Seminar - Paralleles Rechnen auf Grafikkarten

Seminarvortrag:
Übersicht über die Programmierung von Grafikkarten

Marcus Schaber

05.05.2009

Betreuer: J. Kunkel, O. Mordvinova

Gliederung

- Allgemeine Programmierung
  Parallelität, Einschränkungen, Vorteile
- Grafikpipeline
  Programmierbare Einheiten, Hardwarefunktionalität
- Programmiersprachen
  Übersicht und Vergleich, Datenstrukturen, Programmieransätze
- Shader
  Shader Standards
- Beispiele
  Aus Bereich Grafik

Programmierung
Einschränkungen

- Anpassung an Hardware
  Kernels und Streams müssen erstellt werden.
  Daten als „Vertizes“, Erstellung geeigneter Fragmente notwendig.

- Rahmenprogramm
  Programme können nicht direkt von der GPU ausgeführt werden.
  Stream und Kernel werden von der CPU erstellt und auf die Grafikkarte gebracht

Programmierung
Parallelität

- „Sequentielle“ Sprachen
  Keine spezielle Syntax für parallele Programmierung.
  Einzelne shader sind sequentielle Programme.
- Parallelität
  wird durch gleichzeitiges ausführen eines Shaders auf unterschiedlichen Daten erreicht
- Hardwareunterstützung
  Verwaltung der Threads wird von Hardware unterstützt

Programmierung
Typische Aufgaben

- **Geeignet: Datenparallele Probleme**
  Gleiche/ähnliche Operationen auf vielen Daten ausführen,
  wenig Kommunikation zwischen Elementen notwendig
- Arithmetische Dichte:
  Anzahl Rechenoperationen / Lese- und Schreibzugiffe
- **Klassisch: Grafik**
  Viele unabhängige Daten: Vertizes, Fragmente/Pixel
  Operationen: gleicher shader für jedes Vertex/Fragment

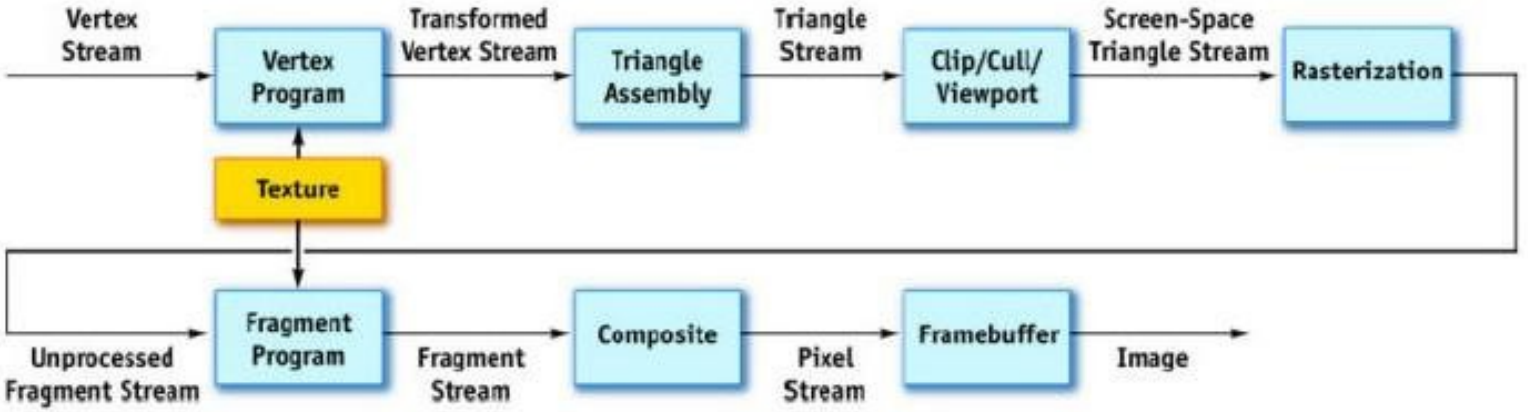
  Z.B. Auflösung 800x600 = 480.000 Pixel,
  keine Kommunikation untereinander
  → maximale Arithmetische Dichte

Programmierung
Typische Aufgaben GPGPU

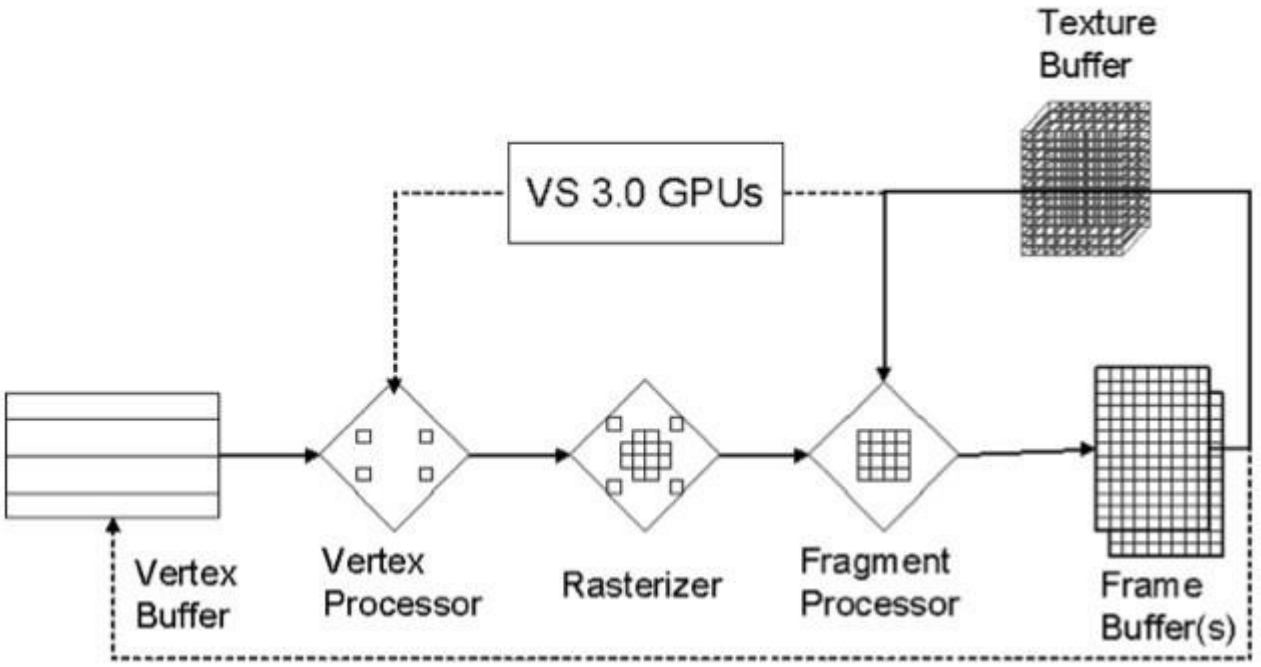
- GPGPU: Z.B. Gleichungssystem lösen
  Bearbeiten einer Zeile kann parallel ausgeführt werden,
  dann ist Schreibzugriff notwendig
- Allgemein Gitter-Methoden (aus der Numerik)
  Repräsentation des Gitters durch Textur
- Nutzung von Texturen generell praktisch
  Lesen/Schreiben in Textur wird von Grafikkarte unterstützt.
  Das Ausgabeziel des Renderers kann auch eine Textur sein.

Die Grafikpipeline

- Schwierig passende Grafik zu finden
  „Unified Shader“ machen die Aufteilung der programmierbaren Stufen überflüssig



Die Grafikpipeline



Grafikpipeline

Programmierbare Einheiten

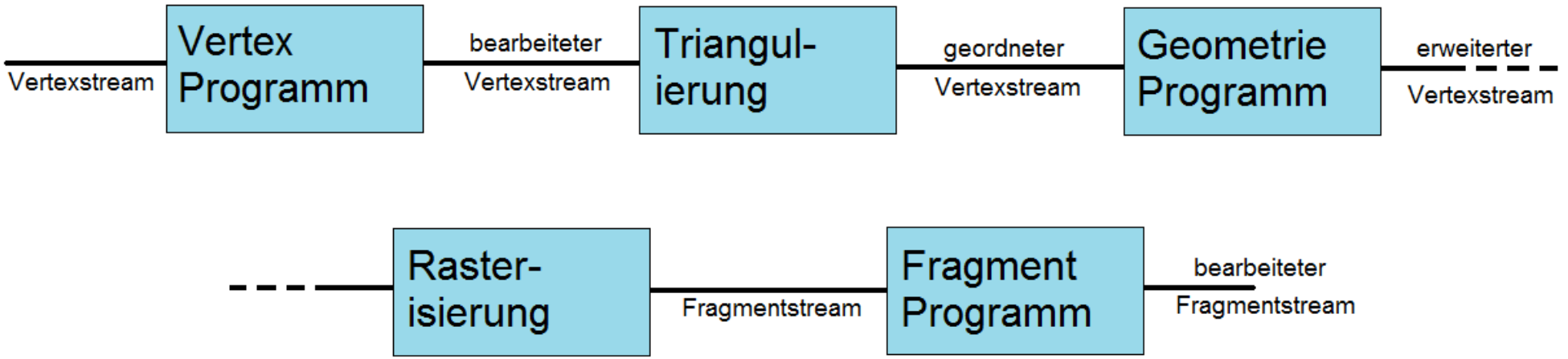
3 Programmierbare Pipelinestufen:

- Vertex processing
  Operiert auf Vertizes im Stream
- Primitive processing
  Erstellt für jedes eingehende Primitiv weitere neue
- Fragment processing
  Operiert auf Fragmenten aus dem Rasterizer

Grafikpipeline

Programmierbare Einheiten

Für die Programmierung ist diese Sichtweise nützlich

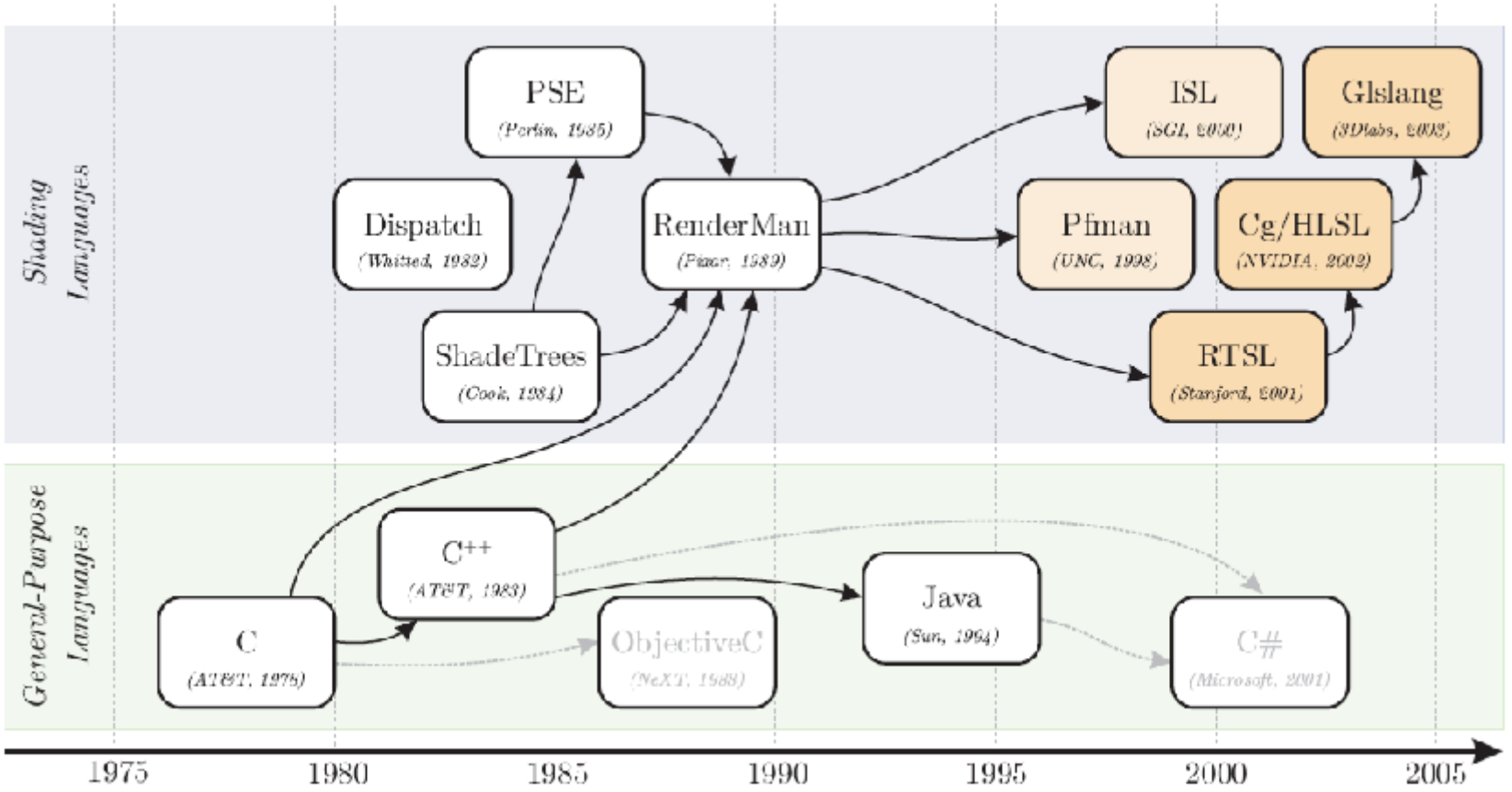


Programmiersprachen
Übersicht

- High-Level für Grafik:
  CG (NIVIDIA), HLSL (DirectX), GLSL (OpenGL), RenderMan (Pixar)
- High-Level für GPGPU:
  CUDA (NVIDIA), Brook

Programmiersprachen

Entwicklung



Programmiersprachen

Datentypen

- Datentypen

  Skalare, Vektoren, Matrizen und arrays. Meist float, int, bool.
  Sampler für Texturen.
  Erstellung eigener Typen möglich. („struct“)

```
float4 vec = {0,0,0,0};
float4x4 matrix;
Sampler2D texSampler;

struct VS_in {
    float4 Pos : POSITION;
    float4 normal : NORMAL;
    float4 texCoord : TEXCOORD0;
}
```

Programmiersprachen

Funktionen und Kontrollstrukturen

```
// eine Funktion im GPU Programm
float4 calcColor(float3 normal, float3 direction, float4 color) {
      float4 res = dot(direction, normal) * color;
      return res;
}

float3 directions[numLights];  // werden vor Ausführung...
float4 colors[numLights];      // ...des shaders gesetzt
// fragment shader
float4 PS {
      float4 color = {0,0,0,0};
      for (int i = 0; i<numLights; i++) {
            color += calcColor(normal, directions[i], colors[i]);
      }
      return color;
}
```

Programmiersprachen

Einbindung in Hochsprachen

- **Spracherweiterung:**
  Erweiterung einer vorhandenen Sprache, wie C.
  Spezieller Compiler notwendig.
- **Eigenständige Sprache:**
  Kompilieren unabhängig möglich.
  Aufruf durch CPU-Programm.
  Einbindung meist in der Form (vereinfacht):
  - Programm kompilieren
  - Programm laden
  - Programm verwenden

Programmiersprachen

Brook

- Erweiterung von C
- Stream und Kernel als Sprachelemente
- Sprachunterstützung für Reduktionen mit Schlüsselwort „reduce“
- Restriktionen bei Standard C.
  Bestimmte Schlüsselwörter, Bibliotheksaufrufe eingeschränkt, Pointer eingeschränkt, keine Rekursion

Programmiersprachen

Brook Beispiel

- Stream:

```
float a<> [3] [2];          // stream aus 3x2 float arrays
float b<300,200>;           // legt Anzahl der Elemente im stream fest
```

- Kernel:

```
void kernel foo (float a<>,out float b<>,float p)
{
    b = a + p;
}
// a ist input-stream, b output-stream, p konstanter Parameter.
```

Der Kernel wird für jedes Element im Stream ausgeführt.
Kernelaufruf wie bei einer normalen C Funktion.

Programmiersprachen

Brook Beispiel

- Reduktion: Summenbildung

```
float a<> [3] [2];
void kernel sum (float a<>,reduce float result) {
      result += a;
}
```

- Restriktionen beim Kernel

Keine Pointer und Adressoperatoren
keine Funktionspointer
Keine Bibliotheksaufrufe (malloc, printf, ...)
Keine Zugriffe auf globalen Speicher

Programmiersprachen
CG

- „C for Graphics“
- Eigenständige Sprache
- Vertex, Fragment und „Geometrie“ Programme
- Syntax sehr ähnlich wie C
- Unabhängig von Graphik-API
- Compiler erstellt shader für DirectX oder OpenGL

Programmiersprachen

CG Beispiel

- Einbindung:

```
CGprogram program = cgCreateProgramFromFile(
    context, CG_SOURCE, filename, profile, "main", NULL);
cgGLLoadProgram(program);
cgGLBindProgram(myVertexProgram);
cgGLEnableProfile(CGProfile profile);
//Programm wird jetzt beim Rendern benutzt
cgGLDisableProfile(CGProfile profile);
```

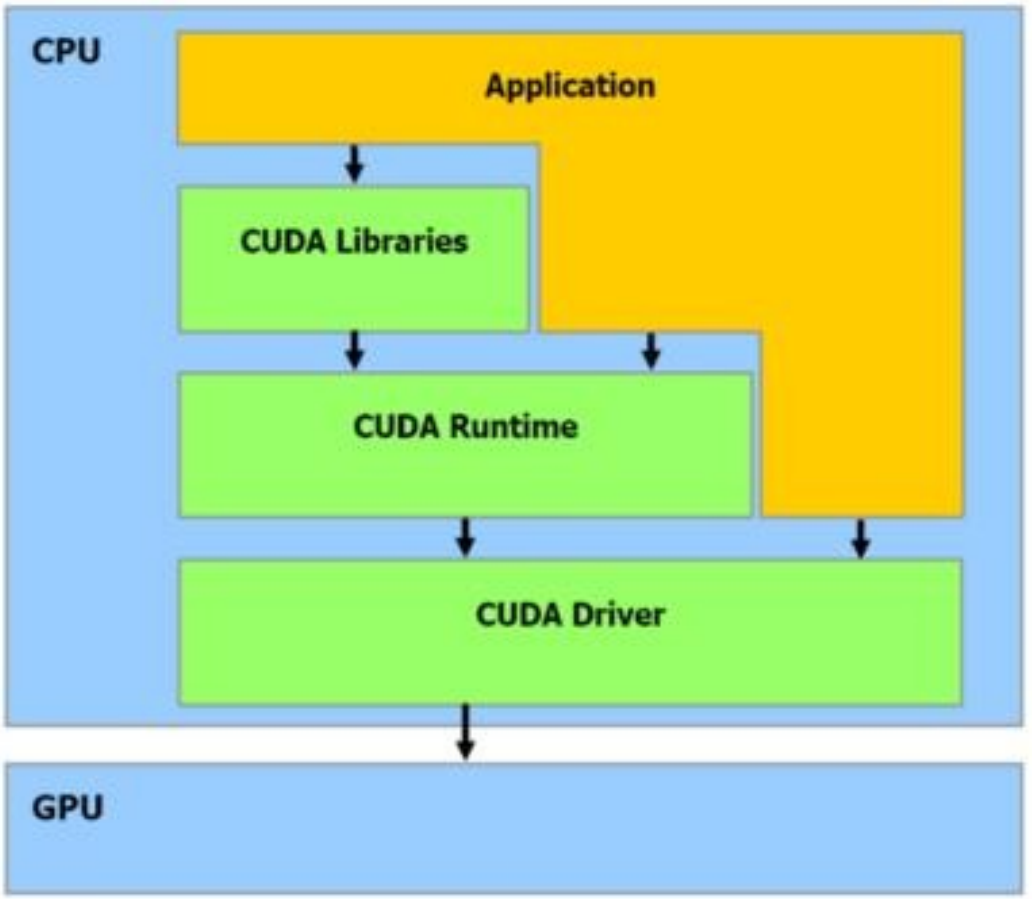
Programmiersprachen
CUDA

- „Compute Unified Device Architecture“
- Erweiterung von C++
- Nutzbar ab NVIDIA Grafikkarten der GeForce 8 Serie und Quadro FX 5600
- Für Wissenschaftliche Berechnungen gedacht

Programmiersprachen
CUDA

- Grafikkarte kann allgemeiner genutzt werden
  - Speicherzugriff:
    cudaMalloc((void **)&host_data, DATA_SZ)
    cudaMemcpy(device_data, host_data, DATA_SZ,
    cudaMemcpyHostToDevice
  - Syntax für parallele Programmierung vorhanden
  - Kernelaufruf:
    scalarProdGPU<<<128, 256>>>(d_C, d_A, d_B, VECTOR_N, ELEMENT_N);

Programmiersprachen
CUDA



Programmiersprachen
GLSL

- „OpenGL Shading Language“
- Eigenständige Sprache
- Für Vertex und Fragment Programme
- Teil des OpenGL Standards

Programmiersprachen
HLSL

- „High Level Shader Language“
- Eigenständige Sprache
- Teil von DirectX
- Vertex, Fragment und „Geometrie“ Programme
- Syntax sehr ähnlich wie C

Programmiersprachen

HLSL Effect-System

Eine gemeinsame Umgebung für Shader, Variablen, Konstanten, Funktionen und Datentypen.

Einstellungen des Renderes können in techniques beschrieben werden.

```
technique10 Render {
	pass P0 {
		SetVertexShader( CompileShader( vs_4_0, VS() ) );
		SetGeometryShader( CompileShader( gs_4_0, GS() ) );
		SetPixelShader( CompileShader( ps_4_0, PS() ) );
		SetBlendState( NoBlending, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF );
		SetDepthStencilState( EnableDepth, 0 );
	}
}
```

Shader

- Standard wird durch DirectX vorgegeben
  ATI und NVIDIA arbeiten mit Microsoft zusammen.
  Mit jeder neuen Shader-Version werden neue Anforderungen an die Hardware gestellt.

Pixel-Shader	Vertex-Shader	Bemerkungen
1.0	1.0	für 3dfx Rampage vorgesehen, kam jedoch nie auf den Markt
1.1	1.0	kleinster gemeinsamer Nenner für alle DirectX-8-Grafikchips
1.2		3DLabs-Grafikchips P9 und P10
1.3		Nvidia-GeForce-4-Ti-Serie, Matrox Parhelia und SiS Mirage 2
1.4	1.0	Radeon-8000-Serie und XGI-Volari-V3-Serie
2.0	2.0	kleinster gemeinsamer Nenner für alle DirectX-9-Grafikchips
2_A		Nvidia-GeForce-FX-Serie
2_B	2.x	ATI Radeon X7xx und X8xx
3.0	3.0	gemeinsamer Nenner für modernere DirectX-9-Grafikchips
4.0	4.0	Unified Shader Model

Shader

Shader Standards

	Shader 1.x	**Shader 2.0**	**Shader 3.0**	**Shader 4.0**
Vertex Instructions	128	256	512	65.536
Pixel Instructions	4+8	32+64	512	65.536
Vertex Constants	96	256	256	16 x 4.096
Pixel Constants	8	32	224	16 x 4.096
Vertex Temps	16	16	16	4.096
Pixel Temps	2	12	32	4.096
Vertex Inputs	16	16	16	16
Pixel Inputs	4+2	8+2	10	32
Render Targets	1	4	4	8
Vertex Textures	-	-	4	128
Pixel Textures	8	16	16	128
2D Texture Size	-	-	2.048 x 2.048	8.192 x 8.192

Shader

Vertex shader Beispiel

```
// Eingabe Vetex-Stream
struct VS_INPUT {
	float3 Pos : POSITION;
	float3 Norm : NORMAL;
	float2 Tex : TEXCOORD0;
};
// Vertex-shader
PS_INPUT VS( VS_INPUT input ) {
	PS_INPUT output = (PS_INPUT)0;

	output.Pos = mul( float4(input.Pos,1), World );  // Objekt-Koordinaten zu Welt-Koodrinaten

	// Vertizes verschieben
	output.Pos.x += sin( output.Pos.y*0.1f + Time ) * Waviness;

	output.Pos = mul( output.Pos, View );          // Kameraperspektive
	output.Pos = mul( output.Pos, Projection );    // Projektion auf Ebene
	output.Norm = mul( input.Norm, World );        // Vertex-Normale zu Weltkoordinaten
	output.Tex = input.Tex;                        // Texturkoordinaten durchreichen

	return output;
}
```

Shader

Pixel shader Beispiel

```
struct PS_INPUT {
	float4 Pos : SV_POSITION;
	float3 Norm : TEXCOORD0;
	float2 Tex : TEXCOORD1;
	float3 ViewR : TEXCOORD2;
};

float4 PS( PS_INPUT input) : SV_Target {
	float fLighting = saturate( dot( input.Norm, vLightDir ) );

	// environment map Texturwert lesen
	float4 cReflect = g_txEnvMap.Sample( samLinearClamp, input.ViewR );

	// Texturwert lesen
	float4 cDiffuse = g_txDiffuse.Sample( samLinear, input.Tex ) * fLighting;

	// beide Texturwerte addieren
	float4 cTotal = cDiffuse + cReflect;

	return cTotal;
}
```

Shader

Geometrie shader Beispiel

```
float Explode;  // wird vom Hauptprogramm gegeben

// Eingabestream in den Geometrie-shader
struct GSPS_INPUT {
	float4 Pos : SV_POSITION;
	float3 Norm : TEXCOORD0;
	float2 Tex : TEXCOORD1;
};

// Geometrie-shader
void GS( triangle GSPS_INPUT input[3], inout TriangleStream<GSPS_INPUT> TriStream ) {
	GSPS_INPUT output;
	for( int i=0; i<3; i++ ) {
		// Werte für neues Vertex festlegen
		output.Pos = input[i].Pos + float4(Explode, 0.0, 0.0, 0.0) * 10;
		output.Pos = mul( output.Pos, View );
		output.Pos = mul( output.Pos, Projection );
		output.Norm = input[i].Norm;
		output.Tex = input[i].Tex;
		TriStream.Append( output );  // neues Vertex in den Stream einfügen
	}
}
```

Abschluss

Fragen?

Quellen

- Artikel von Julian
- wikipedia
- GPU Gems 2
- Microsoft DirectX SDK
- Brook Spec v0.2; Ian Buck; October 31, 2003
- NVIDIA CUDA SDK Samples