GPU-Based Particle Engine

DA08_0405

DIPLOMANDEN
Tobias Egartner, Danijel Milenkovic

BETREUENDER DOZENT
Marcus Hudritsch
FHBB Muttenz

EXPERTE
Lutz Latta
EA Games Los Angeles

AUFTRAGGEBER
Wolfgang Engel
Wings Simulation Hattingen

Fachhochschule beider Basel Nordwestschweiz FHBB | Departement Industrie | Abteilung Informatik

Muttenz, Januar 2004

Vorwort

Egal, was Sie tun, tun Sie es mit Leidenschaft - diese Aussage von unserem Dozenten Marcus Hudritsch hat uns durch die ganze Diplomarbeit begleitet und geprägt.
Wir möchten uns bei Herrn Hudritsch bedanken, dass er uns immer wieder motiviert hat, und sich für zeitgemässe Projekt- und Diplomarbeiten einsetzte.

Vor allem möchten wir uns auch bei Herrn Wolfgang Engel bedanken, der uns eine anspruchsvolle Mission gab, und uns immer mit Rat und Tat zur Seite stand.

Danken wollen wir auch Herrn Lutz Latta, dafür dass er die Stelle als Experte angenommen hat, trotz seiner anspruchsvollen und zeitraubenden Arbeit bei EA GAMES.

Zudem möchten wir uns bei allen Mitstudierenden bedanken, welche Interesse an unserer Arbeit gezeigt haben, und uns somit motiviert und unterstützt haben.

Abstract

Das Programmieren *von Graphics Processing Unit (GPU)* wird immer populärer und gehört zu den neusten Entwicklungen der Computer Grafik. Um Programme auf der *GPU* laufen zu lassen, braucht es die *Shader*.

Die Herausforderung der vorliegenden Arbeit besteht darin ein Partikel System zu erstellen, das mit möglichst viel Unterstützung der *GPU* läuft. Auf diese Weise können Ressourcen gespart und ein enormer Performance-Gewinn erzielt werden.

Die Diplomarbeit, *GPU-Based Particle Engine,* wurde von Wolfgang Engel, *Special Effects Programmer* bei *Wings Simulation GmbH* und Autor von mehreren Büchern über Shader Programming, initiiert.
Marcus Hudritsch, Dozent an der *Fachhochschule beider Basel,* der uns bereits im letzten Studienjahr bei der Projektarbeit zur Seite stand, betreute diese Diplomarbeit.

Als Grundlage diente die Publikation *Building a Million Particle System* von Lutz Latta, welche in der ersten Phase intensiv studiert wurde. Gleichzeitig musste ein Framework erstellt werden, das auf dem *DirectX 9.0 SDK* basierte.

Die Umsetzung der Diplomarbeit umfasst zwei wichtige Etappen: das Programmieren eines *Daten Streams*, der mittels *Vertex Texturen* realisiert wurde und das Entwickeln der Physikberechnungen, die auf die Partikel wirken.

Bei der fertig erstellten Demo konnten mehrere Effekte erzielt werden. Zudem lässt sich das Verhalten der Partikel in Echtzeit manipulieren. Weiter konnte durch einen *Benchmark-Test* der Performance-Gewinn aufgezeigt werden.

Im Rahmen einer Weiterführung dieser Diplomarbeit könnte die Umsetzung der *Partikel Engine* in der digitalen Industrie vom grossen Nutzen sein. Dies aus dem Grund, weil man einerseits die gewonnenen Ressourcen anderweitig verwenden kann und anderseits eine bessere Visualisierung von Grafikeffekten erzielt werden kann.

Inhaltsverzeichnis

1. Einleitung 6
1.1. Aufgabenstellung 6
1.2. Motivation 7
1.3. Theoretische Orientierung 8
2. Hintergrund 9
2.1. Was ist ein Partikelsystem 9
2.1.1. Point Sprites 12
2.1.2. Emitter 13
2.1.3. Parameter 15
2.2. Stream Architektur 17
2.2.1. Texturen erstellen 17
2.2.2. Unterschied zwischen InStream und OutStream 18
2.2.3. Senden der Texturen an den Shader 18
2.2.4. Verwenden der Texturen im Shader 18
2.2.5. Ablauf eines Shader Programmes 19
2.3. Physik 21
2.3.1. Position und Geschwindigkeit 21
2.3.2. Berechnung 22
3. Implementation 24
3.1. Shader 24
3.2. Partikelstruktur 25
3.3. Emitter 26
3.4. Stream Architektur 29
3.4.1. Erstellen von Texturen (Stream) 29
3.4.2. Texturen an den Shader senden 30
3.4.3. Texturen im Shader verwenden 30
3.4.4. Parameterablauf im Shader 32
3.5. Physikberechnung 34
3.5.1. Lineare Positionsberechnung 34
3.5.2. Modifizieren der Geschwindigkeit 35
3.5.3. Einfache Kollision 36

4. Resultate 37
4.1. Benchmark 38
4.2. Fazit 40
4.3. Aussichten 41
5. Tutorial 43
5.1. Konstruktor 43
5.2. Regen / Schnee / Hagel 44
5.3. Explosion 45
5.4. Zusätzliche Informationen über die in DirectX integrierte GUI 46
6. Anhang 47
6.1. GPU-Programming 47
6.1.1. Technische Definition einer GPU 48
6.1.2. Ergänzungen zur technischen Definition 48
6.2. Definitionen 49
7. Quellenverzeichnis 51
7.1. Literatur 51
7.2. Links 51
7.3. Abbildungen 51

1. Einleitung

„….Ihr habt eine Mission!"

Da heutzutage Computer- und Videospiele immer realitätsgetreuer werden, steigen auch die Ansprüche an die Programmierer bei der Entwicklung von besseren Grafiken. Grosse Herausforderung bildet die Darstellung von volumetrischen Objekten. Hier greifen viele Entwickler zu einer *Partikel Engine*. Dieses System ermöglicht verschiedene Objekte realitätsgetreu darzustellen. Die Schwäche dieses Systems bildet jedoch der starke Ressourcenverbrauch im Prozessor.
Um Ressourcen einzusparen, soll die *Partikel Engine* auf eine Grafikkarte ausgelagert werden. So benötigt zwar die Grafikkarte Ressourcen, aber immer noch weniger als der Prozessor. Auf diese Art steigt die Performance und es ergeben sich neue Möglichkeiten für den Entwickler.

1.1. Aufgabenstellung

Ziel dieser Diplomarbeit ist eine generische, auf *Graphics Processing Unit (GPU)* basierende*, Partikel Engine* zu entwickeln. Mit einem Knopfdruck sollen verschiedene Effekte dargestellt werden wie z.B. Regen, Schnee oder Detonation.

Da in der Game Industrie hauptsächlich DirectX verwendet wird, ist es der Wunsch des Auftraggebers, die *Partikel Engine* mit *High Level Shading Language (HLSL)* zu programmieren. Um dieses Ziel zu erreichen wird *Pixel-* und *Vertex Shader Version 3.0* verwendet. Als Framework wird das *SDK* von *DirectX* verwendet.

1.2. Motivation

Die Entwicklung von *Pixel-* und *Vertexshader* immer populärer wird, erweitern sich auch die Anwendungsgebiete. Die Motivation ist, ein effizienteres Partikelsystem unter Zuhilfenahme einer *GPU* zu implementieren.

Ein nächstes Ziel bildet die Verwendung von einer Grafikkarte, die *Vertex-Texturen* unterstützt. Diese braucht die Engine für den *Daten Stream*, um mehr Daten für die Verarbeitung der Partikel zu berechnen.

Bis jetzt werden die Partikelsysteme in Spielen immer noch auf der *CPU* berechnet. Da laut Theorie die *GPU* bis zu 30-mal schneller, als eine moderne *CPU* sein soll, kann die gewonnene Prozessorzeit anderweitig eingesetzt werden, z.B. für die Berechnung der Spielphysik oder Darstellung mehrerer Partikel. In dieser *Partikel Engine* können ohne weiteres bis zu 262’144 Partikel bei einer interaktiven *Framerate* von 30 fps dargestellt werden.

Konfiguration der Diplomarbeit-Rechner:

HP Vectra:

- Intel P4 2.0 GHz
- 512 MB Rambus
- Windows XP Prof.
- DirectX 9.0c
- 6800GT 256 MB AGP 8x

HP Compaq dx6100mt:

- Intel P4 3.0 GHz
- 1 GB Dual Channel DDR
- Windows XP Prof.
- DirectX 9.0c
- 6600 256 MB PCI-E 16x

1.3. Theoretische Orientierung

Mit der Entwicklung eines Partikelsystems auf *GPU* hat Lutz Latta [6] bereits eine Vorarbeit geleistet. Auf der letzten *GDC* hat Latta die Vorstellung seines Partikelsystems, *Building a Million Particle System,* vorgetragen. Das interessante an dieser Arbeit ist der *Stream* der Daten zwischen der *CPU* und *GPU*. Dieser *Daten Steam* wird mittels einer *Vertex-Textur* simuliert.

Ein weiteres *GPU* Beispiel ist das von *UberFlow* [8]. Die Entwickler, Peter Kipfer, Mark Segal und Rüdiger Westermann verwendeten *GLSL* für ihre *GPU* basierende *Partikel Engine*. Durch die Verwendung von *OpenGL 2.0* konnte das Team für den *Daten Stream* auf den *Super Buffer* zurückgreifen, was unter *DirectX9* nicht möglich ist.

Nebst den *GPU* Versionen gibt es unzählige *CPU* Versionen in *DirectX*. Für die vorliegende Diplomarbeit soll als Grundlage das Partikelsystem von Kevin Harris [11] benutzt werden. Kevin Harris hat in seiner Demo über das Partikelsystem die Komplexität reduziert, dabei griff er auf das Beispiel von *DirectX SDK* zurück. Auf diese Art wurde der Ablauf des Partikelsystems ersichtlicht.

Weiter greift die Diplomarbeit auf die Unterlagen von Daniel S.C. Dalmau [3], die in seinem Buch *Core Techniques and Algorithms, in Game Programming* zu finden sind. Dalmau beschrieb den Ablauf eines Partikelsystems, wobei er anschliessend seine Variante - auf einer *GPU* - basierend darstellte. Nebst dem Ablauf, kamen noch theoretische Grundlagen dazu.

2. Hintergrund

2.1. Was ist ein Partikelsystem

Bevor auf die Definition des Begriffes Partikelsystem eingegangen wird, muss erläutert werden, was ein Partikel ist:

Partikel [10], aus dem Lateinischen *pars* (Teil) ist ein bzw. sind die einfachsten Formen von Teilchen eines Stoffes. Der Ausdruck *Partikel* wird verwendet in der Naturwissenschaft von Physikern, Chemikern, Biologen und Kriminologen.

Mehrere Partikel, die abhängig von Parametern sind, werden zu einem Partikelsystem.

Beim Partikelsystem unterscheidet man zwischen einem *lokalen* und einem *globalen* Partikelsystem. Der Unterschied bei den beiden Systemen besteht darin, dass bei einem lokalen System keine Kollisionen oder andere Interaktionen der einzelnen Partikel berechnet werden. Das ganze System wirkt gleich. Im Gegensatz dazu werden bei einem globalen System die Kollisionen und Interaktionen berechnet.
Die Umsetzung eines globalen Partikelsystems bringt Probleme mit sich darum wird in den Spielen immer noch ein lokales System verwendet.

Die Partikelsysteme stellen Grafikeffekte dar. Bei den Effekten, die man mit Partikelsystemen simuliert, handelt es sich üblicherweise um Objekte, die nicht oder nur sehr schwer durch Polygonmodellen darstellbar sind. Typische Beispiele sind:

- Feuer
- Wasser
- Rauch
- Wolken
- Nebel
- Explosionen
- Wetter (Schnee und Regen)

Bei diesen Objekten ist primär nicht die Oberfläche interessant, sondern das Volumen.
Um die Partikel darzustellen, gibt es verschiedene Möglichkeiten. Ob man nun Punkte oder Linien verwendet, hängt vom Effekt ab. Ein Partikel lässt sich auch mittels *Billboards* [10] darstellen. Ein *Billboard* ist ein texturiertes *Polygon*, dessen Normalvektor immer auf den Betrachter zeigt. Egal, von wo dieser auf das *Billboard* schaut, ob von oben, unten, vorne, links oder rechts, das *Billboard* zeigt immer auf den Betrachter.
Es gibt unterschiedliche Varianten von *Billboards*, die auch auf verschiedene Weise erzeugt werden können. Es wird unterschieden zwischen *sphärischen* und *zylindrischen Billboards*. *Sphärische Billboards* sind Rechtecke, meistens Quadrate. Nebst quadratischen Objekten können zusätzlich Objekte in Form von einer Kugel (Planeten, Bälle oder Billardkugeln) kreiert werden.
Im Gegensatz dazu stellt man bei *zylindrischen Billboards* Objekte dar, die eine zylindrische Form besitzen, wie z.B. einen Baumstamm oder einen Lichtstrahl.

Da in dieser Diplomarbeit Effekte wie Regen, Explosionen und Schnee dargestellt werden, verwendet man die *sphärischen Billboards*.

Die Umsetzung von *sphärischen Billboards* wird mit folgenden Methoden erzeugt:

- *Matrixmanipulation*
  Die Matrixmanipulation ist die einfachste Form für die Berechnung der *Billboads.* Pro *Billboard* werden die einzelnen Matrizen so berechnetet, dass die Positionen des *Billboards* auf 0/0/0 stehen. Bei mehreren Objekten ist diese Methode aber ungeeignet.

- *Softwaretransformation*
  Diese Methode ist wesentlich schneller als die *Matrixmanipulation.* Dennoch ist dieses Verfahren nicht das Schnellste. Bei dieser Methode wird die Arbeit von *OpenGL* übernommen. Das Setzen der *Billboardpositionen* in die Matrizen wird manuell von der Software übernommen.

- *Right- and Up- Vektor* [2]
  Damit die Berechnungen beschleunigt werden können, *werden Right- and Up Vektoren* verwendet. Die beiden Vektoren werden gebraucht um ein Rechteck aufzuspannen. So werden die Positionen der *Billboards* gesetzt.

- *Point Sprites*
  Diese Variante der Berechnung von *Billboards* wurde von *DirectX* eingeführt. Die Berechnung findet auf der Hardware statt. Darum sind die *Point Sprites* sehr gut für die Anwendungen von Partikel Systemen.

2.1.1. Point Sprites

Für die vorliegende Diplomarbeit wird die Darstellung mittels *Point Sprites* gewählt. Der Grund dafür ist die einfache Handhabung und ein Performance-Gewinn, das damit erzielt werden kann.

Point Sprites wurden erstmals in *DirectX 8.0* eingesetzt und bringen vor allem bei der Darstellung von Partikelsystemen einen Vorteil mit sich.
Im Gegensatz zu einem *Quad*, wo die 4 Eckpunkte gespeichert werden müssen, speichert ein *Point Sprite* nur einen einzigen Punkt, nämlich das Zentrum.
Zudem müssen die Texturkoordinaten nicht angeben werden, diese werden automatisch von der Hardware berechnet und erzeugt. Auf diese Weise kann ein Performance-Gewinn erzielt werden.

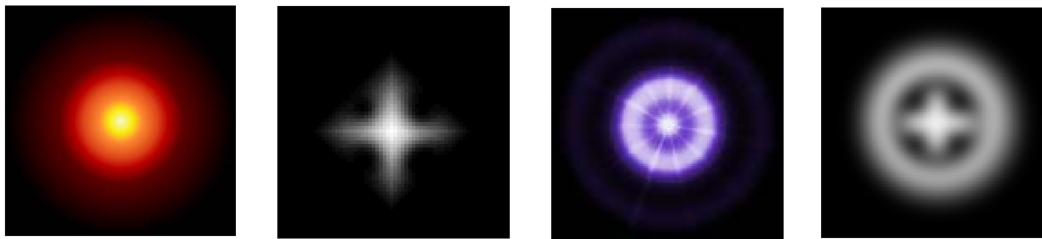

Abb. 1. Mögliche Textur für ein Point Sprite. Diese werden in der Diplomarbeit verwendet.

In *DirectX* kann man nebst der Textur, auch die Grösse der Punkte mittels *Point Sprites* festlegen.
Eine Problematik von *Point Sprites* bildet die Rotation, da nur das Zentrum der Hardware übergeben wird.

2.1.2. Emitter

Bevor sich die Partikel bewegen können, müssen sie geboren werden. Dies geschieht in einem so genannten Emitter. Ein Emitter ist ein Sender, der u.a. die Startpositionen der Partikel bestimmt.
Die Partikel fangen an zu einer bestimmten Zeit zu leben. Diese Einstellung kann man über *Time of Birth (TOB)* steuern. So kann man eine Unregelmässigkeit im System entstehen lassen damit nicht alle Partikel auf einmal loslegen. Je nach Effekt, den man erzielen möchte, braucht es einen anderen Emitter.

In der Literatur[3] unterscheidet man zwischen verschieden Formen des Emitters. In der Diplomarbeit wurden drei Arten von Ermitteln gewählt.

Der Punktemitter

Alle Partikel haben dieselbe Startposition, z.B. x=0, y=0 und z=0.

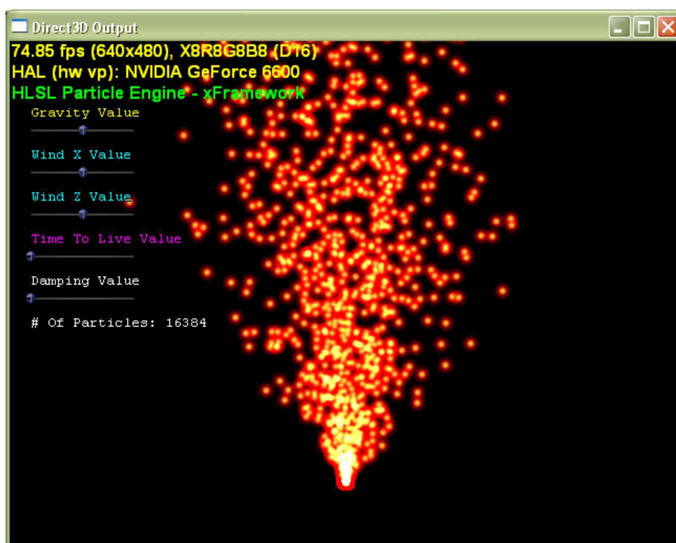

Abb. 2. Punktemitter

Der Flächenemitter

Die Partikel haben auf einer quadratischen Fläche unterschiedliche Startpositionen, welche auf der x- und z-Achse verteilt sind.

Abb. 3. Flächenemitter

Der Kreisemitter

Im Gegensatz zu einer quadratischen Fläche wie beim Flächenemitter bildet beim Kreisemitter die Fläche einen Kreis. Die Grösse des Kreises lässt sich mittels eines Radius' bestimmen.

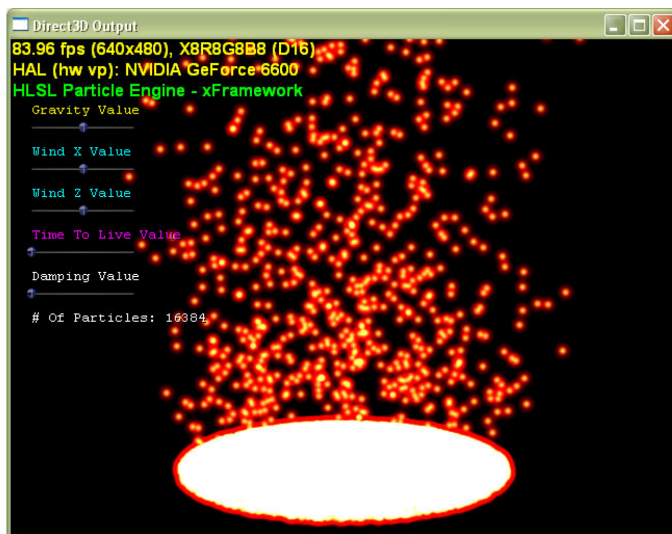

Abb. 4. Kreisemitter

2.1.3. Parameter

Um ein Partikelsystem zu beeinflussen, braucht es neben den oben erwähnten Emitter und *Time of Birth* (TOB) folgende Parameter:

- Ausstossgeschwindigkeit
- Gravität (Kräfteeinfluss)
- Lebensdauer (Time to Life, TTL)
- Dämpfung (Verlangsamung der Partikel)
- Anzahl der Partikel im Gesamtsystem
- Kräfteeinflüsse auf die Partikel

Die Kombinationen dieser Parameter können unterschiedlichste Effekte erzeugen. Zuerst soll auf die Definitionen dieser eingegangen werden. Das Kapitel *Tutorial* setzt sich mit der Anwendung auseinander.

Ausstossgeschwindigkeit
Dieser Parameter bestimmt, wie schnell sich ein Partikel vom Emitter entfernen soll. Ist die Zahl, die die Geschwindigkeit bestimmt relativ hoch, so kann sich das Partikel schneller vom Emitter wegbewegen.

Gravität
Die Gravität ist eine der Hauptkräfte, die auf die Partikel wirkt. Sie bestimmt die Fallgeschwindigkeit der einzelnen Partikel. Die Berechnung für die Gravität ist:

$v = \frac{1}{2} \cdot g \cdot time^2$ ($g = 9.81\ m/sec$)   time = Zeit zwischen 2 Frames

In der vorliegenden *Partikel Engine* kann man die Gravität mittels Parameter beeinflussen. Je höher die Gravität umso stärker wirkt sie auf die Partikel.
In Kombination mit der Ausstossgeschwindigkeit kann der *schiefe Wurf* beeinflusst werden.

Lebensdauer

Die so genannte *Time of Life* bestimmt das Leben eines Partikels. Nach dem Ablaufen der Lebensdauer hat man zwei Möglichkeiten:

- Zurück zum Ursprung und das Ganze nochmal.
- Es geschieht nichts mehr, das Partikel ist tot.

Hier kann man wieder eine Zufälligkeit ins System bringen, damit die Partikel unterschiedlich lang leben.

Dämpfung

Wenn die Partikel mit einer Ebene oder einem Objekt kollidieren, sollen sie sich verlangsamen bzw. dämpfen.
Die Berechnung, die für die Dämpfung angewendet wird, lautet:

v = konst. * alt v   v = Geschwindigkeit

D.h. dass die Geschwindigkeit herunter skaliert wird. Das Ganze kann natürlich auch in umgekehrte Richtung funktionieren, in dem die Geschwindigkeit herauf skaliert wird.

Anzahl der Partikel im System

Je nach Effekt, den man anstreben möchte, braucht es eine unterschiedliche Anzahl an Partikel in der *Engine*. Die Anzahl kann man durch grössere bzw. kleinere *Vertex-Texturen* beeinflussen.
Da die vorliegende *Partikel Engine* quadratische Texturen annimmt, ist die Anzahl der Partikel vordefiniert. Das Spektrum bewegt sich zwischen
2, 4, 8, 16, 32, 64,...., 262144 Partikeln pro Textur.

Kräfteeinfluss auf die Partikel

In dieser Diplomarbeit wird nicht nur die Gravität als Kraft, die auf die Partikel wirkt, eingesetzt, sondern auch der Wind.

2.2. Stream Architektur

In diesem Kapitel werden die einzelnen Punkte für einen *Daten Stream*, der mittels *Vertex-Texturen* vollständig auf der *GPU* läuft, erklärt. Ausser der Initialisierung der Daten, welche auf der *CPU* stattfindet.

2.2.1. Texturen erstellen

Die erstellten Texturen sind virtuell; sie werden auf ein *Quad (Stream Generator)* gerendert.

Vorgehensweise für die Erstellung eines *Stream Generator*:

1. Erstellen eines Float Arrays (Size = Width * Height * Nr. Of Channels).
2. Float Array mit Daten füllen (Position, TTL, TOB... usw).
   Bei drei Channels (rgb) werden die Positionen r=x, g=y, b=z gesetzt.
   Bei TTL und TOB werden nur zwei Channels einer Textur belegt.
3. Der Inhalt des Float Arrays wird auf ein Quad gerendert.

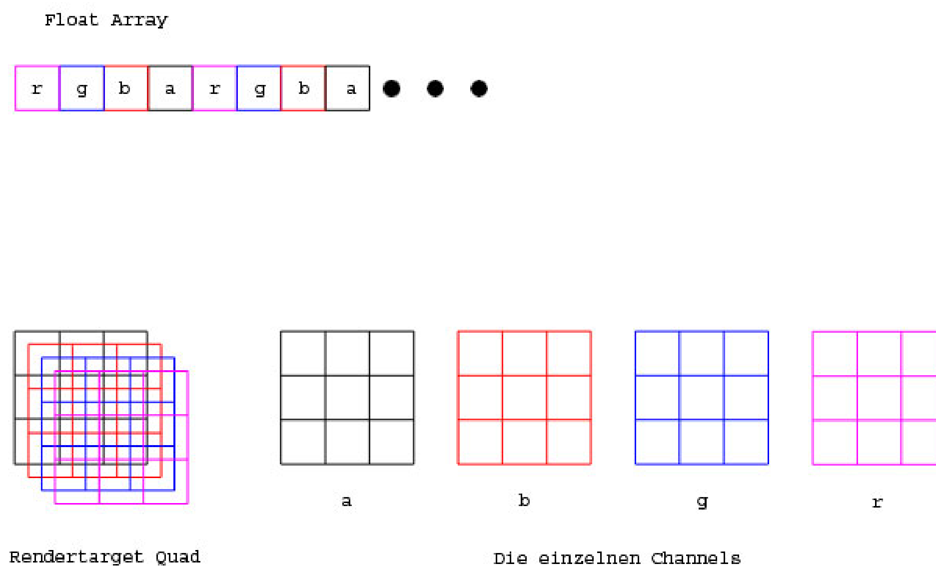

Abb. 5. Vertex Texturen

2.2.2. Unterschied zwischen InStream und OutStream

InStream und *OutStream* sind Vertex-Texturen.
InStream beschreibt den Zustand des *Stream Generators* bzw. den Inhalt des *Quads* zum Zeitpunkt 0 (Übergabe vom *Startpointer* des *Quads* an den *Shader*).
OutStream beschreibt den Inhalt bzw. Zustand des *Quads* nachdem die Berechnungen im *Shader* stattgefunden haben (*ShaderOutput*). So werden die Texturen zirkulär verwendet.

2.2.3. Senden der Texturen an den Shader

Der Zeiger*,* der auf den Anfang des *Quads (Stream Generator)* zeigt, wird an den *Shader* gesendet, damit auf den Inhalt des *Quads* (Float Werte) zugegriffen werden kann.

2.2.4. Verwenden der Texturen im Shader

Da die Position jedes Partikels (Ein Partikel ist ein *Point Sprite*, ein *Point Sprite* benötigt ein *Vertex*) für sich bearbeitet werden soll, muss jedes Partikel wissen, wo seine Parameter abgelegt sind. Diese befinden sich natürlich auf dem *Quad*. Der Zugriff findet mittels Texturkoordinate (U,V) statt.

Ein *Vertex* hat folgende Parameter:

- Position
- Farbe
- Texturkoordinaten
- Für das Rendern relevante Einstellungen

Der Zugriff auf einen Wert eines Pixels (*RGBA*) findet im *Shader* wie folgt statt: Texturkoordinaten von jedem *Vertex* werden in einer separaten Textur gespeichert. Zuerst werden die Texturkoordinaten des zu bearbeitenden *Vertex* ausgelesen. Danach werden die Positionen von diesem *Vertex* bearbeitet dessen Position an der Stelle *U,V,* gespeichert ist.

2.2.5. Ablauf eines Shader Programmes

Nun soll auf die Steuerung des Partikelsystems eingegangen werden. Die Steuerung findet über die Parameter (TTL, TOB) statt.

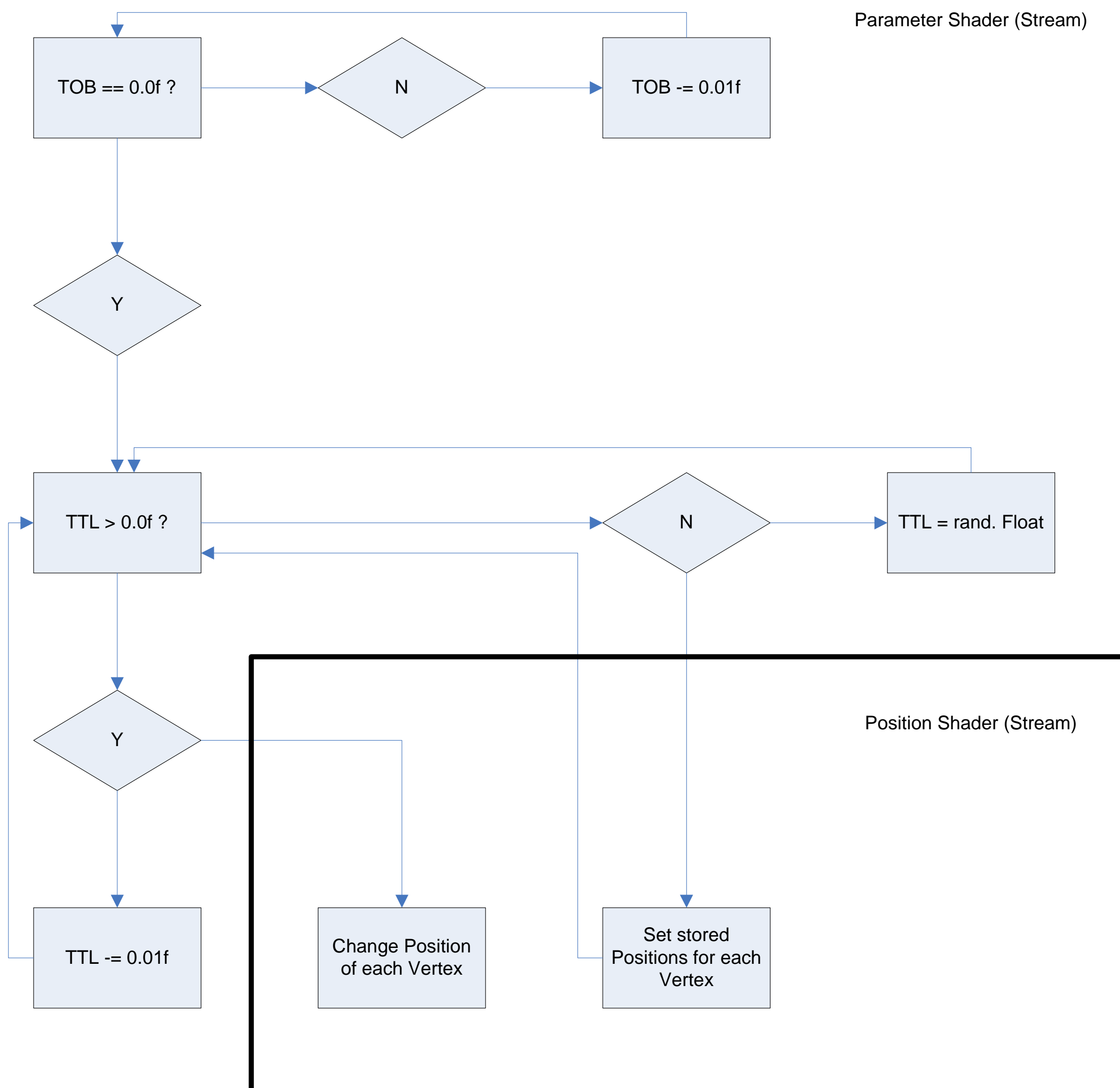

Abb. 6. Vertex Texturen

Der folgende Ablauf beschreibt die Abbildung 7:

1. Überprüfe für jedes Vertex (Partikel) ist TOB == 0.0f? Ist das Partikel auf die Welt gekommen?

   NEIN: 0.01f von TOB abziehen, neues TOB schreiben.

   JA: TOB == 0.0f gehe zu Schritt 2.

2. Ist TTL > 0.0f? Lebt das Partikel?

   NEIN: Das Partikel lebt nicht mehr! Setzte einen neuen Zufallswert für TTL, setzte wieder die Ursprungsposition. Gehe zurück zu Schritt 2.

   JA: Ziehe 0.01f von TTL ab (Partikel sind nicht unsterblich, daher soll ihre Lebensdauer verringert werden), neues TTL speichern. Gleichzeitig findet die Berechnung der neuen Position unter Berücksichtigung der Physik statt. Gehe zurück zu Schritt 2.

2.3. Physik

In jeder modernen 3D-Enigne ist eine Physik vorhanden. Diese kümmert sich um die Objekte, die einen Einfluss auf die Umgebung haben sollen. Das aktuellste Beispiel für eine ausgezeichnete Physik ist *Half Life 2.*

2.3.1. Position und Geschwindigkeit

Für das Partikelsystem werden Berechnungen wie Gravität, Dämpfung, Wind in der Position und Geschwindigkeit vorgenommen. Ein weiterer Schritt sind nun die Berechnungen des Systems. Nebst einer Position muss auch die Geschwindigkeit berücksichtigt werden. Die Kräfte, die auf die Partikel wirken, sind Vektoren, wie z.B. der Wind. Diese Vektoren werden aufsummiert, um die neue Richtung des Partikels festzulegen.

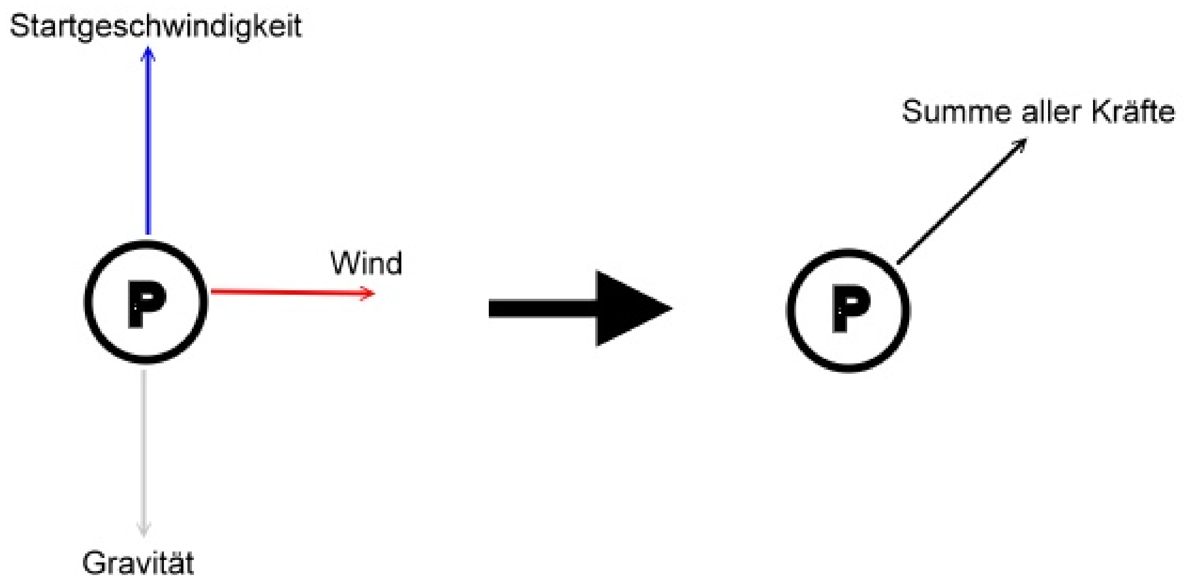

Abb. 7. Die Summe der Kräfte, die auf ein Partikel (P) wirken

2.3.2. Berechnung

Die allgemeine Gleichung für die Berechnung der Position der Partikel lautet:

Neue Position = Alte Position + Geschwindigkeit * Zeit

Doch die einzelnen Parameter, wie Gravität und Wind wirken nun auf das System ein.
Von dort aus folgt jedes Partikel seinem individuellen Abschussvektor mit seiner Geschwindigkeit.
Die effektive Richtungsänderung folgt durch die Geschwindigkeit. Diese wird im *Shader* berechnet. Und dort kommen auch all die Kräfte zum Zuge, die auf einen Partikel wirken.
Für jedes Frame wird die Geschwindigkeit neu berechnet und die Gravität, sowie der Wind wird dazu addiert.

Geschwindigkeit += Wind + Gravität * Zeit

So kann für jedes Partikel eine eigene Bewegung berechnet werden.

Im Partikelsystem der Diplomarbeit wird zusätzlich eine minimale Kollisionsabfrage dargestellt. Diese berechnet für jedes Partikel, wann es auf eine Ebene angekommen ist. Weiter soll das Partikel liegen bleiben oder in die entgegen gesetzter Richtung fliegen. So kann man z.B. Wassertropfen simulieren, die auf dem Boden abspringen.

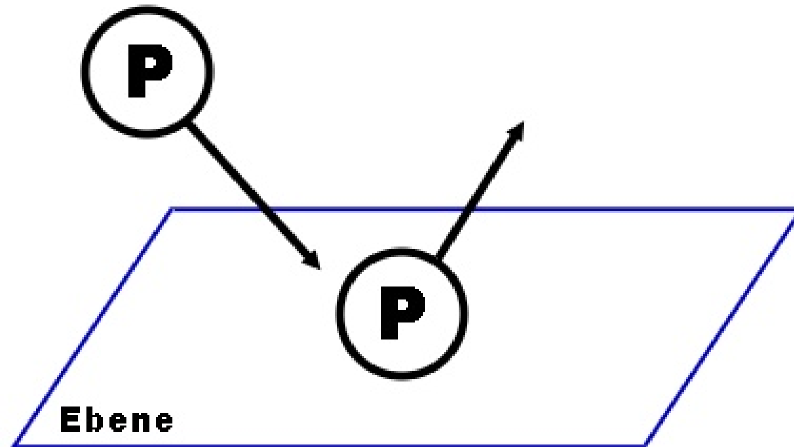

Abb. 8. Kollision

Wie weit die Partikel von der Ebene abspringen, hängt von einem Parameter ab, nämlich der Dämpfung. Diese wird in der Geschwindigkeit dazu multipliziert.
So kann man den Widerstand eines Objektes bestimmen bzw. simulieren.

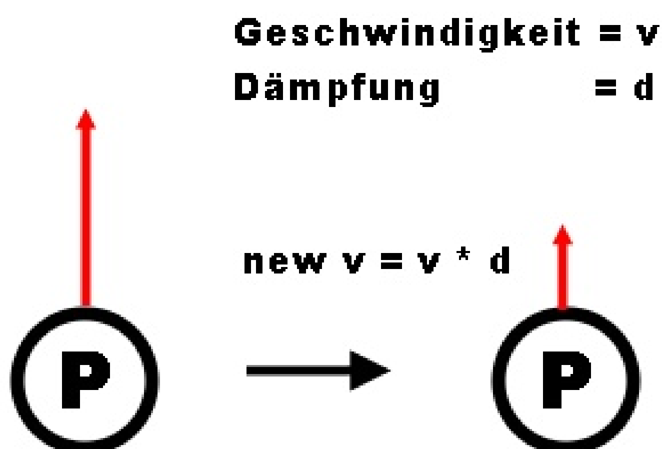

Abb. 9. Dämpfung

3. Implementation

Das Kapitel zeigt auf, wie die *Partikel Engine* die Theorie von Kapitel 2 umsetzt. Es wird auf Strukturen, Initialisierung und Berechnungen eingegangen. Code welcher auf der *CPU* ausgeführt wird ist **grau** hinterlegt, *GPU* Code ist **orange**.

3.1. Shader

Für die *Shader* Programmierung braucht es noch ein paar Einstellungen. Um einzelne Daten zwischen der *CPU* und *GPU* zu wechseln, werden *Uniforms* verwendet.

```
float 4x4   worldviewProj : worldviewProjection; // WVP Matrix
float ttl;                                      // Time to Life
float timedelta;                                // Delta Zeit
float time;                                     // absolute Zeit
float gforce;                                   // G-Kraft
float windx;                                    // Wind in X-Richtung
float windz;                                    // Wind in Z-Richtung
float damp;                                     // Faktor für Dämpfung
```

Damit die *Shader* die Daten verwenden können, müssen dem *Shader* Strukturen übergeben werden. Diese Daten werden dann zwischen den *Vertex*- und *Pixel Shader* kommuniziert.

```
struct VS_OUTPUT
{
      float4 position   : POSITION;       //Postionswerte
      float2 texCoord   : TEXCOORD0;      //Textur Koordinaten
};

struct PS_OUTPUT
{
      Float4 color       : COLOR;         //Farbwerte
}
```

Nach der Definierung der *Uniforms* und Strukturen, werden die *Shader* programmiert. Am Schluss müssen noch die *Shader* kompiliert werden. Dabei besteht die Möglichkeit, die Versionen der *Shader* anzugeben.

```
technique StreamPos
{
	pass Pass0
	{
		VertexShader = compile vs_3_0 MyVertexShader();
		//Version 3.0 wird kompiliert
		PixelShader = compile ps_3_0 MyPixelShader();
	}
}
```

3.2. Partikelstruktur

Ein Partikel entspricht einem *Vertex*, da dieser nur einen Punkt pro Partikel benötigt. (Zentrum der für das Partikel zu verwendenden Textur).

```
struct Vertex
{
	D3DXVECTOR3		posit;		// position
	D3DCOLOR		color;		// farbe
	float			u, v;		// texturkoordinaten
	…
};
```

3.3. Emitter

Bei dem Emitter müssen unterschiedliche Werte gesetzt werden, die an *Shader* gesendet und bearbeitet werden. Die Funktion *getRandomMinMax* ist dafür programmiert worden, dass unterschiedliche Werte für die Partikel erzeugt werden.

In *m_vCurPos* werden die Startpositionen für alle Partikel festgelegt.
Bei *m_StartVel* werden die Startgeschwindigkeiten in den einzelnen Achsen (x, y, z) gesetzt, so können unterschiedliche Geschwindigkeiten erzeugt werden.
Bei *m_vCurVel* werden die laufenden Geschwindigkeiten, die im Shader bearbeitet werden, gespeichert.

Punktemitter
Wichtig bei diesem Emitter ist *m_vCurPos.* Diese soll die gleichen Startwerte besitzen.

```
for(int i = 0; i < getPMax(); ++i)
{
      //Startposition wird gesetzt
      g_particles[i].m_vCurPos      = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
      g_particles[i].m_vStartVel    = D3DXVECTOR3(getRandomMinMax(-5.0f, 5.0f),
                                                  getRandomMinMax(5.0f, 30.0f),
                                                  getRandomMinMax(-5.0f, 5.0f));
      g_particles[i].m_vCurVel      = D3DXVECTOR3(getRandomMinMax(-5.0f, 5.0f),
                                                  getRandomMinMax(1.0f, 5.0f),
                                                  getRandomMinMax(1.0f, 5.0f));
}
```

Kreisflächenemitter

Bei diesem Emitter ist der Radius wichtig, der zwischen der x- und z-Achse entsteht.

```
for(int i = 0; i < getPMax(); ++i)    //für alle Partikel PMax()
{
      do
      { // Die Radiusgrösse wird gesetzt
        x = (getRandomMinMax(0.0f, 1.0f) - 0.5f) * 2 * radius;
        z = (getRandomMinMax(0.0f, 1.0f) - 0.5f) * 2 * radius;
      }
      while( z > sqrt(radius * radius - x * x) ||
             z < -sqrt(radius * radius - x * x)
      g_particles[i].m_vCurPos   = D3DXVECTOR3(x, 0.5f, z);
      g_particles[i].m_vStartVel = D3DXVECTOR3(getRandomMinMax(-5.0f, 5.0f),
                                               getRandomMinMax(5.0f, 30.0f),
                                               getRandomMinMax(-5.0f, 5.0f));
      g_particles[i].m_vCurVel   = D3DXVECTOR3(getRandomMinMax(-5.0f, 5.0f),
                                               getRandomMinMax(1.0f, 5.0f),
                                               getRandomMinMax(1.0f, 5.0f));
}
```

Flächenemitter

Die Startpositionen der Partikel variieren auf einer Ebene von –20 - +20, welche sich auf der x- und z-Achse befindet.

```
for(int i = 0; i < getPMax(); ++i)
{
      g_particles[i].m_vCurPos   = D3DXVECTOR3(getRandomMinMax(-20.0f, 20.0f),
                                               getRandomMinMax(40.0f, 40.01f),
                                               getRandomMinMax(-20.0f, 20.0f));

      g_particles[i].m_vStartVel = D3DXVECTOR3(getRandomMinMax(-5.0f, 5.0f),
                                              getRandomMinMax(1.0f, 2.0f),
                                              getRandomMinMax(-5.0f, 5.0f));

      g_particles[i].m_vCurVel   = D3DXVECTOR3(getRandomMinMax(-5.0f, 5.0f),
                                              getRandomMinMax(1.0f, 5.0f),
                                              getRandomMinMax(1.0f, 5.0f));
}
```

3.4. Stream Architektur

Um die Daten für die Partikel zu senden, braucht es die *Vertex-Texturen* [9].

3.4.1. Erstellen von Texturen (Stream)

Alle Texturen, welche für den *Stream* verwendet werden, werden nach dem gleichen Grundprinzip erstellt, hier ein Beispiel für die Initialisierung der Positionen:

```
for (unsigned long j=0; j < InPositions->getHeight(); ++j)
{
	for (unsigned long i=0; i < InPositions->getWidth(); ++i)
	{
		(*InStream)[4 * j * InStream.getWidth() + 4 * i + 0] =
		g_particles[j * InStream->getHeight() + i].m_vCurPos.x;

		(*InStream)[4 * j * InStream.getWidth() + 4 * i + 1] =
		g_particles[j * InStream->getHeight() + i].m_vCurPos.y;

		(*InStream)[4 * j * InStream.getWidth() + 4 * i + 2] =
		g_particles[j * InStream->getHeight() + i].m_vCurPos.z;

		(*InStream)[4 * j * InStream.getWidth() + 4 * i + 3] = 1.0f;
	}
}
InStream->commit_changes();
```

In diesem Beispiel werden die RGBA Kanäle mit x, y, z und w gefüllt.

InStream ist ein *texInputStream* Objekt. Nähere Angaben sind in folgender Datei zu finden:

xTexture.cpp.
commit_changes() führt ein memcpy() aus, die Floatwerte werden in ein D3DLOCKED_RECT gerendert.

3.4.2. Texturen an den Shader senden

Um die Texturen verwendet zu können, müssen diese an den *Shader* gesendet werden. Dieser Vorgang findet mit folgender Funktion statt:

```
sendTexture("Texture Name", Texture Pointer());
```

Kurzum, diese Funktion liefert den *Pointer*, welcher auf den Anfang der Textur zeigt, an den *Shader*.

3.4.3. Texturen im Shader verwenden

Im *Shader* selbst muss die Textur zuerst noch eingebunden werden. Dazu werden verschiedene Texturfilter gesetzt. Da keine Interpolation erwünscht ist (fehlerhafte Werte), werden die Filter auf *Pointsampling* gesetzt.

```
texture  InStream;
sampler  InStr = sampler_state
{
      Texture     = <InStream>;
      // Texturparameter
      MinFilter = POINT;
      MagFilter = POINT;
      MipFilter = POINT;

      AddressU  = Clamp;
      AddressV  = Clamp;
};
```

Um auf die Positionen der *Vertizes* zuzugreifen, benötigt es noch zwei Zeilen Code:

```
float2 addr = tex2D(Ind, float2(IN.texCoord);
float4 data = tex2D(InStr, addr);
```

Data enthält nun die einzelnen am Anfang gesetzten Werte (x, y, z und w). Die Positionen der einzelnen *Vertizes* können jetzt im *Pixel Shader* beliebig verändert werden. Für die Ausgabe der neuen Positionen ist der *Vertex Shader* zuständig. Dort werden für die Ausgabe (mit Berücksichtigung der *WorldViewProjection-Matrix*) die Positionen aus dem *InputShader* verwendet.
Damit dies funktionieren kann, wird zu einem späteren Zeitpunkt im *main.cpp* der *InputShader* zum *OutputShader*. Somit ist der gesamte Verlauf ist zirkulär.

```
float4 addr = tex2Dlod(Ind, float4(IN.texCoord,0,1);
flaot4 data = tex2Dlod(InStr, addr);
```

OUT.position = mul(worldViewProj, data.xyzw);

Der *Vertex Shader* gibt nun die vorher im *Pixel Shader* neu berechneten Positionen aus.

Der Zugriff auf eine Textur über den *Pixel Shader* findet mit der Funktion *tex2D()* statt. Im *Vertex Shader* hat diese Funktion einen anderen Namen, nämlich *tex2Dlod().* Darauf wurde in der Dokumentation von *DirectX* nicht hingewiesen.

3.4.4. Parameterablauf im Shader

Nun folgt ein Codebeispiel zum generellen Ablauf im *Shader*.

```
// 1.
float2 addr = tex2D(Ind, float2(IN.texCoord));
      float4 para = tex4D(InParam, addr);

      // 2.
      if( para.g > 0.0f)
      {
            OUT.color.g = para.g - 0.01f;
      }

      // 3.
      else if( para.r > 0.0f)
      {
            OUT.color.r = para.r - 0.001f;
            OUT.color.a = 0.0f;
      }

      // 4.
      else if( para.r == 0.0f)
      {
            OUT.color.r = ttl;
            OUT.color.b = 1;
      }
```

Nun soll auf die wesentlichen Aspekte im Programm-Code hingewiesen werden.

1. Dieser Code wird benötigt, um auf die Inhalte der diversen Texturen zuzugreifen.
2. Ist TOB grösser 0.0f wurde das Partikel noch nicht geboren, also wird TOB in kleinen Schritten dekrementiert, da die Partikel nicht unsterblich sind.
3. Ist TTL grösser 0.0f, so lebt das Partikel noch. Damit dieses jedoch nicht ewig lebt, wird die Lebensdauer ebenfalls dekrementiert.
4. Sollte dieser Fall eintreten ist das Partikel tot. Der Anwender kann jedoch beeinflussen, ob es wiedergeboren werden soll. Dies geschieht über das Zuweisen eines neuen TTL Wertes (OUT.color.r = ttl;).

3.5. Physikberechnung

Da nun die Texturen mit den Daten an den *Shader* gesendet wurden, kommt es zur Verwendung. Wie schon beim *Time to Life* und *Time of Birth* werden die Positions-Daten verrechnet.

3.5.1. Lineare Positionsberechnung

Als erstes soll eine lineare Bewegung der Partikel erzeugt werden. Dazu braucht es einen *Richtungsvektor*.

Dieser *Vektor* wird nun zur aktuellen Position dazu gerechnet und beide ergeben zusammen die neue Position. Natürlich braucht diese Berechnung noch die Differenz der Zeit zwischen zwei *Frames*, die im *timedelta* gespeichert werden.

Sollen nun die Partikel einer physikalischen Funktion folgen, kommt noch die modifizierte Geschwindigkeit dazu.

```
if (para.r > 0.0f && para.g == 0.0f
{
      dir.x = data.r + mvel.x * timedelta;
      dir.y = data.g + mvel.y * timedelta;
      dir.z = data.b + mvel.z * timedelta;
}
```

3.5.2. Modifizieren der Geschwindigkeit

Nun bewegen sich alle Partikel linear in die Richtung, die der Emitter vorgegeben hat.
Wenn dazu noch die modifizierte Geschwindigkeit kommt, lässt sich eine beliebige physikalische Funktion darstellen. Nebst der Gravität ist der Wind ein solcher Parameter, der in die modifizierte Geschwindigkeit dazu gerechnet wird.

```
float3 wind;

wind.x = windx * 0.01f; //Wind Stärke
wind.y = windy * 0.01f;
wind.z = windz * 0.01f;
```

Diese Parameter werden als *Uniforms* dem *Shader* übergeben.

```
mvel.x += wind.x;
mvel.y += gforce + wind.y * timedelta;
mvel.z += wind.z;
```

Dadurch, dass sich die *gforce* und *wind* in der *Partikel Eninge* verändern lässt, kann eine unterschiedliche Richtung der Partikel bestimmt werden.
Weiter müssen die Parameter der *Enigne* in der Geschwindigkeit einbezogen werden.

```
if (para.b = = 1.0f)
{
      mvel.x = vel.x;
      mvel.y = vel.y;
      mvel.z = vel.z;
}
```

Die Geschwindigkeit muss wieder zurück gesetzt werden, falls die Partikel tot sind und diese wieder von vorne beginnen zu leben.

3.5.3. Einfache Kollision

Eine einfache Kollision kann mittels einer Abfrage der Position der Partikel durchgeführt werden. Wenn der Boden eine Ebene (y = -20) ist, so überprüft die Engine, ob ein Partikel schon auf den Boden aufgeprallt ist.
In der vorliegenden Diplomarbeit werden die Partikel in die entgegengesetzte Richtung bewegt.

```
if (pos.y <= -20.0f)
{
	mvel.x = 0;
	mvel.y = abs (mvel.y) * (damping * 0.1);
	mvel.z = 0;
}
```

Im oberen Code Beispiel wird weiter die Dämpfung (*damping*) berücksichtigt.

4. Resultate

Mit der vorliegenden Diplomarbeit wurde eine Partikel Engine erstellt, die auf der GPU funktioniert. Das Ergebnis wurde in Form einer Demo dargestellt. Dabei ist es möglich in Echtzeit Veränderungen am Verhalten der Partikel vorzunehmen. Die Modifizierungen geschehen über eine *GUI*, welche mittels *DirectX* erstellt wurde.
Da die Schnittstelle zwischen der *CPU* und *GPU* relativ klein gehalten wurde, ist es möglich, mit wenigen Änderungen des *Konstruktors*, neue Effekte zu erzielen.

Folgende Effekte konnten mit der entwickelten *Partikel Engine* erzeugt werden:

- ein Vulkan, der Lava ausspuckt
- Ein Schneesturm, der unterschiedlich stark sein kann
- Regen, analog zum Schnee

Damit die Demo so entstehen konnte, spielen mehrere Implementationen eine Rolle. Angefangen beim *Framework*, dem *Data Stream* und der Anwendung der *Vertex-* und *Pixel Shader 3.0.*

Der Aufbau des x*Frameworks* setzt auf dem *Framework* von *DirectX 9.0 SDK* auf. Doch muss diese Plattform an die Bedürfnisse der vorliegenden *Partikel Engine* angepasst werden. Vor allem die Handhabung der *Vertex-Texturen* ist ein wichtiger Punkt.
Auch die Anwendung von *Vertex- und Pixel Shader 3.0* ist erfolgreich in der *Engine* implementiert worden. Der Beweis dafür ist der *Data Stream* der mittels *Vertex-Texturen* realisiert wurde.

4.1. Benchmark

Bei der Programmierung der *Partikel Engine* werden zwei verschiedene Grafikkarten verwendet. So ist das Erstellen eines *Benchmarks* sinnvoll, um die Unterschiede der *Framerate* (*fps*), bei variierenden Grössen der Texturen, zu erkennen.

	GeForce 6800GT 256 MB 4x AGP P4 2 GHz 512 MB Rambus	GeForce 6600 256 MB PCI-Express 16x P4 3 GHz 1 GB Dual-Channel DDR	
Textur	FPS	FPS	Anzahl Partikel
2x2	364	254	4
4x4	363	253	16
8x8	362	252	64
16x16	359	240	256
32x32	343	229	1024
64x64	289	175	4096
128x128	146	97	16384
256x256	65	35	65536
512x512	16	8	262144

Abb. 10. Die zwei verschiedenen Grafikkarten

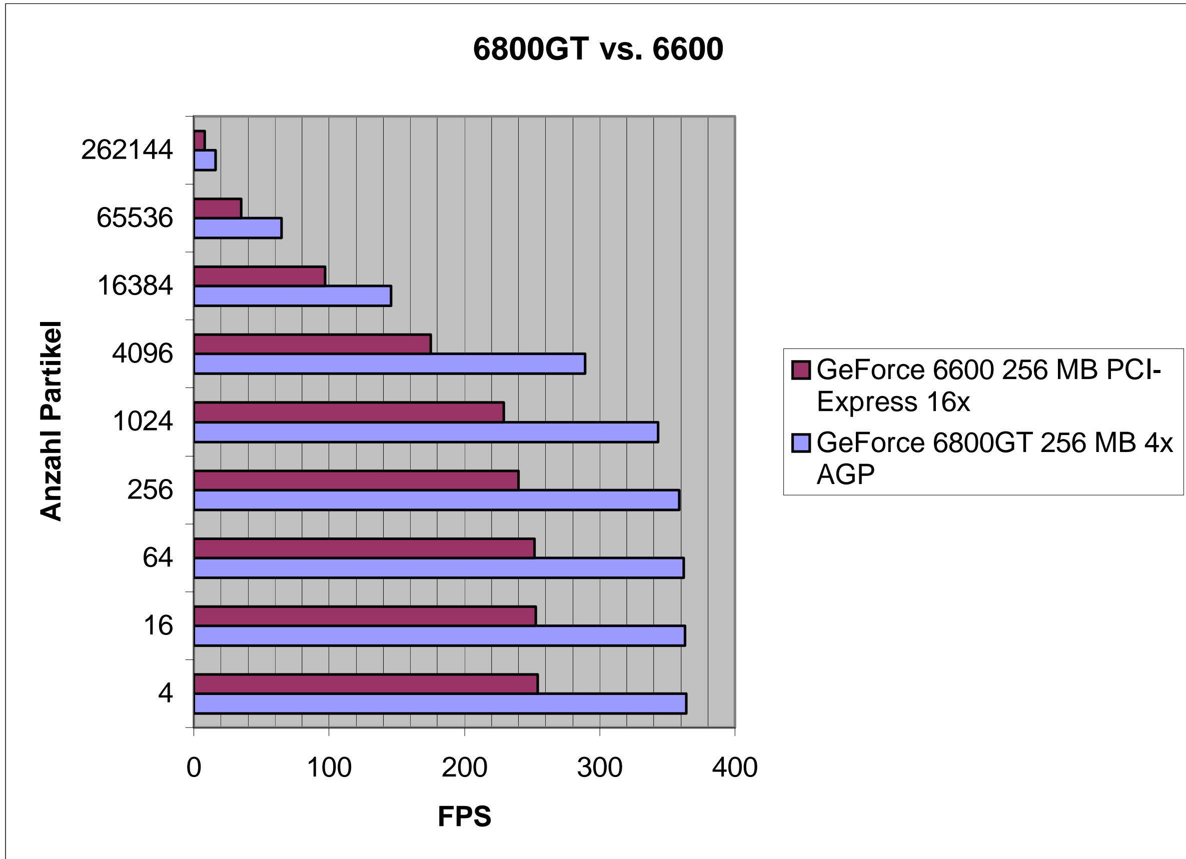

Abb. 11. Benchmark

Die Abbildung 11 zeigt die Unterschiede zwischen den beiden Grafikkarten auf, wobei der *BUS* keine grosse Rolle spielt, da moderne Grafikkarten noch nicht die grössere zur Verfügung gestellte Bandbreite nutzen können. Nichtsdestotrotz gibt es geringfügige Einbussen bei der Leistung der *6800-er* Grafikkarte, da diese nur im *AGP 4x* Modus betrieben wird.

4.2. Fazit

Mit der vorliegenden Diplomarbeit wurde ein Versuch unternommen neuste Technologien mit bekannten 3D Darstellung zu verbinden. Trotz anfänglichen Problemen mit der neuen Version von *Vertex- und Pixel Shader 3.0* verlief die Diplomarbeit relativ zügig.
Bis jetzt gibt es nur wenige Arbeiten, die sich mit einer *Partikel Engine* auf der GPU und in *HLSL* befassen. Bei Problemen konnte also nur beschränkt auf Literatur zurückgegriffen werden.
Damit diese *Engine* auch generisch entwickelt werden kann, müssen mehrere Überlegungen im Aufbau eines solchen Partikelsystems gemacht werden.

Doch all diesen Schwierigkeiten zu trotz, konnte eine funktionierende Implementierung von der *Partikel Engine* programmiert werden.

4.3. Aussichten

Die entwickelte *Partikel Engine* bietet weitere Möglichkeiten für Ergänzungen, Erweiterungen und Verbesserung. Damit die *Partikel Engine* noch generischer wird, wären folgende Punkte sinnvoll:

- **Emitter**
  Der Emitter müsste ein selbstständiges Objekt mit einer Position, Ausrichtung und eventuell Abschusswinkel sein.
  In dieser Diplomarbeit existiert der Emitter als Ansammlung von Partikeln bzw. deren Startpunkten.

- **Kollisions- Abfrage**
  In dieser *Engine* wurden eine oder mehrere Ebenen geschnitten. Sinnvoller wäre, wenn die Partikel auf ein *Textur-Basierendes Height Field* reagieren würden. So könnte im Demo eine beliebige Szene erstellt werden, wo sich die Partikel dementsprechend verhalten.

- **Point Sprites Veränderung**
  Da die Berechnung in Hardware geschieht und mittels *Render_States* gesetzt wird, sehen die *Point Sprites* immer alle gleich aus, egal wie nahe sich der Betrachter von ihnen befindet. Nun werden die *Point Sprites* aber im *Shader* berechnet. Daher müsste die Darstellung manuell im *Shader* eingestellt werden.
  Eine weitere Veränderung der *Point Sprites* ist die Rotation. Diese ist nicht ganz einfach unter den *Point Sprites*. Eine weitere Möglichkeit wäre die Rotierung von den *Point Sprites*.

- **Veränderung der Farbe**
  Da im Shader die Positionen im *color.rgb* gespeichert werden und auch so zurückgeben werden, fällt die Veränderung der Farbe weg. So könnte eine Shader geschrieben werden, die Berechnung, bzw. die Farbverwaltung der Partikel übernimmt.

- **Verbesserte Schnittstelle**
  Das Einlesen der *Parameter* könnte über ein *ASCII File* erfolgen. Man könnte die *Partikel Engine* anpassen, so dass diese später in eine 3D-Engine eingebunden werden kann.

- **GUI**
  Da das *xFramework* auf dem DirectX 9.0 Sample Framework aufbaut, wurde die neue DirectX *GUI* verwendet. Diese ist jedoch noch nicht ganz ausgebaut, bzw. es wurden auch schon besser dokumentierte Projekte von MS hergestellt. Gewisse Objekte, zum Beispiel *Slider,* können nur integer (ganzzahlige Werte) zurückgeben.

5. Tutorial

In diesem Kapitel werden die Schnittstellen der *Partikel Engine* aufgezeigt und wie diese zu modifizieren sind.
Durch die eng gehaltene Schnittstelle ist es möglich mit wenigen Einstellungen verschiedenste Effekte zu erzielen.

5.1. Konstruktor

Erklärung zum benutzerdefinierten *Konstruktor*, der wie folgt aussieht:

```
particle = new xParticle(128, point, L"particlenew.bmp", "Particle",
                         shader, -9.81f, 0.0f, 3.0f,50.0f);
```

128
Die Texturgrösse ist 128 x 128, was 16'384 Partikeln entspricht (pro Pixel ein Partikel).

Point
Enum, Name für den Emittertyp, der verwendet werden soll (point, plane und circle stehen zur Auswahl).

L"particlenew.bmp"
Die für die Partikel zu verwendende Textur. Texturen befinden sich im Projektordner.

"Particle"
Name der *technique* (in der Shaderdatei zu finden) welche verwendet werden soll.

shader
Einem Partikelsystem muss ein Shader übergeben werden (shader->d(L"dx9_hlsl.fx")).

-9.81f
Gravität

0.0f
Geschwindigkeit

3.0f
TTL (Time to Life)

50.0f
TOB (Time of Birth)

Mit Hilfe dieses benutzerdefinierten *Konstruktors* können auch weitere Effekte erstellt werden.

5.2. Regen / Schnee / Hagel

```
particle = new xParticle(256, plane, L"portal9.bmp", "Particle", shader,
                        -9.81f, 0.0f, 3.0f, 50.0f);

//Die Grösse der Plane kann in xParticle.cpp verändert werden:

case 2:
for(int i=0; i < getPMax(); ++i)
{
      g_particles[i].m_vCurPos = D3DXVECTOR3(getRandomMinMax(-20.0f, 20.0f),
                                                              40.0f,
                                             getRandomMinMax(-20.0f, 20.0f));
```

Auf der x- und z-Achse werden die Partikel zufällig auf einer Fläche von X = -20 - 20 und Z = -20 - 20 erstellt, die Höhe (Y) ist 40.

5.3. Explosion

```
case 0:
for(int i=0; i < getPMax(); ++i)
{
      g_particles[i].m_vStartVel = D3DXVECTOR3(getRandomMinMax(-5.0f,
      5.0f), getRandomMinMax(-5.0f, 5.0f), getRandomMinMax(-5.0f,
      5.0f));
      …
}

particle = new xParticle(256, plane, L"portal9.bmp", "Particle", shader,
                         -9.81f, 0.0f, 3.0f, 50.0f);
```

m_vStartVel bestimmt die Richtungen, in die die Partikel los fliegen sollen.

So können Werte gewisser *Parameter* während der Laufzeit geändert werden (bzw. die Werte werden während der Laufzeit an den *Shader* gesendet).
Die Änderungen können links an den beschriebenen *Slidern* vorgenommen werden.

Abb. 12. Sliders

5.4. Zusätzliche Informationen über die in DirectX integrierte GUI

Gravity
Der Wertebereich von *Gravity* geht von *-20 - +20*. Die Schwerkraft, welche auf die Partikel wirkt, kann man selber bestimmen.

Wind
Die x- und z-Achse können unabhängig voneinander geregelt werden. Die Ablenkung der Partikel und deren Stärke bestimmt man selber.
Der Wertebereich des Windes geht von *-10 - +10*.

TTL
Time To Live wird über eine Zufallsfunktion bestimmt. Der kleinstmögliche Wert ist 2.0f. Der grösstmögliche Wert kann über den *Slider* bestimmt werden. Der Wertebereich des *TTL Sliders* geht von *3 - 10*.

Damping
Zum Schluss wird bestimmt, wie sich die Partikel nach dem Aufprall verhalten sollen, und ob die Partikel gleich an Ort und Stelle zum Stehen kommen sollen, oder ob sie noch etwas herumhüpfen sollen. Weiter lässt sich am *Damping Sider* die Stärke des Verhaltens einstellen.

6. Anhang

6.1. GPU-Programming

Moderne Grafikprozessoren werden heute meist als *GPU (Graphics Processing Unit)* oder auch *VPU (Visual Processing Unit)* bezeichnet. So eine *GPU* bzw. *VPU* kommt meistens auf Grafikkarten zum Einsatz, doch schon seit ein paar Jahren findet man diese modernen Grafikprozessoren auch auf Hauptplatinen.

Die Aufgabe des Grafikprozessors ist rechenintensive Prozesse der 3D-Computergrafik zu übernehmen. Der Ablauf der Prozesse ist in der folgenden Abbildung dargestellt. Dadurch wird der Hauptprozessor (*CPU - Central Processing Unit*) entlastet.

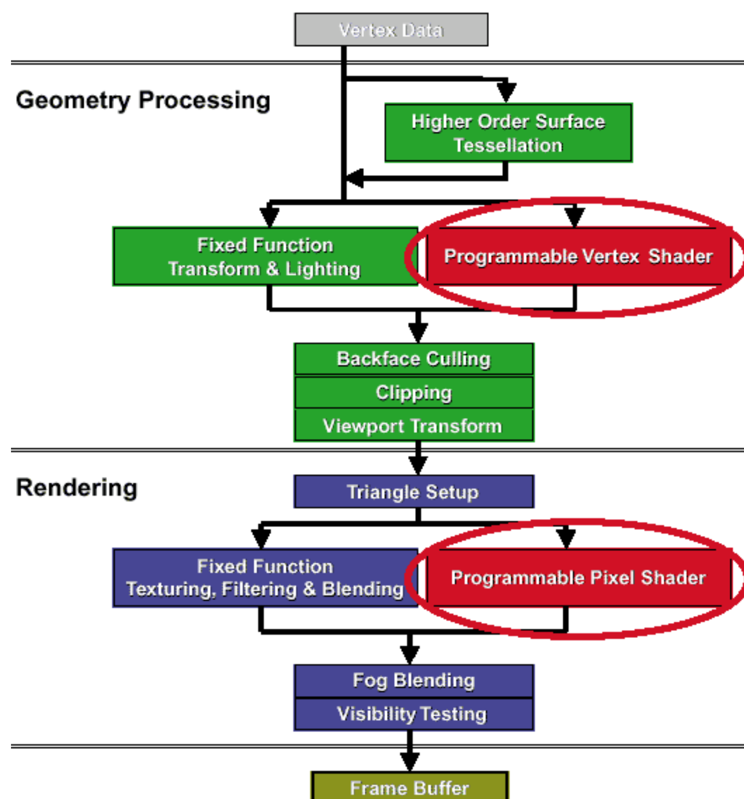

Abb. 13. Graphics Pipeline

Die bei der Abbildung 13 markierten Bereiche, Programmable Vertex und Programmable Pixel Shader, sind von zentraler Bedeutung für diese Diplomarbeit.

6.1.1. Technische Definition einer GPU

GPU ist ein Prozessor mit integrierten *Transform-, Lighting-, Triangle-Setup, Triangle-Clipping- und Rendering-Engines*, der in der Lage ist, Millionen rechenintensive Prozesse bzw. Polygone pro Sekunde zu verarbeiten.
GPU ist ein echter Prozessor auf dem Niveau eines *CPU*s. Beispielsweise enthält die Graphikkarte *Geforce3* von *Nvidias* 57 Millionen Transistoren bei einem 0,15 Mikrometer Fertigungsprozess. Im Gegensatz dazu enthält der Prozessor *Pentium 4* von *Intel* gerade mal 42 Millionen Transistoren bei 0,18 Mikrometer.

Da Fertigungsprozesse mit höherer Dichte schnellere Transistoren ermöglichen, ist zumindest physikalisch betrachtet die *Geforce3* der leistungsstärkere Prozessor.
Der Leistungszuwachs bei Grafikprozessoren hat mit einem jährlichen Faktor von 2,4 sogar das *Mooresche Gesetz* übertroffen und auch die Funktionalität der Karten hat sich deutlich vergrössert. So kann nahezu jede Komponente der *GPU* neu programmiert werden und sämtliche Berechnungen lassen sich mit vollen 32bit Gleitkommwerten ausführen.

Da ein Grafikbefehl gleichzeitig mit mehreren Daten (einem Vektor) arbeitet, kann *GPU* eher als *SIMD* (*single instruction multiple data*) Parallelprozessor beschrieben werden, was diesen auch für wissenschaftliche Anwendungen interessant macht.

6.1.2. Ergänzungen zur technischen Definition

Das wichtigste Feature ist die Möglichkeit die *GPU* zu programmieren. Auf diesem Gebiet fand letztes Jahr eine Projektarbeit (*GPU Programming*) an der Fachhochschule beider Basel (FHBB) statt.

Heutzutage gibt es zwei Hochsprachen für *GPU*s, welche sich im letzten Jahr durchgesetzt haben:

- HLSL (CG Nachfolger) High Level Shading Language Microsoft
- GLSL OpenGL Shading Language OpenGL ARB

6.2. Definitionen

- *Polygon*:
  Ein *Polygon*, griechisch für Vieleck, ist ein Begriff aus der Geometrie.

- *Vertex*:
  Aus dem Lateinischen, Plural: *Vertizes*. In der 3D-Computergrafik ist ein *Vertex* ein Scheitelpunkt einer Primitiven. Er enthält neben einer Positionsangabe in Form eines Vektors meistens noch einige andere Angaben wie z.B. eine Farbe, Transparenz oder eine zweite Positionsangabe, die für andere Zwecke verwendet werden können (z.B. Texturkoordinaten).

- *Primitive*:
  Grunddatentypen aus der 3D-Computergrafik wie z.B. ein Dreieck, welches aus drei Vertizes besteht.

- *Transistor*:
  Die Bezeichnung ist eine Kurzform für die englische Bezeichnung *transfer resistor*, die den Transistor als einen durch Strom steuerbaren Widerstand beschreiben sollte.

- *Mooresche Gesetz*:
  Die Beobachtung, dass sich durch den technischen Fortschritt die Komplexität von integrierten Schaltkreisen alle 18 Monate verdoppelt, heisst Mooresches Gesetz.

- *SIMD*
  SIMD-Computer, auch bekannt als *Array-Prozessoren*, dienen der schnellen Ausführung gleichartiger Rechenoperationen auf mehrere gleichzeitig eintreffende oder zur Verfügung stehende Eingangsdatenströme. Sie werden vorwiegend in der digitalen Bildverarbeitung (*JPEG, MPEG2, DCT*) eingesetzt. Viele moderne Mikroprozessoren (*PowerPC* und *x86*) besitzen inzwischen *SIMD*-Erweiterungen, das heisst spezielle zusätzliche Befehlssätze, die mit einem Befehlsaufruf gleichzeitig mehrere gleichartige Datensätze verarbeiten (*MMX, SSE2* und *3DNow!*).

Die bekanntesten Hersteller von *GPU*s bzw. *VPU*s sind (mit der momentan aktuellsten Grafikkarte und verfügbaren Schnittstelle):

- *ATI* — *X800XT PE* — *PCI-Express 16x und AGP 8x*
- *Nvidia* — *6800 Ultra* — *PCI-Express 16x und AGP 8x*
- *XGI* — *Volari Duo V8 Ultra* — *AGP 8x*

Zurzeit unterstützt nur *Nvidia Pixel Shader (PS)* und *Vertex Shader (VS)* der Version *3.0. ATI* und *XGI* unterstützten die Version 2.0.

7. Quellenverzeichnis

7.1. Literatur

[1] Hudritsch, Marcus; „Computergrafik II“, Vorlesungsskript

[2] Engel, Wolfgang; “ShaderX2, Shader Programming Tips and Tricks with DirectX 9“, 2004, WordWare

[3] Dalmau, Daniel S.C.; “Core Techniques and Algorithms in Game Programming“, 2004, New Riders

[4] Bourg, David M.; “Physics for Game Developers”, 2002, O’Reilly

[5] Deloura, Marc; “Spieleprogrammierung Gems 1”, 2002, mitp

[6] Latta, Lutz; “Building a Million Particle System”, 2004

[7] Lander, Jeff; “The Ocean Spray in Your Face”, 1998

[8] Kipfer, Peter; Segal, Mark; Westermann, Rüdiger; “UberFlow: A GPU-Based Particle Engine“, 2004

[9] Aaron, Lefohn; “GPU Data Formatting and Addressing”, 2004

7.2. Links

[10] Wikipedia.com; http://wiki.delphigl.com/

[11] Wikipedia.de; de.wikipedia.org

[12] Harris, Kevin; „dx9_particle_system“; www.codesampler.com

[13] Microsoft; msdn.microsoft.com/direcx

7.3. Abbildungen

Abb. 1 Texturen; „dx9_particle_system”
Abb. 13 Graphics Pipeline http://www.multi-hardware.com

Ehrlichkeitserklärung

Hiermit bestätigen die unterzeichnenden Autoren dieses Berichts, dass alle nicht klar gekennzeichneten Stellen von Ihnen selbst erarbeitet und verfasst wurden.

Ort und Datum	Unterschrift	
Muttenz, 23.12.2004	T. Egartner	D. Milenkovic