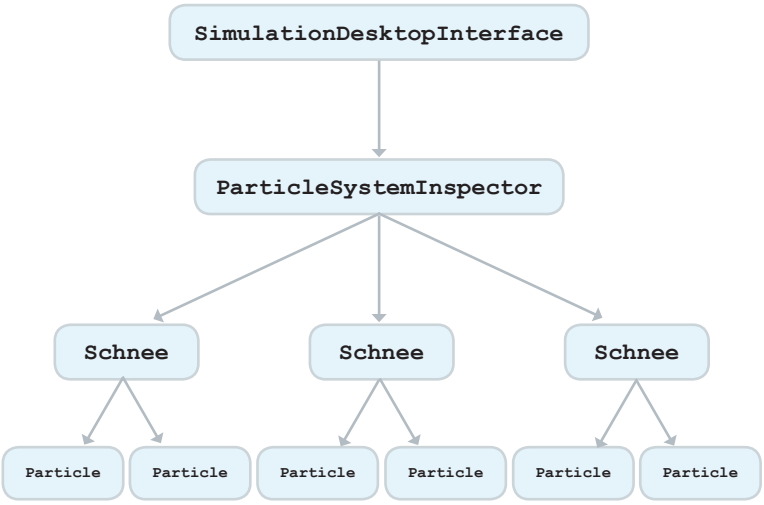

Abbildung 10: Hierarchie

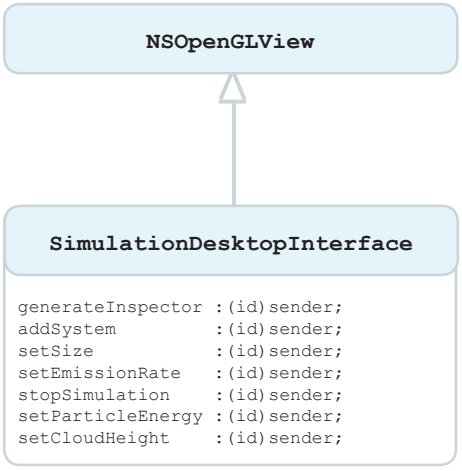

Abbildung 11: Die Klasse `SimulationDesktopInterface`

Die Klasse `SimulationDesktopInterface` ist zum einen für die Darstellung der OpenGL-Szene zuständig und stellt zum anderen die Schnittstelle zwischen der Benutzeroberfläche und dem restlichen Programm dar. Sie ist von Apples frei verfügbarer Klasse `NSOpenGLView` abgeleitet und erbt somit sämtliche Attribute und Funktionen, die zur Darstellung einer OpenGL-Szene nötig sind. Alle darüber hinaus eigens kreierten Funktionen sind für die vom Anwender gewünschten Aktionen, die mittels der verschiedenen Buttons der Benutzeroberfläche ausgelöst werden können, zuständig. So können z.B. neue Partikelsysteme hinzugefügt und die Partikel-Emissionsrate oder die Grösse der Emissionsfläche verändert werden.

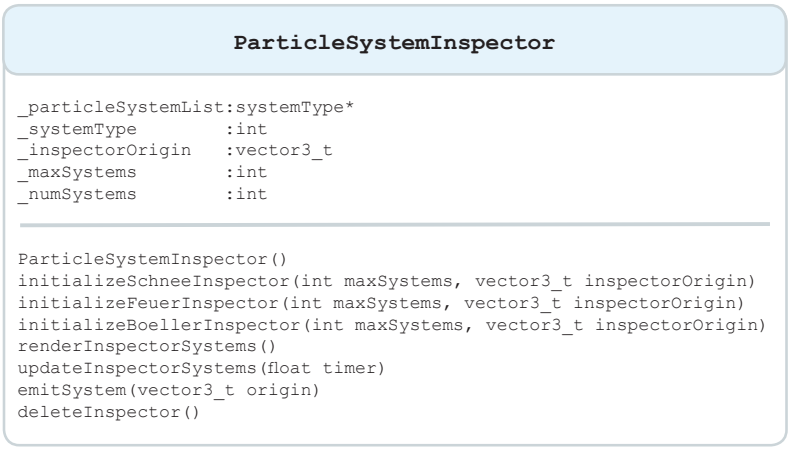

Abbildung 12: Das Template `ParticleSystemInspector`

Der `ParticleSystemInspector` ist für die Verwaltung der Partikelsysteme verantwortlich. Er besteht hauptsächlich aus einem Ursprung in der 3D-Szene, einem Datenfeld für eine bestimmte Anzahl an Partikelsystemen und Funktionen zum Steuern der Partikelsysteme. Der `ParticleSystemInspector` wurde mittels eines Templates realisiert. Wie gleich genauer geschildert wird, gibt es unterschiedliche Partikelsystem-Typen - durch die Typunabhängigkeit des Templates wird Quellcode gespart, da die meisten Funktion nur einmalig implementiert werden müssen und dann auf alle Partikelsystem-Typen angewendet werden können. Zudem bleibt das Programm überschaubarer und es kann im Nachhinein ohne großen Aufwand um weitere Partikelsystem-Typen erweitert werden.
Ein `ParticleSystemInspector` kann nur Partikelsysteme des gleichen Typs verwalten.

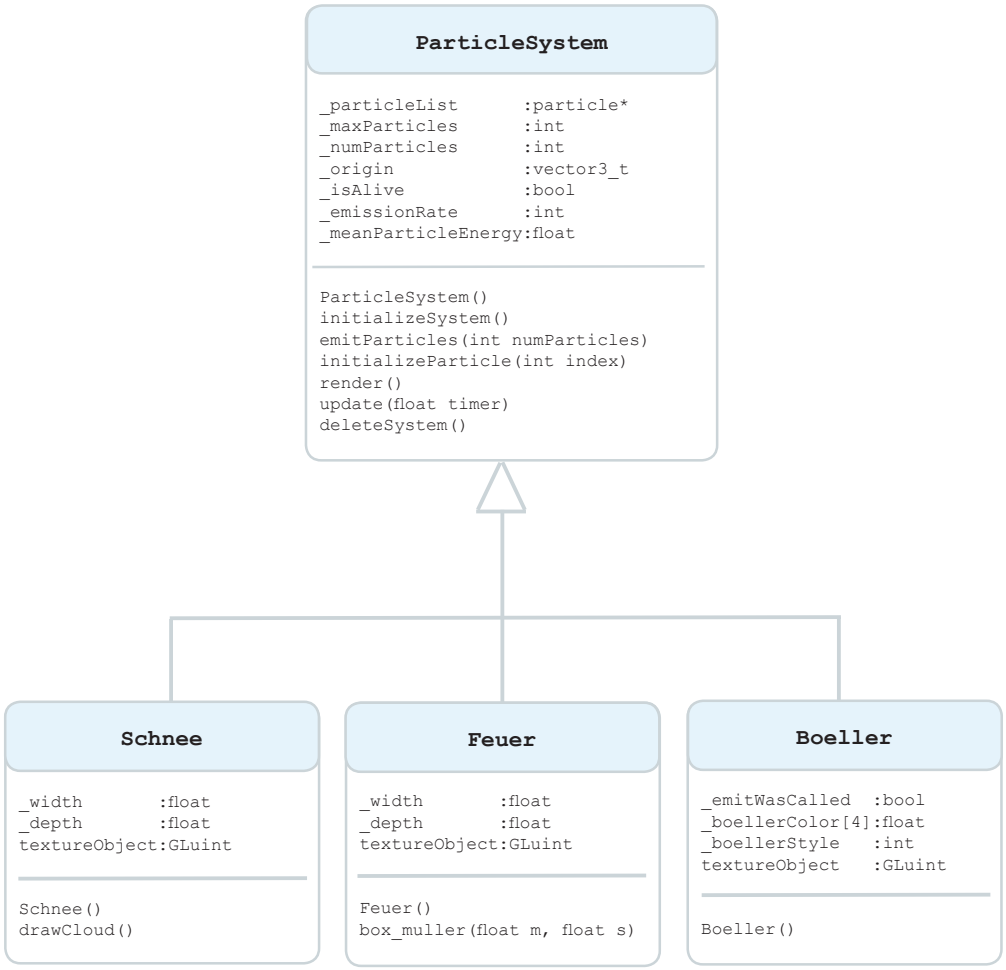

Abbildung 13: Die Klassen `ParticleSystem`, `Schnee`, `Feuer` und `Boeller`

Ein Partikelsystem wiederum ist eine abstrakte Klasse, von der alle konkreten Partikelsystem-Typen abgeleitet sind. Es besteht hauptsächlich aus einer Datenstruktur zum Speichern einer maximalen Anzahl an Partikeln, einem Ursprung und virtuellen Funktionen zum Steuern der Partikel, die in den abgeleiteten Klassen `Schnee`, `Feuer` und `Boeller` implementiert sind. Das jeweils Simulationstyp-spezifische Verhalten wird hierbei hauptsächlich in den Funktionen `initializeParticle` und `update` bestimmt.

Bei der Erzeugung eines Partikelsystems wird durch die Funktion `initializeSystem` ein Array mit der maximalen Anzahl an Partikeln erstellt und die Textur generiert. Dann wird mehrmals die Funktion `update` aufgerufen. Beim ersten Aufruf werden lediglich durch die Funktion `emitParticles` Partikel emittiert und dort wiederum durch `initializeParticle` mit Werten belegt, die das Verhalten der Partikel während der weiteren Simulation bestimmen. Bei jedem weiteren Aufruf der `update`-Funktion werden die Partikel anhand ihrer gesetzten Attribute bewegt und verändert. Die Funktionen `render` und `update` werden durch den in der Klasse `SimulationDesktopInterface` gesetzten Timer kontinuierlich aufgerufen.

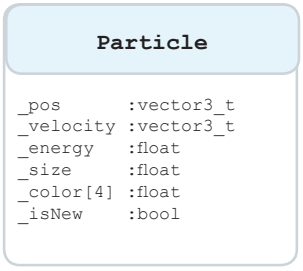

Abbildung 14: Die Struktur `Particle`

Die kleinste eigens entwickelte Datenstruktur stellt `Particle` dar. Ein Partikel besteht lediglich aus einigen Attributen wie Position, Bewegungsrichtung, Farbe und Größe und wurde mittels einer Struktur realisiert. Die Werte der Attribute, die letztlich das Verhalten der einzelnen Partikel und somit das Erscheinungsbild der Simulation bestimmen, hängen vom Partikelsystem-Typ ab. Die Bewegungsrichtung einer Schneeflocke etwa zeigt von oben nach unten, die eines Feuerfunken in entgegengesetzte Richtung.

5.2 Ablauf

Das hier vorgestellte Ablauf-Diagramm (beginnend mit Abbbildung 15 auf Seite 21) beschränkt sich auf Grund der Übersichtlichkeit auf den wesentlichen Ablauf - einige „Neben-Events“, wie das Verändern der Emissionsrate, werden nicht dargestellt.

Bei Start des Programmes wird ein Timer gesetzt, der veranlasst, dass die Szene über die Funktion `drawRect` alle 0.05 Sekunden neu berechnet und gezeichnet wird, unabhängig davon, ob eine Simulation aktiv ist oder nicht. Ist keine Simulation aktiv, ist lediglich ein schwarzes Fenster zu sehen. Wurde ein Button der Benutzeroberfläche gedrückt, wird das entsprechende Event ausgelöst. Das kann das Starten einer neuen Simulation sein, das Löschen einer vorherigen oder das Emittieren eines neuen *Schnee*- oder *Feuer*-Partikelsystems (Abbildung 16).

Eine neue Simulation wird über einen der Buttons *Schnee*, *Feuer* oder *Feuerwerk* gestartet. Ist noch eine Simulation aktiv, wird diese gestoppt - somit ist es nicht möglich, verschiedene Simulationstypen gleichzeitig laufen zu lassen (Abbildung 17 auf Seite 21). Hierbei wird ein neuer Inspector des entsprechenden Typs erzeugt: Sein Ursprung wird gesetzt, ein Partikelsystem initialisiert und innerhalb der Funktion `_particleSystemList[0].update`, die fünf mal aufgerufen wird, Partikel emittiert und bewegt (Abbildung 18 auf Seite 23).

Beim nächsten Aufruf der Funktion `drawRect` nach Generierung eines neuen Inspectors wird das Partikelsystem mit seinen Partikeln durch die Funktion `drawAndUpdateSystems` gezeichnet und wieder neu berechnet. Bei der Simulation von Schnee oder Feuer wird ein weißer bzw. roter Boden gezeichnet und folgend die `render`-Funktion des Partikelsystems aufgerufen. Dort werden alle Partikel als `GL_POINTS` gezeichnet und bei einer Schnee-Simulation als Bonus noch eine wunderschöne Wolke. Die Funktion `updateInspectorSystems` ruft die `update`-Funktion der Partikelsysteme auf, innerhalb der die Partikel bewegt und evtl. gelöscht werden. Bei einer Schnee- oder Feuer-Simulation werden anschließend neue Partikel emittiert, wodurch eine „Endlos-Simulation“für jedes Partikelsystem geschaffen wird. Anders verhält es sich bei der Simulation eines Feuer-

werks: es werden keine neuen Partikel emittiert - sind alle bei Boeller-Initialisierung generierten Partikel verglüht, wird das System gelöscht. Hier erfolgt die „Endlos-Simulation“durch kontinuierliches Erzeugen neuer Boeller: wurde ein zufällig generierter Frame erreicht, wird eine neues System bzw. ein neuer Böller generiert (Abbildungen 19 auf Seite 24 und 20 auf Seite 25.

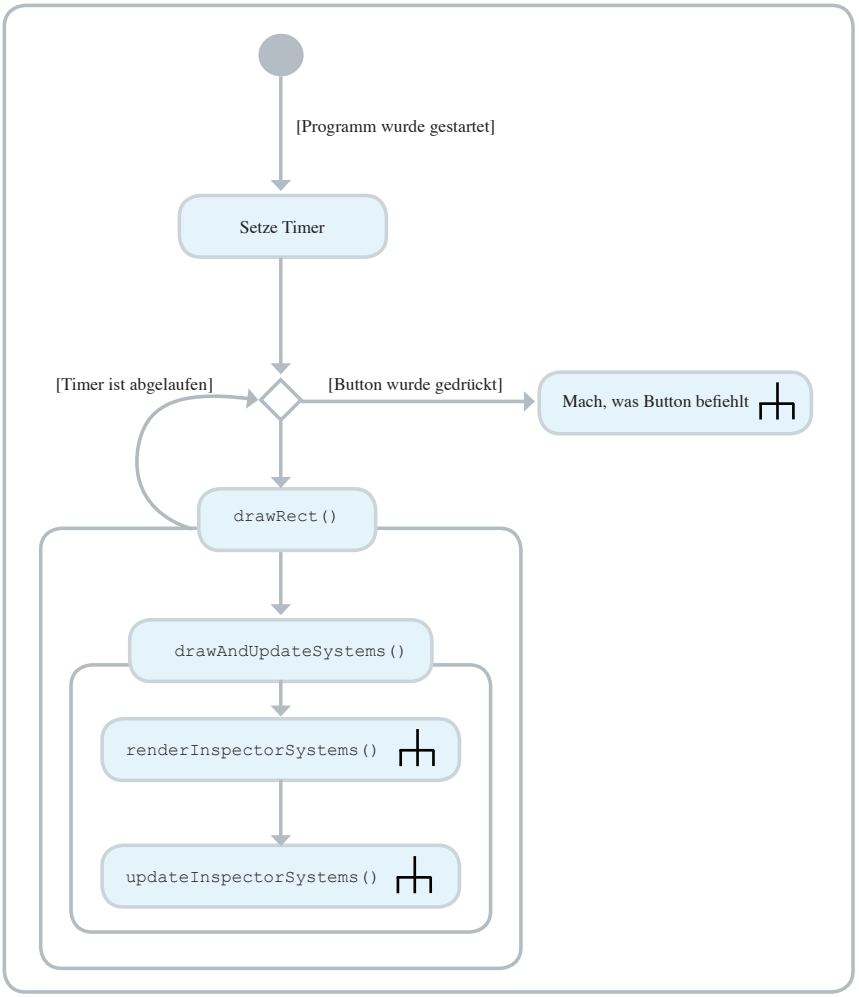

Abbildung 15: Programm-Ablauf

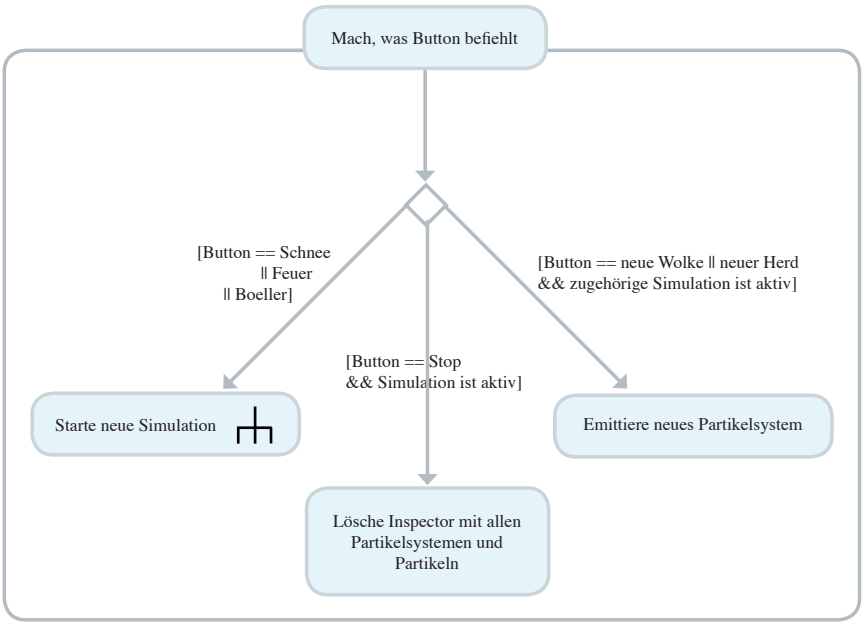

Abbildung 16: Button-Events

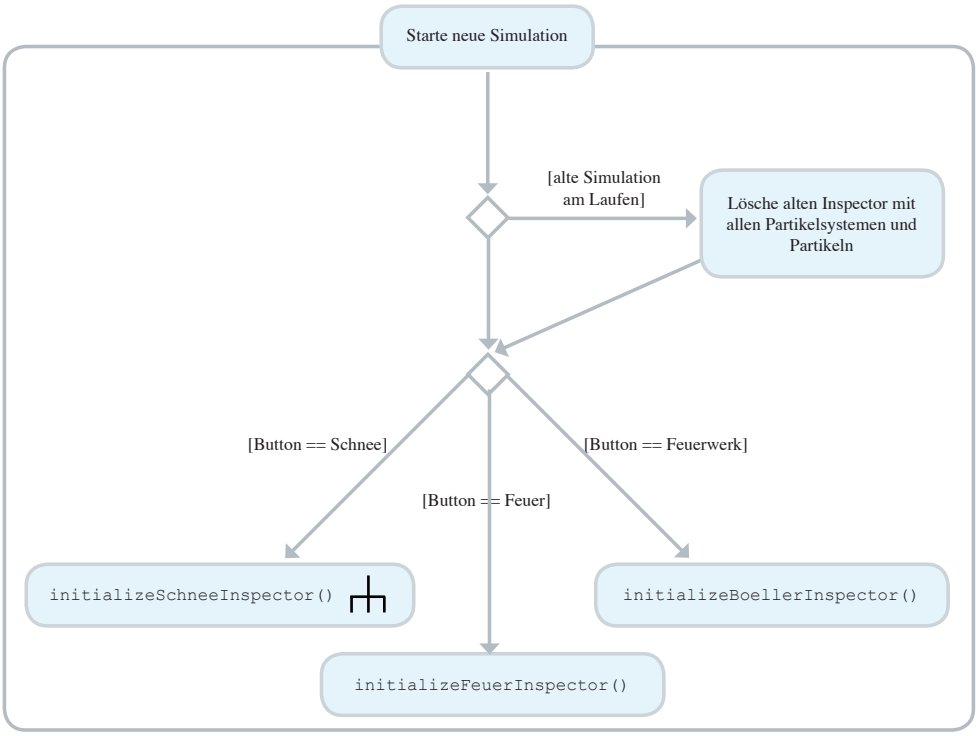

Abbildung 17: Das Starten einer neuen Simulation

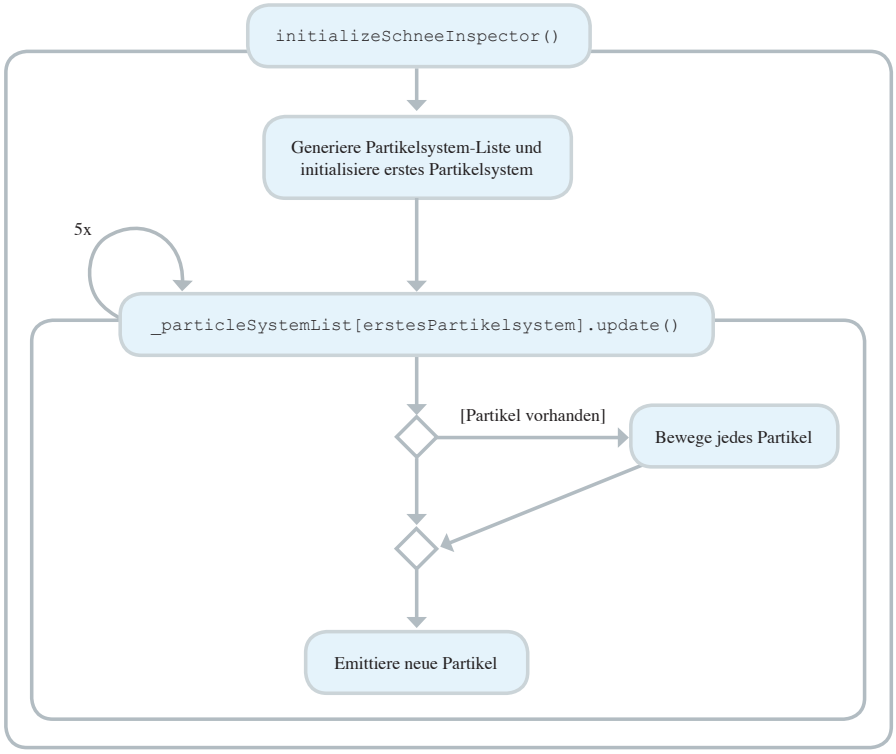

Abbildung 18: Generierung eines neuen Inspectors mit einem Partikelsystem

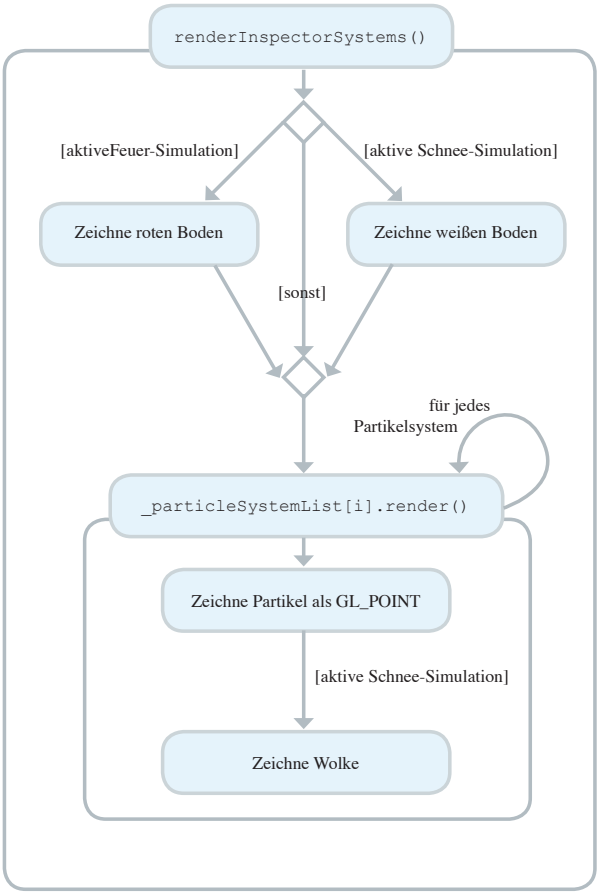

Abbildung 19: Das Rendern der Partikelsysteme

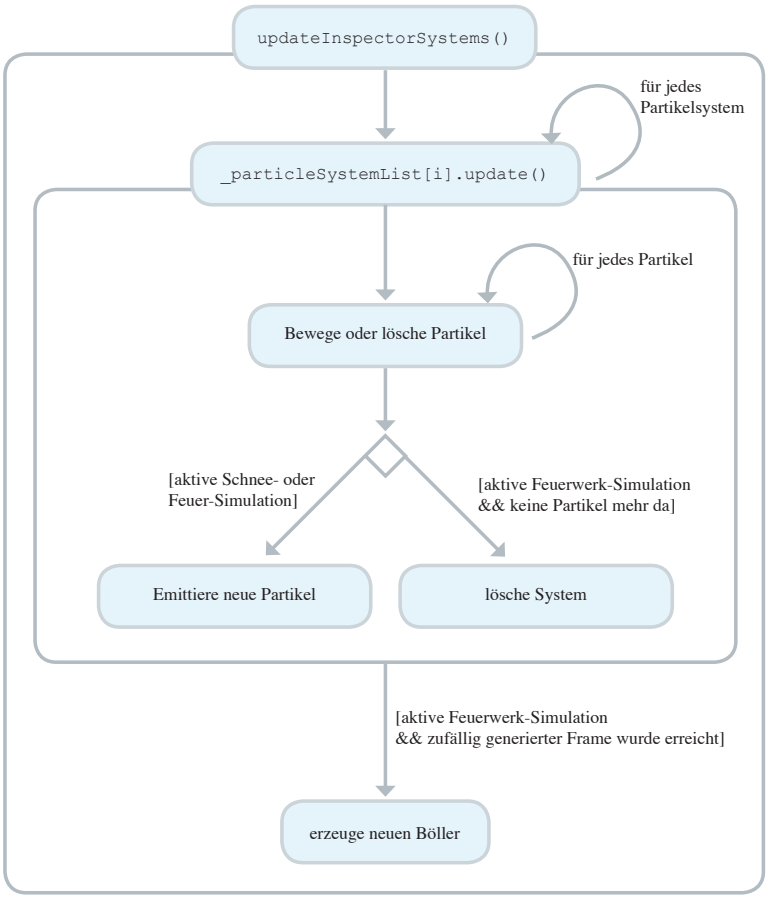

Abbildung 20: Das Updaten der Partikelsysteme

6 Entwicklungsumgebung

****

Das in diesem Rahmen entwicklte Programm wurde mit Hilfe der Entwicklungsumgebung Xcode 2.0 von Apple Inc. realisiert. Die Gestaltung der Benutzeroberfläche geschah mittels InterfaceBuilder 2.5. InterfaceBuilder, ebenfalls von Apple Inc., ist ein Programm, das für die Gestaltung von Apple-Software entwickelt wurde und dem Software-Entwickler sämtliche Obeflächen-Elemente wie Buttons oder Slider im Apple-Stil zur Vefügung stellt. Die Verbindung zwischen Programm-Code und Benutzeroberfläche geschah mittels Cocoa 6.4. Cocoa ist eine API, die für die Entwicklung von Apple-Software mit Benutzeroberfläche eingesetzt werden kann. Programmiert wurde hauptsächlich in C++. Lediglich die Schnittstelle zwischen Programm und Benutzerobefläche wurde mittels Objective-C++, also einem Mix aus Objective-C (der primären Programmiersprache zur Entwicklung von Cocoa-Software), und C++ umgesetzt. Zudem wurde OpenGL und GLUT zur graphischen Beschreibung und Darstellung der Szene verwendet.

7 Implementierung

****

In diesem Kapitel wird auf die Implementierung des obigen Entwurfes eingegangen. Alle konkreten Partikelsystem-Typen sind von der Basis-Klasse `ParticleSystem` abgeleitet. Der Programm-Ablauf der einzelnen Typen unterscheidet sich oft nur in Details - daher wird die Klasse `Schnee` genau erläutert und in den Klassen `Feuer` und `Boeller` meist nur auf die Unterschiede eingegangen. Bei Unklarheiten bzgl. des Programmaufbaus kann in der entsprechenden Funktion der Klasse `Schnee` nachgeschlagen werden. Zudem sei an dieser Stelle darauf hingewiesen, dass die ersten Gehversuche bzgl. der Partikelsystem-Programmierung mit Hilfe des Buches *OpenGL Game Programming* [Hawkins] gemacht wurden. Im Laufe der Zeit hat sich ein eigenständiges Programm entwickelt - trotzdem werden dem versierten *OpenGL Game Programming*-Leser einige Sequenzen vertraut vorkommen.

Wie Kapitel *Entwurf* zu entnehmen, besteht das fertige Programm *Partikel, Partikel!* aus fünf Klassen, einem Template, einer Struktur und drei globale Funktionen. Der Aufbau dieses Kapitels ist, soweit möglich, dem Ablauf des Programms nachempfunden. Der Einstieg in das Programm geschieht durch die Klasse `SimulationDesktopInterface`, die die Schnittstelle zwischen Benutzeroberfläche und restlichem Programm darstellt. Alle Funktionen dieser Klasse sind für Ereignisse, die durch Elemente der Benutzeroberfläche ausgelöst werden, zuständig. Die drei Funktionen `addBoellerSystem`, `drawSystems` und `stopOldSimulation` sind Funktionen, mit denen einige Funktionen der Klasse `SimulationDesktopInterface` arbeiten, die sich jedoch nicht der Schnittstelle zwischen Benutzer und Programm zuordnen lassen und sind daher in der gleichen Datei ausserhalb der Klasse global angelegt. Folgend wird das Template `ParticleSystemInspector` sowie die abstrakte Klasse `ParticleSystem` mit all ihren abgeleiteten Klassen `Schnee`, `Feuer` und `Boeller` vorgestellt. Zu guter Letzt wird die Struktur `Particle` erläutert.

7.1 Globale Variablen

****

Mittels der folgenden Zeilen wird für jeden Simulations-Typ ein Inspector global deklariert:

```
ParticleSystemInspector <Schnee> schneeInspector;
ParticleSystemInspector <Feuer> feuerInspector;
ParticleSystemInspector <Boeller> boellerInspector;
```

7.2 Die Klasse `SimulationDesktopInterface`

****

7.2.1 `generateInspector`

Die Funktion `generateInspector` wird aufgerufen, sobald einer der Buttons *Schnee*, *Feuer* oder *Feuerwerk* der Benutzeroberfläche betätigt wird. Von welchem Sender `generateInspector` ausgelöst wurde, wird mittels `(id)sender` gespeichert. Abhängig von `(id)sender` wird ein `ParticleSystemInspector` vom entsprechenden Simulations-Typ initialisiert und sein Ursprung und die maximale Anzahl an Partikelsystemen, die er gleichzeitig verwalten kann, gesetzt. War zum Aufruf-Zeitpunkt noch eine alte Simualtion aktiv, wird diese mittels `stopOldSimulation` beendet.

```
- (IBAction) generateInspector: (id) sender{

    if (simulationIsActive == true){
       stopOldSimulation();
    }
    else {
       simulationIsActive == true;
    }
    systemType = [sender tag];
    countSystems = 1;

    if (systemType == 0){
       inspectorOrigin = vector3_t(0.0, 3.0, 0.0);
       schneeInspector.initializeSchneeInspector(5, inspectorOrigin);
    }
    else if (systemType == 1){
       inspectorOrigin = vector3_t(0.0,0.0,0.0);
       feuerInspector.initializeFeuerInspector(5, inspectorOrigin);
    }
    else if (systemType == 2){
       inspectorOrigin = vector3_t(0.0,0.5,-14.0);
       boellerInspector.initializeBoellerInspector(20, inspectorOrigin);
    }
    [self setNeedsDisplay: YES];
}
```

7.2.2 `drawRect`

`drawRect` wird von der weiter unten beschriebenen Funktion `awakeFromNib` alle 0.05 Sekunden aufgerufen. Von hier aus wird über den Aufruf der Funktion `drawAndUpdateSystems` das Rendern und Aktualisieren der Szene gesteuert. Nachdem die Szene neu berechnet wurde, wird sie in Kamera- und Pixel-Daten umgerechnet.

```
- (void) drawRect: (NSRect) rect{

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (0.0, 1.0, 8.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0);
    drawAndUpdateSystems();
    glViewport(0, 0, (GLsizei) rect.size.width, (GLsizei) rect.size.height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0f, 1.0f, 1.0f, 100.0f);
    [[self openGLContext] flushBuffer];

}
```

7.2.3 addSystem

`addSystem` wird aufgerufen, sobald einer der Buttons *Neue Wolke* oder *Neuer Herd* betätigt wird. `(id)sender` bezeichnet wieder den Aufrufer bzw. Sender-Button. Es wird ein neuer, zufälliger Ursprung generiert und ein neues Partikelsystem emittiert. Mittels `addSystem` können nur Systeme vom Typ `Schnee` oder `Feuer` generiert werden, da die Böller-Generierung automatisch mittels der Funktion `addBoellerSystem` erfolgt. Diese Trennung beruht auf der Tatsache, dass es sich bei `addBoellerSystem` um eine programminterne Funktion handelt und die Funktion `addSystem` eine Schnittstelle zwischen Programm und Benutzeroberfläche darstellt.

```
- (IBAction)addSystem:(id)sender{

    if ([sender tag]== 20 && systemType == 0){
       ++countSystems;
       vector3_t newSystemOrigin;
       newSystemOrigin.x = inspectorOrigin.x + FRAND * 5 + 0.5;
       newSystemOrigin.y = inspectorOrigin.y;
       newSystemOrigin.z = inspectorOrigin.z + FRAND * 5;
       schneeInspector.emitSystem(newSystemOrigin);
    }
    else if ([sender tag]== 21 && systemType == 1){
       ++countSystems;
       vector3_t newSystemOrigin;
       newSystemOrigin.x = inspectorOrigin.x + FRAND * 5;
       newSystemOrigin.y = inspectorOrigin.y;
       newSystemOrigin.z = inspectorOrigin.z + FRAND * 5;
       feuerInspector.emitSystem(newSystemOrigin);
    }

}
```

7.2.4 setSize

`setSize` wird aufgerufen, sobald ein *Größe*-Slider betätigt wird. Es wird die Breite und die Höhe der Emissionsfläche auf den gesendeten Wert gesetzt.

```
- (IBAction)setSize:(id)sender{

    float value = (float)[sender floatValue];

    if ([sender tag]== 30 && systemType == 0){
       schneeInspector._particleSystemList [countSystems - 1]._width = value;
       schneeInspector._particleSystemList [countSystems - 1]._depth = value;
    }
    if ([sender tag]== 31 && systemType == 1){
       feuerInspector._particleSystemList[countSystems - 1]._width = value;
       feuerInspector._particleSystemList[countSystems - 1]._depth = value;
    }
}
```

7.2.5 setEmissionRate

`setEmissionRate` wird aufgerufen, sobald ein *Emissionsrate*-Slider betätigt wird. Es wird die Anzahl der pro Frame zu emittierenden Partikel gesetzt. Die Emissionsrate einer Wolke liegt zwischen 0 und 50 pro Frame, die eines Feuer-Herdes zwischen 0 und 200.

```
- (IBAction)setEmissionRate:(id)sender{

    int emissionRate = (int)[sender intValue];
    if ([sender tag]== 40 && systemType == 0){
       schneeInspector._particleSystemList[countSystems-1]._emissionRate = emissionRate;
    }
    if ([sender tag]== 41 && systemType == 1){
       feuerInspector._particleSystemList[countSystems-1]._emissionRate = emissionRate;
    }
}
```

7.2.6 setParticleEnergy

`setParticleEnergy` wird aufgerufen, sobald der Slider *Partikel-Lebenszeit* betätigt wird. Die Energie bzw. Lebensdauer der einzelnen Partikel wird auf den gesendeten Wert gesetzt. Anwendung findet diese Funkion nur bei dem Simulations-Typ `Feuer`, da die Energie der Boeller automatisch generiert wird und bei einer Schneeflocke dieses Attribut keine Verwendung findet.

```
- (IBAction)setParticleEnergy: (id)sender{

    int energy = (int)[sender intValue];

    if ([sender tag]== 51 && systemType == 1){
       feuerInspector._particleSystemList[countSystems-1]._meanParticleEnergy = energy;
    }

}
```

7.2.7 setCloudHeight

`setCloudHeight` wird aufgerufen, sobald der Slider *Wolkenhöhe* betätigt wird. Die Höhe der Wolke wird auf den gesendeten Wert gesetzt. Diese Funktion findet nur Anwendung beim Simulations-Typ `Schnee`.

```
- (IBAction)setCloudHeight: (id)sender{

    float height = (float)[sender floatValue];

    if ([sender tag]== 60 && systemType == 0){
       schneeInspector._particleSystemList[countSystems-1]._origin.y = height;
    }

}
```

7.2.8 stopSimulation

`stopSimulation` wird aufgerufen, sobald einer der *Stop*-Buttons betätigt wird. Sofern die zugehörige Simulation gerade aktiv ist (`if ([sender tag]== 1x && systemType == x)`), wird der aktive Inspector und seine zugehörigen Partikelsysteme gelöscht. `systemType = 4` bezeichnet den „Leerlauf“-Simulations-Typ. Bevor Simulations-spezifische Einstellungen vorgenommen werden, wird in jeder Funktion anhand des `systemTyps` geprüft, welche Simulation aktiv ist. Wurde nun `systemType = 4` gesetzt, ist sichergestellt, dass (egal welche Funktion ausgelöst wird), keine Programm-beeinflussenden Aktionen vorgenommen werden bis nicht ein neuer Inspector generiert wurde.

```
- (IBAction)stopSimulation:(id)sender{

    if ([sender tag]== 10 && systemType == 0){
       schneeInspector.deleteInspector();
       systemType = 4;
       countSystems = 0;
       glDisable(GL_FOG);

    }
```

```
    else if ([sender tag]== 11 && systemType == 1){
       feuerInspector.deleteInspector();
       systemType = 4;
       countSystems = 0;
       glDisable(GL_FOG);
    }
    else if ([sender tag]== 12 && systemType == 2){
       boellerInspector.deleteInspector();
       systemType = 4;
       countSystems = 0;
       frame = 0;
       meanFrame = 0;
    }
    countSystems = 0;
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);

}
```

7.2.9 awakeFromNib

Diese Funktion wird automatisch bei Programmstart ausgelöst. Es wird ein Timer gesetzt, der alle 0.05 Sekunden die Funktion `drawRect`, die wiederum die zu zeichnende Szene aufruft und neu berechnet.

```
- (void) awakeFromNib{

    NSTimer* timer;
    timer = [NSTimer scheduledTimerWithTimeInterval:0.05 target: self
            selector:@selector(drawRect:) userInfo:nil repeats:YES];

}
```

7.3 Globale Funktionen

****

7.3.1 drawAndUpdateSystems

`drawAndUpdateSystems` wird über `drawRect` durch den von `awakeFromNib` gesetzten Timer alle 0.05 Sekunden aufgerufen. Je nach gesetztem Simulations-Typ wird der aktive Inspector erst gezeichnet und dann die Geometrie für den nächsten Aufruf neu berechnet. Bei dem Simulations-Typ *Feuerwerk* kommt eine zufällige Generierung eines neuen Boellers hinzu. Es wird alle 15 Frames eine Zufallszahl im Bereich von 0 bis 10 generiert. Diese Zufallszahl plus den aktuellen Frame bestimmt den Frame, in dem ein neuer Boeller generiert werden soll. `addSystemFrame` speichert diesen Frame. Ob der Frame erreicht wurde und ein neuer Boeller generiert werden soll, wird mittels `if (frame-meanFrame == 15)` getestet.

```
static void drawAndUpdateSystems (){

    if (systemType == 0){
       schneeInspector.renderInspectorSystems();
       schneeInspector.updateInspectorSystems(0.05f);
    }
    else if (systemType == 1){
       feuerInspector.renderInspectorSystems();
       feuerInspector.updateInspectorSystems(0.05f);
    }
    else if (systemType == 2){
       boellerInspector.renderInspectorSystems();
       boellerInspector.updateInspectorSystems(0.05f);

       if (frame-meanFrame == 15){
          meanFrame = frame;
          srand((unsigned)time(NULL));
          addSystemFrame = meanFrame + rand() % 10;
       }
       if (frame == (addSystemFrame)){
          addBoellerSystem();
       }
       ++frame;
    }

}
```

7.3.2 addBoellerSystem

Mittels addBoellerSystem wird ein neuer Boeller eines Feuerwerks erzeugt. Es wird ein neuer, zufälliger Ursprung generiert. Dieser Ursprung befindet sich, relativ zum Inspector-Ursprung, in einem Bereich von jeweils -6.0 bis 6.0 in x- bzw. y-Richtung und in gleicher z-Position wie der Inspector.

```
addBoellerSystem{

    if (systemType == 2){

       ++countSystems;
       vector3_t newSystemOrigin;

       newSystemOrigin.x = inspectorOrigin.x + FRAND * 6;
       newSystemOrigin.y = inspectorOrigin.y + FRAND * 6;
       newSystemOrigin.z = inspectorOrigin.z;

       boellerInspector.emitSystem(newSystemOrigin);
    }

}
```

7.3.3 stopOldSimulation

---

stopOldSimulation wird aufgerufen, sofern während einer bereits aktiven Simulation eine neue gestartet wird, ohne dass diese vorher durch den Nutzer mittels des *Stop*-Buttons beendet wurde. Beendet der Nutzer hingegen eine aktive Simualtion mittels des *Stop*-Buttons, wird die Funktion stopSimulation und nicht die hier beschriebene stopOldSimulation ausgelöst. Diese Trennung zwischen „passivem“ und „aktivem“ Simulations-Stop ist darin begründet, dass es sich bei dem „passiven Stop“ um einen Programm-internen Ablauf handelt, bei dem „aktiven Stop“ hingegen um einen Eingriff durch den Anwender und somit der Schnittstelle zwischen Benutzeroberfläche und Programm bzw. der Klasse SimulationDesktopInterface zugeordnet ist. Zudem müssen beim aktiven Stop einige Variablen auf einen default-Wert zurückgesetzt werden, was beim passiven Stop durch die direkt im Anschluss erfolgenden neue Simulation weg fällt (siehe stopSimulation).

```
void stopOldSimulation(){

    if (systemType == 0){
       schneeInspector.deleteInspector();
       glDisable(GL_FOG);
    }
    else if (systemType == 1){
       feuerInspector.deleteInspector();
```

```
        glDisable(GL_FOG);
    }
    else if (systemType == 2){
        boellerInspector.deleteInspector();
        frame = 0;
        meanFrame = 0;
    }
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);

}
```

7.4 Das Template `ParticleSystemInspector`

****

7.4.1 Attribute

`systemType* _particleSystemList`
Zeiger auf einen bei Deklaration des Inspectors festgelegten Partikelsystem-Typ. Wird im weiteren Verlauf zu Zeiger auf erstes Element in Array mit `_maxSystems` Partikelsystemen vom Typ `systemType`.

`int _systemType`
Beschreibt den Simulations-Typ. 0 steht für Schnee, 1 für Feuer und 2 für Boeller.

`vector3_t _inspectorOrigin`
Ursprung des Inspectors in Weltkoordinaten.

`int _maxSystems`
Maximale Anzahl an Partikelsystemen, die der Inspector aufnehmen kann.

`int _numSystems`
Zählt während der Simulation die aktiven Partikelsysteme und dient in Kombination mit `_maxSystems` als Richtwert, ob ein neues Partikelsystem emittiert werden darf.

7.4.2 `ParticleSystemInspector`

In dem Konstruktor wird lediglich `_numSystems` auf Null gesetzt.

```
template <typename systemType> ParticleSystemInspector <systemType> ::
ParticleSystemInspector (){

    _numSystems = 0;
}
```

7.4.3 `initializeSchneeInspector`

In der Funktion `initializeSchneeInspector` werden diverse Attribute der Klasse `Schnee` mit Start-Werten belegt.
Bei der Initialisierung eines Inspectors wird direkt ein Partikelsystem erzeugt und `_numSystems` auf Eins gesetzt. `_numSystems` zählt die aktiven Partikelsysteme eines Inspectors. Soll ein neues System emittiert werden, wird `_numSystems` mit der maximal zulässigen Anzahl an Partikelsystemen des Inspectors verglichen um zu testen, ob noch Speicherplatz zur Verfügung steht.
Mittels `_particleSystemList = new systemType[_maxSystems]` wird ein Array mit

_maxSystems Elementen vom Typ systemType, in diesem Fall Schnee, definiert. Die Belegung von systemType mit dem Typ Schnee erfolgt bei der globalen Deklaration der Variablen schneeInspector.

Der Ursprung _origin des ersten Partikelsystems des Inspectors wird mit einem festen übergebenen Wert belegt. Breite und Höhe der Emissionsfläche werden auf 2.5 gesetzt, wobei sich dieser Wert im Nachhinein mittels des Buttons *Größe* des Simulations-Types Schnee verändern lässt.

Es folgen OpenGL-Kommandos für die Darstellung von Nebel. Die Farbe des Nebels fogColor[] ist hierbei der Farbe des Himmels glClearColor angeglichen, wodurch ein gleichmäßiger Übergang von der weißen Grundfläche, die später gezeichnet wird, hin zur Farbe des Himmels erreicht wird.

In den letzten zwei Zeilen wird das erste Schnee-Partikelsystem fertig zum Zeichnen gemacht. Bei der Initialisierung des ersten und bisher einzigen Partikelsystems des Inspectors wird ein Array mit der maximal zulässigen Anzahl an Partikeln pro Partikelsystem erzeugt. Folgend wird mehrmalig die update-Funktion der Klasse Schnee aufgerufen. Dort wird bei jedem Aufruf eine bestimmte Anzahl an Partikeln emittiert (bzw. im Partikel-Array aktiviert), diese mit einsatzfähigen Start-Werten belegt und die Schnee-Textur geladen.

```
template <typename systemType> void ParticleSystemInspector <systemType> ::
initializeSchneeInspector (int maxSystems, vector3_t inspectorOrigin){

    _systemType = 0;
    _maxSystems = maxSystems;
    _numSystems = 1;
    _particleSystemList = new systemType[_maxSystems];
    _inspectorOrigin = inspectorOrigin;
    _particleSystemList[0]._origin = _inspectorOrigin;
    _particleSystemList[0]._width = 2.5;
    _particleSystemList[0]._depth = 2.5;

    glClearColor(0.8, 0.8, 1.0, 1.0);
    float fogColor[]={0.8, 0.8, 1.0, 1.0};
    glEnable(GL_FOG);     glFogfv(GL_FOG_COLOR, fogColor);
    glFogf(GL_FOG_MODE, GL_EXP2);
    glFogf(GL_FOG_DENSITY, 0.06f);
    _particleSystemList[0].initializeSystem();
    for (int i=0; i<5; i++) {_particleSystemList[0].update(0.01f);}
}
```

7.4.4 initializeFeuerInspector

Der Ablauf von initializeFeuerInspector gleicht dem von initializeSchneeInspector, weshalb dieser hier nicht näher erläutert wird. Lediglich die Werte für den Nebel werden auf rötlich-schwarze Töne gesetzt.

```
template <typename systemType> void ParticleSystemInspector <systemType> ::
initializeFeuerInspector (int maxSystems, vector3_t inspectorOrigin){

    _systemType = 1;
    _maxSystems = maxSystems;
    _numSystems = 1;
    _particleSystemList = new systemType[_maxSystems];
    _inspectorOrigin = inspectorOrigin;
    _particleSystemList[0]._origin = _inspectorOrigin;
    _particleSystemList[0]._width = 2.3;
    _particleSystemList[0]._depth = 2.3;

    glClearColor(0.2, 0.0, 0.0, 1.0);
    float fogColor[]={0.2, 0.0, 0.0, 1.0};
    glEnable(GL_FOG);
    glFogf(GL_FOG_MODE, GL_EXP2);
    glFogfv(GL_FOG_COLOR, fogColor);
    glFogf(GL_FOG_DENSITY, 0.06f);

    _particleSystemList[0].initializeSystem();

    for (int i=0; i<5; i++) {_particleSystemList[0].update(0.01f);}
}
```

7.4.5 initializeBoellerInspector

Der Ablauf von `initializeBoellerInspector` gleicht ebenfalls dem von `initializeSchneeInspector`, weshalb dies an dieser Stelle ebenfalls nicht näher erläutert wird. Lediglich die Kommandos für den Nebel fallen weg.

```
template <typename systemType> void ParticleSystemInspector <systemType> ::
initializeBoellerInspector (int maxSystems, vector3_t inspectorOrigin){

    _systemType = 2;
    _maxSystems = maxSystems;
    _numSystems = 1;
    _particleSystemList = new systemType[_maxSystems];
    _inspectorOrigin = inspectorOrigin;
    _particleSystemList[0]._origin = _inspectorOrigin;
    glClearColor(0.0, 0.0, 0.0, 1.0);

    _particleSystemList[0].initializeSystem();

    for (int i=0; i<5; i++) {_particleSystemList[0].update(0.01f);}

}
```

7.4.6 renderInspectorSystems

Die Funktion `renderInspectorSystems` durchläuft alle aktiven Partikelsysteme im Array und ruft deren `render`-Funktion auf. Ist der Inspector vom Typ `Schnee` oder `Feuer`, wird zusätzlich mittels `glBegin(GL_QUADS)` ein weißes bzw. rotes Quadrat gezeichnet, das den Boden der Szene darstellt.

```
template <typename systemType> void ParticleSystemInspector <systemType> ::
renderInspectorSystems (){

    if (_systemType == 0){

       glColor4f(1.0f, 1.0f, 1.0f, 1.0f);

       glBegin(GL_QUADS);
          for (int x = -50; x<50; x+=5){
          for (int z = -50; z<50; z+=5){
             glVertex3i(x, 0, z );
             glVertex3i(x, 0, z+5);
             glVertex3i(x+5, 0, z+5);
             glVertex3i(x+5, 0, z );
          }
          }
       glEnd();
    }
    else if (_systemType == 1){
       glColor3f(1.0f, 0.0f, 0.0f);

       glBegin(GL_QUADS);
          for (int x = -50; x<50; x+=5){
             for (int z = -50; z<50; z+=5){
             glVertex3i(x, 0, z);
             glVertex3i(x, 0, z+5);
             glVertex3i(x+5, 0, z+5);
             glVertex3i(x+5, 0, z);
             }
          }
       glEnd();
    }

    for (int i=0; i<_numSystems; i++){
       _particleSystemList[i].render();
    }
}
```

7.4.7 updateInspectorSystems

Die Funtion `updateInspectorSystems` durchläuft alle zu Beginn des Funktionsaufrufs aktiven Partikelsysteme im Array und ruft deren `update`-Funktion auf. Nach der `update`-Funktion wird geprüft, ob das Partikelsystem nach wie vor aktiv ist. Dies geschieht mittels der boolschen Variable `_isAlive`, welche in der `update`-Funktion auf `false` gesetzt wird, sobald es im Partikelsystem keine aktiven Partikel mehr gibt. Zur Zeit findet dieser Fall nur beim Simulations-Typ *Feuerwerk* Anwendung, da bei den anderen Typen (*Schnee* und *Feuer*) in jedem Frame so viele neue Partikel emittiert werden, dass es nie zu einem Leerlauf kommt. Bei einem Feuerwerk hingegen verglühen die Partikel eines Boellers (jeder Boeller ist ein Partikelsystem) und es werden keine neuen Partikel mehr emittiert (mehr dazu siehe die jeweiligen `update`-Funktionen der Simulations-Typen). Wurde `_isAlive` auf `false` gesetzt, wird das letzte aktive Partikelsystem im Array (also das an der Stelle `_numSystems-1`), an die Stelle des aktuellen, gerade erloschenen Partikelsystems verschoben und `_numSystems` dekrementiert. Durch diesen Mechanismus müssen immer nur so viele Speicherplätze im Array durchlaufen werden, wie Partikelsysteme aktiv sind.

```
template <typename systemType> void ParticleSystemInspector <systemType> ::
updateInspectorSystems (float timer){

    for (int i = 0; i < _numSystems; ){
       _particleSystemList[i].update(timer);
       if (_particleSystemList[i]._isAlive == false){
          _particleSystemList[i]= _particleSystemList[--_numSystems];
       }
       else{
          ++i;
       }
    }
}
```

7.4.8 emitSystem

Sofern die maximale Anzahl an erlaubten Partikelsystemen noch nicht ausgeschöpft ist, wird ein neues System im Array des Inspectors erzeugt. Der Ursprung wird auf den übergebenen Wert gesetzt, `_isAlive` auf `true` und `_numSystems` um eins erhöht. Wie auch bei der Erzeugung des ersten Partikelsystems innerhalb der Funktionen `initializeFeuerInspector`, `initializeSchneeInspector` oder `initializeBoellerInspector` wird durch die `initializeSystem`-Funktion ein Array mit der maximal zulässigen Anzahl an Partikeln pro Partikelsystem erzeugt. Folgend wird mehrmalig die `update`-Funktion der jeweiligen Klasse aufgerufen. Bei jedem Aufruf wird eine bestimmte Anzahl an Partikeln emittiert (bzw. im Partikel-Array aktiviert), diese mit einsatzfähigen Start-Werten belegt und eine Textur geladen (Geneueres siehe `update`-Funktion der Klassen `Schnee`, `Feuer` und `Boeller`).

```
template <typename systemType> void ParticleSystemInspector <systemType> ::
emitSystem (vector3_t origin){

    if (_numSystems < _maxSystems){

       _particleSystemList[_numSystems].initializeSystem();
       _particleSystemList[_numSystems]._origin = origin;
       _particleSystemList[_numSystems]._isAlive = true;

       for (int i=0; i<5; i++) {
          _particleSystemList[_numSystems].update(0.01f);
       }

       ++_numSystems;
    }
}
```

7.4.9 deleteInspector

Sofern das Array mit den Partikelsysemen existiert, wird die `deleteSystem`-Funktion für jedes Partikelsystem aufgerufen und letztlich das Array gelöscht.

```
template <typename systemType> void ParticleSystemInspector <systemType> ::
deleteInspector (){

    if (_particleSystemList){

       for (int i = 0; i<_numSystems; i++){
          _particleSystemList[i].deleteSystem();
       }

       delete[]_particleSystemList;
       _particleSystemList = 0;
    }
    _numSystems = 0;
}
```

7.5 Die abstrakte Klasse `ParticleSystem`

****

7.5.1 Attribute

`particle* _particleList`
Zeiger auf ein Partikel, der bei Initialisierung des Partikelsystems zu Zeiger auf erstes Element des Partikel-Arrays wird.

`int _maxParticles`
Maximale Anzahl an Partikeln, die das Partikelsystem aufnehmen kann.

`int _numParticles`
Speichert die Anzahl der aktiven Partikel eines Partikelsystems.

`bool _isAlive`
Sind alle Partikel eines Systems gestorben, wird `_isAlive` auf `false` gesetzt und das System gelöscht.

`vector3_t _origin`
Ursprung des Partikelsystems in Weltkoordinaten.

`int _emissionRate`
Bestimmt die Anzahl der zu emittierenden Partikel pro Frame

`float _meanParticleEnergy`
Durchschnittliche Partikel-Energie, der bei Initialisierung der einzelnen Partikel ein Zufallswert hinzuaddiert wird.

7.5.2 `ParticleSystem`

Im Konstruktor der Klasse `ParticleSystem` geschieht nichts.

```
ParticleSystem :: ParticleSystem(){

}
```

7.5.3 `initializeSystem`

`initializeSystem` erstellt, sofern es noch nicht existiert, ein Array mit der maximal zulässigen Anzahl an Partikeln.

```
void ParticleSystem :: initializeSystem(){

    if (!_particleList){
       _particleList = new particle[_maxParticles];
    }
    _numParticles = 0;
}
```

7.5.4 `emitParticles`

Virtuelle Funktion, deren Implementierung in den abgeleiteten Klassen zu finden ist. Allgemeine Aufgabe ist das Emittieren neuer Partikel bzw. das Aktivieren neuer Partikel in der Partikel-Liste. Wird von der Funktion `update` aufgerufen.

7.5.5 `initializeParticle`

Virtuelle Funktion, deren Implementierung in den abgeleiteten Klassen zu finden ist. Allgemeine Aufgabe ist das Initialisieren der neu emittierten Partikel. Es werden Attribute gesetzt, die das Verhalten der Partikel während der weiteren Simulation bestimmen. Aufruf durch die Funktion `emitParticles`.

7.5.6 `render`

Virtuelle Funktion, deren Implementierung in den abgeleiteten Klassen zu finden ist. Allgemeine Aufgabe ist das Zeichnen aller Partikel in der Partikel-Liste.

7.5.7 `update`

Virtuelle Funktion, deren Implementierung in den abgeleiteten Klassen zu finden ist. Allgemeine Aufgabe ist das Neu-Berechnen der aktiven Partikel anhand ihrer durch `initializeParticle` gesetzten Attribute.

7.5.8 `deleteSystem`

`deleteSystem` setzt `_numParticles` auf Null.

```
void ParticleSystem :: deleteSystem(){

      _numParticles = 0;
}
```

7.6 Die Klasse `Schnee`

****

7.6.1 Attribute

`int _width` und `int _depth`
Breite bzw. Tiefe der Emissionsfläche

`GLuint textureObject`
Schnee-Textur

7.6.2 `Schnee`

Im Konstruktor werden einige Attribute auf Start-Werte gesetzt. Die Werte von `_width`,`_depth` und `_emissionRate` können im Nachhinein von der Benutzeroberfläche aus verändert werden.

```
Schnee :: Schnee (){

    _maxParticles = 5000;
    _particleList = 0;
    _isAlive = true;
    _width = 2.0f;
    _depth = 2.0f;
    _emissionRate = 5;
}
```

7.6.3 `initializeSystem`

In der Funktion `initializeSystem` werden Parameter für die Textur der Schneeflocken eingestellt und durch den Aufruf der Funktion `initializeSystem` der Eltern-Kasse `ParticleSystem` eine Partikel-Liste mit `_maxParticles` Partikeln erstellt. Die Textur muss im TGA-Format vorliegen. Die Wahl fiel zu Gunsten dieses Formates aus, da Aufbau und Handhabung recht einfach sind und die Möglichkeit besteht, einen Alpha-Kanal zu verwenden. Der verwendete TGA-Loader `gltLoadTGA` wurde der *OpenGL SuperBible* [Hawkins] entnommen. Die Textur wird als Point-Sprite an die einzelnen Partikel bzw. Punkte gebunden. Point-Sprites sind zum Texturieren von Partikeln optimal, da sie nur einen Punkt, realisiert mittels `GL_POINTS`, benötigen. Die Größe der Textur entspricht hierbei der Größe des Punktes. Durch den ersten Aufruf der Funktion `glTexEnvi` wird die Größe der Textur der Größe des Punktes, der später texturiert werden soll, angepasst. Durch den zweiten Aufruf wird die Farbe des zu texturierenden Objektes beim Rendern komplett durch die Textur-Farbe ersetzt.
Auf der Textur ist lediglich ein weisses Quadrat zu sehen. Die Schneeflocke entsteht erst

Abbildung 21: Alpha-Kanal der Schnee-Textur

durch Kombination des Alpha-Kanals, der sozusagen als eine Art „Schablone“ angesehen werden kann, mit entsprechenden Einstellungen in der render-Funktion. Der Alpha-Kanal besteht aus Werten zwischen 0 und 255, wobei der größte Teil aus den Werten 0=schwarz oder 255=weiss besteht.

```
void Schnee :: initializeSystem (){

    glGenTextures(1, &textureObject);
    GLbyte *pBytes;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;

    glBindTexture(GL_TEXTURE_2D, textureObject);

    glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);
    pBytes = gltLoadTGA(‘‘feuerwerk.tga’’, &iWidth, &iHeight, &iComponents,
&eFormat);

    glTexImage2D(GL_TEXTURE_2D, 0, iComponents, iWidth, iHeight, 0, eFormat,
                                                GL_UNSIGNED_BYTE, pBytes);
    free(pBytes);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

    glTexEnvi(GL_POINT_SPRITE, GL_COORD_REPLACE, GL_TRUE);
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

    ParticleSystem :: initializeSystem();
}
```

7.6.4 emitParticles

`emitParticles` initialsiert `numParticles` neue Partikel in der Partikel-Liste, sofern noch genügend Speicherplatz vorhanden ist. Diese Funktion wird in jedem Frame aufgerufen, wodurch kontinuierlich eine gewisse Anzahl an Partikeln emittiert wird.

```
void Schnee :: emitParticles (int numParticles){

    while (numParticles && (_numParticles < _maxParticles)){
       initializeParticle (_numParticles++);
       --numParticles;
    }
}
```

7.6.5 initializeParticle

An dieser Stelle werden die Grundlagen für das Simulations-Typ-spezifische Verhalten der einzelnen Partikel gelegt. Ein Schnee-Partikel bekommt eine zufällige Größe zwischen 0.0 und 39.0. Die y-Position des Partikels ist durch die Höhe des System-Ursprungs bestimmt. Die x- bzw. z-Position ist ein Zufallswert auf der Emissionsfläche. Die Bewegungs-Richtung und Geschwindigkeit des Partikels `_velocity` setzt sich aus den globalen Variablen `mean_Snowflake_Velocity = (0.0, -0.6, 0.0)` und `mean_Snowflake_Velocity_Tolerance = (0.15, 0.3, 0.15)` zusammen. `FRAND` ist ein Zufallswert zwischen -1.0 und 1.0. Seine Berechnung erfolgt innerhalb der Datei vectorlib.h, die dem Buch *OpenGL Game Programming* entnommen wurde [Hawkins]. Ergebnis der Berechnung ist also ein Vektor, der in eine zufällige Richtung rund um `mean_Snowflake_Velocity` zeigt wobei die maximale Abweichung durch `mean_Snowflake_Velocity_Tolerance` gesetzt ist. Die endgültige Bewegung der Schneeflocke entsteht nun durch wiederholtes Aufrufen der Funktion `update`, in der die Flocke jeweils entlang des hier berechneten Vektors `_velocity` bewegt wird (siehe `update`).

```
void Schnee :: initializeParticle(int index){

    _particleList[index]._size = rand()/(float)RAND_MAX + rand() % 38;
    _particleList[index]._pos.x = _origin.x + FRAND * _width;
    _particleList[index]._pos.y = _origin.y;
    _particleList[index]._pos.z = _origin.z + FRAND * _depth;

    _particleList[index]._velocity.x = mean_Snowflake_Velocity.x + FRAND *
                                       mean_Snowflake_Velocity_Tolerance.x;
    _particleList[index]._velocity.y = mean_Snowflake_Velocity.y + FRAND *
                                       mean_Snowflake_Velocity_Tolerance.y;
    _particleList[index]._velocity.z = mean_Snowflake_Velocity.z + FRAND *
```

```
                                        mean_Snowflake_Velocity_Tolerance.z;
}
```

7.6.6 drawCloud

In der Funktion `drawCloud` werden mittels `glutSolidSphere` drei Kugeln gezeichnet, diese skaliert, transliert und beleuchtet. Jedes mal, wenn ein neues Partikelsystem erzeugt wird, wird automatisch um den Ursprung des Partikelsystems eine neue Wolke generiert.

```
void Schnee :: drawCloud(){

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientLight);
    glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, ambientLight);
    glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
    glMaterialfv(GL_FRONT, GL_SPECULAR, specref);
    glMateriali (GL_FRONT, GL_SHININESS, 128);

    glPushMatrix();
       glTranslatef(_origin.x, _origin.y - 0.3, _origin.z + _depth);
       glScalef(8.5, 1.0, 1.0);
       glutSolidSphere((_width/10.0), 20, 5);
    glPopMatrix();

    glPushMatrix();
       glTranslatef(_origin.x, _origin.y + (_width * 0.1) - 0.3,
                                                  _origin.z + _depth);
       glScalef(6.0, 1.0, 0.7);
       glutSolidSphere((_width/10.0), 20, 5);
    glPopMatrix();

    glPushMatrix();
       glTranslatef(_origin.x, _origin.y + (_width * 0.2) - 0.3,
                                                  _origin.z + _depth);
       glScalef(3.0, 1.0, 0.5);
       glutSolidSphere((_width/10.0), 20, 5);
    glPopMatrix();

    glDisable(GL_LIGHT0);
    glDisable(GL_LIGHTING);
    glDisable(GL_COLOR_MATERIAL);
}
```

7.6.7 `render`

In der Funktion `render` werden Einstellungen für Blending, Point-Sprites und Alpha-Test vorgenommen sowie die Textur für die Schneeflocken geladen. Durch den eingeschalteten Alpha-Test werden nur die Bereiche gezeichnet, deren Alpha-Wert größer als 0.1 ist - also alle Bereiche, die in der schwarz-weiss-Darstellung des Alpha-Kanals nicht schwarz sind (siehe `initializeSystem`). Durch das zusätzlich aktivierte Blending werden die differenzierten Alpha-Werte im Bereich von 0.1 bis 1.0 berücksichtigt. Es findet eine Vermengung des blauen Himmels und der weissen Schneeflocke statt, wobei die Höhe des Alpha-Wertes die Gewichtung bestimmt: Je höher der Wert im Alpha-Kanal, desto mehr Gewicht hat das Weiss der Textur.
Folgend wird an jeder aktuellen Partikel-Position mittels `GL_POINTS` ein Punkt gezeichnet und auf diesen die Textur gelegt.

```
void Schnee :: render(){

    vector3_t partPos;

    glEnable(GL_POINT_SPRITE);

    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, textureObject);

    glEnable(GL_ALPHA_TEST);
    glAlphaFunc(GL_GREATER, 0.1);

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    for (int i=0; i<_numParticles; ++i){

       glPointSize(_particleList[i]._size);

       glBegin(GL_POINTS);
          partPos = _particleList[i]._pos;
          glVertex3f(partPos.x, partPos.y, partPos.z);
       glEnd();
    }

    glDisable(GL_TEXTURE_2D);
    glDisable(GL_POINT_SPRITE);
    glDisable(GL_ALPHA_TEST);
    glDisable(GL_BLEND);

    drawCloud();
}
```

7.6.8 update

Durch ständiges Aufrufen der update-Funktion wird die Bewegung der Partikel realisiert. Bei jedem Aufruf wird jedes Partikel um den Faktor timer in Richtung _velocity bewegt.

Beim Simulations-Typ *Schnee* stirbt ein Partikel, sobald es den Boden berührt, also seine y-Position kleiner Null ist. Ist dies der Fall, wird das letzte aktive Partikel der Partikel-Liste an Position _numParticles-1 an die Stellle des aktuellen, gestorbenen Partikels geschrieben. Anschließend werden mittels der Funktion emitParticles neue Partikel erzeugt. Der Wert von _emissionRate kann hierbei durch den Slider *Emissionsrate* veränder werden.

```
void Schnee :: update(float timer){

    for (int i = 0; i < _numParticles; ){

       _particleList[i]._pos = _particleList[i]._pos + _particleList[i]._velocity
* timer;

       if (_particleList[i]._pos.y <= 0){
          _particleList[i]= _particleList[--_numParticles];
       }
       else{
          ++i;
       }
    }
    emitParticles(_emissionRate);
}
```

7.6.9 deleteSystem

Sofern eine Textur vorhanden ist, wird diese gelöscht. Durch Aufruf der deleteSystem-Funktion der Eltern-Klasse wird _numParticles auf Null gesetzt.

```
void Schnee :: deleteSystem(){

    if (glIsTexture(textureObject)){
       glDeleteTextures(1, &textureObject);
    }

    ParticleSystem :: deleteSystem();
}
```

7.7 Die Klasse `Feuer`

****

7.7.1 Attribute

`int _width` und `int _depth`
Breite bzw. Tiefe der Emissionsfläche

`GLuint textureObject`
Feuer-Textur

7.7.2 `Feuer`

Es werden wesentlich mehr Partikel als bei der Schnee-Simulation zur Verfügung gestellt, da sich erst ab einer gewissen Anzahl an Partikeln der Eindruck eines „großen Ganzen“ erwecken lässt. Die Größe der Emssionsfläche ist kleiner, wobei sich dieser Wert im Nachhinein durch die Benutzeroberfläche verändert lässt. Zudem findet hier das Partikel-Attribut `_meanParticleEnergy` Verwendung. `_meanParticleEnergy` beschreibt die durchschnittliche Lebenszeit eines Partikels, der bei Initialisierung noch ein zufälliger Wert hinzuaddiert wird (siehe `initializeParticle`). Dieser Wert wird in jedem Frame dekrementiert und dient als Vergleichswert, ob das Partikel noch am Leben ist oder gelöscht werden kann (siehe `update`).

```
Feuer :: Feuer (){

    _maxParticles = 12000;
    _particleList = 0;
    _isAlive = true;
    _width = 2.0f;
    _depth = 2.0f;
    _emissionRate = 400;
    _meanParticleEnergy = 10.0;
}
```

7.7.3 `initializeSystem`

Bis auf eine Ausnahme gleicht der Ablauf dem von `initializeSystem` der Klasse `Schnee`: Es werden Parameter für die Textur der Funken eingestellt und durch den Aufruf der Funktion `initializeSystem` der Eltern-Kasse `ParticleSystem` eine Partikel-Liste mit `_maxParticles` Partikeln erstellt.

Die Textur entsteht wieder aus einem weissen Quadrat und einem Alphakanal, der als Funken-„Schablone“ dient. Im Gegensatz zur Schnee-Simulation scheint durch Wahl des Parameters `GL_TEXTURE` der Funktion `glTexEnvi` die Farbe der Partikel durch die Stellen

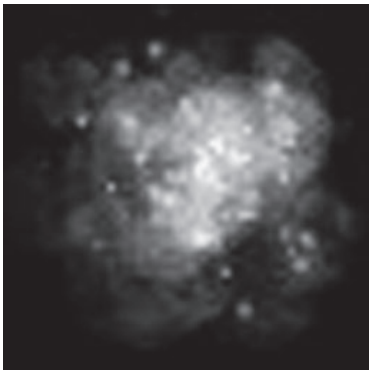

Abbildung 22: Alpha-Kanal der Feuer-Textur

der Textur hindurch, die den Alpha-Test in der `render`-Funtion bestehen.

```
void Feuer :: initializeSystem (){

    glGenTextures(1, &textureObject);
    GLbyte *pBytes;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;
    glBindTexture(GL_TEXTURE_2D, textureObject);

    glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);
    pBytes = gltLoadTGA(``Feuer.tga'', &iWidth, &iHeight, &iComponents, &eFormat);

    glTexImage2D(GL_TEXTURE_2D, 0, iComponents, iWidth, iHeight, 0,
                                          eFormat, GL_UNSIGNED_BYTE, pBytes);
    free(pBytes);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

    glTexEnvi(GL_POINT_SPRITE, GL_COORD_REPLACE, GL_TRUE);
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_TEXTURE);

    ParticleSystem :: initializeSystem();
}
```

7.7.4 emitParticles

Der Ablauf gleicht dem von `emitParticles` der Klasse `Schnee`:
Es werden `numParticles` neue Partikel in der Partikel-Liste initialisiert, sofern noch genügend Speicherplatz vorhanden. Diese Funktion wird in jedem Frame aufgerufen, womit kontinuierlich eine gewisse Anzahl an Partikeln emittiert wird.

```
void Feuer :: emitParticles (int numParticles){

    while (numParticles && (_numParticles < _maxParticles)){
       initializeParticle (_numParticles++);
       --numParticles;
    }
}
```

7.7.5 box_muller

Die Funktion `box_muller` ist [taygeta] entnommen. Sie berechnet einen Gauss-verteilten Zufallswert. Die übergebenen Werte `m` und `s` stellen den Erwartungswert bzw. die Standardabweichung dar. Sie wird aus der Funktion `initializeParticle` heraus aufgerufen.

```
float Feuer :: box_muller(float m, float s){

    float x1, x2, w, y1;
    static float y2;
    static int use_last = 0;

    if (use_last){
       y1 = y2;
       use_last = 0;
    }
    else{
       do {
          x1 = 2.0 * rand()/(float)RAND_MAX - 1.0;
          x2 = 2.0 * rand()/(float)RAND_MAX - 1.0;
          w = x1 * x1 + x2 * x2;
       } while ( w >= 1.0 );
       w = sqrt( (-2.0 * log( w ) ) / w );
       y1 = x1 * w;
       y2 = x2 * w;
       use_last = 1;
    }
    return( m + y1 * s );
}
```

7.7.6 initializeParticle

Wie auch bei der Simulation von Schnee werden hier sämtliche Attribute der Partikel mit Werten belegt, die das Verhalten der Partikel während der Simulation bestimmen. Für jedes Partikel werden in Abhängigkeit der Größe der Emissionsfläche zwei Gauss-verteilte Zufallswerte mit einem Erwartungswert von 0.0 generiert. Die Position des Partikels errechnet sich durch Addition des System-Ursprungs und der Zufallswerte. Die höchste Partikel-Dichte ist nun um den System-Ursprung und nimmt „glockenförmig“ bei Entfernung von diesem ab.

Größe, Energie, Farbe und Geschwindigkeit des Partikels sind wiederum abhängig von seiner Position bzw. dem Zufallswert `a`. Größe, Lebenszeit und Geschwindigkeit nehmen hierbei mit größer werdender Distanz zum Ursprung ab. Die Farbe der Partikel im Zentrum des Feuers haben einen gelblichen Ton und gehen mit größer werdendem Abstand ins Rötliche über.

```
void Feuer :: initializeParticle (int index){

    float a = box_muller(0.0, _width/5);
    float b = box_muller(0.0, _depth/5);

    _particleList[index]._pos.x = _origin.x + a;
    _particleList[index]._pos.y = _origin.y;
    _particleList[index]._pos.z = _origin.z + b;

    _particleList[index]._size = rand() % 13 - 10 * fabs(a);

    _particleList[index]._energy = _meanParticleEnergy - 1.5 * fabs(a);

    _particleList[index]._color[0]= 1.0;
    _particleList[index]._color[1]= 1.0 - fabs(a)/2.0;
    _particleList[index]._color[2]= 0.5 - fabs(a)/2.0;
    _particleList[index]._color[3]= 1.0;

    _particleList[index]._velocity.x = mean_Firespark_Velocity.x + FRAND *
                                mean_Firespark_Velocity_Tolerance.x;
    _particleList[index]._velocity.y = fabs (mean_Firespark_Velocity.y +
                   FRAND * mean_Firespark_Velocity_Tolerance.y - 0.5*fabs(a));
    _particleList[index]._velocity.z = mean_Firespark_Velocity.z + FRAND *
                                mean_Firespark_Velocity_Tolerance.z;
}
```

7.7.7 render

Der Ablauf gleicht dem dem von `render` der Klasse `Schnee`. Point-Sprites, Texturierung, Alpha-Test und Blending werden aktiviert und folgend alle Partikel als texturierte Punkte gezeichnet.

```
void Feuer :: render (){

    vector3_t partPos;

    glEnable(GL_POINT_SPRITE);

    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, textureObject);

    glEnable(GL_ALPHA_TEST);
    glAlphaFunc(GL_GREATER, 0.1);

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    for (int i=0; i<_numParticles; ++i){

       glPointSize(_particleList[i]._size);

       glBegin(GL_POINTS);
          partPos = _particleList[i]._pos;

          glColor4f( _particleList[i]._color[0],
             _particleList[i]._color[1],
             _particleList[i]._color[2],
             _particleList[i]._color[3]);

          glVertex3f(partPos.x, partPos.y, partPos.z);
       glEnd();
    }

    glDisable(GL_TEXTURE_2D);
    glDisable(GL_POINT_SPRITE);
    glDisable(GL_ALPHA_TEST);
    glDisable(GL_BLEND);
}
```

7.7.8 update

Alle Partikel werden entlang des Vektors _velocity bewegt. Der Grün- und Blau-Wert wird dekrementiert und somit wird der Farbton des Partikels rötlicher. Die Größe des Partikels wird in Abhängigkeit von seiner Energie (die wiederum von seiner Position abhängig ist) verändert: je geringer die Energie, desto größer scaleDown und desto schneller verkleinert sich das Partikel. Durch diesen Mechanismus schwirren Partikel, die sich nicht im „Kernbereich"des Feuers befinden, nicht so lange einsam in der Gegend herum. Sobald die Energie erloschen ist oder das Partikel kleiner als 0.0 ist, wird es aus der Liste gelöscht. Anschließend werden neue Partikel emittiert.

```
void Feuer :: update (float timer){

    for (int i = 0; i < _numParticles; ){

       _particleList[i]._pos = _particleList[i]._pos +
                           (_particleList[i]._velocity * timer * 3);

       _particleList[i]._color[1]-= 0.01;
       _particleList[i]._color[2]-= 0.05;

       float scaleDown = 1 - (_particleList[i]._energy/15.0);

       if (_particleList[i]._size >= scaleDown){
          _particleList[i]._size -= scaleDown;
       }

       _particleList[i]._energy -= 1.0f;

       if (_particleList[i]._energy <= 0.0f || _particleList[i]._size <= 0.0){
          _particleList[i]= _particleList[--_numParticles];
       }

       else{
          ++i;
       }
    }
    emitParticles(_emissionRate);
}
```

7.7.9 deleteSystem

Der Ablauf gleicht dem von `deleteSystem` der Klasse `Schnee`:
Sofern eine Textur vorhanden, wird diese gelöscht. Durch Aufruf der `deleteSystem`-Funktion der Eltern-Klasse wird `_numParticles` auf Null gesetzt.

```
void Schnee :: deleteSystem(){

    if (glIsTexture(textureObject)){
       glDeleteTextures(1, &textureObject);
    }

    ParticleSystem :: deleteSystem();
}
```

7.8 Die Klasse `Boeller`

****

7.8.1 Attribute

`GLuint textureObject`
Funken-Textur

`float _boellerColor[4]`
Farbe aller Partikel eines Böllers. Wird in `initializeSystem` zufällig generiert.

`int _boellerStyle`
Bestimmt die Art und Weise, wie die Partikel eines Boellers initialisiert werden. Es gibt drei verschiedene Typen: bei Typ eins bilden die Partikel einen Kreis mit hoher Partikeldichte, bei Typ 2 und 3 eine Kugel mit wesentlich geringerer Partikel-Dichte (siehe `initializeParticle`).

7.8.2 `Boeller`

Im Konstruktor werden alle zu Beginn benötigten Attribute auf Start-Werte gesetzt. `_boellerStyle`, dessen Wert eine zufällige Zahl im Bereich von 0 bis 3 ist, bezeichnet verschiedene Böller-Typen, deren Aussehen sich durch unterschiedliche Initialisierung der Partikel unterscheiden (siehe `initializeParticle`). Im Gegensatz zu den vorherigen Simulaions-Typen wird an dieser Stelle kein Wert für `_emissionRate` gesetzt. Dieser Wert wird zufällig in der Funktion `update` gesetzt.

```
Boeller :: Boeller (){

    _maxParticles = 5000;
    _particleList = 0;
    _isAlive = true;
    _meanParticleEnergy = 25.0;
    _boellerStyle = rand() % 4;
}
```

7.8.3 `initializeSystem`

Bis auf eine Ausnahme gleicht der Ablauf dem von `initializeSystem` der Klasse `Boeller`: die Rahmenbedingugen für die Textur werden gesetzt und die Funktion `initializeSystem` der Eltern-Klasse `ParticleSystem` aufgerufen. Zudem wird für jeden Böller eine Farbe generiert: Für jeden der RGB-Werte wird eine Zufallszahl zwischen 0.0 und 1.0 erzeugt und diesem Wert 0.3 hinzu addiert. Da OpenGL jeden Farb-Wert über 1.0 auf 1.0 abrundet, wird durch diesen Weg für jeden Farbkanal ein Wert zwischen 0.3 und

1.0 erzeugt. Durch das Entfernen der dunklen Töne im Bereich von 0.0 bis 0.3 bekommen die Partikel eine bessere Leuchtkraft.
Die Feuerwerks-Textur ist die gleiche, wie die Schnee-Textur.

```
void Boeller :: initializeSystem (){

    _emitWasCalled = false;

    _originX = _origin;

    glGenTextures(1, &textureObject);
    GLbyte *pBytes;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;

    glBindTexture(GL_TEXTURE_2D, textureObject);

    glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);
    pBytes = gltLoadTGA(``Feuerwerk.tga'', &iWidth, &iHeight, &iComponents,
&eFormat);

    glTexImage2D(GL_TEXTURE_2D, 0, iComponents, iWidth, iHeight, 0,
                                        eFormat, GL_UNSIGNED_BYTE, pBytes);
    free(pBytes);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

    glTexEnvi(GL_POINT_SPRITE, GL_COORD_REPLACE, GL_TRUE);
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_TEXTURE);

    ParticleSystem :: initializeSystem();

    _boellerColor[0]= rand()/(float)RAND_MAX + 0.3;
    _boellerColor[1]= rand()/(float)RAND_MAX + 0.3;
    _boellerColor[2]= rand()/(float)RAND_MAX + 0.3;
}
```

7.8.4 emitParticles

Bis auf eine Ausnahme gleicht der Ablauf dem von `emitParticles` der Klasse `Schnee`: `emitParticles` initialsiert `numParticles` neue Partikel in der Partikel-Liste, sofern noch genügend Speicherplatz vorhanden. Im Gegensatz zu den anderen Simulationstypen wird hier die Funktion jedoch nur ein mal nach Initialisierung des Boellers aufgerufen: Für jedes Partikelsystem wird sie ein Mal aufgerufen und dann während der Lebzeit des Systems nicht mehr. Wird also bei den ersten beiden Typen kontinuierlich eine gewisse Anzahl an Partikeln zum Leben erweckt, bekommt ein Boeller zu Beginn eine gewisse Anzahl an Partikeln zur Verfügung gestellt, die im Laufe der Simulation verglühen und somit stirbt das System (siehe `update-Funktion` der jeweiligen Klassen).

```
void Boeller :: emitParticles(int numParticles){

    while (numParticles && (_numParticles < _maxParticles)){
       initializeParticle(_numParticles++);
       --numParticles;
    }
}
```

7.8.5 initializeParticle

Wie auch in den `initializeParticle`-Funktionen der beiden anderen Simulations-Typen werden hier sämtliche Einstellungen vorgenommen, die Verhalten, Aussehen und Lebenszeit der einzelnen Partikel bestimmen, wobei der Ablauf hier etwas komplexer ist: jedes Partikel kekommt eine Energie im Bereich von 55.0 bis 60.0 und die Böller-Farbe, die bei Initialisierung des Böllers zufällig generiert wurde. Zu Beginn ist die Position aller Partikel der Böller-Ursprung, von dem sie dann entlang des `_velocity`-Vectors weg bewegt werden. Die Komponenten von `_velocity` setzen sich aus `mean_Boellerspark_Velocity` und einem Zufallswert zusammen. `mean_Boellerspark_Velocity` ist ein zufällig generierter Vektor, der in irgendeine Richtung im Raum zeigt und dessen Komponenten alle im Bereich von -1.0 bis 1.0 liegen. Der hinzugefügte Zufallswert für die endgültige Richtung und Geschwindigkeit des Partikels hängt von `_boellerStyle` ab. Die Partikel der ersten beiden Typen bilden gemeinsam eine größer werdende Kugel. Der Unterschied liegt nur in der Geschwindigkeit, mit der sich die Partikel bewegen. Die Partikel des dritten Böller-Typs hingegen bewegen sich kreisförmig vom Ursprung weg. Um die Wahrscheinlichkeit der Generierung des dritten Böller-Typs zu erhöhen, wird dieser erzeugt, wenn die Zufallszahl `_boellerStyle` 2 oder 3 ist.

```
void Boeller :: initializeParticle(int index){

    _particleList[index]._energy = _meanParticleEnergy + 30 + FRAND * 5;

    _particleList[index]._color[0]= _boellerColor[0];
    _particleList[index]._color[1]= _boellerColor[1];
```

```
    _particleList[index]._color[2]= _boellerColor[2];
    _particleList[index]._color[3]= 1.0;

    _particleList[index]._pos.x = _origin.x;
    _particleList[index]._pos.y = _origin.y;
    _particleList[index]._pos.z = _origin.z;

    int winkel1 = rand() % 360;
    int winkel3 = rand() % 360;

    vector3_t mean_Boellerspark_Velocity (cos(winkel1), sin(winkel1), cos(winkel3));

    if (_boellerStyle == 0){
       _particleList[index]._velocity.x =
                                mean_Boellerspark_Velocity.x + FRAND * 15;
       _particleList[index]._velocity.y =
                                mean_Boellerspark_Velocity.y + FRAND * 15;
       _particleList[index]._velocity.z =
                                mean_Boellerspark_Velocity.z + FRAND * 15;
    }
    else if(_boellerStyle == 1){
       _particleList[index]._velocity.x =
                                mean_Boellerspark_Velocity.x + FRAND * 2;
       _particleList[index]._velocity.y =
                                mean_Boellerspark_Velocity.y + FRAND * 2;
       _particleList[index]._velocity.z =
                                mean_Boellerspark_Velocity.z + FRAND * 2;
    }
    else if(_boellerStyle == 2 || _boellerStyle == 3){
       _particleList[index]._velocity.x =
                                mean_Boellerspark_Velocity.x * 2;
       _particleList[index]._velocity.y =
                                mean_Boellerspark_Velocity.y * 2;
       _particleList[index]._velocity.z = 0.0;
    }
}
```

7.8.6 render

Diese Funktion unterscheidet sich nur in wenigen Details von der `render`-Funktion der Klasse `Schnee`. Es werden die gleichen Einstellungen für Blending, Point-Sprites und Alpha-Test vorgenommen sowie die Textur für die Funken geladen. Folgend wird an jeder aktuellen Partikel-Position mittels `GL_POINTS` ein Punkt gezeichnet und auf diesen die Textur gelegt.

Im Gegensatz zur Größe der Schneeflocken, die einmalig bei Initialisierung erzeugt wird, wird die Größe der Böller-Funken für jedes Partikel bei jedem Aufruf der `render`-Funktion neu berechnet. Durch dieses Verfahren wird der Eindruck von flackernden, aufblitzenden Funken erweckt. Durch entsprechende Einstellungen bei der Erzeugung der Textur (siehe `initializeSystem`) scheint die Farbe der Partikel `_color` durch die Textur hindurch und es entsteht ein kunterbuntes Feuerwerk, ohne dass unzählige verschiedenfarbige Texturen erstellt werden müssen.

```
void Boeller :: render(){

    vector3_t partPos;

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    glEnable(GL_ALPHA_TEST);
    glAlphaFunc(GL_GREATER, 0.0);

    glEnable(GL_POINT_SPRITE);

    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, textureObject);

    for (int i=0; i<_numParticles; ++i){

       float _size = (rand()/(float)RAND_MAX) + (rand() % 20);
       glPointSize(_size);

       glBegin(GL_POINTS);
          partPos = _particleList[i]._pos;
          glColor3f( _particleList[i]._color[0],
             _particleList[i]._color[1],
             _particleList[i]._color[2]);
          glVertex3f(partPos.x, partPos.y, partPos.z);
       glEnd();
    }
    glDisable(GL_ALPHA_TEST);
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_POINT_SPRITE);
    glDisable(GL_BLEND);
}
```

7.8.7 update

Die update-Funktion dieser Klasse ist anders aufgebaut als die der anderen Klassen: Anstatt kontinuierlich bei jedem Aufruf neue Partikel zu emittieren und somit eine „endlos“-Simulation zu erzeugen, wird für jeden Böller einmalig eine zufällige Anzahl an Partikel erzeugt. Diese Anzahl hängt vom Böller-Typ ab, wobei einer Mindest-Partikel-Anzahl von 70 bzw. 200 ein Zufallswert von 0 bis 799 hinzuaddiert wird. Bei jedem weiteren Aufruf der update-Funktion wird zuerst mittels if(_numParticles == 0) geprüft, ob noch Partikel am Leben sind. Ist dies nicht der Fall, wird _isAlive auf false gesetzt, was wiederum zur Folge hat, dass das Partikelsystem beim nächsten Update der Partikelsystem-Liste durch den Inspector gelöscht wird (siehe update von Inspector). Sind noch Partikel am Leben, werden sie entlang ihres _velocity-Vektors vom System-Ursprung weg bewegt. Die Geschwindigkeit, mit der sie sich bewegen, hängt hierbei von ihrem Abstand zum Ursprung ab: Befinden sie sich in einem Umkreis kleiner 5.0, werden sie relativ schnell hinaus geschossen. Ab einem Abstand größer 5.0 verlangsamt sich die Geschwindigkeit. Anschließend wird die Energie des Partikels dekrementiert und, wenn es erloschen ist, aus der Liste gelöscht.

```
void Boeller :: update(float timer){

    if (_emitWasCalled == false){

       int emitFirecracker;
       srand((unsigned)time(NULL));

       if(_boellerStyle == 1){
          emitFirecracker = 70 + rand() % 800;
       }
       else{
          emitFirecracker = 200 + rand() % 800;
       }
       emitParticles(emitFirecracker);
       _emitWasCalled = true;
    }
    else {

       if(_numParticles == 0){
          _isAlive = false;
       }
       else{
          for (int i = 0; i < _numParticles; ){

          float a = _particleList[i]._pos.x - _origin.x;
          float b = _particleList[i]._pos.y - _origin.y;
          float c = _particleList[i]._pos.z - _origin.z;
          float d = sqrt(a*a + b*b + c*c);

          if (d >= 5.0){
```

```
            _particleList[i]._pos =
                 _particleList[i]._pos +
                 _particleList[i]._velocity *
                 log(_particleList[i]._energy) * 0.01;
          }
          else{
             _particleList[i ]._pos =
                 _particleList[i]._pos +
                 _particleList[i]._velocity *
                  timer * (_particleList[i ]._energy/5.0);
          }
          _particleList[i]._energy -= 1.0f;

          if (_particleList[i]._energy <= 0.0f){
             _particleList[i]= _particleList[--_numParticles];
          }
          else{
             ++i;
          }
          }
       }
    }
}
```

7.8.8 deleteSystem

Der Ablauf gleicht dem von `deleteSystem` der vorangegangenen Partikelsysteme: Sofern eine Textur vorhanden ist, wird diese gelöscht. Durch Aufruf der `deleteSystem`-Funktion der Eltern-Klasse wird `_numParticles` auf Null gesetzt.

```
void Schnee :: deleteSystem(){

    if (glIsTexture(textureObject)){
       glDeleteTextures(1, &textureObject);
    }

    ParticleSystem :: deleteSystem();
}
```

7.9 Die Struktur `Particle`

****

`vector3_t _pos`
Aktuelle Position des Partikels. Sie wird bei jedem Aufruf der update-Funktion entlang des Richtungs- und Geschwindigkeits-Vektors `_velocity` verändert.

`vector3_t _velocity`
Bewegungsrichtung eines Partikels. Setzt sich aus einer Durchschnittsrichtung und einem Zufallswert zusammen.

`float _energy`
Lebenszeit des Partikels. Dient sie als Richtwert, wann das Partikel „verglüht“ ist und gelöscht werden kann.

`float _size`
Größe des Partikels. Wird verwendet, wenn das Partikel in der render-Funktion als `GL_POINTS` gezeichnet wird

`float _color[4]`
Farbe des Partikels mit Alphakanal.

8 Ergebnisse

****

Das finale Programm bietet die Möglichkeit, eine Schnee-, Feuer- oder Feuerwerks-Simulation zu betrachten und einige Werte, die das Aussehen der Simulation bestimmen, zu verändern.

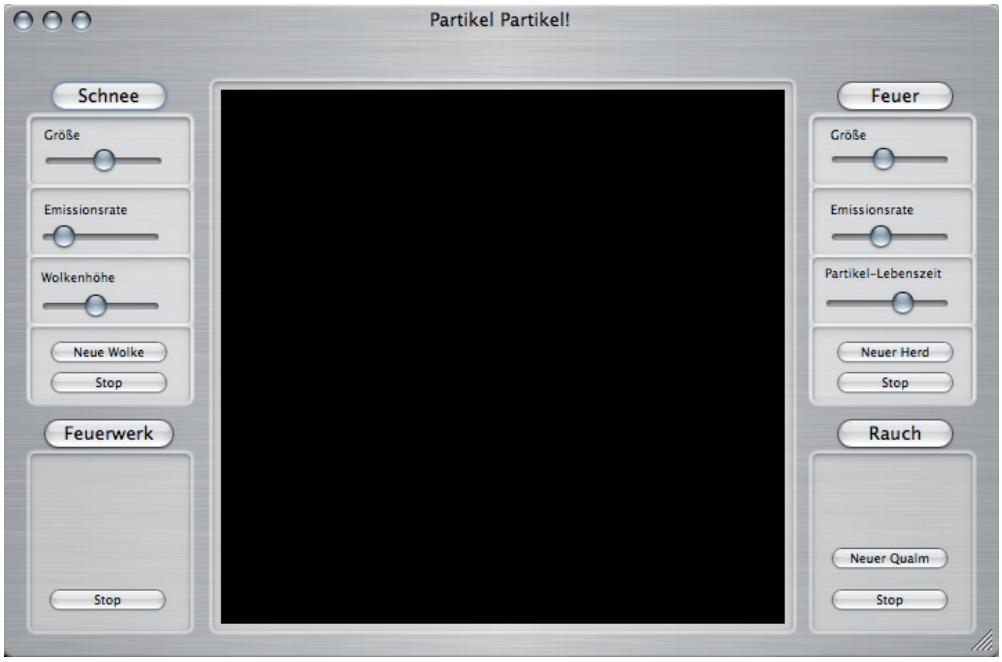

Abbildung 23: Benutzeroberflaäche

Die Benutzeroberfläche besteht hierbei zum einen aus einem sich im Mittelpunkt befindenden Fenster zur graphischen Darstellung der Szene und zum anderen aus vier um das Fenster herum arrangierte Bereiche zum Steuern der verschiedenen Simulations-Typen. Mittels der Buttons *Schnee*, *Feuer* oder *Feuerwerk* wird die entsprechende Simulation gestartet. Da die Simulation von Rauch im Rahmen der Studienarbeit nicht implementiet wurde, ist dieser Button funktionslos. Hat man sich für einen Simulations-Typ entschieden, sind alle, nicht diesem Simulations-Typ angehörigen Steuerelemente, die sich unterhalb des Buttons befinden, „außer Gefecht gesetzt“. Lediglich das Starten eines neuen Simulations-Types ist möglich, wobei in diesem Fall die derzeitige Simulation automatisch gestoppt wird. Soll eine Simulation gestoppt werden, ohne eine andere Simulation zu starten, gelingt dies mit Hilfe des zugehörigen „Stop“-Buttons. Ist dies geschehen, ist lediglich eine schwarze Hintergrundfläche zu sehen.
Im Folgenden ist die Bedienung nach Simulations-Typen sortiert näher erläutert:

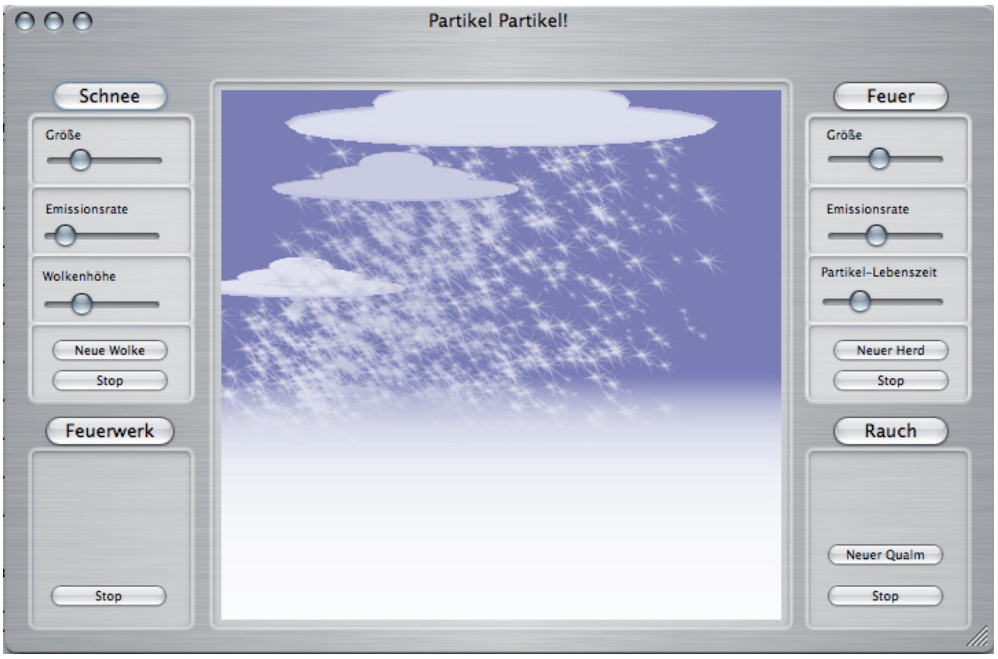

Abbildung 24: Die finale Schnee-Simulation

8.1 Schnee

Wie bereits erwähnt wird die Simulation von Schnee mittels des Buttons *Schnee* gestarten. Es erscheint eine Wolke, aus der heraus Schneeflocken auf einen weißen Boden fallen. Der Ursprung der Wolke befindet sich an der Position (0.0, 3.7, 0.0) wobei der y-Wert, also die Höhe der Wolke, im Nachhinein mittels des Sliders *Wolkenhöhe* verändert werden kann. Die obere Grenze wurde hierbei so gewählt, dass es möglich ist, die Wolke so weit nach oben zu verschieben, dass sie selbst nicht mehr zu sehen ist und statt dessen nur noch der verschneite Horizont. Zudem kann die Größe der Wolke verändert werden. Die Emissionsrate liegt zu Beginn bei 5 Partikeln pro Aufruf der Funktion `emitParticles` (also pro Frame) und lässt sich im Nachhinein ebenfalls verändern. Eine weitere Wolke wird mittels des Buttons *Neue Wolke* erstellt, wobei es ab diesem Zeitpunkt nicht mehr möglich ist, Parameter der vorherigen Wolke zu verändern - die Möglichkeit, Parameter zu verändern gilt also nur für die zuletzt erzeugte Wolke. Die Wahl der variablen, von der Benutzeroberfläche aus veränderlichen Parameter beschränkt sich hierbei auf einige, nach eigener Auffassung wesentliche Parameter und ist keineswegs erschöpfend (mehr hierzu sie Kapitel Fazit).

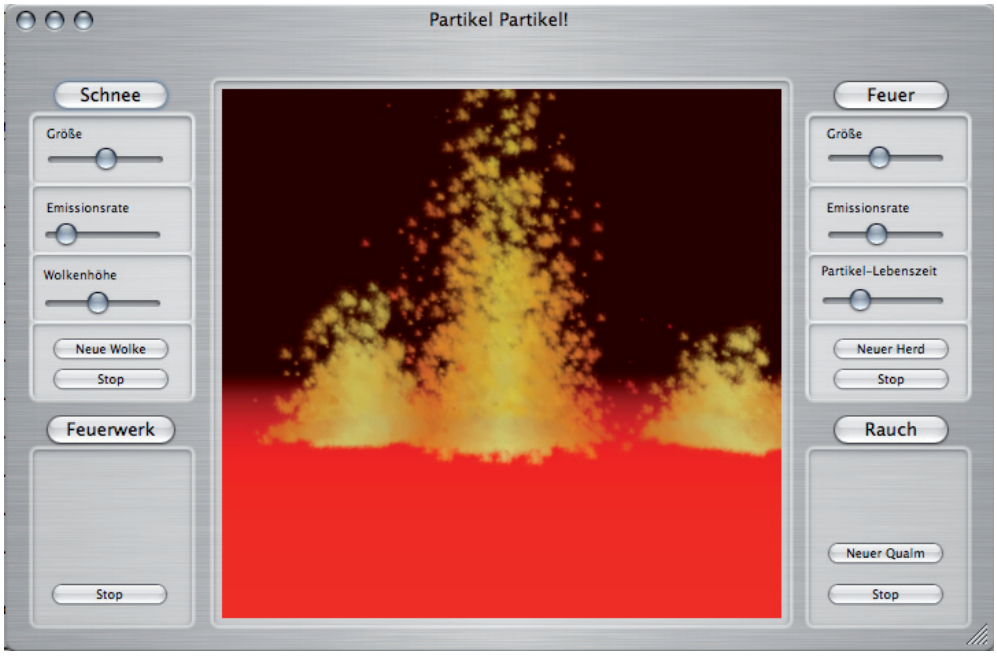

Abbildung 25: Die finale Feuer-Simulation

8.2 Feuer

Die Simulation von Feuer ist der Simulation von Schnee sehr ähnlich, weshalb die Bedienugselemente einer Feuer-Simulation, bis auf eine Ausnahme, denen der Schnee-Simulation gleichen. An dieser Stelle wird nur auf den Unterschied eingegangen: Anstelle des Sliders *Wolkenhöhe* ist ein Slider *Partikellebenszeit* zu sehen. Die Partikellebenszeit bestimmt die Höhe des Feuers.

8.3 Feuerwerk

Bei der Simulation eines Feuerwerks hat der Benutzer kaum Mitspracherecht - es ist ihm lediglich möglich, die Simulation zu starten und zu stoppen. Der Grund liegt m Aufbau des Feuerwerks: Bei den anderen Simulations-Typen läuft die Emission neuer Partikel eines einzelne Partikelsystems in einer „Endlos-Schleife“, die erst aufhört, sobald die Simulation aktiv durch den Benutzer beendet wird - die Wolke etwa schneit ununterbrochen. Bei einem Feuerwerk hingegen werden kontinuierlich neue Partikelsysteme, also Böller, erzeugt. Sobald alle Partikel eines Böllers erloschen sind, wird er automatisch gelöscht. Hier wird die „Endlos-Schleife“ also durch ständiges, automatisches Generieren neuer Partikelsysteme erreicht, und nicht durch das ständige Generieren neuer Partikel der vorhandenen Partikelsysteme. Sämtliche, während des Programmablaufs benötigten Werte werden zufällig erzeugt oder sind feste Default-Werte.

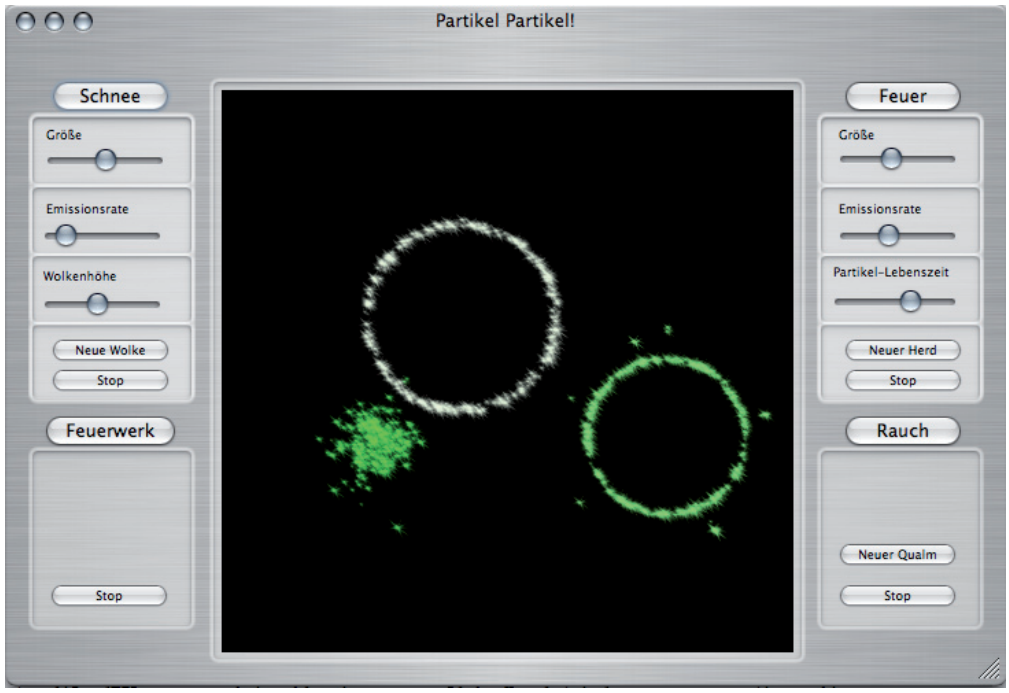

Abbildung 26: Die finale Feuerwerks-Simulation

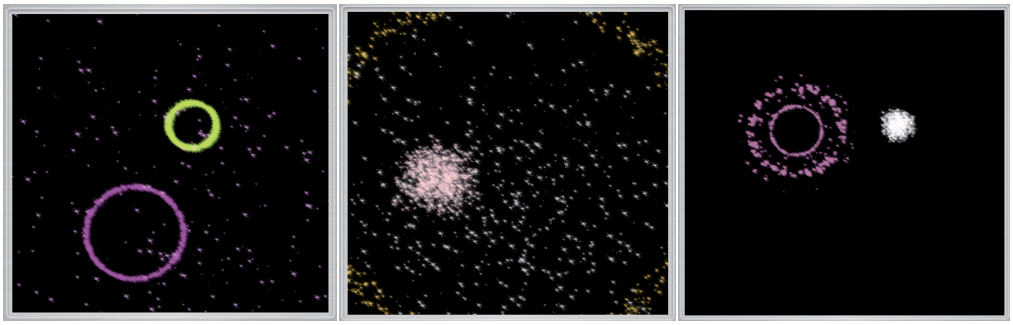

Abbildung 27: Noch ein paar bunte Boeller

9 Fazit

****

Das im Rahmen dieser Studienarbeit entwickelte Programm *Partikel, Partikel!* entspricht allen zu Beginn gesetzten Anforderungen: Es liefert ein Grundgerüst zur Implementierung verschiedener Simulations-Typen und demonstriert die Funktionsfähigkeit anhand der drei umgesetzten Typen *Schnee*, *Feuer* und *Feuerwerk*. Das Grundgerüst ist hierbei derart flexibel gehalten, dass sich das Programm im Nachhinein ohne Beeinträchtigung der bereits existierenden Typen um neue Typen erweitern lässt.

Darüber hinaus gibt es Punkte, die man verbessern könnte:
Die Bewegung der einzelnen Partikel beruht auf äußerst simplen Regeln - um sie realistischer bzw. komplexer erscheinen zu lassen, könnten Effekte wie Schwerkraft oder Wind mit einfliessen.

Auch das Aussehen der Partikel ist durchaus noch ausbaufähig: Die Feuerpartikel etwa könnten, wie in Kapitel *GPU Gems: Fire in the „Vulcan"Demo* demonstriert, eine animierte Textur bekommen. Über diesen Weg erreicht man mehr Dynamik in der Simulation, ohne unzählige Partikel mit komplexen Verhaltensregeln steuern zu müssen.

Zur Zeit kann man die 3D-Szene nur aus einem festen Blickwinkel betrachten - eine variable Kameraführung würde dem Ganzen mehr Abwechslung verleihen.

Zudem kann der Grundaufbau des Programmes überdacht werden: Das hier entwickelte Programm veranschaulicht die Simulation verschiedener konkreter Phänomene, die von einer Benutzeroberfläche aus mittels verschiedener Buttons in einem bestimmten Rahmen gesteuert werden können. Der Anwender kann also die Simulation nach Wahl des Types nur in einem kleinen, vorgegebenen Rahmen beeinflussen. Ein anderer Ansatz wäre das Bereitstellen verschiedenster Steuerelemente, ohne diese einem konkreten Typ zuzuordnen. Die finalen Simulationen ergeben sich letztlich durch äusserst geschicktes Austarieren der einzelnen Variablen durch den versierten Anwender.

Ein vernachlässigter Aspekt stellt die Performance dar: Der Quellcode wurde nur nach dem Kriterium der Lauffähigkeit erstellt und nicht optimiert. Zudem finden alle Bewegungen und Emissionen *pro Frame* statt - auf einem anderen Rechner mit anderer Leistung würden die Simulationen nicht so laufen, wie geplant. Vor einer Portierung müsste das Programm daher auf *pro Zeiteinheit* umgestellt werden.

10 Literatur

****

[GPUGems] Randima Fernando:
GPUGems.
NVIDIA Corporation, Online-Ausgabe.
Stand: 15.07.2008.
URL: http://developer.nvidia.com/ object/gpu_gems_home.html.
Abgerufen am 27. August 2008.

[taygeta] Taygeta Scientific Inc.:
URL: http://www.taygeta.com/random/gaussian.html
Monterey, 2001.
Abgerufen am 12. Oktober 2008.

[McAllister] David K. McAllister:
Particle System API. Version 2.0.
Stand: März 2007.
URL: http://www.particlesystems.org/. Abgerufen am 27.August 2008.

[Pesek] Yvo Pesek:
Simulation von Feuer mit Hilfe eines Partikelsystems.
Studienarbeit Universität Koblenz, Januar 2006.

[SEWK] Irma Sejdic, Dominik Erdmann, Daniel Wilhelm, Christoph Kurz:
Partikelsysteme und Schwärme.
Seminar Programmierung von Grafikkarten Universität Kassel, August 2006.

[Maxon] MAXON (Hrsg.):
Cinema 4D R9.52 - Referenz-Handbuch.
MAXON Computer GmbH, Friedrichsdorf 2005.

[vanDerBurg] John van der Burg:
Building an Advanced Particle System.
Stand: Juni 2000.
URL: http://www.gamasutra.com/features/20000623/vanderburg_01.htm.
Aufgerufen am 27. August 2008.

[Pätzold] Philipp Pätzold:
Entwicklung eines Partikelsystems auf Basis moderner 3D-Grafikhardware.
Studienarbeit Universität Koblenz, September 2005.

[Brockhaus] *Der Brockhaus. Computer und Informationstechnologie. Fachlexikon für Hardware, Software, Multimedia, Internet, Telekommunikation.*
Bibliographisches Institut & F.A. Brockhaus AG, Mannheim und Leipzig, 2003.

[Maier] Sarah Maier:
Darstellung von Spezialeffekten durch Partikelsimulaion.
Studienarbeit Universität Koblenz, 2003.

[Hawkins] Kevin Hawkins, Dave Astle, Andre LaMothe:
OpenGL Game Programming.
Prima Tech, ??? 2002.

[Parent] Rick Parent:
Computer Animation. Algorithms and Techniques.
Morgan Kaufmann Publishers, San Francisco 2002.

[Hein] Johannes Hein:
Partikelsysteme.
Proseminar Computergrafik Universität Ulm, 2002.

[vanDerBurg] John van der Burg:
Building an Advanced Particle System.
Stand: Juni 2000.
URL: http://www.gamasutra.com/features/20000623/vanderburg_01.htm.
Aufgerufen am 27. August 2008.

[ECP] David S. Ebert, Wayne E. Carlson, Richard E. Parent:
Solid spaces and inverse particle systems for controlling the animaition of gases and fluids.
In: *The Visual Computer*, 10(4):179-190, 1994.

[Sims] Karl Sims:
Particle Animation and Rendering Using Data Parallel computation.
In: *ACM SIGGRAPH Computer Graphics*, 24(4):405-413, August 1990.

[Reynolds] Craig W. Reynolds:
Flocks, Herds and Schools: A Distributed Behavioral Model.
In: *ACM SIGGRAPH Computer Graphics*, 21(4):25-34, Juli 1987.

[ReevesBlau] William T. Reeves, Ricki Blau:
Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems.
In: *ACM Transactions on Graphics*, 19(3):313-322, Juli 1985.

[Reeves] William T. Reeves:
Particle Systems - A Technique for Modeling a Class of Fuzzy Objects.
In: *ACM Transactions on Graphics*, 2(2):91-108, April 1983.

[Norton] Alan Norton:
Generation and display of geometric fractals in 3-D.
In: *ACM SIGGRAPH Computer Graphics*, 16(3):61-67, Juli 1982.

[PBS] PBS Public Broadcasting System:
Carl Sagans Cosmos Series. (Fernseh-Serie), 1980.

[CHPCH] C. Csuri, R. Hackathorn, R. Parent, W. Carlson, M. Howard:
Towards an interactive high visual complexity animation system.
In: *ACM SIGGRAPH Computer Graphics*, 13(2):289-299, August 1979.

[Bertelsmann] *Das Bertelsmann Lexikon in zehn Bänden.*
Bertelsmann Lexikon-Verlag, Gütersloh 1975.