UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Gegenüberstellung herkömmlicher Shader-Hochsprachen und der Metasprache Sh bei der GPU Programmierung

Studienarbeit
im Studiengang Computervisualistik

vorgelegt von
Michelle Kristin Martin

Betreuer: Dipl.-Inform Matthias Biedermann
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im September 2006

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 von Michelle Kristin Martin

Thema: **Gegenüberstellung herkömmlicher Shader-Hochsprachen und der Metasprache Sh bei der GPU Programmierung**

Shader werden in der GPU-Programmierung sowohl in Bereichen der Bildverarbeitung für grafische Effekte, als auch, im Rahmen von GPGPU, als zusätzlicher Prozessor mit der Fähigkeit zur parallelen Datenverarbeitung eingesetzt. Realisiert werden diese Shader häufig mit Hochsprachen, wie z.B. Cg oder GLSL. Diese Programme werden dabei in der Regel separat vom Hauptprogramm entwickelt. Dies bedeutet auf der einen Seite weiteren Aufwand, um den Shader in das Hauptprogramm zu integrieren, zur Grafikkarte zu laden und um alle erforderlichen Parameter anzubinden. Auf der anderen Seite erschwert es auch die Fehlersuche, da Syntaxfehler nicht so einfach zu lokalisieren sind. Änderungen im Shaderprogramm, die andere Parameter erfordern, verursachen weiteren Aufwand, der besonders beim schnellen Testen von Ideen und Konzepten behindernd wirkt.
Eine Alternative bietet die Metasprache Sh. Die in Sh geschriebenen Shader werden direkt im Hauptcode des Programms entwickelt, basierend auf der C++-Syntax, die den meisten Programmierern bereits bekannt ist. Weiterhin werden durch die Aufhebung der Trennung zwischen Shader und Hauptprogramm syntaktische Fehler bereits beim Compilieren entdeckt. Die Bibliothek organisiert optional auch die Anbindung aller Daten an die Grafikkarte und macht Änderungen an den Programmen und schnelles Testen neuer Ideen möglich. Darüber hinaus bietet Sh weitere Funktionalitäten, wie z.B. Emulation auf der CPU und algebraische Möglichkeiten, mehrere Shader miteinander zu kombinieren.

Ziel der Arbeit ist, neben der Einarbeitung in die Shader Metaprogrammierung mittels Sh, die Implementierung einer Beispielanwendung zur Demonstration der besonderen Spracheigenschaften von Sh. Ein weiterer Schwerpunkt soll die Gegenüberstellung bekannter Hochsprachen wie GLSL oder Cg und Sh, sowie ein Ausblick auf weitere Entwicklungen sein. Dabei werden essentielle Spracheigenschaften, insbesondere Sprachumfang, Anwenderfreundlichkeit und Robustheit sowie nachträgliche Optimierbarkeit verglichen und die Ergebnisse dokumentiert.

Die Schwerpunkte der Arbeit im Überblick sind:

1. Recherche zu Programmiersprachen und -konzepten für GPUs sowie Einarbeitung in Sh
2. Entwurf und Implementation einer Beispielapplikation zur Demonstration von Sh
3. Gegenüberstellung gängiger Shaderhochsprachen im Hinblick auf weiterführende Konzepte von Sh
4. Dokumentation der Ergebnisse.

Koblenz, den 15.03.2006
Betreuer: Dipl.-Inform. Matthias Biedermann

– Prof. Dr. Stefan Müller –

Erklärung

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

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

Gegenüberstellung herkömmlicher Shader-Hochsprachen und der Metasprache Sh bei der GPU Programmierung

Inhaltsverzeichnis

Verzeichnis der Code Listings

Abbildungsverzeichnis

1. Einleitung

Mit der Einführung von Windows Vista und DirectX 10 und damit auch Shader Model 4.0[1] befinden wir uns mittlerweile in der 3. Generation der programmierbaren Grafikhardware, die es gestattet, die Standard-Pipeline[2] auf den Grafikkarten durch selbst geschriebene Shader Programme teilweise zu verändern und zu ersetzen.

Schon lange ist es nicht mehr notwendig, Shader direkt in Assembler zu schreiben, da sich speziell für diesen Zweck entwickelte Hochsprachen etabliert haben. Die populärsten darunter sind GLSL für die OpenGL API, HLSL für DirectX und das von NVidia entwickelte Cg. Die wachsende Unterstützung durch die Grafikchips, der sogenannten GPUs[3], und die beständige Weiterentwicklung des zugrunde liegenden Shader Models haben dazu geführt, dass die Sprachen in Umfang und Komplexität deutlich gewachsen sind. Neben dem hauptsächlichen Einsatzgebiet in der Computergrafik hat sich dadurch die allgemeine GPU-Programmierung entwickelt, die sogenannte GPGPU[4], die sich den Prozessor der Grafikkarte als zusätzlichen Prozessor für allgemeine Berechnungen zunutze macht.

1.1. herkömmliche Shader-Hochsprachen

Die Sprachen Cg, GLSL sowie HLSL haben einen ähnlichen zugrunde liegenden Aufbau. Zunächst einmal lehnen sich alle an der C/C++ Syntax an, um die Programmierung der Shader so intuitiv wie möglich zu gestalten. Darüber hinaus verfügen alle über Datentypen, um Vektoren und Matrizen verschiedener Größe darzustellen und bieten für diese Datentypen auch mathematische Funktionen an, die teilweise von der Grafikhardware direkt unterstützt werden. Die Namen der Funktionen sind dabei selbsterklärend und meist in allen Sprachen ähnlich. Die Shader werden in aller Regel extern vom Hauptprogramm in separaten Dateien deklariert, die eingeladen werden müssen, oder als Strings innerhalb des Programmes.

Der Aufbau der Shader selbst ist ebenfalls vergleichbar. Alle Ein- und Ausgabeparameter eines Shader müssen angegeben werden, sowie die genutzten Register, wie Position, Farbe und Texturkoordinaten manuell an Variablen geknüpft werden. Gegebenenfalls muss noch das jeweils vom Vertex oder Pixel Shader genutzte Shader Model angegeben werden.

Die größten Unterschiede finden sich in der Anbindung der Shader an das Hauptprogramm, sowohl beim Einladen, Kompilieren und Linken des Shaders selbst, als auch bei der Verknüpfung der Variablen und Parameter. Diese müssen manuell vom Hau

ptprogramm aus einzeln vor jedem Renderaufruf aktualisiert werden.

1 Shader Model: definiert die von der Grafikkarte unterstützten Fähigkeiten, z.B. das in den Shadern zur Verfügung stehende Instruktionsset, Anzahl der Texturzugriffe und maximale Anzahl der Anweisungen. Seit Shader Model 3.0 steht dynamische Flusskontrolle zur Verfügung. Das unterstützte Shader Model hängt von der Grafikkarte ab.

2 Standard-Pipeline: Arbeitsablauf der Grafikkarte einschließlich der Geometrie-Transformation, Per-Vertex-Beleuchtung und einfachen Texturzugriffen, solange der Nutzer sie nicht durch eigenen Shader ersetzt

3 GPU – Graphics Processing Unit; der Prozessor der Grafikkarte

4 GPGPU – General Purpose Graphics Processing Unit; die Zweckentfremdung der GPU als Co-Prozessor für allgemeine Aufgaben[1]

Zudem unterstützt DirectX mit HLSL (High Level Shading Language) die Nutzung sogenannter fx-Dateien. Diese Dateien können nicht nur mehrere Shader enthalten, sondern es können auch Techniken definiert werden, in denen diese in mehreren Render-Durchläufen in einer festgelegten Reihenfolge aufgerufen werden. So lassen sich insbesondere Multipassing Shader bereits relativ übersichtlich entwickeln.

Die auf OpenGL spezialisierte Sprache GLSL ist möglicherweise die mächtigste der vorgestellten Sprachen. Sie ist im Vergleich zu HLSL und Cg noch relativ jung. Gerade durch die hohe Spezialisierung bietet sie viele Funktionen, die die Programmierung erleichtern, wie z.B. den direkten Zugriff auf OpenGL interne Daten im Shader, oder die automatische Geometrietransformation.

Die Sprache Cg (C for Graphics) bietet als einzigste ein höheres Abstraktionsniveau, da sie nicht fest an eine der beiden Grafik APIs gebunden ist. Der von NVidia entwickelte Compiler kann die in Cg geschriebenen Shader sowohl für OpenGL in ARB Assembler übersetzen, als auch in ein DirectX taugliches Format. Die Sprache hat aber auch deutliche Nachteile. Zum einen kann in ihr nicht wie in GLSL auf die OpenGL States zugegriffen werden. Dies bedeutet zusätzlichen Aufwand im Hauptprogramm. Und zum anderen kann sie, da sie vom Grafikkartenhersteller NVidia entwickelt wird, die neuesten Fähigkeiten der jeweils aktuellen Grafikkarten des Konkurrenten ATI nicht nutzen. Trotz dieser Einschränkungen ist die Sprache im professionellen Bereich weit verbreitet, insbesondere bei Firmen aus dem Bereich der Spiele-Entwicklung[2].

1.2. das Konzept der Metaprogrammierung

Wie bereits erwähnt sind Shader-Hochsprachen für gewöhnlich an eine bestimmte, zugrunde liegende Umgebung gebunden. So ist GLSL auf die OpenGL Grafik API beschränkt, während DirectX seine Shader in HLSL verlangt. Unterstützt ein Programm beide Grafik APIs, müssen alle verwendeten Shader jeweils in den entsprechenden Sprachen vorliegen. Eine Ausnahme dazu bildet die Sprache Cg, die trotz ihrer Nachteile sehr etabliert ist. Der Grund hierfür liegt auch in der Unterstützung beider Grafik-APIs.

Dem Prinzip einer spezifischen Sprache gegenüber steht das Prinzip der Metaprogrammierung: das eigentliche Programm wird dabei nicht direkt in der Zielsprache oder für eine bestimmte Zielumgebung geschrieben, sondern stattdessen in einer "Metasprache" definiert und dann automatisiert in das gewünschte bzw. unterstützte Zielsystem übersetzt. Dies bedeutet, dass solche Programme ohne Mehraufwand in verschiedenen Systemen eingesetzt werden können.

Die Metasprache Sh ist ein erster Versuch das Konzept der Metaprogrammierung auf die Shaderprogrammierung anzuwenden. Sh wurde 2003 als Forschungsprojekt unter Michael McCool an der Universität von Waterloo entwickelt. Es bietet die Möglichkeit, Shader sowohl für allgemeine GPGPU Anwendungen als auch für Grafikanwendungen in einer einfachen Metasprache zu schreiben und diese dann in verschiedene Sprachen zu übersetzen.

In dieser Arbeit wird die Metasprache Sh näher vorgestellt und untersucht. Im zweiten Kapitel wird zunächst ein theoretischer Überblick über den Funktionsumfang des Sh Toolkits gegeben. Ergänzend dazu beinhaltet das dritte Kapitel mehrere Tutorials, die den praktischen Einsatz von Sh demonstrieren sollen. Im Rahmen dieser Arbeit wurde darüber hinaus eine Beispielapplikation implementiert, die einige Spracheigenschaften von Sh visualisiert. Näheres dazu findet sich im vierten Kapitel. Im fünften Kapitel wird abschließend grundlegend untersucht, wie sich die Metasprache in der direkten Gegenüberstellung mit einer herkömmlichen Shader-Hochsprache verhält, insbesondere im Hinblick auf Anwendbarkeit und Performanz.

2. Überblick über Sh

Das "Sh GPU Programming Toolkit" wurde 2003 an der Universität Waterloo von Michael McCool entworfen. Unter seiner Leitung wurde es in der Abteilung für Computergrafik als Forschungsprojekt weiterentwickelt und implementiert.

Das Projekt befindet sich noch immer in Entwicklung und verändert sich fortlaufend. Dabei soll Sh dauerhaft unabhängig bleiben von Plattform und Grafik API[5], und außerdem Grafikkartenhersteller neutral sein. Derzeit wird es unter einer Open Source Lizenz veröffentlicht und kann in der jeweils aktuellsten Version unter *http://libsh.sourceforge.net/* heruntergeladen werden. Einen detaillierteren Einblick in das Toolkit bietet das von Michael McCool und Stefanus Du Toit veröffentlichte Buch "*Metaprogramming GPUs with Sh*", das 2004 unter dem A.K. Peters Verlag erschienen ist [3].

Sh wurde als Metasprache für die GPU-Programmierung in C++ entworfen und ihre Entwickler sahen als Einsatzgebiet nicht nur die Nutzung der GPU für graphische Anwendungen vor, sondern auch die Nutzung für allgemeine Programmierung im Sinne eines Co-Prozessors. Dabei wurde ein besonderes Augenmerk auf die syntaktische Ähnlichkeit zur Programmiersprache C++, in die sich Sh einbettet, gelegt. Zur Zeit unterstützt Sh Windows, Unix und OS X, sowie 32-Bit und 64-Bit Systeme.

2.1. Sh als Mathematik-Bibliothek

Das Fundament von Sh bildet eine komplett eigenständige, optimierte Mathematik-Bibliothek, die häufig genutzte Variablentypen und mathematische Funktionen auf diesen Variablen definiert. Diese Bibliothek kann somit auch unabhängig von der Shaderentwicklung genutzt werden.

Die wichtigsten Variablentypen sind Vektoren und Matrizen. Alle Typen sind als Templates realisiert, so dass man Vektoren mit einer beliebigen Anzahl von Komponenten, bzw. Matrizen mit beliebiger Dimension erstellen kann. Der zugrunde liegende Datentyp der Komponenten, wie z.B. eine Fließkommazahl (float), ist wählbar. Tabelle 2.1 zeigt die unterstützten Datentypen.

Für die am häufigsten genutzten Typen, wie z.B. Fließkommazahl-Vektoren der Größe 1 bis 4, sowie Matrizen der Größe 1x1 bis 4x4 gibt es vordefinierte Bezeichner, um den Schreibaufwand zu reduzieren.

Alle Sh-eigenen Variablentypen beginnen mit der Präfix "Sh" und enden, angelehnt an die OpenGL Bezeichnungen, mit einer Anzahl-Typ Kombination. Einige Beispiele:

`ShVector1f`	float-Vektor mit nur einer Komponente
`ShMatrix3x4ui`	eine unsigned int-Matrix mit 3 Zeilen und 4 Spalten
`ShVector3d`	double-Vektor mit drei Komponenten

5 API – Application Programming Interface; hier: Schnittstelle für die Programmierung von Computergrafik, z.B. OpenGL und DirectX / Direct3D

Sh-Typ	Kürzel	Beschreibung (für eine 32-Bit Umgebung)
char	b	8 Bit Ganzzahlwert
unsigned char	ub	8 Bit Ganzzahlwert ohne Vorzeichen
short	s	16 Bit Ganzzahlwert
unsigned short	us	16 Bit Ganzzahlwert ohne Vorzeichen
int	i	32 Bit Ganzzahlwert
unsigned int	ui	32 Bit Ganzzahlwert ohne Vorzeichen
float	f	32 Bit Fließkommawert
double	d	64 Bit Fließkommawert
ShFracByte	fb	8 Bit Kommawert zwischen -1 und 1
ShFracUByte	fub	8 Bit Kommawert zwischen 0 und 1
ShFracShort	fs	16 Bit Fließkommawert zwischen -1 und 1
ShFracUShort	fus	16 Bit Fließkommawert zwischen 0 und 1
ShFracInt	fi	32 Bit Fließkommawert zwischen -1 und 1
ShFracUInt	fui	32 Bit Fließkommawert zwischen 0 und 1
ShHalf	h	Fließkommawert mit halber Präzision (Sh intern: double)

Listing 2.1: von Sh unterstützte Datentypen

Der Zugriff auf einzelne Komponenten eines Vektors, bzw. Tupels kann über den []-Operator und einen Index geschehen, oder auf ein Teil-Tupel mittels des ()-Operators, bei dem die Reihenfolge der Indizes beliebig gewählt werden kann. Die Anzahl der Daten muss dabei den Komponenten des Zielvektors entsprechen:

```
ShVector3f a;
ShVector2f b;

// die z- und x-Komponente werden in b kopiert
b = a(2, 0);

// die y-Komponente von b wird in alle
// Komponenten von a kopiert
a = b(1, 1, 1);
```

Listing 2.2: Zugriff auf Vektoren (Tupel) in Sh

Auf ähnliche Art und Weise geschieht auch der Zugriff auf Matrizen. Mit einem doppelten ()-Operator können Teilmatrizen gewählt werden, während einzelne Skalare, sowie Zeilen und Spalten mittels des []-Operators gewählt werden. Alle Indizes beginnen bei 0.

```
ShMatrix4x4f a;
ShMatrix3x3f b;
ShAttrib1f c;

// die obere linke 3x3-Teilmatrix wird kopiert
b = a(0, 1, 2)(0, 1, 2);

// der 2. Wert der 4. Spalte wird kopiert
c = b[1][3];
```

Listing 2.3: Zugriff auf Matrizen in Sh

Sh bietet sowohl den Typ *ShVector* als auch den Typ *ShPoint* an. Dabei schränkt es aber die auf den Tupeln erlaubten Operationen nicht abhängig von diesem semantischen Typ ein[6]. So sind auf *ShPoint* ebenso Addition und skalare Multiplikation definiert, wie auf *ShVector*. Wird ein 3-Komponenten Tupel mit einer 4x4 Matrix multipliziert, so wird automatisch eine 4. homogene Koordinate erzeugt. Im Falle des Typs Punkt ist diese Koordinate 1, im Falle eine eines Vektors ist sie 0. Falls die Unterscheidung zwischen Vektor und Punkt irrelevant ist, steht der Typ *ShAttrib* als generisches Tupel zur Verfügung.

Auf alle Typen sind mathematische Funktionen und Operatoren vor-definiert. Neben den aus C/C++ bekannten Methoden sind dies vor allem für Vektor- und Matrizenrechnung notwendige Funktionen wie Kreuz- und Skalarprodukt, Normalisierung und Invertierung, aber auch Funktionen zur Berechnung von Beleuchtungs-Koeffizienten und Reflektionen. Ebenso sind die Berechnen der 3. Wurzel eines Wertes, der Summe und dem Produkt der Komponenten eines Vektors und des Logarithmus zur Basis 2, 10 und *e* bereits enthalten. Darüber hinaus gibt es aber auch fortgeschrittenere Methoden, beispielsweise zur Berechnung von Bernstein-Polynomen, Bézier- und Hermite-Splines, sowie diversen Rausch-Funktionen, wie der Perlin- und der Worley-Funktion. Trigonometrische Funktionen wie Sinus und Kosinus arbeiten auf den Sh Vektoren jeweils komponentenweise.

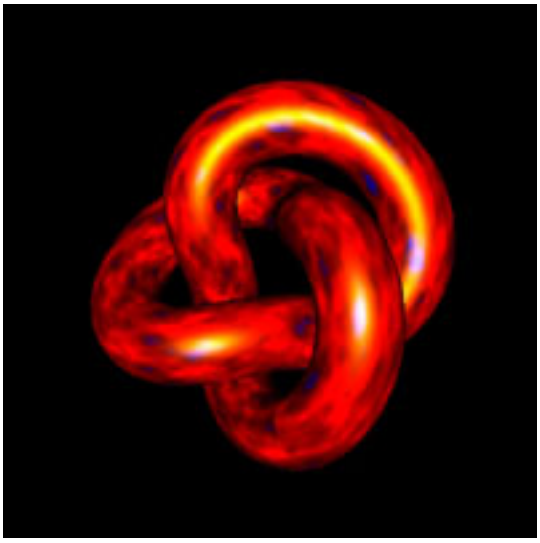

Abbildung 2: Shader mit Worley Funktion

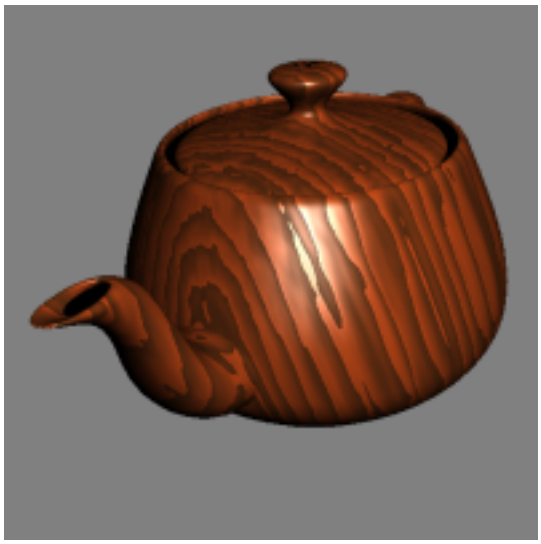

Abbildung 1: Holz Shader mit Perlin Funktion

2.2. Sh für die Shadermetaprogrammierung

Während die Mathematikbibliothek die notwendige Grundlage bietet, liegt das eigentliche Einsatzgebiet von Sh in der Shader-Metaprogrammierung. Und hier sind gegenüber anderen Shader-Hochsprachen einige Besonderheiten zu finden:

Sh stellt Schnittstellen für verschiedene APIs zur Verfügung. Der Nutzer kann nach dem gleichen Grundmuster Shader Programme in Sh schreiben und in verschiedenen Umgebungen einsetzen. Für jedes von Sh unterstützte Zielsystem, zur Zeit der Erstellung dieser Arbeit waren dies ARB-Assembler und GLSL, existiert ein sogenanntes "Backend", das den Sh Code in den entsprechenden

6 Punkte und Vektoren verhalten sich unter affinen Transformationen unterschiedlich

Shader Code der API übersetzt, und auch alle sonstigen benötigten Schnittstellen-Funktionen für das Zielsystem spezialisiert und implementiert. Ein weiteres Backend mit dem Namen "Sm", das eine Software-Emulation der GPU Hardware darstellt, ist zur Zeit in Arbeit. Eine Unterstützung von Direct3D existiert bisher noch nicht.

Die Möglichkeiten von Sh beschränken sich dabei nicht auf das Erstellen von Vertex- und Fragment-Shadern für die Programmierung grafischer Effekte. Die Entwickler wollten mit dem Toolkit auch eine einfach zu nutzende Schnittstelle für die Programmierung von Streaming Programmen (Näheres unter Kapitel 2.4. Sh für GPGPU) schaffen. Daher besitzt Sh für diesen Zweck einen speziellen Modus, um den Grafikprozessor im Rahmen der GPGPU nutzbar zu machen.

Um im Sinne der Metaprogrammierung alle Schnittstellen möglichst allgemein zu halten und dadurch später verschiedene Zielsysteme nutzen zu können, ist eine starke Abstraktion und Automatisierung notwendig. Dazu gehört zunächst die Deklaration des Shader Codes, der in C++ Syntax geschrieben und vom Sh Toolkit in den Code des gewählten Backends, z.B. GLSL, übersetzt wird. Darüber hinaus ist die gesamte Verknüpfung des Shaders mit der Grafikkarte, also das Hochladen, das Compilieren, Linken und das Aktivieren des Programms, automatisiert. Dies schließt auch die Anbindung der Shadervariablen mit ein, deren Deklaration und Aktualisierung von Sh übernommen wird.

2.2.1. Deklaration und Übersetzung

In den meisten Shader Hochsprachen werden die Shader Programme separat vom Hauptprogramm entwickelt und in Textdateien gespeichert. Vor der Nutzung müssen diese Dateien im Programm geladen und dann mit einem Programmobjekt verknüpft werden. Sh Shader können an jeder Stelle direkt im Code des Hauptprogramms deklariert und verwendet werden. Mit den Makros *SH_BEGIN_PROGRAM* und *SH_END* wird der Anfang und das Ende eines Shaders festgelegt, angelehnt an die glBegin() und glEnd() Befehlsstruktur in OpenGL. Der eigentliche Code steht zwischen diesen beiden Anweisungen. Kenntnisse in einer bestimmten Shader-Hochsprache sind nicht notwendig - die Shader werden in Sh geschrieben, dessen Syntax in großen Teilen der C++ Syntax entspricht. Alle Anweisungen in einem solchen Sh-Block werden, ähnlich wie die Befehle in einer OpenGL Display Liste[7], nicht unmittelbar ausgeführt, sondern in einem Objekt vom Typ ShProgram gespeichert, das später aufgerufen werden kann.

```
ShProgram prg = SH_BEGIN_PROGRAM("gpu:fragment")
{
    ShInputColor3f inCol;
    ShOutputColor3f outcol;
    outCol = inCol;
} SH_END;
```

Listing 2.4: Deklaration und Zuweisung eines simplen Fragment Shaders in Sh

Als Parameter wird dem *SH_BEGIN_PROGRAM* Makro die Art des Programms angegeben. Im Falle eines Fragment Shaders wäre dies der Ausdruck *"gpu:fragment"*. Sh definiert zur Erleichterung bereits Konstanten für Erstellung von Vertex- und Fragment Shader Programmen vor: *SH_BEGIN_VERTEX_PROGRAM* und *SH_BEGIN_FRAGMENT_PROGRAM*. Diese Makros benötigen keine weiteren Parameter.

7 Display Liste: speichert eine Reihe OpenGL Anweisungen in kompilierter Form für eine spätere Ausführung[4]

Sh übersetzt den Code anschliessend automatisch im jeweilig gewähltem Backend. Die einmal in eine Hochsprache übersetzten Programme können natürlich auch ausgegeben werden und direkt oder nach manuellen Optimierungen in einem anderen Programm unter der entsprechenden API weiterverwendet werden.

Im einzelnen ist der Ablauf wie folgt: Zunächst wird der Sh Code vom C++ Compiler regulär kompiliert. Dadurch wird dem Sh Parser viel Arbeit abgenommen, da bereits der Compiler das Parsen arithmetischer Ausdrücke übernimmt. Beim Ausführen der kompilierten Datei werden die übersetzen Anweisungen jedoch nicht ausgeführt, sondern – ausgelöst durch das zuvor im Code gesetzte Makro – von Sh abgefangen, eingesammelt und in eine Assembler-ähnliche Zwischenrepräsentation übersetzt. Diese "*Intermediate Representation*" wird dann an das gewählte Backend zur Übersetzung in die Zielsprache weitergegeben.

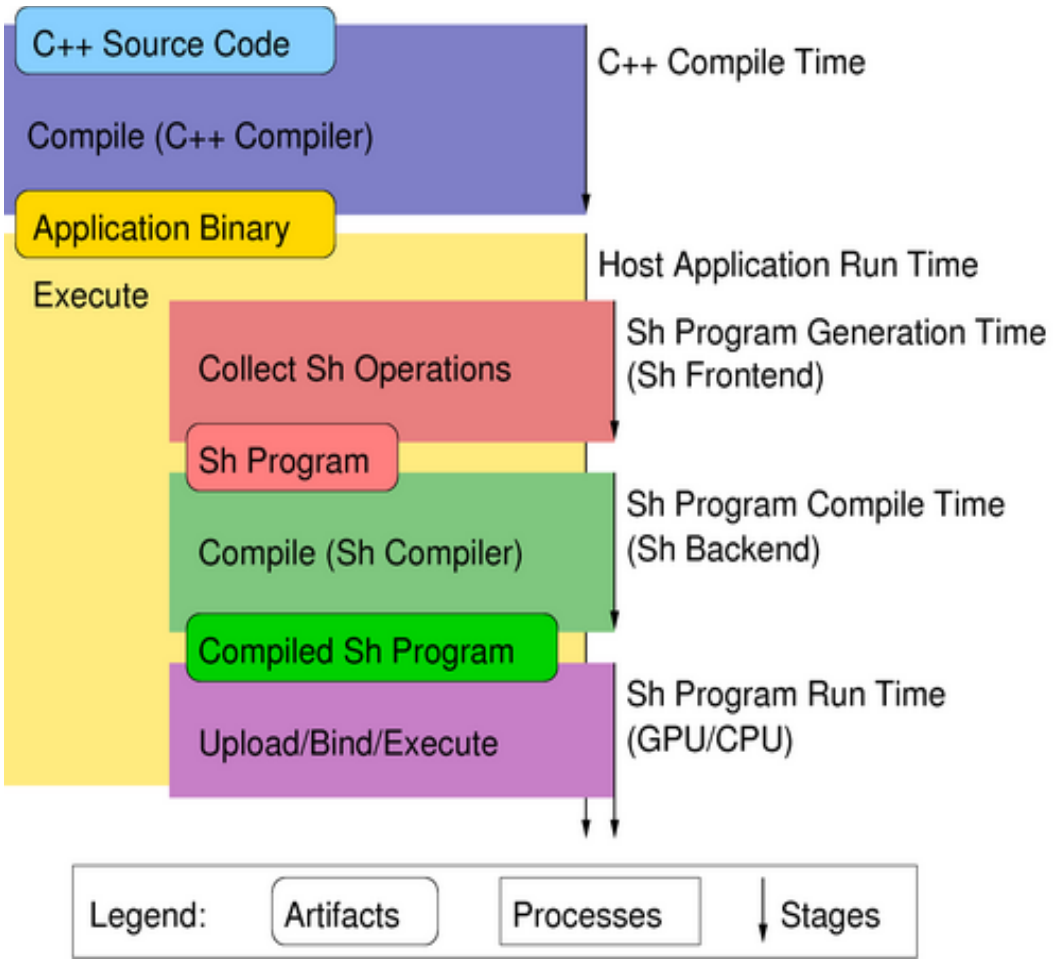

Abbildung 3: Ablauf der Erzeugung eines Sh Programms

Neben der vertrauten Syntax bietet dieses Prinzip der Shaderdeklaration innerhalb des Programmcodes den nicht zu unterschätzenden Vorteil, dass Tipp- und auch Syntaxfehler bereits zur Kompilierzeit erkannt und in der bevorzugten Entwicklungsumgebung angezeigt werden können.

```
// Auswahl des GLSL Backends
ShSetBackend("glsl");

// Shader Deklaration
ShProgram prg = SH_BEGIN_FRAGMENT_PROGRAM
{
    ShInputColor3f inCol;
    ShOutputColor3f outCol;
    outCol = inCol;
} SH_END;

// Ausgabe des GLSL Codes auf die Konsole
prg.node()->code()->print(std::cout);
```

Listing 2.5: Backendwahl, Shader-Deklaration und Ausgabe des Codes in GLSL

2.2.2. Anbindung

Da das Kompilieren, Linken und Aktivieren eines Shaders Programms je nach der genutzten Plattform sehr unterschiedlich verläuft, abstrahiert Sh auch diesen Teil vollständig und versteckt die einzelnen Schritte vor dem Anwender. Da der eigentliche, spezialisierte Code der entsprechenden Sprache bereits von Sh erzeugt wurde, benötigt der Anwender hier keine Fehlerrückmeldung vom jeweiligen System, da sich das Toolkit bereits um fehlerfreien Code kümmert. Laut Entwickler Michael McCool ist bei dieser Abstraktion theoretisch auch ein Direct3D/HLSL Backend denkbar. Wird kein Backend ausgewählt, sucht sich Sh automatisch ein passendes und vom System unterstütztes Backend heraus.

Ist ein Shader einem ShProgram zugewiesen, so kann dieses jederzeit beliebig mit dem Befehl *shBind()* aktiviert und mit *shUnbind()* wieder deaktiviert werden. Eine Unterscheidung zwischen Vertex und Fragment Shader macht Sh hierbei nicht, da bereits bei der Deklaration der Shader festgelegt wurde, um was für einen Typ es sich jeweils handelt. Dies erleichtert es, Vertex und Fragment Shader vollkommen unabhängig voneinander einzusetzen.

```
// Vertex und Fragment Shader aktivieren
shBind(fragShader);
shBind(vertShader);

draw_Stuff_1();

// Fragment Shader deaktivieren
shUnbind(fragShader);

draw_Stuff_2();

// alle Shader deaktivieren
shUnbind();

draw_Stuff_3();
```

Listing 2.6: Beispiel zur Nutzung von shBind() und shUnbind()

2.2.3. Attribute, Parameter und Texturen

In Sh gibt es grundsätzlich drei Arten von Variablen, die in einem Programm genutzt werden können: semantische, uniforme und temporäre Variablentypen. Ihre Benutzung ist weitestgehend intuitiv und erfordert nur wenig Vorwissen.

2.2.3.1. semantische Variablen

Die semantischen Variablen sind nur für die Verwendung in Vertex und Fragment Shadern von Bedeutung und stehen nicht in Streaming Programmen zur Verfügung. Sie repräsentieren die Eingabe- und Ausgabedaten der Vertices. Sie entsprechen in etwa *varying variables*[8], oder auch den Vertexattributen[9]. Dazu zählen unter anderem Farbe und Position des Pixels, bzw. des Vertices. Sie werden von Sh automatisch mit entsprechenden Registern verknüpft.

Die semantischen Variablen werden in der Regel zu Beginn eines Shader Programms aufgelistet, es ist aber auch möglich, sie erst später im Code zu deklarieren, bevor sie eingesetzt werden. Der Namenskonvention (siehe hierzu auch Kapitel 2.1 Variablentypen) folgend beginnen sie mit dem Präfix "Sh" und enden mit einer Kombination aus Typ und Tupelgröße. Als Mittelstück erhalten sie den Bezeichner "Input", "Output" oder "InOut", der festlegt, ob diese Variable ausschließlich als Eingabe, als Ausgabe oder für beides genutzt werden soll. Im letzteren Fall werden die Daten automatisch vom entsprechenden Eingaberegister an das entsprechende Ausgaberegister durchgereicht, ohne dass eine manuelle Zuweisung notwendig ist.

ShInputNormal3f	Eingabe-Normale des Vertices
ShInputPosition4f	Eingabe-Position des Vertices
ShInputColor3f	Eingabe-Farbe
ShInputTexCoord2f	Eingabe-Texturkoordinaten
ShOutputPosition4f	Ausgabe-Position
ShOutputColor3f	Ausgabe-Farbe
ShOutputTexCoord2f	Ausgabe-Texturkoordinaten

Listing 2.7: Liste der semantischen Variablen für den Vertex Shader

8 Varying Variables: GLSL Bezeichnung für Variablen, die vom Vertex Shader interpoliert an den Fragment Shader gereicht werden[5]

9 Vertexattribute: nur lesbare Werte, die eine Verbindung zwischen dem Shader und der OpenGL-VertexAPI darstellen und nur in einem Vertex Shader nutzbar sind [5]

```
// Shader Deklaration 1
ShProgram frag_1 = SH_BEGIN_FRAGMENT_PROGRAM
{
    ShInputColor3f inCol;
    ShOutputColor3f outcol;
    outCol = inCol;
} SH_END;

// Shader Deklaration 2
ShProgram frag_2 = SH_BEGIN_FRAGMENT_PROGRAM
{
    ShInOutColor3f col;
} SH_END;
```

Listing 2.8: die Shader frag_1 und frag_2 haben eine identische Funktion

Alle semantischen Ausgabevariablen des Vertex Shaders werden interpoliert und können im Fragment Shader weiter verwendet werden. Neben den vorgegebenen Werten, wie z.B. Farbe oder Position kann die Liste der Werte durch eigene Variablen leicht erweitert werden und so die Interpolation zwischen den Vertices im Fragment Shader ausgenutzt werden. Zusätzliche Variablen, die als semantische Variablen deklariert werden, also "Input" oder "Output" im Mittelteil haben (siehe hierzu auch Kapitel 2.1 Variablentypen), werden über die vorhandenen Multitexturkoordinatenregister weiter gegeben. Die Reihenfolge der Deklaration bestimmt dabei die Nummer des Texturkoordinatenregisters. Dadurch können Werte wie z.B. eine interpolierte Normale leicht weitergegeben werden.

```
ShProgram vert = SH_BEGIN_VERTEX_PROGRAM
{
    ShInOutColor3f col;
    ShInputNormal3f inNorm;
    ShOutputVector3f outNorm;
    outNorm = inNorm;
} SH_END;

ShProgram frag = SH_BEGIN_FRAGMENT_PROGRAM
{
    ShInOutColor3f col;
    ShInputVector3f norm;
} SH_END;
```

Listing 2.9: der Fragment Shader liest vom Vertex Shader interpolierte Werte ein.

Im obigen Beispiel hätte die Zeile "`ShInOutNormal3f norm;`" im Vertex Shader zu einem identischen Ergebnis geführt, obwohl es im Vertex Shader eigentlich kein eigenes Ausgaberegister für die Normale gibt. Sh wählt in einem solchen Fall für die Ausgabe automatisch eines der Texturkoordinatenregister.

2.2.3.2. uniforme Variablen

Uniforme Variablen sind Werte, die im Hauptprogramm zugewiesen und vor der Ausführung zum Shader hochgeladen werden. Dabei kann es sich z.B. um zusätzliche Lichtparameter oder um

Texturen handeln, oder um alle Arten von Koeffizienten und sonstigen Steuerwerten, die außerhalb des Shaders gesetzt werden müssen. Die Werte dieser Variablen sind für jeden Aufruf des Shaders innerhalb eines Durchlaufs konstant.

Bei Sh müssen solche Variablen nicht, wie in anderen Hochsprachen üblich, gesondert im Shader deklariert werden. Jede im Hauptprogramm an der Stelle der Shader Deklaration verfügbare Variable kann direkt im Shader Programm verwendet werden. Einzige Voraussetzung ist, dass es sich um eine Sh-Variable handelt. Während in herkömmlichen Programmen einzeln für jede Variable über sogenannten "glue code" der Transfer der Daten zur Grafikkarte initiiert werden muss, kümmert sich Sh automatisch um die Verwaltung aller uniformen Variablen. Der Benutzer muss nicht für jede im Shader genutzte Variable zusätzlichen Code im Hauptprogramm einfügen, der diese regelmäßig überprüft und hochlädt. Alle verwendeten, das heißt derzeit gebundenen, uniformen Variablen werden, sofern notwendig, automatisch aktualisiert, wenn die Funktion *shUpdate()* aufgerufen wird. Dies macht es besonders leicht die Anzahl der genutzten Parameter zu verändern, was bei der Prototypen-Entwicklung sehr komfortabel und zeitsparend ist.

In der in dieser Arbeit zum Einsatz kommenden Version von Sh ist der Aufruf von *shUpdate()* tatsächlich nicht notwendig, da die Implementation eine uniforme Variable automatisch auf der Grafikkarte aktualisiert, sobald ihr Wert im Hauptprogramm geändert wird. In zukünftigen Versionen soll diese Funktionalität jedoch aus Performanzgründen eingestellt werden. Wird die Sh-eigene Texturverwaltung genutzt (siehe 2.2.3.4. Texturen), so muss *shUpdate()* bei einer Änderung aufgerufen werden, da diese einen Sonderfall darstellen und nicht vorzeitig aktualisiert werden.

```
// Deklaration einer Variable
ShAttrib1f param;

void declareShaders()
{
   // Deklaration eines Shaders
   ShProgram frag = SH_BEGIN_FRAGMENT_PROGRAM
   {
       ShInputColor3f inCol;
       ShOutputColor3f outcol;
       outCol = inCol * param;  // Zugriff auf die Variable
   } SH_END;
}
```

Listing 2.10: Beispiel für den Zugriff auf eine uniforme Variable in einem Shader

2.2.3.3. temporäre Variablen

Temporäre Variablen sind nur lokal im Shader Programm verfügbare Variablen. Da sich Sh beinahe vollständig an die C++ Syntax hält, erfolgt die Deklaration dieser Variablen identisch mit der Deklaration temporärer Variablen in einer Funktion.

2.2.3.4. Texturen

Eine besondere Form uniformer Variablen sind Texturen. Das Sh Toolkit bietet als primäre Lösung eine eigenes Texturformat an, um Plattformunabhängigkeit zu gewährleisten. Dabei stellt Sh eine Funktion zum Einladen von Bitmaps im PNG Format zur Verfügung. Aus Bildern können Texturen vom Typ *ShTexture* erzeugt werden, auf die in einem Shader wie auf jede übliche uniforme

Variable direkt zugegriffen werden kann. Als Datenquelle können Bilder dienen, die mittels des Toolkits oder auch anderweitig eingeladen werden. Auch das Erstellen von Mipmaps[10], das Setzen von Filtern und der Clamp[11]-Einstellungen wird von *ShTextures* unterstützt.

```
// Deklaration einer Variable
ShImage image;
// Einladen eines Bildes
ShUtil::load_PNG(image, "monalisa.png");

// Erzeugen einer Textur aus den Daten
ShTexture2D<ShColor3f> myTexture(image.width(), image.height());
myTexture.memory(image.memory());
```

Listing 2.11: Beispiel für das Einladen eines Bildes und das Erzeugen einer Textur

Die wichtigsten von Sh angebotenen Texturtypen sind *ShTexture1D*, *ShTexture2D, ShTexture3D* und *ShTextureRect*. Der Zugriff auf die Texturen im Shader kann auf mittels der zwei üblichen Operatoren geschehen, "[]" und "()", die jeweils ein Koordinatentupel in der Dimension des Bildes oder eine Dimension größer erwarten. Hat das Tupel eine Dimension mehr als die Textur, wir die zusätzliche Koordinate als homogene Komponente interpretiert, die beim Zugriff heraus dividiert wird. Der []-Operator skaliert die Koordinaten nicht, sondern erwartet Werte zwischen 0 und der tatsächlichen Breite, bzw. Höhe der Textur. Der ()-Operator hingegen arbeitet mit Koordinaten aus dem Intervall [0,1].

```
ShProgram frag = SH_BEGIN_FRAGMENT_PROGRAM
{
    ShInputColor3f inCol;
    ShInputTexCoord2f tex;
    ShOutputColor3f outCol;
    outCol = inCol * myTexture(tex);
} SH_END;
```

Listing 2.12: Texturzugriff innerhalb eines Shaders

Falls die Verwaltung der Texturen durch Sh ungewollt ist, z.B. wenn Sh in ein bestehendes Projekt eingebaut werden soll, oder wenn ein Shader mit verschiedenen Texturen arbeiten können soll, so muss der Texturzugriff mit einem kleinen Umweg direkt mit einem Texturen-Register verknüpft werden. Dies funktioniert ausschließlich unter OpenGL. Näheres dazu im Tutorial-Teil unter 3.6. Sh in bestehende Projekte einbauen. Weitergehende Informationen zu Texturen, Arrays und ihrer Anwendung können in "Metaprogramming GPUs with Sh"[3] nachgelesen werden.

10 MipMapping: eine Technik, die die Darstellungsqualität von Texturen verbessert, wenn diese kleiner als in der ursprünglichen Größe dargestellt werden – spart oft auch zusätzliche Rechenzeit[6]

11 Clamping-Einstellungen: definieren, wie auf eine Textur zugegriffen wird, falls die Texturkoordinaten ausserhalb der Range liegen

2.2.4. Besonderheiten

2.2.4.1. OpenGL States

In der OpenGL Hochsprache GLSL ist es möglich, direkt auf OpenGL States[12] wie die Lichtparameter oder die Transformationsmatrizen zuzugreifen. GLSL bietet sogar einen Befehl für den Vertex Shader an, der die Vertex-Transformationen entsprechend der Standard-Pipeline übernimmt. Da diese Funktionalitäten nicht in allen Hochsprachen vorhanden sind, bietet Sh keine allgemeine Abstraktion für diese an. Dies bedeutet, dass sich das Hauptprogramm darum kümmern muss, die entsprechenden Matrizen auszulesen und den Shadern als uniforme Variablen zur Verfügung zu stellen. Da dies allerdings mit einigem Aufwand und Performance-Einbußen einhergeht, bietet Sh eine Möglichkeit, OpenGL States manuell zu verknüpfen.

Dabei handelt es sich um Meta-Informationen, die an eine uniforme Variable angehängt werden können, die bewirken, dass diese bei der Übersetzung in den Zielcode in einer vorgegebenen, selbst definierten Text übersetzt wird. So kann eine Variable vom Typ *ShMatrix4x4f* mittels Meta-Informationen dazu gezwungen werden, immer in den Ausdruck "*gl_ModelViewProjectionMatrix*" übersetzt zu werden, was dem State in GLSL entspricht. Diese Variable kann dann ohne jede weitere Initialisierung genutzt werden. Wird jedoch ein anderes Backend gewählt, ist zu beachten, dass die erzwungenen Übersetzungen dieser Variablen ebenfalls manuell geändert werden müssen. Für das ARB Backend wäre der entsprechende Ausdruck z.B. "*state.matrix.mvp*".

```
// Deklaration einer Matrix
ShMatrix4x4f m_MoViewProj;

void setupShaders()
{
   // Verknüpfung mit einem OpenGL State
   m_MoViewProj.meta("opengl:readonly", "true");
   m_MoViewProj.meta("opengl:state", "state.matrix.mvp");

   // Nutzung der Matrix in einem Vertex Programm
   ShProgram vertSH = SH_BEGIN_VERTEX_PROGRAM
   {
       ShInOutPosition4f pos;
       pos = m_MoViewProj * pos;
   } SH_END;
}
```

Listing 2.13: Verknüpfung einer uniformen Variable mit einem OpenGL State

2.2.4.2. Kontrollkonstrukte

Da der C++ Compiler alle Anweisungsblöcke vorkompiliert, werden auch die Befehle zur Flußkontrolle, wie if-Anweisungen und Schleifen, übersetzt. Da der Sh Compiler diese dann nicht zur Bearbeitung bekommen würde, kann Sh in diesem Bereich nicht vollständig auf Abweichungen von der C++ Syntax verzichten.

Sh löst dieses Problem durch die Definition von Makros, die die entsprechenden C++-Anweisungen ersetzen. Diese Makros entsprechen in Namen und Verhalten ihren C++ Gegenstücken. Wird

12 Open GL State: OpenGL ist nach dem Prinzip eines Zustandsautomaten (state machine) aufgebaut; als States werden z.B. die aktuelle Zeichenfarbe oder auch die aktuellen Transofrmationsmatrizen bezeichnet

innerhalb eines Sh Programms dennoch versehentlich das C++ Konstrukt verwendet, so wird ein Compiler Fehler erzeugt.

Die folgende Tabelle zeigt die C++ Konstrukte und ihre korrespondierenden Sh Makros auf. Eine *switch*-Anweisung stellt Sh nicht zur Verfügung. Die geschweiften Klammern sind bei den Sh Makros optional und dienen lediglich der besseren Lesbarkeit.

`if(Bedingung_1)`
`{`
`  ...`
`} else if (Bedingung_2)`
`{`
`  ...`
`} else`
`{`
`  ...`
`}`
	`SH_IF(Bedingung_1)`
	`{`
	`  ...`
	`} SH_ELSEIF(Bedingung_2)`
	`{`
	`  ...`
	`} SH_ELSE`
	`{`
	`  ...`
	`} SH_ENDIF;`

`While(Bedingung)`
`{`
`  ...`
`}`
	`SH_WHILE(Bedingung)`
	`{`
	`  ...`
	`} SH_ENDWHILE;`

`do`
`{`
`  ...`
`} until(Bedingung);`
	`SH_DO`
	`{`
	
	`} SH_UNTIL(Bedingung);`

`for(int i = 0; i < 10; i++)`
`{`
`  ...`
`}`
	`ShAttrib1i i;`
	`SH_FOR(i = 0, i < 10, i++)`
	`{`
	`  ...`
	`} SH_ENDFOR;`

Listing 2.14: Kontrollkonstrukte in C++ und die korrespondierenden Sh Makros

Auch die Anweisungen *break* und *continue* werden von Sh mittels Makros unterstützt. Im Gegensatz zu ihren C++ Entsprechungen bekommen diese Makros die Bedingung, die bei ihrer Ausführung erfüllt sein muss, direkt übergeben: *SH_BREAK(Bedingung)* und *SH_CONTINUE(Bedingung)*. Ausführlichere Erklärungen zur Implementierung der Kontrollkonstrukte finden sich in Michael McCools Buch[3].

2.2.4.3. Optimierung

Sh beinhaltet einen eigenen Optimierer für seine Programme. Dieser soll auf verschiedene Arten den Code effizienter machen, indem beispielsweise Variablenzuweisungen aus Schleifen heraus gezogen und davor gesetzt werden. Außerdem sollen redundante und überflüssige Instruktionen entfernt werden. Dieser Teil von Sh steckt allerdings noch in den Kinderschuhen und hat viele noch nicht implementierte Anteile. In besonders ungünstigen Fällen kann die Optimierung sogar zu zusätzlichen Rechenoperationen führen.

Die Optimierung der Programm findet automatisch direkt nach der Definition eines Shaders statt. Beim Verbinden zweier Shader mittels Shader Algebra (siehe Kapitel 2.5. Shader Algebra) wird

erneut eine Optimierung durchgeführt. Es ist keine explizite Anweisung nötig und die Optimierung kann auch nicht nachträglich verändert werden. Der Grad der Optimierung kann vom Benutzer global angepasst werden. Sh stellt hierzu die Funktion

void ShContext::current()->optimization(int level)

zur Verfügung. Ein Level von 0 kann gewählt werden, um jegliche Optimierung vollständig zu verhindern. Der Standardwert für den Grad der Optimierung ist 2. Besonders relevant ist der Optimierer im Zusammenhang mit der Shader Algebra, siehe auch Kapitel 2.5. Shader Algebra.

2.2.4.4. automatische Codegenerierung

Da Sh zur Metaprogrammierung sehr unterschiedlicher Hardware eingesetzt werden können soll, muss das Toolkit die unterschiedlichen Fähigkeiten dieser Hardware erkennen und gegebenenfalls besondere Eigenschaften nutzen, oder fehlende softwareseitig ausgleichen. Stellt eine Hardware eine Funktion, wie z.B. die Trilineare Interpolation von Texturen, nicht zur Verfügung, generiert Sh in diesem Fall automatisch zusätzlichen Code, um diese Funktion nachzubilden.

Zum Beispiel wird auf den meisten Grafikkarten der reguläre Texturzugriff nativ nur mit Koordinaten aus dem Intervall [0,1] unterstützt, mit Ausnahme der rechteckigen Texturtypen, die in der Regel nativ mit [0, Breite] x [0, Höhe] indiziert werden. Sh stellt jedoch für jeden Texturtyp beide Indizierungsarten zur Verfügung (siehe Kapitel 2.2.3.4. Texturen) und fügt eventuell notwendigen Code zur Skalierung der Texturkoordinaten automatisch hinzu. Dadurch kann der Code eines Programms ohne das Wissen des Anwenders wachsen. Weitere Beispiele umfassen trigonometrische Funktionen, wie z.B. die Sinusfunktion, die in OpenGL ARB Vertex Programmen nicht zur Verfügung stehen und von Sh durch polynomische oder rationale Funktionen approximiert werden.

2.3. Sh für Graphik

Mit Sh können Vertex- und Fragment-Shader Programme geschrieben und eingesetzt werden. Der theoretische Aufbau und Einsatz selbst entwickelter Shader wurde bereits in den vorangehenden Abschnitten erläutert. Sh bietet jedoch auch Funktionen zum automatisierten Erstellen von Shadern an, die bestimmte Aufgaben übernehmen, wie z.B. die Geometrietransformation oder Beleuchtung. Diese Shader können direkt eingesetzt werden, oder mittels Shader Algebra (siehe Kapitel 2.5. Shader Algebra) kombiniert und erweitert werden.

Die Funktion *shVsh()* erstellt z.B. einen Vertex Shader, der die Geometrie transformiert und die Tangenten berechnet, sowie die Texturkoordinaten und die Normale an den Fragment Shader ausgibt. Erweiterungen dieser Funktion sind in Planung, um Lichtvektorberechnungen mit aufzunehmen. Auch für Fragment Programme bietet Sh Funktionen an, um z.B. Blinn-Phong und andere Beleuchtungsmodelle und auch Bump Mapping zu berechnen. Auf die einzelnen Funktionen werde ich hier jedoch nicht weiter eingehen, da dies den Rahmen dieser Arbeit sprengen würde. Ein detaillierte Erklärung findet sich im Referenzteil des Buches[3], der auch online im Wiki der offiziellen Sh Website[7] verfügbar ist.

2.4. Sh für GPGPU

Shader werden auf modernen Grafikkarten nicht für jedes Pixel oder Vertex nacheinander aufgerufen. Vielmehr verfügen ihre Prozessoren über die Fähigkeit, das gleiche Programm parallel

auf mehreren Daten auszuführen. Ein Grafikchip mit 8 Pixel-Pipelines beispielsweise kann theoretisch in der Zeit, die für einen einzigen Durchlauf des Fragment-Shaders benötigt wird, 8 Pixel bearbeiten. Die neuesten GPUs verfügen über 16, 24 oder sogar 32 Pixel-Pipelines und die Anzahl wächst mit jeder neuen Grafikkartengeneration.

Diese Fähigkeit, Instruktionen auf mehreren Daten parallel ausführen zu können, macht die GPU als Co-Prozessor für rechenintensive Aufgaben interessant. GPGPU steht für *General Purpose Graphics Processing Unit*, also für den Einsatz des Prozessors der Grafikkarte für allgemeine Aufgaben, jenseits der Computergrafik.

Ein *Streaming Programm* ist ein solches Programm, das auf einem großen Datensatz ausgeführt wird, wobei das Programm dabei für jedes Element durchgeführt werden muss. Ein klassisches Einsatzgebiet für ein solches Programm sind Dreiecks-Schnittpunkttests in einem Raytracer[13], bei dem die gleichen Berechnungen mit einer großen Anzahl von Kamerastrahlen durchgeführt werden müssen. Umgesetzt werden solche Programme in der Regel mittels eines erweiterten Befehlssatzes der CPUs für SIMD[14] wie z.B. SSE[15], oder mittels Hyperthreading[16] oder Mehrprozessorsystemen.

Da es sich bei der GPU ebenfalls um einen SIMD Prozessor handelt, ist die Idee des Einsatzes von Shadern für Streaming Programme naheliegend. Durch die große Anzahl der Pipelines ist so unter gewissen Voraussetzungen eine enorme Geschwindigkeitssteigerung möglich. Die Umsetzung erfordert jedoch ein wenig Aufwand, da es nicht möglich ist, ein Programm direkt auf dem Grafikprozessor ausführen zu lassen. Alle Daten müssen zunächst in eine Textur oder uniforme Variablen gespeichert und ein entsprechender Shader muss in einer geeigneten Hochsprache geschrieben und verknüpft werden. Um den Shader dann tatsächlich auszuführen, ist zudem das Zeichnen von einfacher Projektions-Geometrie samt der damit verbunden Aufrufe einer Grafik API notwendig. Abschliessend müssen die Daten noch von der Grafikkarte ausgelesen werden.

Die Erschaffer von Sh zielten bei der Entwicklung darauf ab, mit dem Toolkit auch ein leicht zu bedienendes Werkzeug für *Streaming Shader* zu kreieren, das dem Nutzer so viel Arbeit wie möglich abnimmt. Sh kümmert sich dabei um die notwendige Aufbereitung der Daten als Texturen, sowie um sämtliche Kommunikation mit der Grafik API. Da selbst moderne Grafikkarten "branching", also datenabhängige Schleifen und Verzweigungen im Code, nicht unbedingt effizient unterstützen, erzeugt Sh gegebenenfalls mehrere Durchläufe, sogenanntes Multipassing, um die Ausführung des Streaming Programms möglichst effizient zu gestalten. Der Nutzer muss nur noch die notwendigen Daten zur Verfügung stellen und ein Programm in Sh schreiben, das auf diesen ausgeführt werden soll.

2.4.1. Channels und Streams

Um mit Sh ein Streaming Programm zu schreiben, müssen Sh zunächst die Eingabedaten, auf denen die Berechnung ausgeführt werden soll, zur Verfügung gestellt werden. Dies geschieht über einen Buffer. Ein zweiter Buffer wird für die Ausgabe benötigt, um die Ergebnisse der Berechnungen zu speichern.

Gespeichert werden die Eingabedaten unter Sh in sogenannten *ShChannels*. Ein *ShChannel* ist am ehesten zu vergleichen mit einem Array oder einem Vektor und er repräsentiert die Daten für genau

13 Raytracing: bilderzeugendes Verfahren, bei dem Sichtstrahlen in eine 3D Szene projiziert werden. Der Schnittpunkt zwischen den Kamerastrahlen und der vorliegenden Geometrie bestimmt das Aussehen eines Bildpunktes.

14 SIMD – Single Instruction Multiple Data; steht für die parallele Ausführung einer Anweisung auf mehreren Daten gleichzeitig

15 SSE – Streaming SIMD Extensions; Befehlssatzerweiterung der x86 Architektur von Intel, der die Ausführung von Tasks auf einem Prozessor durch Parallelisierung beschleunigen soll; weitere Versionen: SSE2 und SSE3[8]

16 Hyperthreading: von Intel entwickelte Technologie, die hardwareseitiges Multithreading mit nur einem Prozessor unterstützt, indem einem virtuellen Kern eigene Arbeitsregister zurVerfügung gestellt werden.

einen Eingabe- oder Ausgabeparameter. Wenn ein Streaming Programm also drei Eingabewerte und zwei Ausgabewerte hat, so werden insgesamt fünf *ShChannels* benötigt, wobei in dreien davon sinnvolle Eingabe-Daten stehen müssen.

Die Reihenfolge, in der die *ShChannel* beim Aufruf mit dem Streaming Shader verknüpft werden, bestimmt auch die Parameter, denen sie zugewiesen werden, d.h. der Reihenfolge, in der die Eingabe- und Ausgabeparameter im Shader definiert sind.

Mehrere *ShChannel* können jeweils mit dem Operator "&“ zu einem S*hStream* für die Eingabe oder die Ausgabe zusammengefasst werden, der dann direkt mit dem Streaming Programm aufgerufen werden kann. Die Reihenfolge, mit der die *ShChannel* mit dem *ShStream* verknüpft werden, entspricht der Reihenfolge, mit der sie beim Aufruf mit dem Programm verknüpft werden. Hat ein Stream nur einen einzigen Eingabe- oder Ausgabeparameter, ist es überflüssig, einen *ShStream* aus dem jeweiligen *ShChannel* zu erstellen. Genauere Erklärungen der einzelnen Befehle in Kapitel 3.2. Tutorial: erster Streaming Shader

```
float data1[] = { 1.0, 0.5, -0.5 };
float data2[] = { 1.0, 0.5, -0.5 };

// Erstellen des ersten ShChannels
ShHostMemoryPtr m1 = new ShHostMemory(sizeof(float) * 3, data1, SH_FLOAT);
ShChannel<ShAttrib3f> in1(m1, 1);

// Erstellen des zweiten ShChannels
ShHostMemoryPtr m2 = new ShHostMemory(sizeof(float) * 3, data2, SH_FLOAT);
ShChannel<ShAttrib3f> in2(m2, 1);

// die beiden Channel werden zu einem Stream zusammengefügt
ShStream inData = in2 & in1;
```

Listing 2.15: Erstellen zweier Eingabe-Channels und Verknüpfung zu einem *ShStream*;

Falls das Programm Ausgabewerte hat, so muss ein *ShChannel* für jeden Ausgabeparameter vorbereitet werden, in den die Ergebnisse geschrieben werden. Vor dem Zugriff auf die Ergebnisse muss der Speicher jedoch für den Zugriff gesichert werden. Dazu ist ein Synchronisationsbefehl notwendig, der sicherstellt, dass die Daten im Speicher vorhanden und aktuell sind. Listing 2.16 zeigt das Vorbereiten eines Ausgabe-Channels, das Aufrufen des Shaders und das anschliessende Synchronisieren des Speichers.

```
// Erstellung eines Ausgabe-Channels
float data[3];
ShHostMemoryPtr m3 = new ShHostMemory(sizeof(float) * 3, data, SH_FLOAT);
ShChannel<ShAttrib3f> outCh(m3, 1);

// Aufrufen eines Streaming-Shaders mit dem ShStream aus Listing 2.15
outCh = myShader(inData);

// Synchronisation des Speichers des Ausgabe-Channels
m3->hostStorage()->sync();
```

Listing 2.16: Ablauf beim Aufruf eines Streaming Shaders

2.4.2. Streaming Shader

Das Schreiben eines Streaming Shaders entspricht in etwa dem Schreiben einer gewöhnlichen C++ Funktion. Bei der Deklaration wird dem Sh-Makro als Parameter anstelle des Vertex oder Fragment Targets lediglich "*gpu:stream*" übergeben. Ausserdem ist in einem solchen Programm der Zugriff auf semantische Variablen der Grafik API nicht möglich. Ansonsten unterscheidet sich die Programmierung nicht von der eines üblichen Shaders in Sh.

Auch die Handhabung funktioniert ähnlich einer gewöhnlichen C++ Funktion, da Sh den Aufruf eines Streaming Shaders wie eine reguläre Funktion unterstützt. Auch andere Methoden zum Aufruf sind möglich. Ein Streaming Shader kann erst aufgerufen werden, wenn mit jedem seiner Eingabeparameter eine Datenquelle verknüpft wurde.

```
// gegeben seien:
ShChannel<ShAttrib3f> ch1;
ShChannel<ShAttrib3f> ch2;
ShChannel<ShAttrib3f> ch3;


// Aufruf eines Shaders mit nur einem Eingabeparameter
ch3 = myShader(ch1);
ch3 = myShader << ch1;


// Aufruf eines Shaders mit zwei Eingabeparametern
ch3 = myShader << (ch1 & ch2);
ch3 = myShader << ch1 << ch2;

ShStream stream = ch1 & ch2;
ch3 = myShader(stream);


// Aufruf eines Shaders mit einem Eingabe- und zwei Ausgabeparametern
(ch2 & ch3) = myShader(ch1);

ShStream stream = ch2 & ch3;
stream = myShader(ch1);
```

Listing 2.17: Beispiele für die verschiedenen Aufrufmethoden von Streaming Shadern

2.5. Shader Algebra

Einzigartig im Vergleich zu allen anderen Hochsprachen ist die von Sh eingeführte Shader Algebra. Hierbei handelt es sich um "Rechenregeln", nach denen Shader miteinander kombiniert werden können um neue, komplexere Shader zu erschaffen. So ist es beispielsweise möglich, einen Vertex-Shader, der die Geometrie transformiert und einen zweiten, der die Texturkoordinaten an den Fragment Shader durchreicht, zu einem Shader zusammen zu addieren, der beides tut.

Sh bietet für diesen Zweck zwei Operatoren für den Typ *ShProgram* an: den *Combine*- und den *Connect*-Operator, die beide dazu dienen Shader Programme zusammenzufügen, sich aber in der Art der Zusammenfügung unterscheiden.

2.5.1. Der Combine Operator

Der *Combine* Operator ist am ehesten mit der direkten Konkatenation zweier Shader zu vergleichen. Sein zugehöriger Operator ist "&". Nach seiner Definition werden alle Eingabeparameter-Definitionen zusammengefasst, ebenso alle Ausgabeparameter und alle Anweisungen aus beiden Programmen werden ausgeführt. Als Ergebnis entsteht ein Shader Programm, dass die gleiche Anzahl an Ein- und Ausgabeparameter wie die eingeflossenen Shader zusammen hat und das die gleiche Funktionalität erfüllt wie die einzelnen Programme.

Listing 2.19 zeigt den GLSL Code der beide einzelnen und des aus der Kombination resultierenden Vertex Shaders.

```
// Deklaration eines Vertex Programms, der die Farbe
// der Vertices durchreicht
ShProgram v1 = SH_BEGIN_VERTEX_PROGRAM
{
       ShInOutColor3f col;
} SH_END;


// Deklaration eines Vertex Shaders, der die
// Texturkoordinaten durchreicht
ShProgram v2 = SH_BEGIN_VERTEX_PROGRAM
{
       ShInOutTexCoord2f tex;
} SH_END;

// Kombination der beiden Vertex Programme
ShProgram combo = v1 & v2;
```

Listing 2.18: Beispiel für den Einsatz des *Combine* Operators

```
// GLSL Code des ersten Shaders
void main()
{
    gl_TexCoord[0].xy = gl_MultiTexCoord0.xy;
}

// GLSL Code des zweiten Shaders
void main()
{
    gl_FrontColor.xyz = gl_Color.xyz;
}
// Kombination der beiden Vertex Programme
void main()
{
    gl_FrontColor.xyz = gl_Color.xyz;
    gl_TexCoord[0].xy = gl_MultiTexCoord0.xy;
}
```

Listing 2.19: der Code der obigen Shader ausgegeben in GLSL

Sinnvolle Anwendungen für den *Combine* Operator sind Vertex und Streaming Shader, da sich mit der Verbindung die Anzahl der Ausgabeparameter erhöht, was bei einem Fragment Shader nicht möglich ist.

2.5.2. Der Connect Operator

Der *Connect* Operator ist am ehesten mit einer Pipeline zu vergleichen. Sein Operator ist entsprechend "<<". Beim Verbinden zweier Shader mit dem *Connect* Operator werden die Ausgabeparameter des ersten Shaders an die Eingabeparameter des zweiten weitergereicht. So können die Programme die Ergebnisse anderer Programme verwenden. Ist die Anzahl der Ausgabeparameter des ersten Programms größer als die Anzahl der Eingabeparameter des zweiten, so werden die überflüssigen Parameter ignoriert. Ist die Anzahl geringer, entsteht ein Programm, das weitere Eingabewerte benötigt. Handelt es sich also zum Beispiel um einen Streaming Shader, so kann das Programm nicht ausgeführt werden, bis eine weitere Datenquelle mit ihm verknüpft wurde. Dies kann auch ein weiteres *ShProgram* sein, das mittels des *Connect* Operators verknüpft wird. Im Falle eines Fragment oder Vertex Shaders wird der entsprechende Parameter an eines der vorhandenen, noch nicht genutzten Register gebunden. Dabei können ungewollte Effekte entstehen, wenn in diesen Registern keine sinnvollen Daten stehen.

```
// Streaming Shaders zu Berechnung von sin x
ShProgram sin_X = SH_BEGIN_PROGRAM("stream")
{
    ShInputAttrib1f a;
    ShOutputAttrib1f b;
    b = sin(a);
} SH_END;

// Streaming Shaders zu Berechnung von 1/x
ShProgram rez_X = SH_BEGIN_PROGRAM("stream")
{
    ShInputAttrib1f a;
    ShOutputAttrib1f b;
    b = 1.0f / a;
} SH_END;


// Verbindung mit dem Connect Operator

// Ergebnis: 1 / sin(x)
ShProgram s1 = rez_X << sin_X;

// Ergebnis: sin(1/x)
ShProgram s2 = sin_X << rez_X;
```

Listing 2.20: Beispiel für den Einsatz des *Connect* Operators

Sinnvolles Anwendungsgebiet für den *Connect* Operator sind vornehmlich Streaming Shader, die diesen zur Schachtelung von Funktionen verwenden können, und Fragment Shader, die verschiedene Effekte nacheinander ausführen sollen.

2.5.3. Optimierung

Besonders relevant ist der Sh eigene Optimierer im Zusammenhang mit der Shader Algebra, insbesondere beim Einsatz des *Connect* Operators. Produziert das erste Programm mehr Ausgabeparameter, als das zweite Programm an Eingabeparametern benötigt, so werden diese bei der Verknüpfung zwar ignoriert, der Code zum Berechnen dieser Parameter bleibt aber erhalten. In diesem Fall springt der Optimierer ein und versucht, die nun überflüssig gewordenen Anweisungen zu kürzen.

Wird die Shader Algebra mit Vertex oder Fragment Shadern genutzt, werden zwei spezialisierte Formen des *Combine* und des *Connect* Operators wichtig: *namedConnect* und *namedCombine*. Bei der Übersetzung eines solchen Shaders werden alle Eingabe- und Ausgabeparameter an entsprechende Register gebunden, wobei kein Register doppelt genutzt werden kann. Werden also zwei Shader miteinander kombiniert, die in ihren Ein- oder Ausgabewerten ein gleiches Register nutzen, so wird im resultierenden Shader dieses Register nur von einem der Shader genutzt werden können, und die Parameter des zweiten werden mit anderen Registern verknüpft. In zwei Fragment Shadern, die beide auf die Texturkoordinaten im ersten Register zugreifen, führt dies dazu, dass der zweite Shader als Eingabeparameter nicht die Texturkoordinaten aus dem ersten Register, sondern die aus dem zweiten bekommt, also möglicherweise unbrauchbare Werte.

Eine Möglichkeit, einen bestimmten Eingabe- oder Ausgabeparameter zwingend mit einem bestimmten Register zu verknüpfen, gibt es in Sh nicht. Allerdings gibt es eine Möglichkeit, in vorherigen Shadern verwendete Variablen weiterzuverwenden. Dazu müssen die definierten Eingabevariablen zusätzlich in ihren Meta-Informationen einen Namen zugewiesen bekommen und die Shader dann mit den oben genannten modifizierten Operatoren zusammengefügt werden. Dabei werden Eingabeparameter des zweiten Shaders, die den gleichen Meta-Namen haben wie ein Eingabeparameter aus dem ersten Shader, nicht mit einem neuen, ungenutzten Register verknüpft, sondern stattdessen mit dem gleichen Register, mit dem die Variable im ersten Shader verknüpft wurde.

Für das Zuweisen eines Meta-Namens zu einer Variable gibt es verschiedene Möglichkeiten. Die Einfachste ist, diesen direkt bei der Deklaration anzugeben. Dazu gibt es in Sh das Makro *SH_NAMEDECL()*, dass einer Sh Variable einen Namen zuweist. Dieser kann frei gewählt werden. Wird kein expliziter Name angegeben, so wird unter Verwendung von *SH_DECL()* der Bezeichner der Variable selbst als Name gewählt.

```
ShProgram temp = SH_BEGIN_VERTEX_PROGRAM
{
    ShInputColor3f SH_DECL(col);
    ShInputTexCoord2f SH_NAMEDECL(texCoords, „tex0“);

    // auch Initialisierungen sind möglich
    ShInputVector1f SH_DECL(someVal) = 5.0f;
}
```

Listing 2.21: Beispiel für das Zuweisen eines Meta-Namens zu einer Variable

3. Tutorial Teil

Dieses Kapitel befasst sich mit der praktischen Anwendung des Sh Toolkits. Zum Zeitpunkt der Erstellung dieser Arbeit finden sich im Internet praktisch keinerlei Anleitungen in die Benutzung von Sh. Auch die offizielle Website[7] enthält nur wenige Code Beispiele, die darüber hinaus unzureichend kommentiert und teilweise nicht lauffähig sind. Lediglich das von Michael McCool veröffentlichte Buch[3] beinhaltet eine Einführung, geht dabei aber nicht auf Fragen und Details der Praxis ein.

Ein wichtiger Teil dieser Arbeit ist daher ein ausführliches Tutorial, in dem die Installation und Einrichtung von Sh erklärt wird, und auch vollständige Programme für Vertex, Fragment und Streaming Shader entwickelt werden. Alle Tutorials beziehen sich auf die Nutzung von Sh unter dem Windows XP Betriebssystem, als Entwicklungsumgebung kommt Microsoft Visual Studio .NET 2005 zum Einsatz.

Zum Verständnis der Tutorials und zum sinnvollen Arbeiten mit Sh ist Kenntnis der Standard-Pipeline notwendig, insbesondere die Funktionen von Vertex und Fragment Shader. In der auf aktuellen Grafikkarten noch vorhandenen Standard-Pipeline ist es die Aufgabe des Vertex Shaders, die Eckpunkte zu beleuchten, die Texturkoordinaten durchzureichen und die Geometrie zu transformieren. Die Daten, die der Vertex Shader pro Eckpunkt berechnet und wieder ausgibt, werden für jeden Bildpunkt, für den der Fragment Shader aufgerufen wird, interpoliert. Aufgabe des Fragment Shaders ist es die endgültige Farbe des Pixels zu bestimmen und auszugeben. Dazu muss ggf. mithilfe der Texturkoordinaten ein Texturzugriff erfolgen und die Beleuchtung vom Vertex Shader eingerechnet werden. Näheres zur Standard-Pipeline und Shadern findet sich z.B. unter [9].

3.1. Setup von Sh

Unter *http://libsh.sourceforge.net/* kann die aktuellste Version des Sh Toolkits heruntergeladen werden; dies ist zum Zeitpunkt der Erstellung dieser Arbeit Version 0.8.0rc1. Alle Tutorials in diesem Kapitel beziehen sich daher auf diese Version.

Für die Verwendung mit .NET 2003 muss der Quellcode heruntergeladen und selbst kompiliert werden – entsprechende Projektdateien für Visual Studio sind in dem Quellcode-Paket enthalten. Für .NET 2005 ist dies nicht notwendig, da die Bibliotheken bereits vorkompiliert zur Verfügung stehen. Benötigt wird hier nur die Datei:

libsh-0.8.0rc1-msvc8.zip

Nach dem Download wird die Datei „libsh-0.8.0rc1-msvc8.zip“ entpackt. Die im Verzeichnis „/vc8/include“ und „/vc8/lib befindlichen Daten müssen nun in die entsprechenden Verzeichnisse von .NET kopiert werden. Der Zielpfad ist in der Regel:

C:\Programme\Microsoft Visual Studio 8\VC\PlatformSDK

Dabei müssen die Dateien jeweils in das entsprechende Unterverzeichnis „/include“ und „/lib“ kopiert werden. Die dll-Dateien aus dem Hauptverzeichnis müssen zur Ausführung von Sh Programmen in das jeweilige Projektverzeichnis kopiert werden. Alternativ können diese Dateien

auch einmalig in den System32-Ordner von Windows kopiert werden. Diese Variante ist empfehlenswert, damit alle Projekte immer mit den gleichen Versionen arbeiten. Die folgenden DLLs werden benötigt und müssen kopiert werden:

libsh_debug.dll und libsh.dll
libshglsl.dll und libshglsl_debug.dll
libsharb.dll und libsharb_debug.dll

Vorweg müssen noch einige Einstellungen in der Entwicklungsumgebung vorgenommen werden. Ohne diese kommt es beim Kompilieren und Ausführen von Sh Programmen unter Windows zu Fehlern. Um dies nicht für jedes neue Projekt wiederholen zu müssen, empfiehlt es sich, ein leeres Sh-Projekt zu starten, alle Einstellungen vorzunehmen und dann als Basisprojekt für alle weiteren zu nutzen.

1. Erstellen eines neuen, leeren Win32-Projektes (für die Tutorials mit Konsole)
2. Eine main.cpp Datei erstellen und dem Projekt hinzufügen; die Datei darf leer sein.
   Die ist notwendig, um Visual Studio mitzuteilen, daß es sich um ein C/C++ Projekt handelt.
3. Menü „Project“ -> „Properties“ öffnen und „Configuration Properties“ aufklappen
4. „C/C++“ --> „Preprocessor“ im Feld „Preprocessor Definitions“ Folgendes eintragen:
   Debug: WIN32;_DEBUG;NOMINMAX;_USE_MATH_DEFINES
   Release: WIN32;NDEBUG;NOMINMAX;_USE_MATH_DEFINES
5. „C/C++“ --> „Code Generation“ im Feld „Runtime Library“ Folgendes eintragen:
   Debug: Multi-Threaded Debug DLL (/Mdd)
   Release: Multi-Threaded DLL (/MD)
6. „C/C++“ --> „Language“ im Feld „Enable Run-Time Type Info“ in Debug und in Release „Yes (/GR)“ auswählen.
7. „C/C++“ --> „Advanced“ im Feld „Disable Specific Warnings“ Folgendes eintragen:
   Debug und Release: 4003;4251;4244
8. Jetzt muss noch die Sh-Library eingebunden werden.
   Dazu unter „Linker“ --> „Input“ im Feld „Additional Dependencies“ folgende zusätzliche Einträge machen:
   Debug: libsh_debug.lib
   Release: libsh.lib

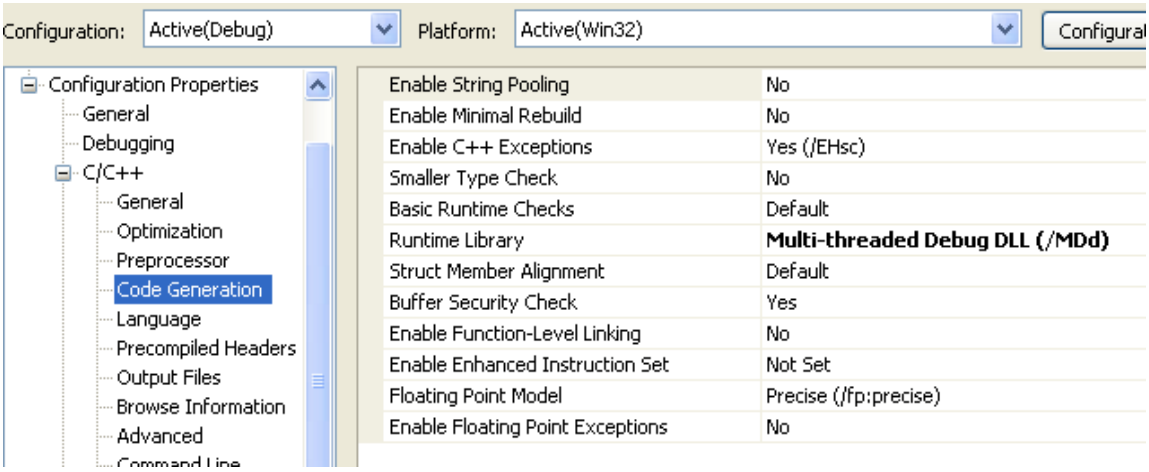

Abbildung 4: Projekt Eigenschaften

Abhängig von der Art des genutzten Projektes sind einige der aufgeführten Einstellungen bereits richtig voreingestellt. Es werden hier dennoch alle notwendigen aufgeführt, um Sh auch mit anderen Projekttypen als einem „Win32 Console Project“ nutzen zu können.

3.2. Tutorial: erster Streaming Shader

Dieses Tutorial dreht sich darum, einen ersten Streaming Shader zu programmieren. Es wird vorrausgesetzt, dass die Entwicklungsumgebung nach den in [3.1.] beschriebenen Schritten vorbereitet wurde.

In dem vorbereiteten Projekt erstellen wir zunächst in der „main.cpp“ Quelldatei eine leere main-Routine. Außerdem binden wir einen Header ein, der es uns erlaubt, später auch die Ergebnisse auf der Konsole auszugeben.

```
#include <iostream>
using namespace std;


void main(int argc, char* argv[])
{
}
```

Listing 3.1: main.cpp

Um Sh auch wirklich einzubinden, muss noch der Sh Header inkludiert und der Namespace ausgewiesen werden.

```
#include <iostream>
using namespace std;
#include <sh/sh.hpp>
using namespace SH;


void main(int argc, char* argv[])
{
     shInit();
     shSetBackend(“glsl“);
}
```

Listing 3.2: Sh Initialisierung

Der Befehl „shInit()“ in der main-Routine initialisiert Sh. Dies muss nur einmalig im Programm gemacht werden. Der nächste Befehl wählt das zu benutzende Backend aus.

Als nächstes definieren wir uns einen ziemlich simplen Streaming-Shader. Der Shader soll eine Eingabe- und eine Ausgabevariable haben und nichts weiter tun, als den eingehenden Wert zu verdoppeln und wieder heraus zu schreiben. Den Shader definieren wir direkt in der main-Routine, die dann folgendermaßen aussieht:

```
void main(int argc, char* argv[])
{
     shInit();
     shSetBackend("glsl");
     ShProgram prg = SH_BEGIN_PROGRAM("gpu:stream")
     {
          ShInputAttrib1f a;
          ShOutputAttrib1f b;
          b = a + a;
     } SH_END;
}
```

Listing 3.3: Streaming Shader Definition

Zunächst wird ein ShProgram Objekt "prg" erstellt, dem der Shader zugewiesen wird. Im Sh-Makro wird als Parameter *"gpu:stream"* übergeben, was Sh dazu bringt, den Shader als Streaming Shader zu verarbeiten. Die ersten beiden Zeilen im Shader deklarieren die Ein- und Ausgabeparameter, hier jeweils einen einfachen float-Wert. In der dritten Zeile wird dem Ausgabeparameter *b* dann sein Wert zugewiesen – wie zuvor schon erwähnt der zweifache Wert von *a*. Die geschweiften Klammen und das Semikolon hinter *SH_END* sind nicht zwingend erforderlich und dienen nur der besseren Lesbarkeit.

Nachdem nun der Shader definiert ist, ist es an der Zeit, sich um die Buffer für die Eingabe- und Ausgabe des Programms zu kümmern. Zuerst bereiten wir den *ShChannel* mit den Eingabedaten vor und füllen ihn mit den Zahlen von 0 bis 127.

```
// ein entsprechend großes Array erschaffen
int dataSize = 128;
float *data = new float[dataSize];

// ... und mit Daten füllen
for (int i = 0; i < dataSize; i++)
     data[i] = i;


// jetzt werden die Daten mit einem ShChannel verknüpft
ShHostMemoryPtr mem_in = new ShHostMemory(sizeof(float) * dataSize, data,
                                          SH_FLOAT);

ShChannel<ShAttrib1f> in(mem_in, dataSize);
```

Listing 3.4: Eingabe-Channel Vorbereitung

ShChannel ist als template implementiert, was bedeutet, bei seiner Erstellung muss man den Datentyp angeben, auf dem er arbeiten soll. In diesem Fall ist das der Datentyp float, bzw. als Sh-Typ ausgedrückt: *ShAttrib1f*. Dieser Datentyp muss mit dem Datentyp der Variable aus dem Shader, der die Daten zugeordnet werden müssen, übereinstimmen. Ist dies nicht der Fall, bricht das Programm mit einem wenig hilfreichen Laufzeitfehler ab.

Dem Konstruktor eines *ShChannels* wird dann eine Speicheradresse übergeben, an dem die Daten des Channels stehen, und die Anzahl der Elemente, in unserem Fall 128. Der Konstruktor akzeptiert allerdings nur Datenpointer im Sh-eigenen Format, also muss zuvor ein Speicher-Pointer im Sh Format aus dem Array erzeugt werden. Dies geschieht in der Zeile davor. Der Konstruktor *ShHostMemory* bekommt als Parameter die Gesamtgröße des allokierten Speicherbereiches, den Pointer auf den Speicherbereich und den Datentyp übergeben.

Als nächstes kommt der Buffer für die Ausgabedaten dran. Sh verlangt, dass der Speicher für die zu schreibenden Daten bereits allokiert ist, also müssen wir dafür sorgen. Auch hier muss wieder aus dem Array zunächst ein Sh Pointer erzeugt und dieser dann an einen Channel übergeben werden.

```
// Allokieren des Speichers für die Ergebnisse
float *outData = new float[dataSize];

// Verknüpfung mit einem ShChannel
ShHostMemoryPtr mem_out = new ShHostMemory(sizeof(float) * dataSize,
                                           outData, SH_FLOAT);
ShChannel<ShAttrib1f> out(mem_out, dataSize);
```

Listing 3.5: Ausgabe-Channel Vorbereitung

Nachdem die Buffer für die Ein- und Ausgabe nun vorbereitet sind, kann der Streaming Shader ausgeführt werden. Ein Streaming Shader wird immer dann ausgeführt, wenn er einem Ausgabebuffer zugewiesen wird und alle seine Eingabeparameter an eine Datenquelle geknüpft sind. Dies klingt komplizierter, als es ist.

```
out = prg << in;
```

Listing 3.6: Aufruf des Streaming Shaders

Der „<<“ Operator knüpft den *ShChannel* mit unseren Eingabedaten an den Shader. Da dieser nur einen Eingabeparameter hat, ist er nun bereit für die Ausführung. Durch den „=“ Operator wird der Shader nun ausgeführt und die Ergebnisse in den vorbereiteten Buffer geschrieben. Alternativ könnten wir auch „out = prg(in);“ schreiben.

Am Ende wollen wir die Daten, die uns die GPU berechnet hat, auch ausgeben. Dazu muss zunächst der Speicher synchronisiert werden, in den geschrieben wurde. Dann lassen wir einfach wieder eine Schleife über die Daten laufen und geben sie aus.

```
mem_out->hostStorage()->sync();

for (int i = 0; i < dataSize; i++)
     cout << outData[i] << " ";

cout << endl;
```

Listing 3.7: Speicher Synchronisation

Das Programm sollte nach erfolgreichem Ausführen die Zahlenreihe „0 2 4 6 8 10 ...“ ausgeben.

Anmerkung 1: Da für die Berechnung der Ergebnisse die GPU genutzt wird, müssen Grafik-Optionen wie Anti-Aliasing ausgeschaltet sein. Falls die Ergebnisse also falsch ausschauen, sollte in den Einstellungen des Grafikkartentreibers gesucht werden, ob diese Features deaktiviert sind. Sollte es daran nicht liegen, könnte eine Erhöhung der Anzahl der zu berechnenden Werte das Problem beheben. Im Test hat sich gezeigt, dass ab Größen von 32 Werten keine Probleme mehr auftreten.

Anmerkung 2: Das Programm stürzt nach erfolgreichem Ausführen und Anzeigen der Ergebnisse auf der Konsole ab. Dies liegt an einem Fehler in den Sh Bibliotheken, deren Destruktoren im Streaming nicht korrekt arbeiten. Dies ist im Issue Tracker auf der Sh Website bereits bekannt und wird voraussichtlich mit der nächsten Version von Sh behoben. Wer nicht solange warten möchte, dem kann ich meine eigene Lösung anbieten: den Source Code herunterladen, in der Datei „GlTextureName.cpp“ im Destruktor die Zeile „glDeleteTextures(1, &m_name);“ auskommentieren und dann das GLSL Backend neu kompilieren.

3.3. Tutorial: erster Fragment Shader

In diesem Tutorial wird ein einfacher Fragment Shader entwickelt, der die auszugebende Farbe manipuliert. Es wird vorausgesetzt, dass die Entwicklungsumgebung nach den in [3.1.] beschriebenen Schritten vorbereitet wurde. Außerdem werden die Glut[17] Biblithek sowie Kenntnisse in der Anwendung von OpenGL für die Grafikausgabe benötigt. Dieses Tutorial setzt auf 3.2. Tutorial: erster Streaming Shader auf, daher werden nicht alle Schritte erneut ausführlich erklärt.

Abbildung 5: der Fragment Shader manipuliert die Farbe des rechten Quadrates

In dem vorbereiteten Projekt erstellen wir zunächst eine „main.cpp“ Quelldatei und erstellen eine leere main-Routine. Dann wird Sh und Glut eingebunden und Glut und Sh initialisiert.

17 Glut: OpenGL Utility Toolkit – vereinfacht die Entwicklung von OpenGL Anwendungen

```
#include <GL/glut.h>
#include <sh/sh.hpp>
using namespace SH;


void main(int argc, char* argv[])
{
     glutInit(&argc, &argv);
     shInit();
     shSetBackend(“arb“);
}
```

Listing 3.8: main.cpp

Die Reihenfolge, in der Glut und Sh initialisiert werden, spielt keine Rolle. Damit der obige Code kompiliert, muss außerdem „glut32.lib“ ins Projekt eingebunden werden und ggf. die Datei „glut32.dll“ ins Projektverzeichnis kopiert werden. Als Backend wählen wir ARB. Der Grund dafür liegt darin, dass das folgende Vertex Shader Tutorial auf diesem aufsetzt und dort das ARB Backend benötigt wird.

Als nächstes erzeugen wir uns ein Glut Fenster und legen eine Display-Routine fest. Danach kann die *glutMainLoop()* gestartet werden.

```
GlutInitWindowSize(400, 400);
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
glutCreateWindow("Sh Shader Program");
glutDisplayFunc(render);
glutMainLoop();
```

Listing 3.9: Glut Initialisierung

Nun muss die Display-Routine „render()“ implementiert werden. Zunächst soll diese noch nicht viel machen, ausser das Fenster mit der Farbe Weiss zu löschen und die Buffer zu tauschen.

```
void render()
{
     glClearColor(1, 1, 1, 0);
     glClear(GL_COLOR_BUFFER_BIT);
     glutSwapBuffers();
}
```

Listing 3.10: rudimentäre render-Routine

Dies sollte bereits kompilieren und ohne Probleme ausführbar sein. Beim Starten wird ein kleines Fenster mit weißem Hintergrund erzeugt. Ansonsten sieht man nichts. Da man ohne Geometrie aber auch keine Shader sichtbar machen kann, schreiben wir als nächstes eine Routine, die uns ein gelbes Quadrat auf den Bildschirm zeichnet.

```
void renderSquare()
{
     glColor3f(1, 1, 0);
     glBegin(GL_QUADS);
          glVertex2f(-0,5f, 0);
          glVertex2f(-0.5f, -0.5f);
          glVertex2f(0, -0.5f);
          glVertex2f(0, 0);
     glEnd();
}
```

Listing 3.11: renderSquare-Routine

Dieses Quadrat lassen wir nun zweimal zeichnen. Später werden wir dann bei einem der beiden den Fragment Shader aktivieren und so den Unterschied besser sehen können. Die *render()* Routine wird also ausgebaut. Beim Starten des Programms werden nun zwei gleichermassen gelbe Quadrate gezeichnet.

```
void render()
{
     glClearColor3f(1, 1, 1, 0);
     glClear(GL_COLOR_BUFFER_BIT);
     glPushMatrix();

     // rendert das erste Quadrat
     glTranslatef(-0.25f, 0, 0);
     renderSquare();

     // rendert das zweite Quadrat
     glTranslatef(1, 0, 0);
     renderSquare()

     glPopMatrix();
     glutSwapBuffers();
}
```

Listing 3.12: erweiterte render-Routine

Jetzt ist es an der Zeit, den Fragment Shader zu definieren, der auf eines der Quadrate angewandt werden soll. Zunächst brauchen wir dafür eine Variable vom Typ *ShProgram*, die wir uns am Anfang der Datei einfach global definieren und *myFirstFragmentShader* nennen.

```
#include <GL/glut.h>
#include <sh/sh.hpp>
using namespace SH;

ShProgram myFirstFragmentShader;
```

Listing 3.13: Deklaration eines Fragment Shaders

Der Shader wird in der main-Routine definiert, direkt nach der Wahl des Backends. Dies ist nicht zwingend erforderlich, aber es teilt die Befehle in der Routine so praktischerweise nach Sh und Glut Befehlen auf. Der Shader selbst soll einfach nur die Farbe der Pixel halbieren.

```
ShSetBackend("arb");

myFirstFragmentShader = SH_BEGIN_FRAGMENT_PROGRAM
{
     ShInputColor3f inCol;
     ShOutputColor3f outCol;

     outCol = inCol * 0.5f;
} SH_END;
```

Listing 3.14: Definition des Fragment Shaders

Zu Beginn des Fragment Shaders legen wir die Eingabe- und die Ausgabevariablen fest, in diesem Fall die semantischen Variablen für das Farbregister, denn in das wollen wir ja schreiben. Die vom Vertex Shader eingehende Farbe wird nun einfach halbiert und in die Ausgangsvariable geschrieben.

Nachdem der Fragment Shader nun festgelegt ist, muss er noch aktiviert werden. Dazu müssen wir die *render()* Routine um die Befehle *shBind()* zum Aktivieren und *shUnbind()* zum Deaktivieren erweitern. Der Shader soll bei jedem Zeichnen nur für das zweite Quadrat eingeschaltet und danach wieder ausgeschaltet werden. Die *render()* Routine sieht dann so aus:

```
void render()
{
     glClearColor3f(1, 1, 1, 0);
     glClear(GL_COLOR_BUFFER_BIT);
     glPushMatrix();

     // rendert das erste Quadrat
     glTranslatef(-0.25f, 0, 0);
     renderSquare();

     // rendert das zweite Quadrat
     glTranslatef(1, 0, 0);

     shBind(myFirstFragmentShader);
     renderSquare()
     shUnbind();

     glPopMatrix();
     glutSwapBuffers();
}
```

Listing 3.15: finale render-Routine

Voilà, der erste Fragment Shader läuft. Das Fenster sollte nun zwei gelbe Quadrate anzeigen, von denen das rechte deutlich dunkler erscheint. Da wir nur einen einzigen aktiven Shader haben, ist es nicht nötig, bei *shUnbind()* einen Parameter anzugeben.

3.4. Tutorial: erster Vertex Shader

In diesem Tutorial soll ein simpler Vertex Shader ähnlich dem in der Standard-Pipeline geschrieben werden, der die Geometrie transformiert und die Farbe durchreicht. Damit man am Ende auch etwas sieht, soll er die Farbe zudem noch verändern. Es wird vorausgesetzt, dass die Entwicklungsumgebung nach den in [3.1.] beschriebenen Schritten vorbereitet wurde. Das Tutorial baut direkt auf dem in 3.3. Tutorial: erster Fragment Shader auf.

Wir benötigen zunächst eine weitere *ShProgram* Variable, die wir am Anfang der Datei deklarieren, wo auch der Fragment Shader bereits deklariert ist.

```
#include <GL/glut.h>
#include <sh/sh.hpp>
using namespace SH;

ShProgram myFirstFragmentShader;
ShProgram myFirstVertexShader;
```

Listing 3.16: Deklaration eines Vertex Shaders

Der Vertex Shader, den wir schreiben wollen, soll die Geometrie transformieren. Dazu benötigen wir im Shader Zugriff auf die Projektions- und die Modelview-Matrix. Wir könnten uns eine Routine schreiben, die diese Matrizen aus OpenGL ausliest und in uniforme Sh Variablen kopiert. In diesem Tutorial wird aber die schnellere Methode angewandt und stattdessen ein OpenGL State

an eine Variable gebunden, wie in Kapitel 2.2.4.1. OpenGL States beschrieben. Dazu erschaffen wir uns in der main-Routine eine entsprechende Variable und weisen sie der ModelViewProjection Matrix zu.

```
shInit();
shSetBackend("arb");

ShMatrix4x4f mvpMatrix;

mvpMatrix.meta("opengl:readonly", "true");
mvpMatrix.meta("opengl:state", "state.matrix.mvp");
```

Listing 3.17: State Binding

Der Grund, warum hier unbedingt das ARB Backend gewählt werden muss, liegt darin, dass ein Fehler in der Implementierung des GLSL Backends beim Zugriff auf Matrizen, die an einen OpenGL State gebunden sind, fehlerhaften GLSL Code produziert. Dieser Fehler wird voraussichtlich ebenfalls in der nächsten Version von Sh behoben sein[18]. Das ARB Backend produziert zwar dafür andere Warnungen, die aber in diesem Fall gefahrlos ignoriert werden können, da wir nur lesend auf die Matrix zugreifen.

Als nächstes kann der Vertex Shader definiert werden. Er soll neben der Transformation der Geometrie auch noch die Farbe weitergeben. Damit seine Arbeit aber nicht völlig unsichtbar bleibt, soll er diese zuvor noch ein wenig verändern, indem er die Farbe auf grün setzt. Wir definieren den Vertex Shader direkt oberhalb des Fragment Shaders in der main-Routine.

```
MyFirstVertexShader = SH_BEGIN_VERTEX_PROGRAM
{
     ShOutputColor3f col;
     ShInOutPosition4f pos;

     pos = mvpMatrix * pos;

     col = ShColor3f(0, 1, 0);
}
```

Listing 3.18: Definition eines Vertex Shaders

Da der Shader als Farbe immer nur grün heraus schreiben soll, ist die eingehende Farbe irrelevant. Daher ist die Variable *col* auch nur als Ausgabewert deklariert. Am Ende des Shaders wird dieser Variable noch eine grüne Farbe zugewiesen. Der Datentyp *ShColor3f* ist dabei nichts anderes als ein anderer Name für einen *ShVector3f*, also ein Tupel mit 3 Komponenten.

Der Wert *pos* ist als Ein- und Ausgabewert deklariert. Er wird aber nicht unverändert weitergegeben, sondern zuvor noch mit der einer vorbereiteten Matrix multipliziert. Sh kümmert sich im Hintergrund darum, dass bei der Berechnung die richtigen Register an der richtigen Stelle eingesetzt werden.

18 State Binding Fehler: Der Übersetzungsfehler in GLSL kann umgangen werden, indem statt einer einzelnen Matrize vier einzelne Vektoren erzeugt werden, denen jeweils eine Spalte der Matrix zugewiesen wird und die im Shader dann auch einzeln mit den Komponenten der Position multipliziert werden – auch andere Lösungen sind denkbar

Zuletzt muss nun auch noch der Vertex Shader vorm Zeichnen aktiviert werden. Dieser soll aber schon beim Zeichnen des ersten Quadrates aktiviert werden und bis zum Ende aktiv bleiben. Die *render()* Routine muss also folgendermaßen erweitert werden:

```
void render()
{
	glClearColor3f(1, 1, 1, 0);
	glClear(GL_COLOR_BUFFER_BIT);
	glPushMatrix();

	shBind(myFirstVertexShader);

	// rendert das erste Quadrat
	glTranslatef(-0.25f, 0, 0);
	renderSquare();

	// rendert das zweite Quadrat
	glTranslatef(1, 0, 0);

	shBind(myFirstFragmentShader);
	renderSquare()
	shUnbind();

	glPopMatrix();
	glutSwapBuffers();
}
```

Listing 3.19: neue render-Routine

Von der ursprünglichen gelben Farbe der Quadrate ist nun nichts mehr übrig geblieben, beide Quadrate sind grün. Das rechte Quadrat, auf dem neben dem Vertex Shader auch noch der Fragment Shader wirkt, ist dabei weiterhin dunkler als das linke. Der Befehl *shBind()* deaktiviert in diesem Fall gleichzeitig sowohl den Vertex als auch den Fragment Shader.

Abbildung 6: der Vertex Shader transformiert die Geometrie und verändert die Farbe

3.5. fortführende Beispiele

3.5.1. Tutorial: Wellen-Effekt

In diesem Tutorial soll die Nutzung von Vertex Shadern und von uniformen Variablen weiter vertieft werden. Ziel ist es, eine Ebene mittels einer Sinusfunktion in Schwingung zu versetzen. Das Tutorial setzt das Wissen aller vorangehenden Tutorials voraus und es werden nur relevante Teile erklärt.

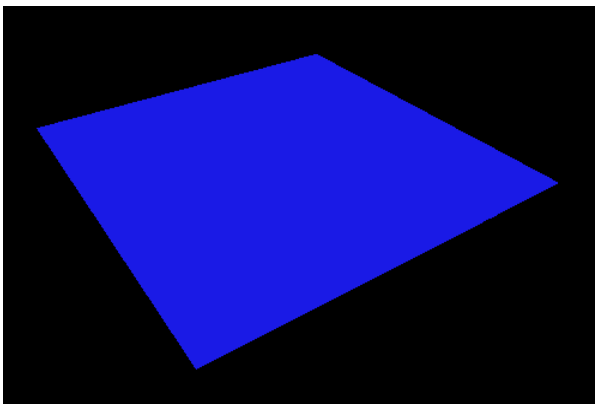

Abbildung 7: flache Ebene

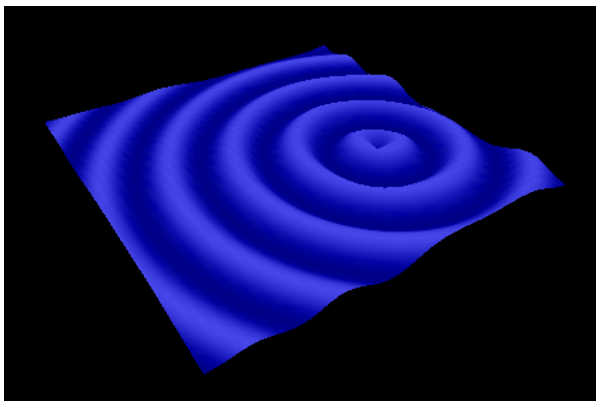

Abbildung 8: verformte und eingefärbte Ebene

Das folgende Listing zeigt die main-Routine des Programms. Sie wurde im Vergleich zu den vorangehenden Tutorials um einige Punkte erweitert. Zum einen wurden die gesamte Sh Initialisierung und die Shader Deklarationen in eine separate Funktion *setupShading()* ausgelagert. Außerdem wurde zusätzlich zur Display Routine bei Glut auch eine Idle Routine angegeben. Beide Funktionen werden später noch besprochen. Da dieses Programm mit 3D Grafik arbeitet, wird am Ende der main-Routine noch die perspektivische Projektion vorbereitet und eine Kamera definiert.

```
void main(int argc, char* argv[])
{
      glutInit(&argc, &argv);

      glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
      glutInitWindowSize(600, 600);
      glutCreateWindow("Ripples");

      glutDisplayFunc(render);
      glutIdleFunc(idle);

      setupShading();

      glMatrixMode(GL_PROJECTION);
      gluPerpective(45, 4.0f/3.0f, 1, 1000);
      glMatricMode(GL_MODELVIEW);
      gluLookAt(-150, 75, -100, 0, 0, 0, 0, 1, 0);

      glutMainLoop();
}
```

Listing 3.20: main-Routine

Als nächstes schauen wir uns mal die Funktion *render()* an. Diese Funktion soll eine blaue Ebene zeichnen, die später vom Vertex Shader verformt und verfärbt werden kann. Dazu muss sie viele kleine Quads zeichnen, denn der Vertex Shader wird einmal für jeden Eckpunkt aufgerufen.

Zu Beginn der Funktion werden die Variablen *quadSize* und *numQuads* festgelegt. Diese bestimmt die Größe eines Teilquadrates der Ebene, und die Anzahl der Teilquadrate. Damit die Ebene immer schön zentriert vor unserer Kamera liegt, folgt ein etwas komplexerer Translationsbefehl, der hier aber nicht weiter von Bedeutung ist. Schliesslich werden die Teilquads, aus denen unsere Ebene besteht, in einer Schleife gezeichnet.

```
void render()
{
   glClearColor(0, 0, 0, 0);
   glClear(GL_COLOR_BUFFER_BIT);

   float quadSize = 2.5f;
   int numQuads = 45;

   glPushMatrix();
   glTranslatef(-numQuads * quadSize * 0.5f, 0,
                    -numQuads * quadSize * 0.5f);

   glBegin(GL_QUADS);
   glColor3f(0.1f, 0.1f, 0.9f);
   for (int x = 0; x < numQuads; x++)
      for(int z = 0; z < numQuads; z++)
      {
         glVertex3f(x * quadSize, 0, z * quadSize);
         glVertex3f(x * quadSize, 0, z * quadSize + quadSize);
         glVertex3f(x * quadSize + quadsize, 0, z * quadSize + quadSize);
         glVertex3f(x * quadSize + quadSize, 0, z * quadSize);
      }
   glEnd();

   glPopMatrix();
   glutSwapBuffers();
}
```

Listing 3.21: render-Routine

Der Vertex Shader, den wir hier entwickeln wollen, soll zwei Aufgaben erfüllen: Zum einen soll er die Position des Vertices, für den er aufgerufen wird, in der y-Komponente verändern, bevor er ihn transformiert. So sollen Ripples entstehen, die von einem Punkt ausgehen, der sich auf der Ebene bewegt. Dazu benötigen wir zunächst eine globale Variable, die diesen Punkt repräsentiert. Außerdem sollen sich die Ripples wellenförmig bewegen, daher wird noch eine zweite uniforme Variable *k* und zwei Variablen zur Zeitmessung benötigt.

```
#include <sh/sh.hpp>
using namespace SH;

int time1, time2;
ShPoint4f p;
ShAttrib1f k;
```

Listing 3.22: globale Variablen

Den Punkt *p* werden wir später auf der Ebene Kreise ziehen lassen, er dient als Ausgangspunkt für die Wellenbewegungen. Aber zunächst einmal zur Shader Deklaration in der Routine *setupShading()*. Neben der Sh Initialisierung und dem Binden einer *ShMatrix* an einen OpenGL State, wird zusätzlich auch noch die Variable *k* mit 0 initialisiert. Dann wird Vertex Shader deklariert und auch gleich aktiviert. Da er dauerhaft aktiv sein soll, muss er nicht in der *render()* Funktion ständig an und aus geschaltet werden.

```
void setupShading()
{
   shInit();
   shSetBackend("arb");
   k = 0.0f;
   ShMatrix4x4f mvp;
   mvp.meta("opengl:readonly", "true");
   mvp.meta("opengl:state", "state.matrix.mvp");

   ShProgram rippleShader = SH_BEGIN_VERTEX_PROGRAM
   {
       ShInoutColor4f col;
       ShInputPosition4f inPos;
       ShOutputPosition4f outPos;
       // Berechne Abstand Vertex – Punkt p
       ShAttrib1f temp = length(inPos – p);
       // verändere die y-Komponente der Position
       inPos[1] += 2.5f * sin(((temp) / 3.0f + k)) * (1–(temp /100.0f));
       // transformiere die Position wie üblich
       outPos = mvp * inPos;
       // verändere die Farbe ein wenig
       col = col * 0.8f + sin(((temp) / 3.0f + k)) * 0.2f;
   } SH_END;

   shBind(rippleShader);
}
```

Listing 3.23: setupShading-Routine

Der Vertex Shader arbeiten nur mit zwei Eingabewerten – der Position und der Farbe. Zunächst wird zwischen der Position des zu bearbeitenden Vertex und unserem Punkt *p* der Abstand berechnet. Dieser Wert dient zusammen mit der zeitlichen Komponente *k* als Eingabe für eine simple Sinusfunktion, die für den schönen Wellen-Effekt sorgt. Die eigentliche Berechnung kann auf viele verschiedene Weisen durchgeführt werden und soll daher hier nicht näher besprochen werden.

Die zweite Aufgabe des Vertex Shader ist die Verfärbung der Ebene. Dazu wird erneut die Sinusfunktion herangezogen. Alle Berechnungen sind lediglich meine Vorschläge und können jederzeit geändert werden, um die Ergebnisse zu untersuchen.

Bevor das Programm nun aber kompiliert und ausgeführt werden kann, fehlen noch ein paar kleinere Ergänzungen. Es muss die Zeit gemessen werden und die Differenz zwischen zwei Frames bestimmt werden. An dieser Stelle kommt die Idle-Routine ins Spiel. Sie übernimmt die Zeitmessung und sorgt auch gleich dafür, das das Bild ständig neu gezeichnet wird.

```
void idle()
{
     time2 = glutGet(GLUT_ELAPSED_TIME);
     k -= 0.015f * (time2 - time1);
     time1 = glutGet(GLUT_ELAPSED_TIME);

     render();
}
```

Listing 3.24: idle-Funktion

Als letztes muss noch die *render()* Routine erweitert werden. Damit der Vertex Shader auch ständig die aktuellen Werte für die uniformen Variablen *p* und *k* bekommt, muss *shUpdate()* aufgerufen werden. Außerdem soll sich der Punkt *p* ja bewegen, auch das geschieht an dieser Stelle.

Wie bereits in Kapitel 2.2.3.2. erwähnt, ist in der in dieser Arbeit zum Einsatz kommenden Version von Sh der Aufruf der Funktion *shUpdate()* tatsächlich nicht erforderlich, da keine Texturen zum Einsatz kommen und die uniformen Variablen bereits bei ihrer Änderung automatisch auch auf der GPU aktualisiert werden. Dieses Tutorial verwendet die Funktion dennoch, um die Kompatibilität mit kommenden Version von Sh zu gewährleisten.

```
void render()
{
   glClearColor(0, 0, 0, 0);
   glClear(GL_COLOR_BUFFER_BIT);

   float quadSize = 2.5f;
   int numQuads = 45;

   p[0] = p[2] = numQuads * quadSize * 0.5f;
   p[1] = 0;
   p[3] = 1;
   p[0] += sin(k / 13.0f) * quadSize * numQuads / 4.0f;
   p[2] += cos(k / 13.0f) * quadSize * numQuads / 4.0f;

   glPushMatrix();
   glTranslatef(-numQuads * quadSize * 0.5f, 0,
                    -numQuads * quadSize * 0.5f);

   shUpdate();

   glBegin(GL_QUADS);
   glColor3f(0.1f, 0.1f, 0.9f);
   for (int x = 0; x < numQuads; x++)
      for(int z = 0; z < numQuads; z++)
      {
         glVertex3f(x * quadSize, 0, z * quadSize);
         glVertex3f(x * quadSize, 0, z * quadSize + quadSize);
         glVertex3f(x * quadSize + quadsize, 0, z * quadSize + quadSize);
         glVertex3f(x * quadSize + quadSize, 0, z * quadSize);
      }
   glEnd();

   glPopMatrix();
   glutSwapBuffers();
}
```

Listing 3.25: render-Routine

Damit ist das Tutorial abgeschlossen. Beim Ausführen des Programms sollte sich nun ein netter Wellen-Effekt über das Quadrat bewegen.

3.5.2. weitere Beispiele

In dem auf der offiziellen Website von Sh[7] zum Download zur Verfügung stehenden Source Code von Sh sind einige Beispiele enthalten. Sie befinden sich im Verzeichnis

/libsh-0.8.0.rc1/examples

Auch auf der Webseite selbst befinden sich einige Beispiele zum Einsatz von Sh für grafische Effekte und für Streaming Zwecke.

3.6. Sh in bestehende Projekte einbauen

In diesem Kapitel sollen noch einige wenige zusätzliche Informationen gegeben werden, die es erleichtern sollen, Sh nachträglich in bestehende Projekte einzubauen. Generell gestaltet sich das nachträgliche Einbinden relativ simpel. Lediglich die beschriebenen Änderungen an den Projekt-

Einstellungen müssen vorgenommen werden, bevor das Projekt nach Einbinden der Sh Bibliotheken wieder kompiliert und ausführt.

Das Angeben eines speziellen Backends ist nicht zwingend erforderlich, aber unbedingt zu empfehlen. Falls Vertex Shader eingesetzt werden sollen ist so die Geometrietransformation durch OpenGL State Binding wesentlich leichter umzusetzen.

Die Funktion shInit() kann ohne Nebenwirkung mehrfach aufgerufen werden, sofern es die Hierarchie des Programms nicht anders möglich macht. Uniforme Variablen, die in Shadern genutzt werden sollen, müssen lediglich am Ort der Shaderdeklaration verfügbar sein; es können also Membervariablen und lokale Variablen genutzt werden.

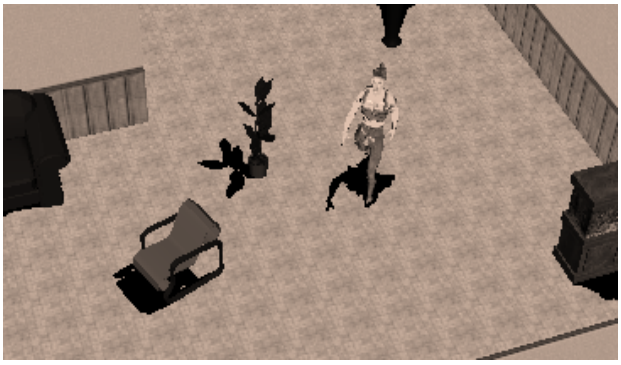

Abbildung 10: Black&White Shader (Arbeitsaufwand: 20min)

Abbildung 9: Spiel "Thieves Twist" Original

Besonders schwierig gestaltet sich beim nachträglichen Einbau von Sh der Umgang mit Texturen. Sh ist grundsätzlich darauf ausgelegt, dass alle Texturen auch über Sh verwaltet werden und in einem Shader auf eine spezifische Textur zugegriffen wird. Da in einem bereits bestehenden Programm die Texturen in aller Regel bereits anderweitig verwaltet werden, oder Shader auf mehrere Objekte mit verschiedenen Texturen angewandt werden sollen, muss dies beim Einsatz von Texturen in Shadern umgangen werden. Durch das Setzen von Meta-Informationen kann man Sh dazu bringen, in den Shadern statt auf eine bestimmte Textur, deren Unit von Sh verwaltet wird, auf eine bestimmte Textur-Unit zuzugreifen. Dadurch muss man das bestehende Programm nicht grundlegend abändern und kann mit nur zwei Zeilen zusätzlichem Code trotzdem im Shader auf die jeweils aktive Textur zugreifen. Listing 3.26 zeigt, wie eine Textur an die Texturunit 0 gebunden werden kann.

```
ShTexture2D<ShColor3fub>  m_texture;
m_texture.meta("opengl:preset", "0");
```

Listing 3.26: Workaround für Texturen

Auf eine derartig vorbereitete Textur kann im Shader dann wie üblich zugegriffen werden (siehe dazu 2.2.3.4. Texturen). Wird ein Shader, in dem ein Texturzugriff vorkommt, auf nicht texturierter Geometrie ausgeführt, führt dies nicht zu einem Fehler.

4. Beispielapplikation

Im Rahmen dieser Arbeit wurde eine Beispielapplikation entwickelt, die einige der besonderen Spracheigenschaften von Sh demonstrieren soll. Vorgestellt wird die Shader Algebra, im Einzelnen sowohl die Verwendung des *Connect*- als auch des *Combine*-Operators. Diese wird in Verbindung sowohl mit grafischen Shadern, als auch mit Streaming Shader gezeigt. Außerdem sollen die Fähigkeiten des Sh-eigenen Optimierers vorgeführt werden.

Sh verfügt über viele weitere, teilweise einzigartige Eigenschaften. Dazu zählen z.B. das automatische Parameter-Update und die einfache Erstellung von Streaming Shadern. Diese bieten jedoch vor allem in der Entwicklung Vorteile und lassen sich nur schwer in einer Beispielapplikation visualisieren.

Zur Implementierung der grafischen Oberfläche habe ich Qt gewählt. Mit der Version 4 veröffentlicht Trolltech[10] Qt für nicht kommerzielle Projekte wieder als Open Source[11]. Da Qt in dieser freien Version keine Unterstützung für Visual Studio .NET beinhaltet und sich Projekte nicht kompilieren lassen, habe ich die von Volker Wiendl verfasste Anleitung[12] und einen inoffiziellen Qt Patch eingesetzt, um Qt auch unter .NET einsetzen zu können. Für die Grafik wurde OpenGL und die freie Character Animation Library Cal3D[13] eingesetzt.

Das Programm ist in drei Präsentation unterteilt, die auf drei Tabs verteilt sind. Der erste Tab zeigt die Anwendung von Shader Algebra mit grafischen Shadern. Auf der rechten Seite des Fensters stehen verschiedene kurze Shader sowohl für die Vertex als auch für die Fragment Einheit zur Auswahl. Per Mausklick können die Shader ausgewählt und aktiviert werden.

Die Vertex Shader werden dabei jedes mal mit dem bereits aktiven Shader kombiniert (*combine*-Operator), während die Fragment Shader verbunden werden (*connect*-Operator). Eine umgekehrte Anwendung der Operatoren ist nicht sinnvoll, siehe dazu auch Kapitel 2.5. Shader Algebra. Der resultierende Shader wird in Kurzform im unteren Teil des Fensters angezeigt. Es kann aber auf Wunsch auch der vollständige generierte GLSL Code der jeweils aktiven Shader angezeigt werden. Das Ergebnis der angewandten Shader wird in einem OpenGL Fenster im linken, oberen Teil des Fensters dargestellt.

Neben der Shader Algebra demonstriert dieses Tab auch zwei der Schwächen von Sh: Zum einen sorgt die von Sh durchgeführte Verknüpfung der Variablen in den Shadern mit den Registern dafür, dass die Standard-Pipeline mit Shader Algebra nicht ohne Weiteres nachgebaut werden kann. Dies wird deutlich, wenn der Fragment Shader komplett deaktiviert wird, aber ein Vertex Shader zusammengestellt wird, der die Geometrie transformiert, beleuchtet und die Texturkoordinaten durchreicht. Das Ergebnisfenster sollte nun eigentlich ein korrekt texturiertes Modell zeigen, jedoch werden die Texturkoordinaten offensichtlich nicht korrekt weitergegeben. Dieses Problem in der Registerbindung bei der Shader Algebra ist auch dafür verantwortlich, dass der Shader „Per Vertex Beleuchtung“ neben der Beleuchtung auch die Transformation der Geometrie beinhalten muss, um zu funktionieren.

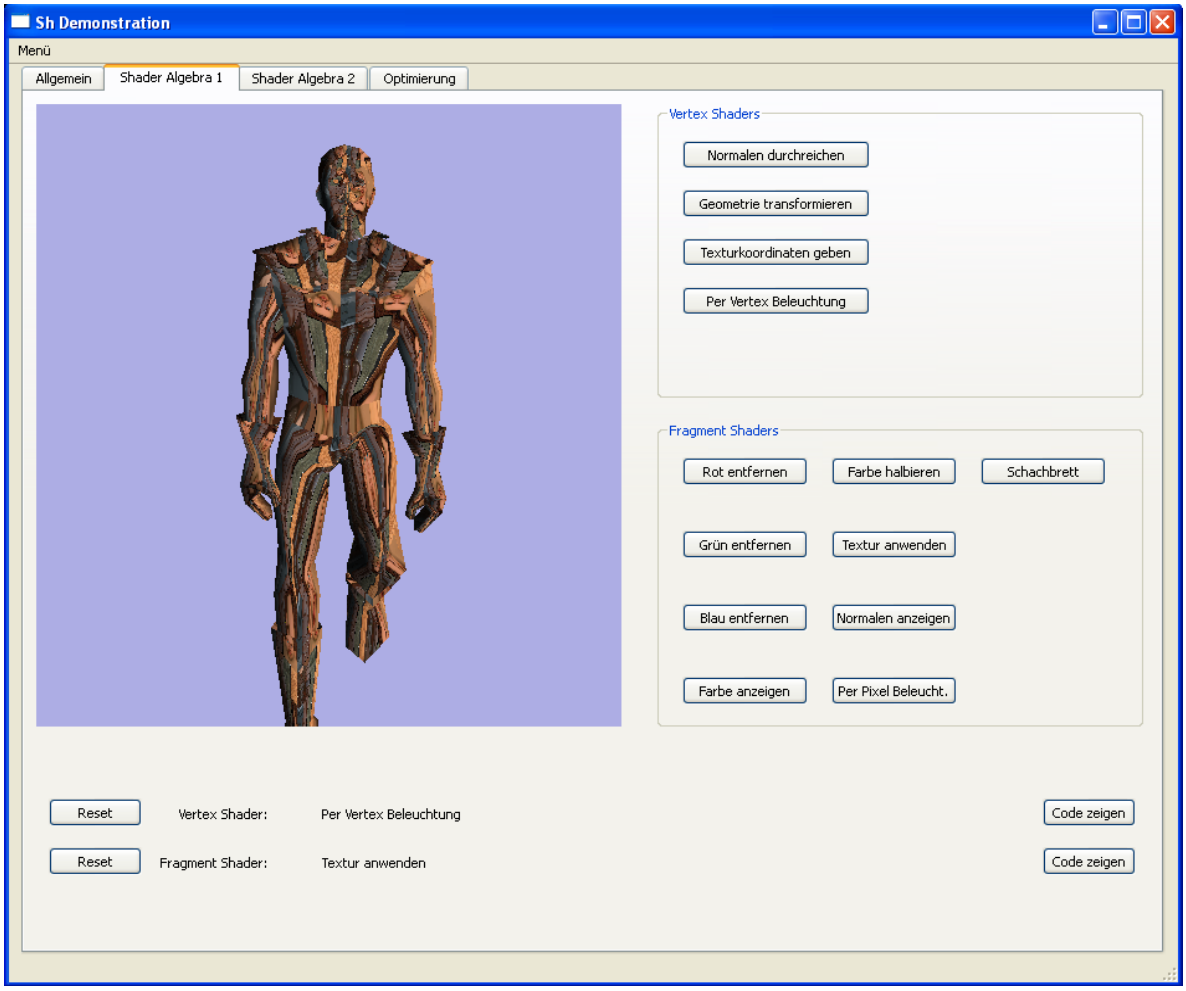

Abbildung 11: Shader Algebra mit grafischen Shadern

Des weiteren wird hier auch die gelegentliche Fehlfunktion des Sh-internen Optimierers sichtbar, wenn als Fragment Shader „Normalen anzeigen“ gewählt wird. Nach der Aktivierung diesen Shaders lässt sich kein weiterer Fragment Shader mehr anhängen, da der Optimierer jeden weiteren Code wegkürzt. Wird die Optimierung deaktiviert, tritt dieser Fehler nicht auf. Auf beide Eigenheiten, die Registerbindung und der interne Optimierer von Sh, wird in Kapitel 5 noch detailliert eingegangen.

Im zweiten Tab wird der *connect*-Operator der Shader Algebra mit Streaming Shadern demonstriert. Dabei wird die GPU genutzt, um unter Einsatz ihrer Parallelfähigkeiten eine mathematische Funktion auf einer großen Menge an Daten zu berechnen. Auf der rechten Seite stehen verschiedene Shader zur Auswahl, die jeweils eine mathematische Funktion in Abhängigkeit der Variable x repräsentieren. Mittels Mausklick können diese hintereinander geschaltet werden. Dadurch erhält eine Funktion als Eingabewert für x jeweils den Ausgabewert der vorangehenden Funktion. Der Wertebereich von x kann ebenfalls auf der rechten Seite festgelegt werden. Im unteren Bereich des Fensters wird die aktuelle, zusammengefügte Funktion dargestellt, die durch den momentan aktiven Shader repräsentiert wird.

Ein Klick auf den Knopf „Neu Berechnen“ bewirkt, dass der Streaming Shader erneut auf den Daten ausgeführt wird. Dabei wird der Shader parallel auf den Daten ausgeführt, siehe dazu auch 2.4. Sh für GPGPU. Im OpenGL-Fenster auf der linken Seite wird das Ergebnis der resultierenden Funktion als Graph dargestellt.

In dieser Präsentation kommt das GLSL Backend zum Einsatz, da das ARB Assembler Backend mit Streaming Shadern mehrere Fehler in der Implementierung aufweist (siehe dazu auch Kapitel 5.5. Sh intern). Der Code des resultierenden Shaders kann hier im Gegensatz zum vorangehenden Tab nicht angezeigt werden, da Sh Streaming Shader nicht direkt übersetzt, sondern intern anders als grafische Shader verwaltet und gegebenenfalls sogar in mehrere Shader aufgeteilt werden, die dann in mehreren Passes ausführt werden, um größtmögliche Effizienz zu erreichen.

Neben dem Wertebereich für *x* kann auch die Schrittgröße angegeben werden. Bei der Berechnung wird dann ein Array, bzw. ein *ShChannel* mit Werten zwischen dem unteren und dem oberen Ende des Wertebereichs gefüllt, wobei der Abstand zwischen den einzelnen Werten der Schrittgröße entspricht. Besonders bei kleinen Wertebereichen ist die Angabe einer möglichst kleinen Schrittgröße wichtig, um den resultierenden Graphen in genügend hoher Auflösung darstellen zu können.

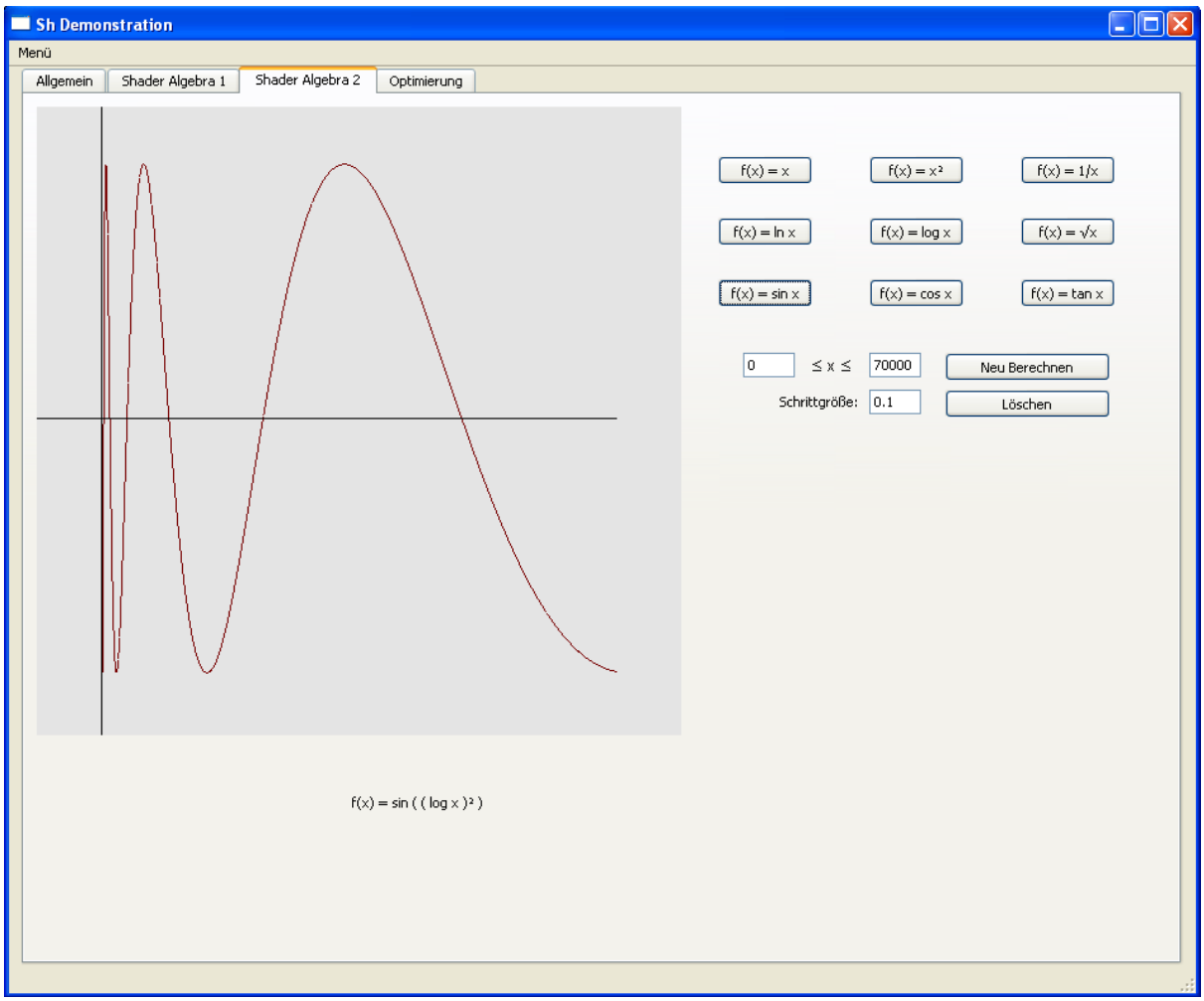

Abbildung 12: Shader Algebra mit Streaming Shadern

Für die Darstellung des Graphen wird das Array, bzw. der *ShChannel* mit den Ausgabedaten grob analysiert und ungefähre Minimal- und Maximal-Werte ermittelt. Anhand dieser Daten wird dann das Koordinatensystem entsprechend angepasst.

Nach der Berechnung einiger Funktion in diesem Tab kommt es gelegentlich beim Beenden des Programms zu einem Absturz der Applikation. Dies ist auf einen Fehler in der Implementation einiger Destruktoren des GLSL Backends zurückzuführen (siehe dazu auch Kapitel 5.5. Sh intern).

Einige der angebotenen Funktionen, wie z.B. *log(x)*, können mit negativen Werten als Eingabeparameter nicht arbeiten, da sie für diesen Wertebereich nicht definiert sind. In einem solchen Fall geben die Shader als Ergebnis einen im Hauptprogramm definierten Code zurück, der später bei der Ausgabe wieder abgefangen wird. Hiermit wird indirekt ein weiterer Vorteil von Sh demonstriert. Durch die Einbettung in den Hauptcode des Programmes kann in den Shadern auf alle definierten Konstanten und Marko-Definitionen zugegriffen werden. So ist eine gute Konsistenz zwischen Hauptprogramm und Shadern gewährleistet, da Änderungen, z.B. an Werten von konstanten Variablen, automatisch in alle Shader übernommen werden.

Eine Zeitmessung der Dauer der Berechnungen enthält dieses Tab nicht. Um eine zuverlässige Zeitmessung zu erhalten, müsste der resultierende Shader mehrfach hintereinander ausgeführt werden. Dies führt zu sekundenlangen Wartezeiten vor der Darstellung der Funktion als Graphen. Zudem müsste zum Zwecke des Vergleiches auch eine CPU Version der Funktionen implementiert werden, sowie ein Konstrukt, das für diese eine Art Shader Algebra emuliert. Dieser Tab soll jedoch nicht zum Geschwindigkeitsvergleich zwischen CPU und GPU dienen, sondern zur Präsentation der Shader Algebra in Verbindung mit Streaming Shadern.

Im dritten Tab des Programms wird der Sh-eigene Optimierer demonstriert. Dazu kann in der oberen linken Hälfte ein Ausgangsshader gewählt werden, dessen Sh Code so, wie er im Hauptprogramm definiert ist, im linken Textfeld angezeigt wird. In der rechten oberen Seite des Fensters kann der gewünschte Grad der Optimierung gewählt werden. Im rechten Textfeld wird dann der von Sh generierte GLSL Code abgebildet.

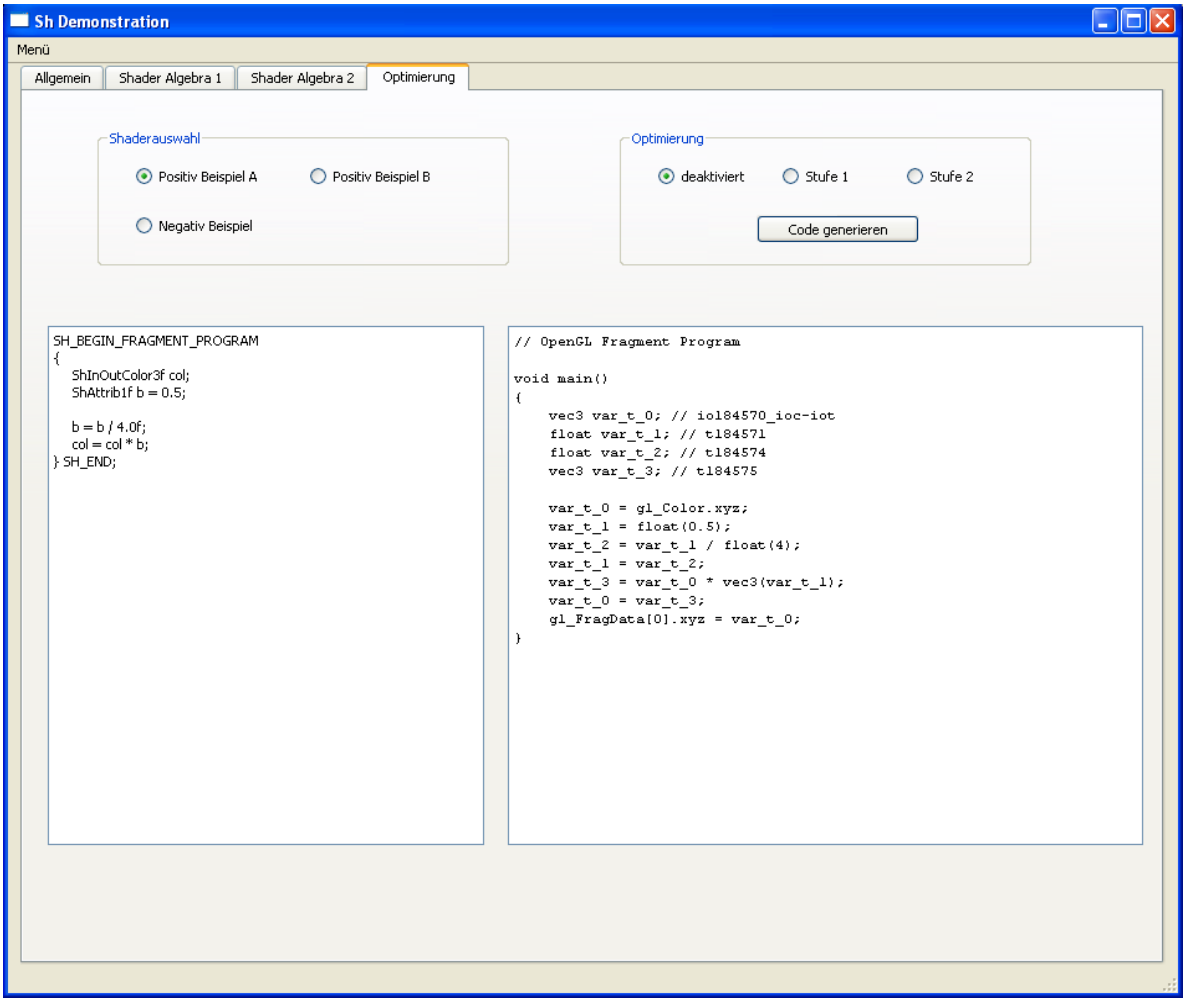

Abbildung 13: Demonstration des Sh-eigenen Optimierers

Da der Optimierer nicht in jedem Fall gute Ergebnisse liefert, sondern unter Umständen sogar zusätzliche Rechenoperationen verursacht, stehen neben den zwei positiven auch ein negatives Beispiel zur Auswahl.

Bemerkenswert ist besonders die höchste Stufe der Optimierung der positiven Shader Beispiele. Hier werden sämtliche Berechnungen der beiden Shader bereits vorher ausgewertet. Im zweiten Shader, in dem ein Vektor zuerst normalisiert und danach seine Länge berechnet wird, kürzt der Optimierer die beiden Operationen komplett weg und setzt direkt eine 1.0 an die Stelle des Ergebnisses.

Im negativen Beispiel wird gezeigt, dass der Optimierer in ungünstigen Fällen auch zusätzliche Rechenoperationen erzeugen kann. In der Übersetzung des entsprechenden Shaders kommen je ein redundanter *normalize()* und ein *cross()* Aufruf vor.

5. Gegenüberstellung

Im Folgenden soll Sh auf seine Anwendbarkeit und Nutzbarkeit hin untersucht werden. Dazu wird die Metasprache unter anderem einer herkömmlichen Shader-Hochsprache gegenübergestellt, um Vorteile und Nachteile zu identifizieren. Auch Sh-eigene Tests sollen durchgeführt werden, um insbesondere die Erwartungskonformität und die Intuitivität der Sprache zu untersuchen.

Als Hochsprache für die Gegenüberstellung habe ich GLSL gewählt. Die Gründe dafür sind zum einen, dass es sich bei GLSL um eine moderne und mächtige Shader Hochsprache handelt, die zudem von Sh bereits mittels eines Backends unterstützt wird. So ist es möglich, zu vergleichen, ob ein direkter Einsatz von GLSL oder ein indirekter über die Metaprogrammierung mittels Sh vorteilhafter ist. Darüber hinaus lehnt sich die GLSL Syntax an die C Syntax an, was auf die meisten der aktuellen Hochsprachen zutrifft, wie z.B. Cg und HLSL. Daher ist der Lernaufwand in der Gegenüberstellung mit Sh, das vollkommen auf der C++ Syntax basiert, vergleichbar. Zudem basiert GLSL auf OpenGL, was zum derzeitigen Entwicklungsstand auch die einzige von Sh unterstütze Grafik API ist.

5.1. Handhabung und Funktionalität

Der Ablauf von der Deklaration eines Shaders bis zu dessen Einsatz vor dem Zeichnen von Geometrie verläuft in beiden Sprachen zwar ähnlich, Sh vereinfacht diesen Prozess jedoch wesentlich und die zum Einsatz kommenden Strukturen erscheinen gegenüber GLSL weniger verwirrend.

In GLSL muss der Source Code eines Shaders in der Regel von einer externen Datenquelle eingelesen werden, d.h. es müssen ggf. zunächst Routinen zum Einlesen von Textdateien geschrieben werden. Dieser Source wird dann mit einem Shaderobjekt verbunden, das über ein Handle verwaltet wird. Anschließend wird dieses kompiliert und dann mit einem Programmobjekt verknüpft. Vertex und Fragment Shader verwenden dabei unterschiedliche Shaderobjekttypen, werden aber an das gleiche Programmobjekt gehängt. Das Programmobjekt kann schließlich gelinkt und danach kann es dann aktiviert werden. Soll nur einer der Shader einzeln deaktiviert oder ausgetauscht werden, muss das Programmobjekt verändert und danach erneut gelinkt werden.

In Sh hingegen werden die Shader direkt im Source Code des Hauptprogramms deklariert, eine Routine zum Einlesen von Textdateien ist nicht notwendig. Bei der Deklaration wird einmalig angegeben, ob es sich um einen Vertex oder einen Fragment Shader handelt. Eine weitere Unterscheidung zwischen den Shadern erfolgt nicht, da beide vom gleichen Typ ShProgram sind. Weitere Programmobjekte sind nicht notwendig und auch das Kompilieren und Linken geschieht bei Sh automatisch. Die Shader können direkt nach der Deklaration durch einen einzelnen Befehl separat aktiviert oder deaktiviert werden.

Neben der Shader Deklaration und Anwendung spielen auch die Variablen und Parameter eine große Rolle. Lokale, temporäre Variablen sind hier zu vernachlässigen, da sich ihre Nutzung in den Sprachen praktisch nicht unterscheidet. Die semantischen und uniformen Parameter sind jedoch von besonderem Interesse.

Auf die semantischen Variablen (siehe 2.2.3.1.), die in GLSL Attribute genannt werden, kann hier über feste Bezeichner zugegriffen werden, die fest mit den entsprechenden Registern verknüpft sind. Varyings werden dabei alle Variablen genannt, die vom Vertex Shader an den Fragment Shader weiter gegeben werden. Selbstverständlich zählen die Bezeichner der semantischen Ausgabevariablen des Vertex Shaders dazu, wie die Farbe und die Position, aber GLSL erlaubt auch die Definition eigener Varying Variablen. Diese müssen gesondert sowohl im Vertex als auch im Fragment Shader deklariert sein und ihre Bezeichner müssen übereinstimmen.

Sh erlaubt die Deklaration semantischer Variablen, die abhängig von ihrem Typ automatisch mit den entsprechenden Registern verbunden werden, während die Bezeichner frei wählbar sind. Varying Variablen hingegen werden von den semantischen Variablen nicht speziell unterschieden und müssen nicht gesondert deklariert werden. Stattdessen werden sie von Sh automatisiert erstellt, wenn im Vertex Shader zusätzliche Ausgabeparameter eingesetzt werden. Weder die Anzahl dieser zusätzlichen Variablen, noch ihre Bezeichner müssen mit den im Fragment Shader deklarierten Eingabeparametern übereinstimmen, was ihre Anwendung stark erleichtert.

Uniforme Variablen stellen die Schnittstelle zwischen dem Hauptprogramm und den Shadern dar und müssen vom Hauptprogramm aus regelmäßig aktualisiert werden. Hier kommt eine der größten Stärken von Sh zum Tragen, da es die Verwaltung und Aktualisierung aller uniformer Variablen in allen deklarierten Shader automatisiert. Mit nur einer Anweisung werden alle uniformen Parameter der gerade aktiven Shader auf den neuesten Stand gebracht. In GLSL muss ein Handle für jede einzelne Variable abgefragt werden. Zudem ist beim Einsatz mehrerer Shader zusätzlicher Code für die Verwaltung notwendig, um nur die Variablen der jeweils aktiven Shader zu aktualisieren. Hier bietet Sh klare Vorteile, insbesondere in der Shader Entwicklung, da sich das Hinzufügen und Entfernen uniformer Variablen einfach und schnell gestaltet.

5.2. Sprachumfang

Bei GLSL handelt es sich um eine sehr mächtige Sprache, die zudem auf die OpenGL Grafik API zugeschnitten ist. Da Sh API-unabhängig bleiben und darüber hinaus auch Sprachen mit einem geringeren Sprachumfang als Ziel unterstützen soll, bietet GLSL in manchen Gebieten Vorteile. Doch Sh kann mit erweiterter Funktionalität aufwarten.

Sehr viel Komfort bietet GLSL beim Zugriff auf Variablen. Zum einen kann bereits durch die Indizierung Swizzling[19] ausgeführt werden, z.B. "gl_Color.zx". Dies kann auch in Sh genutzt werden. Da die Mathematik Bibliothek des Toolkits allerdings Tupel beliebiger Größe erlaubt, erfolgt hier die Indizierung etwas uneleganter über einen Operator. Zum anderen unterstützt GLSL eine Vielzahl an Konstruktoren für einen Variablentyp. So kann ein vec4 durch die Angabe von vier floats erstellt werden, aber auch durch die Angabe zweier vec2 oder eines vec3 und eines einzelnen floats. Durch die Gleichsetzung mit einem einzelnen float können auch alle Komponenten eines Vektors oder einer Matrizenzeile/-spalte gleichzeitig auf einen Wert gesetzt werden. Sh hat dafür keine vergleichbare Funktionalität zu bieten und erfordert somit für die gleichen Anweisungen mehr Schreibaufwand.

Datenabhängige Flusskontrolle, das sogenannte branching, wird von beiden Sprachen unterstützt. Sh hat hier als eingebettete Sprache den Nachteil, die Nutzung der entsprechenden C++-Befehle *if*, *do*, *for* und *while* nicht nutzen zu können, sondern stattdessen den Anwender zur Nutzung von entsprechenden Makros zwingen muss. Dennoch sorgt die Implementation von Sh dafür, dass Schleifen weniger performant umgesetzt werden. Siehe dazu auch Kapitel 5.6. Performanz.

19 Swizzling: Zugriff auf einen Vektor unter Angabe, in welcher Reihenfolge die Komponenten angesprochen werden sollen. Dabei können Komponenten vertauscht oder dupliziert werden. Auch die Anzahl der Komponenten im resultierenden Tupel kann verändert sein.

Weiterhin sorgt die Spezialisierung GLSLs auf OpenGL für zusätzliche Vorteile. So stellt GLSL eine Funktion mit dem Namen *ftransform()* zur Verfügung, die die Geometrietransformation im Vertex-Shader entsprechend der Standard-Pipeline durchführt. Wegen der Abstraktion kann Sh selbstverständlich keine entsprechende Funktion anbieten. Darüber hinaus ermöglicht GLSL den Zugriff auf zusätzliche, eingebaute semantische und uniforme Variablen, wie zum Beispiel die Punktgröße und die Koordinaten der Clipping Plane. Für diese gibt es in Sh keinen entsprechenden Variablentyp. GLSL erlaubt es im Gegensatz zu Sh außerdem, von Ausgaberegistern zu lesen, wie z.B. der Position im Vertex Shader.

Zumindest einen Teil dieser Funktionalitäten kann man durch geschickte Anwendung von Meta-Informationen in Variablen auch unter Sh nutzbar machen. So z.B. der Zugriff auf eingebaute Variablen, wie in Kapitel 2.2.4.1. OpenGL States erklärt.

Die Abstraktion von Sh birgt jedoch auch deutliche Vorteile. So erkennt Sh die zum Einsatz kommende Hardware und passt die Shader entsprechend an, ohne Mehraufwand für den Nutzer zu verursachen. Auch werden, soweit vorhanden, automatisch Erweiterungen eingesetzt, wie z.B. das „NV_fragment_program_option“ auf NVidia Karten. Zudem lehnt sich die Sh Syntax nicht nur an die C++ Syntax an, sondern basiert vollkommen auf ihr, was das Programmieren sehr intuitiv macht. Darüber hinaus bietet Sh zusätzliche und teilweise neuartige Funktionen an. Zum einen gibt es erstmals die Möglichkeit, einzelne Shader mittels der Shader Algebra zu neuen Shader zu kombinieren. Zum anderen beinhaltet Sh viele vordefinierte Funktionen, z.B. Worley oder Perlin, aber auch vorgefertigte Shader z.B. für Bump Mapping, die die Entwicklung beschleunigen. Auch GLSL besitzt verschiedene Noise-Funktionen[20], die aber bislang lediglich in der Spezifikation der Sprache vorhanden und selbst auf aktuellen Grafikkarten noch nicht implementiert sind[21]. Zudem ist das Angebot von Sh wesentlich umfangreicher.

5.3. Fehlersuche

Beim Programmieren treten Fehler auf, am häufigsten sind sicherlich Tipp- und Syntaxfehler. Daher ist es sinnvoll, beide Sprachen auf den Umfang und die Qualität ihrer Fehlermeldungen hin zu untersuchen.

Grundsätzlich schreibt GLSL ein Log, in dem beim Kompilieren oder Linken von Shadern aufgetretene Fehler verzeichnet werden. Dieses Log muss vom Nutzer ausgelesen werden, um an die Informationen zu gelangen. So können Tipp- und Syntaxfehler in Shadern gefunden werden. Bei Fehlern außerhalb des Shadercodes, die aber keine Kompilierfehler erzeugen, wie zum Beispiel das Aktualisieren einer uniformen Variable mit einem Wert vom falschen Typ (z.B. float mit einem Vektor), entsteht bei der Ausführung ein allgemeiner GL_INVALID_OPERATION Fehler.

Sh fragt in seinem GLSL Backend automatisch das Log ab und schreibt es im Debug Modus auf die Konsole. Dort werden auch alle sonstigen OpenGL Fehler ausgegeben. Damit bietet Sh bereits die gleiche Funktionalität wie GLSL. Allerdings verursacht Sh auch eigene Fehlermeldungen bereits zur Kompilierzeit, etwa, wenn in einem Shader statt der Makros für die Flusskontrolle die C++ Variante *if* verwendet wurde. Tipp- und Syntaxfehler werden aufgrund der Einbettung in C++ ebenfalls bereits zur Kompilierzeit erkannt oder durch das Syntax-Highlighting in der Entwicklungsumgebung auch sogar schon früher. Sofern es die Entwicklungsumgebung unterstützt, wird zu in Shadern genutzten Funktionen sogar Hilfe angeboten, wie z.B. Auskunft über Anzahl und Art der Parameter. Treten Fehler bei der Ausführung auf, gibt Sh in einigen Fällen weitere Informationen auf die Konsole aus, z.B. falls kein passendes Backend gefunden wurde. Stürzt das

20 Noise-Funktion: erlaubt es im Vertex und Fragment Shader, eine gewisse Zufälligkeit zu simulieren

21 Noise Implementation: auf Nvidia Karten wird immer der Wert 0.0 zurückgeliefert, auf ATI Karten verlangsamt die Funktion den Shader so erheblich, daß ein Einsatz unmöglich ist – offiziell unterstützen beide die Funktionen nicht. Nur 3DLabs, die Erfinder von GLSL, implementieren die Funktionen auf einigen ihrer Karten.

Programm allerdings ohne eine solche konkrete Hilfestellung ab, so ist der Fehler in der Regel nur schwer zu finden, da der Absturz meist fern von der tatsächlichen Fehlerursache auftritt. Verbindet man z.B. einen Streaming Shader, der nur einem einzelnen float-Eingabeparameter hat, mit einem Channel, der mit 3-Komponenten Vektoren gefüllt ist, so stürzt das Programm erst beim Aufruf des Shaders mit einer völlig nutzlosen Fehlermeldung und in einer ebenso wenig hilfreichen Codezeilenangabe ab.

5.4. Schwierigkeitsgrad

Der Schwierigkeitsgrad einer Sprache ist ein größtenteils subjektiver Wert und hängt stark vom Vorwissen des Nutzers ab. Daher kann dieser Abschnitt vor allem auf objektive Größen zur Beurteilung der Schwierigkeit eingehen. Dabei soll vor allem betrachtet werden, wie leicht oder umständlich sich der Einstieg in eine Sprache gestaltet. Dies hängt z.B. unter anderem von Umfang und Qualität der Lehrmaterial-Angebotes ab. Auch die Dauer bis zum ersten lauffähigen Programm ist von Interesse.

Aufgrund der Verbreitung von GLSL finden sich im Internet zahlreiche Seiten, auf denen man Unterstützung finden kann. Es gibt eine große Anzahl an Tutorials und Code Beispielen. Zudem gibt es eine aktive Community, die in zahlreichen Diskussionsforen Hilfestellung bietet, z.B. bei gamedev.net[14] oder auch opengl.org[15], die ein eigenes Forum für GLSL anbieten. Viele Informationen, Dokumentationen und Code Beispiele finden sich auch auf der Seite von 3DLabs[16]. Zudem ist ein Buch, das sogenannte „Orange Book“[17], über die OpenGL Shading Language erschienen, zur Ergänzung des „Red Book“[18], dem Standardwerk über OpenGL.

Im Vergleich dazu fällt das Angebot rund um Sh wesentlich geringer aus. Außer der offiziellen Webseite finden sich praktisch keine weiteren Informationen zu der Metasprache im Internet. Auf ihrer Seite befinden sich einige Code Beispiele, sowie die mittels DoxyGen[22] erstellte API Dokumentation. Diese ist jedoch leider oftmals nur wenig hilfreich, da der Code und die Funktionen vieler Klassen nur unzureichend oder gar nicht kommentiert sind. Darüber hinaus gibt es ein Sh-eigenes Forum auf gpgpu.org[20], in dem Fragen und Probleme geklärt werden können. Tutorials sind online nicht zu finden. Darüber hinaus wurde von Michael McCool und Stefanus Du Toit ein Buch über Sh[3] veröffentlicht, das auch eine kurze Einführung in die Sprache beinhaltet. Der Referenzteil aus dem Anhang des Buches wurde online zur Verfügung gestellt.

Zusammenfassend ist eindeutig zu sagen, dass das Dokumentation- und Hilfe-Angebot von GLSL deutlich umfangreicher ist als das von Sh. Anfragen im Sh-Forum werden allerdings in der Regel innerhalb weniger Tage von den Entwicklern selbst beantwortet.

Neben der vorhandenen Dokumentation ist auch die Dauer bis zum ersten lauffähigen Programm bei der Beurteilung des Schwierigkeitsgrades von Interesse, da die Experimentierfreudigkeit in einer neuen Sprache bei Einsteigern schnell gedämpft werden kann und im Gegenzug ein schneller Erfolg motivierend wirkt. Für diesen Vergleich habe ich ein Rahmenprogramm in Glut geschrieben, in das von einer Testperson mit guten C++ Kenntnissen jeweils ein Shader in Sh und einer in GLSL eingefügt werden sollte. Dabei wurde eine korrektes Installation von Sh, bzw. Glew[23] vorgegeben, nicht aber die für das Kompilieren von Sh-Projekten notwendigen Projekteinstellungen. Anleitung zum Einfügen der Shadern lieferte jeweils ein Tutorial. Das benötigte Vorwissen ist bei Sh und GLSL in etwa gleich. Kenntnis der C/C++ Syntax sowie der Standard-Pipeline ist vonnöten.

Beim Versuch mit GLSL kam das Tutorial von Lighthouse3D[21] zum Einsatz. Schwierigkeiten traten bei der Abfrage auf, ob das System GLSL unterstütze. Außerdem musste eine Routine zum

22 DoxyGen: Tool zum automatischen Erstellen einer Dokumentation aus Quelltext-Kommentaren[19]

23 Glew: OpenGL Extensions Wrangler Library – OpenGL Bibliothek die Hilfestellung bei der Abfrage und Aktivierung der vom System unterstützen OpenGL Erweiterungen bietet

Einlesen einer Textdatei geschrieben werden. Es dauerte insgesamt etwa 55 Minuten, bis der erste Shader ausgeführt werden konnte.

Da es für Sh bislang keine Tutorials gab, ist eine genaue Zeitangabe schwierig. Aus Mangel an Dokumentation insbesondere für Windows-Systeme und infolgedessen vieler irreführender Fehlermeldungen betrug die Zeit bis zum allerersten lauffähigen Shader etwa anderthalb Tage. Verantwortlich für diese Zeit waren insbesondere die zum Kompilieren und Ausführen von Sh notwendigen Projekt-Einstellungen. Dieser Wert kann jedoch kaum als Vergleichswert dienen. Daher sollen die in Kapitel 3 dieser Arbeit enthaltenen Tutorials die Basis für den Vergleichs-Test mit Sh bilden. Anhand dieser Anleitung war es der Testperson möglich, Sh innerhalb von 25 Minuten in das bestehende Rahmenprogramm einzubauen und einen ersten Shader zu implementieren.

Natürlich sind die gemessenen Zeiten lediglich als Anhaltspunkte zu werten, da die Programmiererfahrung und Vertrautheit im Umgang mit der Entwicklungsumgebung sowie mit Shadern diese Zeiten erheblich beeinflussen können. Dennoch erschien der unter Anleitung erfolgte Einstieg in Sh leichter als der in GLSL.

5.5. Sh intern

Einige Punkte in der gegenwärtigen Sh Implementierung und auch Eigenheiten beim Einsatz des Toolkits bedürfen auch außerhalb des Vergleiches mit einer anderen Hochsprache der Erwähnung.

Zum einen sind dabei Mängel in der Implementierung zu nennen. Das Sh Toolkit befindet sich momentan erst in der Version 0.8. Die Versionsnummer ist ein klares Indiz dafür, dass noch weitere Entwicklung notwendig ist. Dazu zählen unter anderem auch eine vollständigere und korrektere Dokumentation. Sehr deutlich wird dies, wenn das 2004 von den Entwicklern Michael McCool und Stefanus Du Toit veröffentlichte Buch[3] über Sh mit dem tatsächlichen aktuellen Stand des Projektes verglichen wird. Dabei erscheint das Buch stellenweise eher wie eine Wunschliste als eine Dokumentation. Viele Funktionen, die im Detail vorgestellt werden, sind noch immer nicht, oder anders als beschrieben implementiert.

Die Funktion *shVsh()* (siehe Kapitel 2.3.) zum Beispiel soll laut Buch abhängig von den Parametern einen Vertex Shader generieren, der die Geometrie transformiert, den Halbvektor, den Sichtvektor, alle Lichtvektoren und die Normalen in allen Koordinatenräumen (Objekt-, Oberflächen- und View-Koordinatensystem) und die Tangenten berechnet. In der tatsächlichen Implementation stimmen die angegebenen Parameter allerdings nicht mit denen im Buch überein. Darüber hinaus lässt sich die Funktion nicht erfolgreich ausführen und es wird ein fehlerhafter Shader generiert.

Die Shader Algebra, die von Sh neu eingeführte und beworbene Technik, mehrere Shader miteinander zu kombinieren, verursacht weitere Probleme, da sie sich nicht erwartungsgemäß verhält. Der Zuweisungs-Operator der Klasse *ShProgram* ist nicht korrekt implementiert, so dass ein Kopieren eines Programms nicht problemlos möglich ist. Dies führt bei der Shader Algebra dazu, dass man einen Shader nicht mit sich selbst verbinden kann. Eine solche Operation kann aber durchaus sinnvoll sein, insbesondere im Bereich der Streaming Shader.

Des weiteren haben sich die Entwickler dafür entschieden, die Register nicht fest an Variablen zu binden. Bei der Kombination von mehreren Vertex oder Fragment Shadern führt dies zu ungewollten Ergebnissen, da es ohne besondere Angaben nicht möglich ist, dasselbe Register in zwei Shadern zu nutzen. Auch temporäre Variablen und Ergebnisse aus vorherigen Berechnungen, wie z.B. der Vertexposition, können nicht weiterverwendet werden. Der *combine*-Operator ist in seiner ursprünglichen Form für Fragment Shader gar nicht anwendbar. Insgesamt ist die momentane Implementierung der Shader Algebra für grafische Shader eher unbrauchbar.

Der eingebaute Optimierer, der bereits in früheren Kapiteln angesprochen wurde, bedarf ebenfalls noch einiger Verbesserungen. So führte er im Test dazu, dass im Zusammenhang mit der Shader Algebra vermeintlich redundanter Code aus den Shadern gelöscht wurde, der jedoch tatsächlich benötigt wurde.

Während diese Defizite die Nutzung von Sh nicht wesentlich einschränken, so sind in der Implementierung der verschiedenen Backends bereits gröbere Mängel zu finden. Das „CC"-Backend, das die Sh Streaming Shader auf der CPU ausführen und so datenabhängiges „load balancing"[24] einfach möglich machen soll, konnte unter Windows nicht zum Laufen gebracht werden, u.a. auch weil keinerlei Anleitung oder Dokumentation vorhanden ist. Unter Linux soll dieses Backend jedoch funktionieren. Das ARB Assembler Backend verursacht bei Streaming Shadern fehlerhafte oder gar keine Ergebnisse, wenn die Anzahl der Daten zu gering oder zu groß ist. Auch im GLSL Backend finden sich einige Fehler. Während sich Streaming Shader zwar korrekt ausführen lassen, so stürzt das Programm dennoch beim Beenden aufgrund fehlerhafter Destruktoren ab[25]. Bei grafischen Shadern kann es bei der Benutzung von *State Binding* (siehe 2.2.4.1. OpenGL States) in bestimmten Fällen sogar zu der Erzeugung von fehlerhaftem Code kommen. Die Qualität des übersetzten Codes ist dabei generell zu bemängeln (siehe 5.6. Performanz). Zudem treten auf manchen Grafikkarten bei der Verwendung von *for*-Schleifen Fehler auf, bei denen der Shader nicht mehr ausgeführt werden kann und das Programm einfriert.

Insgesamt lässt sich Sh dennoch weitestgehend problemlos sowohl für GPGPU als auch für grafische Zwecke einsetzen. Jedoch ist die Zukunft des Toolkits fraglich. Da es sich um ein Open Source Projekt handelt, kann jedermann zur Entwicklung beitragen. Allerdings ist eine größere Community nötig, um ein solches Projekt am Leben zu halten. Die ursprünglichen Schöpfer jedoch haben sich Ende August 2006 offiziell von dem Projekt zurückgezogen. Aus dem Forschungsprojekt ist mittlerweile die Firma *RapidMind*[22] entstanden, die ein kommerzielles Produkt auf Basis von Sh erschaffen haben. Die sogenannte *RapidMind Development Platform* scheint sich allerdings nach ersten Eindrücken einer Evaluierungsversion nur in wenigen Dingen[26] von Sh zu unterscheiden. Ob die Zahl der Sh-Anhänger groß genug ist, um das Open Source Projekt trotzdem fortzuführen, wird sich zeigen.

5.6. Performanz

Im Folgenden soll die Performanz im direkten Vergleich mit GLSL untersucht werden. Dazu wird ein Geschwindigkeitsvergleich zweier Shader mit gleichem Inhalt angestellt.

Da Sh keine eigene Shader-Hochsprache ist, sondern nur eine Meta-Sprache, die in GLSL übersetzt und ausgeführt wird, ist Sh in der Folge grundsätzlich nicht schneller oder langsamer als die Vergleichssprache. Jedoch kommt zu dieser theoretischen Tatsache hinzu, dass Sh den eigenen Programmcode zunächst einmal übersetzen und entsprechenden GLSL Code erzeugen muss. Während der Übersetzungsvorgang selbst zeitlich kaum ins Gewicht fällt und daher hier nicht weiter berücksichtigt werden wird, so nimmt doch die Qualität der Übersetzung entscheidenden Einfluss auf die Geschwindigkeit des resultierenden Shaders.

Im Test wurde ein Fragment Shader in Sh und auch in GLSL implementiert, der innerhalb einer *for*-Schleife mehrere Vektoroperationen durchführt. Listing 5.1 und 5.2 zeigen den Code der Shader in Sh und in GLSL Die Anzahl der Schleifendurchläufe wurde schrittweise erhöht, bis auf 250 Durchläufe. Aufgerufen wurde der Shader durch das Zeichnen eines einfachen Quads. Um die

24 datenabhängiges load balancing: abhängig von der Anzahl der zu berechnenden Daten wird die Entscheidung getroffen, ob die CPU oder die GPU die Berechnungen durchführen soll

25 Absturz nach Ausführung: Dieser Fehler kann unterdrückt werden, siehe Kapitel 3.2 für die Lösung

26 vor allem Variablentypen und Namespace- und Makro-Bezeichner wurden abgeändert – teilweise nur von SH in RM (=RapidMind); ausserdem wurde das ARB Backend aufgegeben

Zeitmessung zu verbessern, wurden jeweils zwischen 50 und 10000 Passes durchgeführt. Als Hardware kam eine NVidia 7800 GS (AGP Version mit 16 Pipelines) zum Einsatz.

```
ShProgram frag = SH_BEGIN_FRAGMENT_PROGRAM
{
     ShInputColor3f inCol;
     ShOutputCoor3f outCol;

     ShVector3f a(1.1f, 1.2f, 1.3f);
     ShVector3f c(2.1f, 2.2f, 2.3f);
     ShAttrib1f b = 1.0f;

     ShAttrib1i i = 0;
     SH_FOR(i = 0, i < 250, i++)
     {
         b = length(a);
         c = normalize(c);
         a = cross(a, c);
     } SH_ENDFOR;

     outCol = inCol * b;
} SH_END;
```

Listing 5.1: Test-Shader in Sh

```
void main()
{
     vec3 a = vec3(1.1, 1.2, 1.3);
     vec3 c = vec3(2.1, 2.2, 2.3);
     float b = 1.0;

     for(int i = 0; i < 250; i++)
     {
         b = length(a);
         c = normalize(c);
         a = cross(a, c);
     }

     gl_FragColor = gl_Color * b;
}
```

Listing 5.2: Test-Shader in GLSL

Der Code der Shader in GLSL und Sh ist beinahe identisch. Die Ergebnisse der Geschwindigkeitsmessung zeigen jedoch deutliche Unterschiede. Der von Hand implementierte GLSL Shader brauchte im Test durchschnittlich 37% weniger Zeit als Sh, bzw. genauer gesagt, als der von Sh erzeugte Code.

Der Grund für die Differenz in den Geschwindigkeiten liegt in dem von Sh erzeugten Code. Dieser enthält viele unnötige temporäre Variablen und Kopieroperationen. Die große Anzahl an temporären Variablen ist auffällig. Statt 4 Variablen (ohne die Ein- und Ausgabe Variablen) nutzt der erzeugte GLSL Shader 9. Der Aufruf der Funktion *length()* wurde nicht in einen Aufruf der gleichnamigen GLSL Funktion übersetzt, sondern stattdessen wird zusätzlicher Code zur Berechnung der Länge eingefügt. Somit wird verhindert, dass der Vorteil von Hardware-unterstützten Funktionen genutzt werden kann.

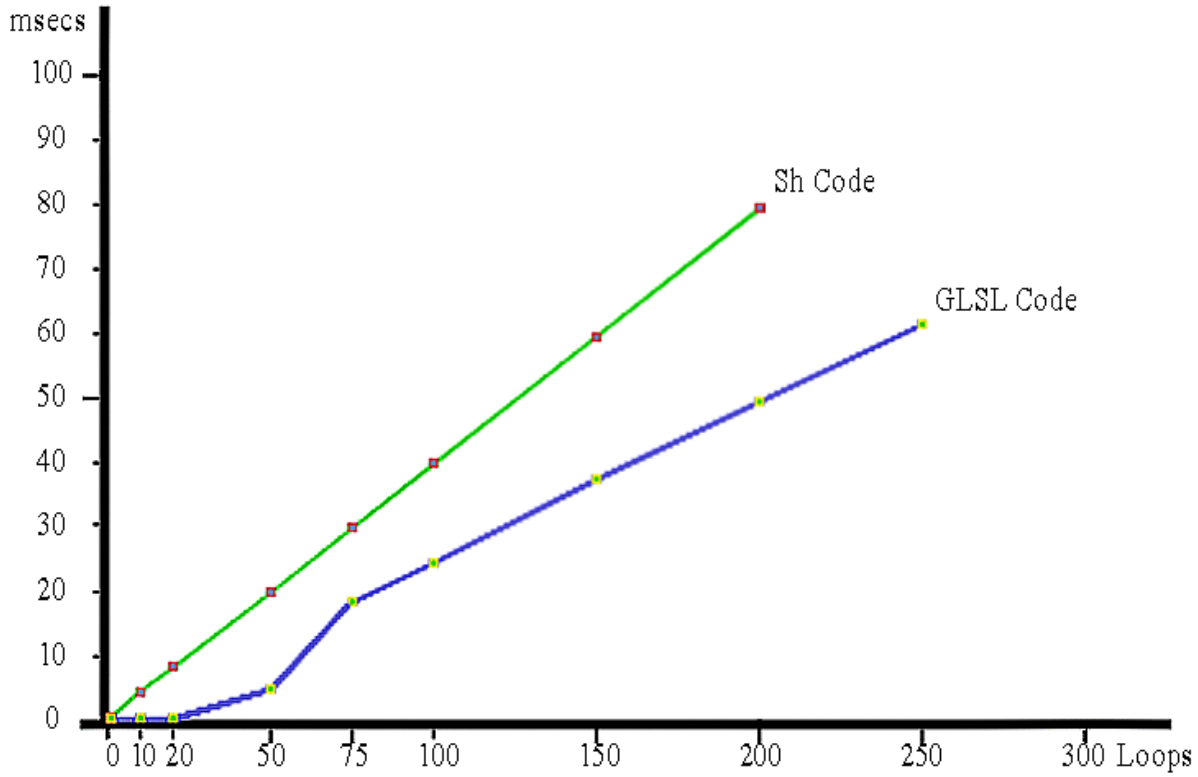

Abbildung 14: Geschwindigkeitsvergleich zwischen Sh und GLSL

Sehr negativ dürfte sich auch der zusätzliche und redundante Aufruf der *normalize()* Funktion auswirken, der beim Übersetzen durch Sh entstanden ist, da es sich hierbei um eine kostspielige Funktion handelt. Listing 5.3 zeigt den durch Sh übersetzten Code – zur Erleichterung des Lesens mit abgeänderten Variablennamen.

Am stärksten auf die Geschwindigkeit drückt allerdings, dass die *for*-Schleife in eine *while*-Schleife übersetzt wird. Nicht nur die Auswertung der Schleifenbedingung erscheint dabei besonders umständlich. Da die Anzahl der Schleifendurchläufe bereits vor dem Kompilieren bekannt ist, kann der GLSL Compiler eine *for*-Schleife komplett entrollen, d.h. die Instruktionen des Schleifeninhaltes entsprechend oft hintereinander schreiben. Dadurch können Sprünge innerhalb des Codes und auch die Auswertung der Schleifenbedingung vermieden werden. Dies ist nur möglich, sofern es die erlaubte Maximalzahl der Instruktionen zulässt. Bei einer *while*-Schleife ist eine Entrollung jedoch aufgrund der dynamischen Abbruchbedingung generell nicht möglich.

Daher ist der handgeschriebene GLSL Code in den Bereichen, in denen die Anzahl der Schleifendurchläufe so niedrig ist, dass die Schleife entrollt werden kann, um ein Vielfaches schneller als der Sh Code. Eine Schleifen-Entrollung kann in Sh nur manuell emuliert werden. Setzt

man statt einer *for*-Schleife mit einer festen Schleifenanzahl einfach die Instruktionen aus dem Schleifenkörper in entsprechend häufiger Anzahl von Hand hintereinander in den Sh Shader, so erzielt dieser auch tatsächlich ähnliche Ergebnisse wie der GLSL Shader.

Im Test erzeugte Sh nach dem Verschieben der Zeile „b = length(a);“ vom oberen an das untere Ende der Schleife sogar neben dem zusätzlichen *normalize()* Aufruf auch noch einen zweiten, zusätzlichen Aufruf der Funktion *cross()*. Diese redundanten Operationen werden durch den Sh-eigenen Optimierer verursacht. Wird der Grad der Optimierung auf 0 gesetzt, also die Optimierung gänzlich unterdrückt, werden keine zusätzlichen Aufrufe generiert. Zudem legen weitere Tests nahe, dass eine solche Verdopplung vornehmlich in Zusammenhang mit Schleifen auftritt.

```
void main()
{
    vec3 a;
    vec3 c;
    float b;
    float i;
    float temp_i;
    float temp;
    vec3 c2;
    const vec3 temp_a = vec3(1.1, 1.2, 1.3);
    const vec3 temp_c = vec3(2.1, 2.2, 2.3);

    a = temp_a;
    c = temp_c;
    b = float(1);
    i = float(0);

    // evaluate loop condition:
    temp_i = float(i < float(5));

    while (bool(temp_i)) {

        // execute loop body:
        temp = float(dot(a, a));
        b = sqrt(temp);
        c2 = normalize(c);
        c = normalize(c);
        a = cross(a, c2);
        i = i + float(1);

        // evaluate loop condition:
        temp_i = float(i < float(5));
    }

    gl_FragData[0].xyz = gl_Color.xyz * vec3(b);
}
```

Listing 5.3: von Sh generierter GLSL Code

In der Implementierung des GLSL Backends, das für die Übersetzung verantwortlich ist, kann somit noch Geschwindigkeit gewonnen werden. Insgesamt ist eine Implementierung eines Shaders von Hand in aller Regel deutlich performanter als der von Sh erzeugte Code, zumindest in der aktuellen Version von Sh.

Neben der teilweise unglücklichen Übersetzung der Shader Programme birgt das Meta-Sprachen Konzept von Sh noch weitere Geschwindigkeitsfallen: Dadurch, dass Sh so vieles vor dem Nutzer versteckt, können ungewollt und unerkannt sehr unperformante Programme entstehen, die sich nur vermeiden lassen, wenn der Anwender sich Einblick in die genauere Funktionsweise von Sh verschafft. Werden beispielsweise die in den Shadern genutzten uniformen Variablen im Hauptprogramm ebenfalls wie gewöhnliche Variablen eingesetzt, für Zwischenergebnisse etc., so entsteht ein Overhead, da bei jeder Änderung der Variable der Wert im Hintergrund von Sh unnötig zur Grafikkarte hochgeladen wird.

Auch an anderen Stellen besteht die Gefahr, versehentlich und unwissentlich die Erzeugung sehr unperformanter Shader Programme zu verursachen. Abhängig von der Hardware generiert Sh zusätzlichen Shader Code, um die gleichen Funktionen auf allen Karten zu gewährleisten, z.B. beim Texturzugriff oder trigonometrischen Funktionen. Da die GPU nur 4er-Tupel direkt unterstützt, Sh aber Tupel beliebiger Größe, kommt es beim Einsatz solcher Tupel in einem Shader ebenfalls eventuell zur Erzeugung von zusätzlichem Code. Der Nutzer erhält darüber keine Rückmeldung und kann somit auch nicht zur Vermeidung dieses Overheads eingreifen.

Wegen all dieser Nachteile ist Sh für einen Einsatz für grafische Zwecke im professionellen Bereich eher eingeschränkt nutzbar. Die Stärken des Toolkits, wie das automatisierte Parameter-Update, können jedoch in der Entwicklung, insbesondere beim Prototyping, von Interesse sein. Für den späteren Einsatz ist eine manuelle Optimierung zumindest in Performance Produkten unumgänglich. Im Bereich GPGPU kann je nach Anwendung der Einsatz von Sh auch danach noch sinnvoll sein, da das Toolkit zum einen gerade im Bereich Datenaufbereitung viel Arbeit abnimmt und die Programme gleichzeitig für viele verschiedene Systeme kompatibel hält.

6. Resume

Mit Sh wurde ein erster Versuch unternommen, das Konzept der Metaprogrammierung für die GPU umzusetzen und die Programmierung von Shadern zu vereinfachen. Dabei sind sowohl die Abstraktion als auch die Automatisierungen gut gelungen.

Bislang unterstützt das Toolkit jedoch nur zwei Backends und ausschließlich die OpenGL Grafik API, was ihren Einsatzbereich deutlich einschränkt. Zudem sorgen Mängel in der Implementation für eine Performanz, die deutlich unter der der vergleichbaren spezifischen Hochsprachen liegt. Dies schließt einen Einsatz in kommerziellen Produkten, die auf die neueste Technologie setzen und auf Geschwindigkeit optimiert werden müssen, wie z.B. Computerspiele, beinahe vollständig aus.

Für das Entwickeln neuer Techniken, also Prototyping, ist Sh jedoch außerordentlich gut geeignet, da besonders die Zahl der uniformen Variablen leicht und ohne Aufwand geändert werden kann. Im Anschluss kann der von Sh generierte Code als Ausgangsbasis für die im finalen Programm zum Einsatz kommenden Shader dienen. Aufgrund der starken Verbreitung von Cg würde ein Backend für diese Sprache oder für HLSL das Sh Toolkit für die Industrie vermutlich interessanter machen.

Ob es jedoch weitere Versionen von Sh geben wird, in denen die Mängel der derzeitigen Version behoben und neue Erweiterungen hinzugefügt werden, ist fraglich. Nachdem sich die ursprünglichen Schöpfer von der Entwicklung zurückgezogen haben, ist die Zukunft des Projekts ungewiss.

Als nachteilig, vor allem in großen Projekten, könnte sich die Einbettung in den Hauptcode erweisen, da diese nach jeder Änderung im Shader ein erneutes Kompilieren des Programms erfordert. Dies ist allerdings beim Einfügen zusätzlicher Variablen, was in der Entwicklungszeit häufig vorkommt, wegen des benötigten "glue codes" mit herkömmlichen Hochsprachen ebenfalls erforderlich. Zudem lässt sich durch sinnvolle Kapselung der Shader der Kompilieraufwand reduzieren.

Die von Sh eingeführte Shader Algebra hat sich aufgrund der vielen Einschränkungen für grafische Shader als weitestgehend unbrauchbar herausgestellt. Und auch ihr Nutzen im Zusammenhang mit Streaming Shadern ist fraglich.

Für die Entwicklung und den Einsatz von Streaming Shadern im Sinne der GPGPU stellt Sh ein mächtiges Werkzeug dar. Insbesondere für Programmierer, die nur über begrenzte Erfahrungen mit Grafik APIs verfügen, bietet Sh die Möglichkeit, dennoch die GPU als Co-Prozessor zu nutzen, da Sh sich um sämtliche Kommunikation mit der Grafik API kümmert und ein einfaches Interface anbietet.

Insgesamt sollten sich Shader mit Sh schneller realisieren lassen, da dem Programmierer viel Arbeit abgenommen wird. Auch macht Sh es Einsteigern leicht, erste Schritte in der Shaderprogrammierung zu versuchen, da viele Hindernisse, wie zum Beispiel das Einladen von Textdateien, Variablendeklarationen und "glue code" aus dem Weg geräumt werden und die frustrierende Fehlersuche durch das Abfangen von Tipp- und Syntaxfehlern zur Kompilierzeit minimiert wird.

Literaturverzeichnis

[1] Wikipedia - Die freie Enzyklopädie (2006) : GPGPU. http://de.wikipedia.org/wiki/GPGPU; Abruf: Juni 2006

[2] Cg (2006) NVidia : FAQ - Häufig gestellte Fragen. http://www.nvidia.de/object/cg_faq_de.html; Abruf: September 2006

[3] McCool, Michael; Du Toit, Stefanus (2004) : Metaprogramming GPUs with Sh. AK Peters Ltd, Wellesley, MA. ISBN 1-56881-229-9

[4] Zurab, Khadikov (2005/06) : OpenGL Display Lists. Referat, Universität Freiburg

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

[6] DGL Wiki - Freies OpenGL Wissen für alle! (2005) Jagdmann, Dirk : MipMaps. http://wiki.delphigl.com/index.php/MipMaps; Abruf: August 2006

[7] Sh Embedded Metaprogramming Language (2003) : A high-level metaprogramming language for modern GPUs. http://www.libsh.org; Abruf: August 2006

[8] Neogrid, das innovative EDV Lexikon (2006) Müller, Michael : SSE, SSE2 und SSE3. http://www.neogrid.de; Abruf: August 2006

[9] Tom's Hardware Guide (2006) THG Lexikon : Shader. http://www.thgweb.de/lexikon/Shader; Abruf: August 2006

[10] Trolltech (2006) : Qt Homepage. http://www.trolltech.com/products/qt; Abruf: September 2006

[11] Qt Reference Documentation (2005) Trolltech : Open Source Edition. http://doc.trolltech.com/4.0/index.html; Abruf: September 2006

[12] Volker Wiendl (2006) : Qt 4.x unter Windows und Visual Studio .NET 2003. Tutorial,

[13] Cal3D (2004-2006) the Gna! people : 3d character animation library. https://gna.org/projects/cal3d/; Abruf: September 2006

[14] GameDev.Net (1999-2006) Community : Diskussionsforen. http://www.gamedev.net/community/forums/; Abruf: August 2006

[15] OpenGL.org (2003-2006) Discussion & Help Forums : OpenGL Shading Language. http://www.opengl.org/cgi-bin/ubb/ultimatebb.cgi?ubb=forum;f=11; Abruf: August 2006

[16] 3DLabs (2006) Developer Relations : OpenGL Shading Language. http://developer.3dlabs.com/openGL2/index.htm; Abruf: August 2006

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

[18] Shreiner, Fave (2006) : OpenGL Programming Guide : The Official Guide To Learning OpenGL, Second Edition. Addison-Wesley, Upper Saddle River, NJ. ISBN 0-321-33573-2

[19] Dimitri van Heesch (1997-2006) DoxyGen : A Documentation System. http://www.stack.nl/~dimitri/doxygen/index.html; Abruf: August 2006

[20] GPGPU - General Purpose Computation on GPUs (2006) Forum Index : Sh Discussion. http://www.gpgpu.org/forums/viewforum.php?f=8; Abruf: September 2006

[21] Lighthouse 3D - A Resource For 3D Programmers (2005) Fernandes, António Ramires : GLSL Tutorial. http://www.lighthouse3d.com/opengl/glsl/; Abruf: August 2006

[22] RapidMind (2006) : RapidMind Development Platform. http://www.rapidmind.net/; Abruf: August 2006