View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Online Publikationen der Universität Stuttgart

Institut für Parallele und Verteilte Systeme

Abteilung Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Studienarbeit Nr. 2436

Dynamische Ausführung von Positionstransformationen mittels OpenGL ES 2.0 Shaderprogrammen

Felix Zehender

Studiengang:	Informatik
Prüfer:	Prof. Dr. Kurt Rothermel
Betreuer:	Dipl.-Inf. Stephan Schnitzer
begonnen am:	28.08.2013
beendet am:	27.02.2014
CR-Klassifikation:	D.1.7, D.3.4, I.3.7

Kurzfassung

Wegen der Forderung nach Isolation und Performanz in eingebetteten Systemen („embedded Systems“) ist ein Konzept notwendig, dass dynamisch bei laufzeitkritischen Grafikanwendungen die Positionstransformation von Vertices mit Hilfe von Vertex Shadern, während der Laufzeit übernimmt. Daher wurde in dieser Arbeit ein Konzept entwickelt, welches die Position einzelner Vertices berechnet, bevor die kompletten Rendering Befehle ausgeführt werden. Dies ist nötig um abschätzen zu können, ob die Deadlines von sicherheitskritischen Anwendungen eingehalten werden. Dazu wird in dieser Arbeit der Vertex Shader während der Laufzeit mit Hilfe von LLVM kompiliert und berechnet, für einen gegebenen Vertex dessen Position.

Inhalt

1 Einleitung

In der heutigen Zeit wirken sich viele Beschränkungen auf das Verhalten eines Programmes aus. In sogenannten eingebetteten Systemen („embedded systems“) gibt es zum Beispiel Einschränkungen im Bauraum, dem Energieverbrauch oder der vorhandenen Hardwareleistung. Trotz solchen Einschränkungen, die auf ein Programm wirken, soll unter anderem die Performanz und die Isolation aufrecht erhalten bleiben. Hinsichtlich dieser Probleme wird in dieser Arbeit ein Konzept vorgestellt, welches die Positionstransformation des Vertexshaders eines OpenGL ES 2.0-Programms dynamisch zur Laufzeit emuliert. Durch die Emulation während der Laufzeit ergeben sich viele Vorteile für laufzeitkritische Grafikanwendungen. Jenen kann es nun zum Beispiel ermöglicht werden, schon vor dem Berechnen des kompletten Rendering Befehls abzuschätzen, ob es sinnvoll ist die Befehle auszuführen [SC13].

Als Grundlage dieser Arbeit wurde Mesa3D [SO14a] gewählt, welches eine funktionsfähige Implementierung von OpenGL ES 2.0 mit Hilfe der Software-Rendering bereitstellt und Shader mittels LLVM kompiliert. Da es in Mesa3D nicht möglich ist getrennt die Position des Vertex mittels des Vertexshaders zu berechnen ohne ihn sofort zu zeichnen, müssen einige Veränderungen am Quellcode vorgenommen werden. In Kapitel 3 werden zunächst einige Grundlagen angesprochen, die ein Einsteigerwissen über Mesa3D und dessen Bausteine vermitteln sollen. Um zu verstehen, wie Mesa3D intern arbeitet, wird in Kapitel 4 das Vorgehen der Shadertransformation bis zum LLVM-Code beschrieben. In diesem Kapitel wird gezeigt, wie Mesa3D mit Shadern arbeitet und wie diese vorbereitet werden müssen um später mittels LLVM kompiliert werden zu können. Außerdem werden noch einige Bausteine von LLVM untersucht, welche vertiefendes Wissen vermitteln sollen. In Kapitel 5 wird das in dieser Arbeit erstelle Konzept eines Programmes dargestellt, dass durch Verwendung von Mesa3D Teilen unabhängig die Position eines Vertex mittels des Vertexshaders berechnet. Hier wird anhand eines Beispiel Vertexshaders dargestellt, wie sich das Programm in den einzelnen Abschnitten verhält und in welchen Schritten der Shader bearbeitet wird. Zusätzlich wird auf interne Mesa3D Strukturen eingegangen, welche Normalerweise für die Berechnung der Position verwendet werden und erläutert inwiefern diese für diese Problemstellung relevant sind. Zum Schluß wird die Vorgehensweise von LLVM, die Mesa nutzt und deren Konstrukte untersucht und erklärt, wie sich die einzelnen Funktionen während der Erstellung des ausführbaren LLVM-Codes verhalten. In Kapitel 6 werden angesprochene Themen nochmal zusammenfassend gegliedert, kurz erörtert und ein Ausblick auf weitere Vorgehensweisen und Nutzung des in dieser Arbeit vorgestellten Konzeptes gegeben.

2 Grundlagen

Die Grundlagen für diese Arbeit bauen auf Mesa3D (Mesa) [SO14a], einem Open-Source Projekt, auf. Dabei nutzt Mesa3D viele andere Open-Source Projekte, wie die „Direct Rendering Infrastructure“ (DRI), welches ein Framework für den direkten Zugriff unter X Windows Systemen auf die Grafikhardware darstellt. In diesem Kapitel werden Bausteine vorgestellt, welche als Grundlage für das erstellte Konzept dienen. Dabei werden Mesa, OpenGL ES 2.0, die LLVM Compiler Infrastructure Projekt und die Shaderprogramme angesprochen.

2.1 OpenGL ES 2.0

OpenGL ES 2.0 ist eine plattformübergreifende API für 2D und 3D Grafiken. Realisiert wird dies durch eine programmierbare 3D Grafikpipeline und ist bezüglich der OpenGL 2.0 Spezifikation definiert. Dabei ersetzt OpenGL ES 2.0 die schon vorhandene „Fixed Function Pipeline“ (FFP) durch eine programmierbare Pipeline, welche größtenteils programmgesteuert ist. Die Pipeline wird zum generieren oder rendern von Bildern benutzt und besitzt mehrere Stufen [BLW08]. In Abbildung 1 wird so eine programmierbare Pipeline dargestellt. Hier wurden die fixen Teile der „Fixed Function Pipeline“ durch frei programmierbare Shader ersetzt, welche durch die orangenen Felder dargestellt werden. So wird aus dem Transformationsteil der FFP, dem „Transform and Lighting“, der Vertex Shader und aus der Fragment Pipeline, welche sich aus „Texture Enviroment“, „Colour Sum“, „Fog“ und „Alpha Test“ in der FFP zusammensetzt, der Fragment Shader. Durch die Einführung solcher Shader werden Berechnungskosten und Energiekosten gesenkt. Open GL ES 2.0 bietet die Möglichkeit, Shader und Programmobjekte zu erzeugen und diese mit der „OpenGL ES Shading Language“ [SI09], welche dieselben Funktionen wie OpenGL 2.0 bietet aber für eingebettete Systeme angepasst wurde, zu beschreiben. Dabei kombiniert OpenGL ES 2.0 die „OpenGL Shading Language“ mit einer modernisierten API aus OpenGL ES1.1 [KH14a].

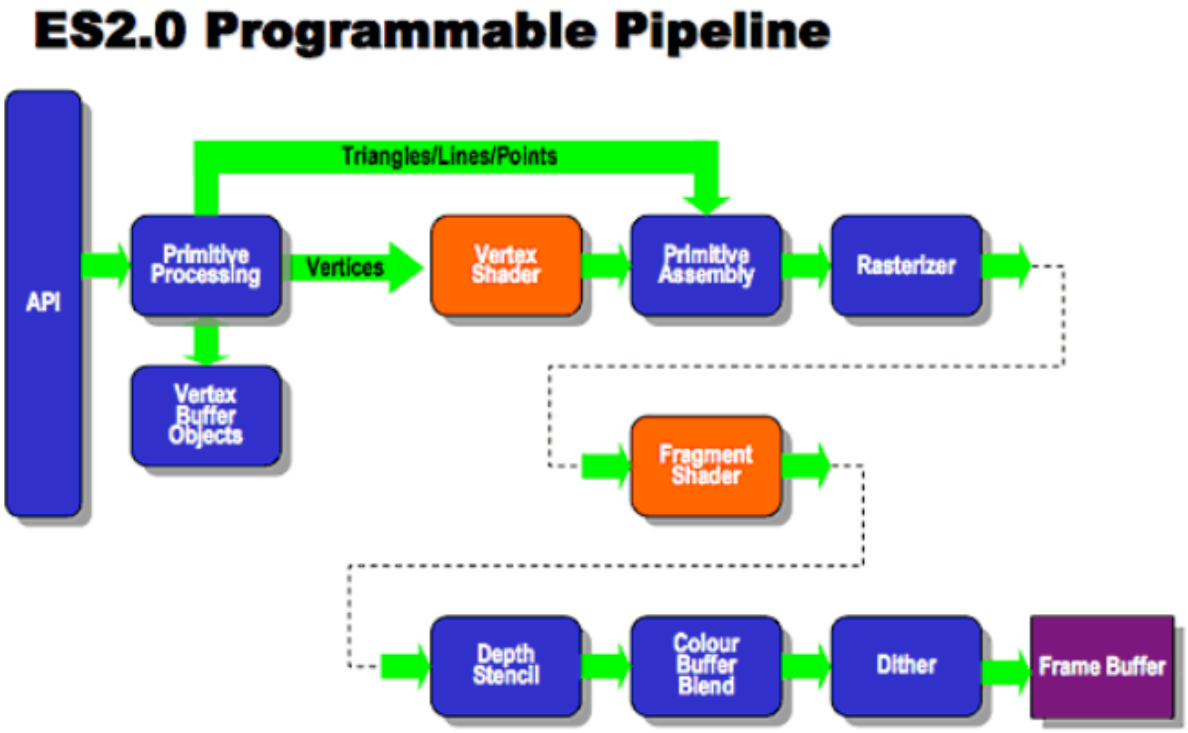

Abbildung 1, ES2.0 Programmierfähige Pipeline [KH14b]

2.2 OpenGL ES 2.0 Shaderprogramme

Die „OpenGL ES Shading Language“ [SI09] besteht aus zwei sehr eng verwandten Programmiersprachen. Zum einen gibt es die für den Vertex Shader und zum anderen die für den Fragment Shader. Diese Sprachen werden zum Erzeugen des jeweils benötigten Shaders für den Prozessor der Pipeline genutzt. Allerdings unterscheidet die „OpenGL ES Shading Language“ nicht differenziert zwischen diesen beiden, sondern bietet sie als eine einzelne Programmiersprache an, die durch ihren Zustand unterschieden wird [SI09].

Die Shader werden zuerst einzeln kompiliert, bevor Vertex- und Fragment Shader verknüpft werden. Aus der GLSL ES Spezifikation ergeben sich 9 Phasen, die den Logischen Kompilierungsablauf darstellen [SI09]:

1. Wenn ein Shader aus mehreren Strings aufgebaut ist, werden diese untereinander verkettet um einen einzelnen großen „Shaderstring“ zu bilden.

2. Der verkettete Input String wird in eine Sequenz von Präprozessor Tokens übersetzt, welcher ein Symbol darstellt, das durch eine Grammatik erstellt wurde. Diese Tokens beinhalten Präprozessor Nummern, Kennungen und kürzen Kommentare des Input Strings auf ein Leerzeichen. Eine Kennung ist dabei ähnlich wie eine Kennung in C, welche eine Sequenz von Buchstaben, Unterstriche und Zeichen sind. Dabei beginnt jede Kennung mit einem Buchstaben oder Unterstrich [FS14]. Die Präprozessor Nummer beschreibt eine Menge aus allen normalen Integer und Floating Point

Konstanten. Der Sinn dahinter ist, dass man den Präprozessor vor der Komplexität der numerischen Konstanten trennt [FS14].

3. Der Präprozessor wird auf den Tokens ausgeführt. Dabei werden zuvor festgelegte Richtlinien ausgeführt und eine Makroexpansion durchgeführt. Solche Makroexpansionen beschreiben dabei das Mappingverfahren beim Aufruf von Makros. Dabei werden die zuvor definierten Makros an die Stellen an denen sie benutzt werden kopiert.

4. Die ausgeführten Präprozessor Tokens werden in normale Tokens übersetzt. Der Unterschied zwischen Präprozessor Tokens und normalen Tokens liegt darin, dass Tokens in der Präprozessorphase nicht zwischen Schlüsselwörtern und Symbolen unterscheidet, sodass Schlüsselwörter nur Symbole sind. Ein weiterer Unterschied liegt darin, dass numerische Werte anders behandelt werden. So erkennt ein Präprozessor Token nicht den Unterscheid zwischen „5“ und „5X“, der normale Token aber schon.

5. Leerzeichen und Zeilenumbrüche des Input Strings werden verworfen.

6. Die Syntax des Strings wird mit Hilfe der GLSL ES Grammatik auf Korrektheit analysiert.

7. Das Ergebnis der Syntaxprüfung wird mit Hilfe von semantischen Regeln der GLSL ES Grammatik auf Korrektheit überprüft.

8. Die bearbeiteten Vertex- und Fragment Shader Strings werden gemeinsam verknüpft. Alle Shader werden nun als ein großer Input Shader String gesehen. Alle Variablen, die in keinem der beiden Shader genutzt werden, werden verworfen.

9. Die Binary wird generiert.

Wie man in den Phasen erkennt, wird der Shader zuerst in Präprozessor Tokens und danach in normale Tokens übersetzt, welche in Grammatiken einem Terminalsymbol entsprechen. Somit ist die Sprache eine einfache Sequenz von Tokens, die aus Schlüsselwörtern, Kennungen, Konstanten oder Operatoren bestehen [SI09].

2.3 Mesa3D

Mesa3D [SO14a] ist ein Open-Source Projekt, welches auf der OpenGL Spezifikation [SA04] aufbaut, um 3D Anwendungen zu rendern. Ursprünglich wurde das Projekt 1993 ohne Namen von Brain Paul ins Leben gerufen, um eine einfache 3D Grafikbibliothek, welche die damals neue OpenGL API nutzen sollte, zu erzeugen. Im Februar 1995 wurde es dann unter dem Namen Mesa 1.0 veröffentlicht. In dieser Arbeit wird die im Februar, 2012 veröffentlichte Version 8.0 verwendet, die nun auch weitere Projekte und Techniken implementiert hat. Einige dieser wurden und werden in dieser Arbeit vorgestellt und etwas

genauer erläutert. Darunter zählt OpenGL ES 2.0 [KH14a], welches eine low-level, leicht gewichtete API für eingebettete Grafiksysteme darstellt, das LLVM Compiler Infrastructure Projekt [CO14], welches eine Sammlung von Compilern und Toolchain Technologien ist, Vertex Shaderprogramme und das Pipelining in Mesa3D [SO14a, SA04, CO14].

2.4 LLVM Compiler Infrastructure Projekt

Das LLVM Compiler Infrastructure Projekt [CO14] wurde als Forschungsprojekt an der Universität von Illinois gestartet. Das Ziel dieses Projektes war es eine moderne, Static Single Assignment [PO06, BM94] (SSA)-basierte Kompilierungsstrategie bereitzustellen. SSA ist dabei eine spezielle Klasse des Intermediate Code (IR), welcher zum Zusammenfassen von Datenabhängigkeiten genutzt wird. SSA wird dabei meistens in sogenannten Just-in-Time (JIT) Compiler genutzt, welche auf High-Level Programmrepräsentationen arbeiten, wie zum Beispiel der von LLVM erzeugte Bit-Code [BM94]. Dabei sollte sie statische und dynamische Kompilierung von willkürlichen Programmiersprachen unterstützen. Heute besteht das Projekt aus vielen Subprojekten, welche auch in kommerziellen und Open Source Projekten genutzt werden, die in dieser Arbeit aber keine weitere Relevanz haben. Solche Subprojekte sind unter anderen Clang [LL14c], dragonegg [LL14d] oder LLDB [LL14e] [CO14, PO06, BM94].

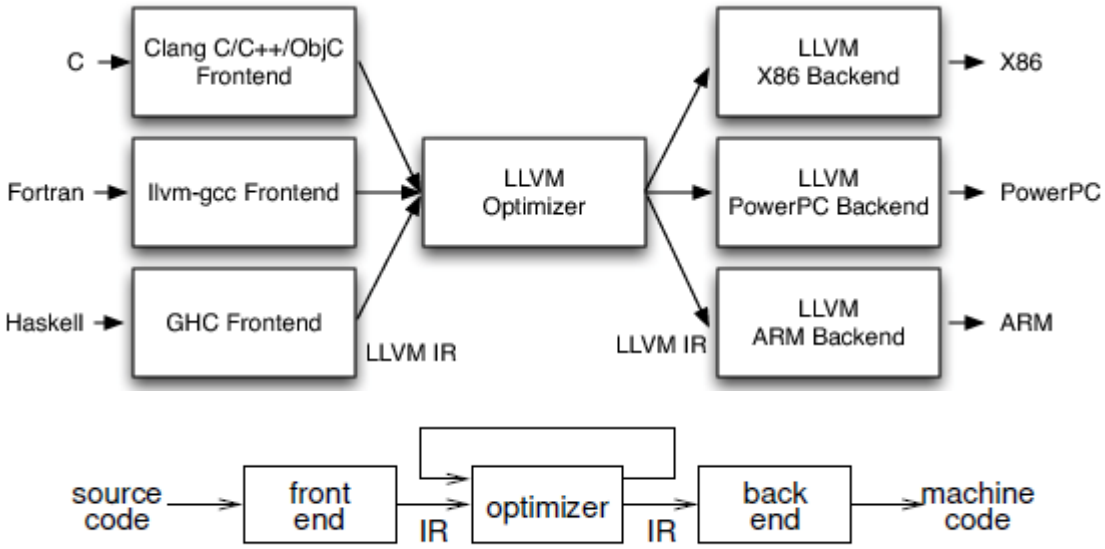

Abbildung 2, Drei-Phasen Design von LLVM [CH14]

LLVM baut, wie man in Abbildung 2 sehen kann, auf einem Drei-Phasen Design auf. Am Eingang (linke Seite Abb. 2) ist ein Frontend zuständig um den Input Code zu parsen und Fehler in ihm zu erkennen. Danach wird der Input Code in LLVM IR Code übersetzt, was meistens über einen Abstract Syntax Tree (AST) geschieht. Dieser stellt eine grammatische Struktur da, welche objektorientierte Programme durch eine Baumstruktur visuell darstellen

kann und speichert dessen relevante Informationen in den Blättern der Baumstruktur [WE97]. Der über den AST erzeugte IR Code durchläuft einige Optimierungen im Optimizer, welcher den Code auf die vom Backend gewünschte Form bringt, um ihn dann in einen Code Generator im Backend zu senden, welcher daraus nativen Maschinen Code erstellt. Durch einfaches Hinzufügen eines neuen Frontends, ist es möglich eine neue Sprache zu unterstützen. Hier ist es dann nicht nötig das Backend oder den Optimizer zu verändern. Der durch den Optimizer erzeugte Code ist sprachenunabhängig und wird durch das demensprechende Ziel-Backend auf die gewünschte Sprache übersetzt [CH14].

3 Ansätze

In diesem Kapitel geht es um die Ansätze die Mesa durchführt, um aus einem Input String einen ausführbaren Shader zu erzeugen. Dabei durchläuft der String verschiedene Phasen, die im letzten Kapitel schon kurz erläutert wurden. Wie genau Mesa diese Phasen durchläuft und welche Hilfsmittel dabei genutzt werden, wird in den nächsten Abschnitten erklärt.

3.1 Shaderprogram Lexing & Parsing

Im letzten Kapitel wurden die Phasen eines Shaderprograms bis zu der Erzeugung des Binaries angesprochen. Es wurde erklärt, dass der Shader in Tokens übersetzt wird um ihn dann anschließend mit der Syntax der Zielsprache auf Korrektheit überprüfen zu lassen. Diese Übersetzung des Input Shaders in eine Sequenz von Tokens geschieht mit Hilfe sogenannter Lexer und Parser. In diesem Kapitel wird erklärt, was solche Lexer und Parser sind und wie sie es schaffen, ein Input String in eine Sequenz von Tokens zu übersetzen. Dabei wird auch das, in Mesa verwendete Tool, Flex genauer betrachtet, welches als Ergebnis den im letzten Kapitel angesprochenen „Abstract Syntax Tree“ erzeugt.

3.1.1 Flex: Der schnelle lexikalische Analysator

Flex [SO08] ist ein Werkzeug, welches benutzt wird um einen Scanner den sogenannten Lexer zu erzeugen. Diese Scanner erkennen durch Textanalysierung das Vorkommen von vorher festgelegten Mustern in Texten. Wenn der Scanner ein solches, vorher festgelegtes Muster erkennt, führt er den dazu vom Nutzer definierten C Code aus. Die Muster, die der Scanner erkennen soll, müssen dabei vorher vom Benutzer festgelegt werden. Dabei legt der Benutzer Regeln für Flex fest, welche in der Form eines Tupels gespeichert werden. Dieser Tupel verbindet das Muster und eine zugehörige Aktion [SO08]. Um den Vorgang besser zu verstehen wird in Abbildung 3 ein einfaches Beispiel dargestellt.

```
        int num_lines = 0, num_chars = 0;

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%

int main()
        {
        yylex();
        printf( "# of lines = %d, # of chars = %d\n",
                num_lines, num_chars );
        }
```

Abbildung 3, Flex Code für zwei Regeln [SO14b]

In Abbildung 3 wird ein möglicher Code-Ausschnitt einer Flex-Implementierung dargestellt. Die „%“-Zeichen markieren den Anfang und das Ende der Regeln, die für bestimmte Muster definiert werden. Der dahinterstehende C-Code, also für dieses Beispiel „++num_lines“ und „++num_chars“, werden bei gefundenem Muster ausgeführt. Als Ergebnis wird bei jedem Fund von einem Zeilenumbruch die Variable „num_lines“ und „num_chars“ erhöht und bei einem Fund von einem anderem Zeichen, repräsentiert durch den regulären Ausdruck „ . “, nur „num_chars“ erhöht[SO14b].

Der Scanner sucht den Text nach den vordefinierten Mustern ab und schaut dort nach Strings, welche die Muster enthalten. Wenn es mehrere Treffer für ein Muster gibt, so entscheidet sich der Scanner für den String, der am meisten Text enthält. Sollte es mehrere Treffer von Strings geben, welche die gleiche Länge an Text enthalten, so entscheidet sich Flex für den String, der zuerst gefunden wurde. Ist ein Treffer gefunden und der Scanner hat sich für den Besten entschieden wird ein sogenannter „Token“ erstellt, welcher nun in der globalen Umgebung der flex-internen Variable „yytext“ verfügbar gemacht wird. Die dazu entsprechende, vorherig definierte Aktion wird ausgeführt und der restliche Input wird auf weitere Treffer gescannt. Jedes Muster hat, wie schon erwähnt, eine dazugehörige Aktion. Diese Aktionen können willkürliche C-Statements sein, die unter anderem spezielle Richtlinien wie „ECHO“, welche den „yytext“ zum Output des Scanners schickt, „BEGIN“, welche den Scanner versetzen kann oder „REJECT“, welche dem Scanner die Anweisung gibt, die zweitbeste Regel welche auf den Input zutrifft zu nutzen, enthalten können. Der durch einen Lexer bearbeitete Code wird als der „geparster“ Code angesehen, [SO08].

3.1.2 AST: Abstract Syntax Tree

Durch die im letzten Abschnitt mit Hilfe von Flex beschrieben Schritte ist es möglich einen „Abstract Syntax Tree“ [JO03], auch kurz AST, zu erstellen. Dabei werden in Flex spezielle Regeln definiert, die aus einem Input Code einen AST erzeugen können. Dieser AST repräsentiert den durch den Input gegebenen Code als Baumstruktur, ohne jedoch syntaktische Feinheiten der Inputsprache zu behandeln. Dadurch wird es möglich verschiedene Sprachen auf ein abstrakteres Level zu bringen indem nur die Semantik, also die Bedeutung des Inputs, betrachtet wird. In Abbildung 4a werden dabei beide Input Codes in der AST-Form gleich interpretiert, ohne die Syntaktischen Feinheiten, wie unter anderem die Zuweisungsregeln durch „:=“ oder „=“ zu beachten. Dabei können solche Bäume durch Hand geschriebene Parser oder wie in Mesa mit Hilfe von Flex erzeugt werden. Um die benötigte feine Gliederung eines solchen ASTs zu erhalten, müssen vor dem Erzeugen des Baumes noch einige Designentscheidungen getroffen werden. Eine solche Entscheidung ist unter anderen, die Gestaltung von binären arithmetischen Operationen. So kann aus einer binären Operation wie „+“, „-“, etc. ein einziger Knoten werden, der die Operationen als Attribute hält oder es kann für jede Operation einen eigenen Knoten erzeugt werden. Diese Entscheidungen werden dabei komplett dem Benutzer überlassen und können je nach Gebrauch oder Vorlieben, vorteilhafter sein. Wurden die Entscheidungen über die Granularität des Programms getroffen, kann der AST nun in die Zielsprache übersetzt werden. Dies geschieht durch das Durchlaufen der Baumstruktur. Häufig kann die Zielsprache nicht direkt abgeleitet werden, da jede Sprache ihre eigenen syntaktischen Feinheiten besitzt. Deswegen ist es sinnvoll den AST zuerst in eine Zwischenform zu bringen (Intermediate Code: IR). Solche Zwischenformen bringen durch ihre noch abstrahiertere Form den Vorteil der Erhöhung der Laufzeitperformanz. Diese wird oft durch die Ausführung des IR-Codes auf Maschinenebene gewonnen. Außerdem führt die Abstraktion auf den Intermediate Code, durch ihren Aufbau, zu einer einfacheren Weiterverarbeitung in die gewünschte Zielsprache [JO03, FR14].

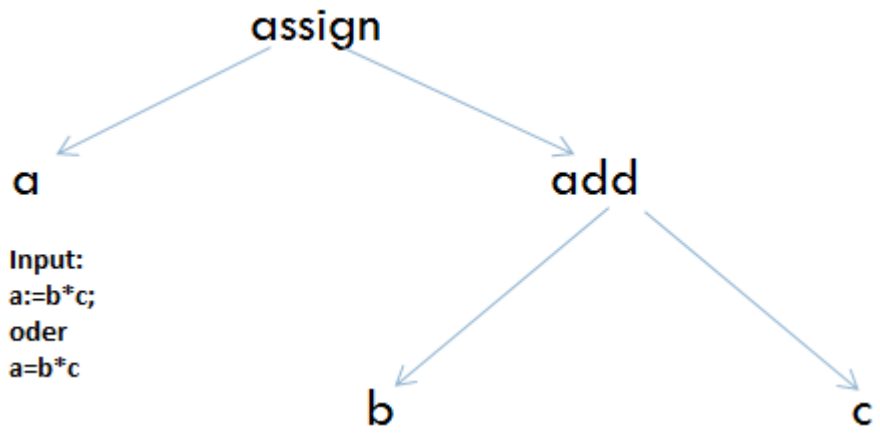

Abbildung 4a, Input Code - Abstract Syntax Tree [LO14]

Mesa unterscheidet zwischen zwei Zwischenformen in die der Input Code überführt wird. Zuerst wird der aus dem Input Code erzeugte AST in einen „Higher Intermediate Code“ (HIR) gebracht, indem noch Matrixoperationen, Strukturanmerkungen des Codes und

Ähnliches vorhanden sind. Auf diesen HIR werden nun noch einige Lowerings ausgeführt, welche zum Beispiel die Matrixoperationen auf Punktmultiplikationen aufsplitten und dadurch den „Lower Intermediate Code“ (LIR) erzeugen [LO14].

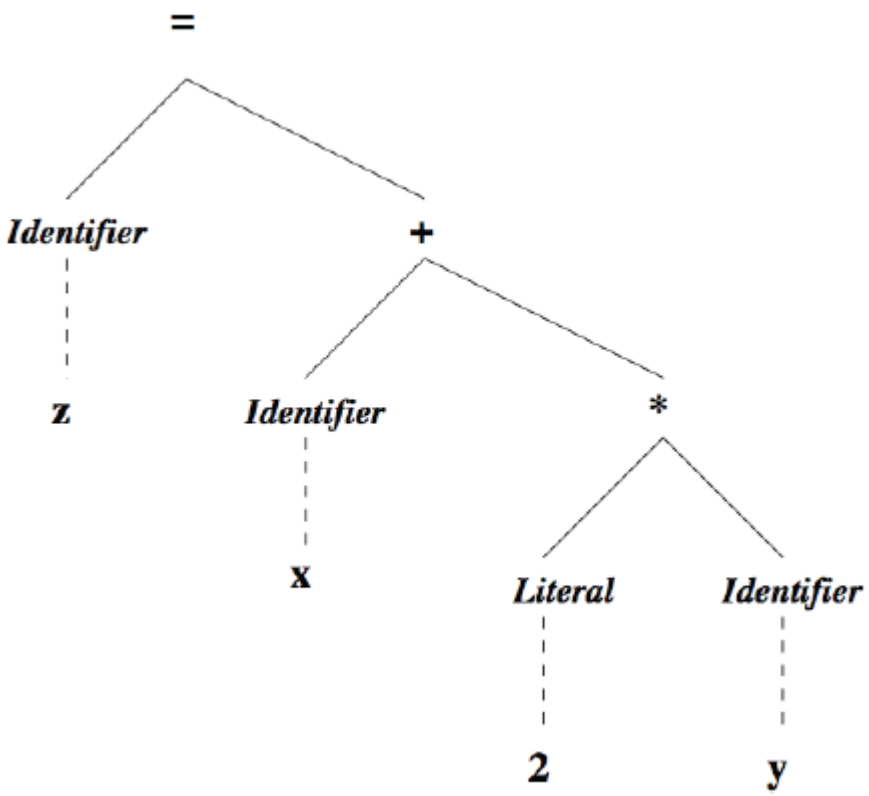

Abbildung 4b, Abstract Syntax Tree für $z = 2 * y$ [WT06]

Ein weiteres Beispiel eines Abstract Syntax Tree ist in Abbildung 4b durch $z = 2 * y$ gegeben. Hier werden nur die wichtigsten Elemente einer Sprache gespeichert. Somit kann die Binary, die im letzten Schritt der Phasen aus [SI09] generiert wird, erzeugt werden. In Abbildung 5 wird nun aus den einzelnen Blättern des Abstract Syntax Tree die Binary erzeugt, indem für jeden Block ein Identifier oder Operator hinzugefügt wird. Aus dem Aufbau der Binary ergibt sich der sogenannte 3-Adress-Code [AR13], welche eine typische Intermediate Code Darstellung ist. Dieser wird so genannt, weil sich die Binary wie in Abbildung 5 immer aus drei Komponenten zusammensetzt: dem „Operator“ und zwei „Terms“.

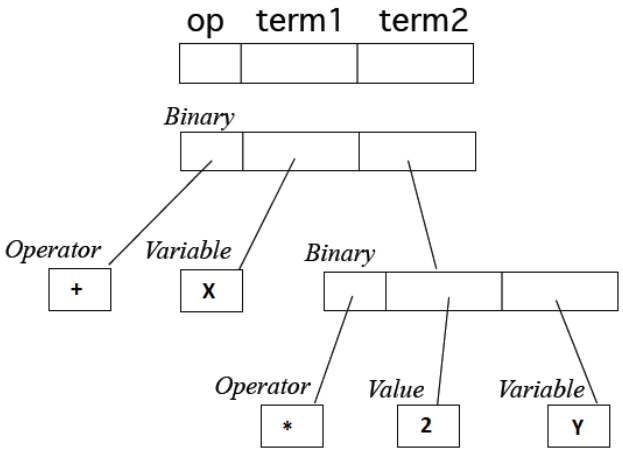

Abbildung 5, Aufbau 3-Adress-Code Binary für $z = 2 * y$ [AR13]

3.2 Kompilieren des Shaders

Im letzten Abschnitt wurde erklärt, wie durch das Parsen durch Flex ein Abstract Syntax Tree entstanden ist und das es bei Mesa nötig ist, diesen in zwei weitere Zwischenformen zu bringen. Diese Zwischenformen werden durch das Kompilieren des Shaders in AST Form erzeugt. In diesem Abschnitt wird erklärt, wie Mesa aus dem AST den Higher Intermediate und Lower Intermediate Code erzeugt und warum diese bei Mesa so wichtig sind.

3.2.1 HIR: Higher intermediate Code

Ausgehend von dem Flex-erzeugtem Abstract Syntax Tree wird in Mesa durch das Kompilieren des AST's ein Higher Intermediate Code erzeugt. Wie schon erwähnt enthält der HIR noch alle Matrixoperationen und Strukturanmerkungen und ist somit der gewöhnliche IR Code. Dieser wird durch das Durchlaufen des Baumes, wie in Abbildung 6, abgeleitet. In Abbildung 6 wird anschaulich aus dem Input Code ein AST erzeugt, indem er die Knoten des Baumes als Code darstellt. Da in Mesa ein abstrakter und kein konkreter Baum erzeugt wird, sind alle Knoten auch nur mit abstrakten Informationen gefüllt. Das heißt, dass vorher gescannter Code, wie man in der Abbildung sieht, vereinheitlicht wird. Dadurch kann aus einem AST sehr einfach ein IR Code erzeugt werden, indem die Knoteninformationen als Quelle dienen. Im Beispiel aus dieser Abbildung ist die max-Funktion nicht dargestellt, da diese durch „Inlined" Funktionen aufgerufen wird. Dabei bedeutet „Inlined", dass Funktionen die als „Inlined" deklariert werden an der Codestelle, an der sie aufgerufen werden, ihren kompletten Inhalt einfügen anstatt die Funktion extern aufzurufen. Der IR Code wird aus dem Baum direkt gewonnen und kann als 3-Adress-Code [AR13] dargestellt werden [FR14].

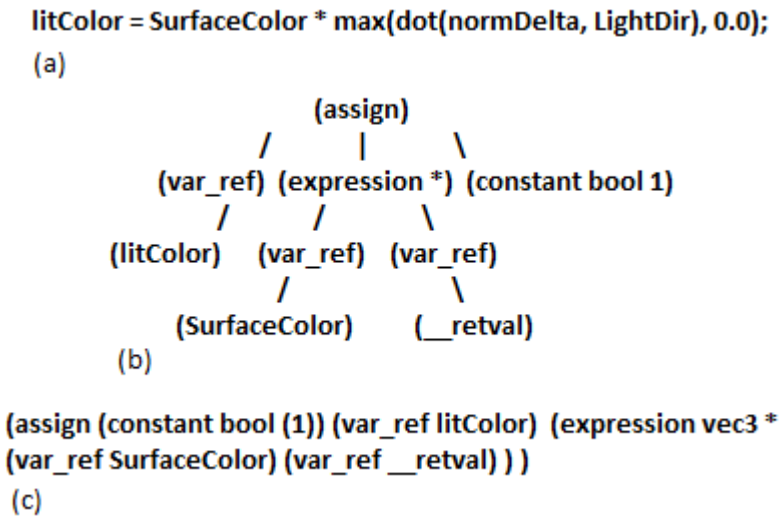

Abbildung 6, (a) Input Code, (b) Abstract Syntax Tree, (c) IR-Code [FR14]

Um den IR-Code aus Abbildung 6 aus dem „Abstract Syntax Tree“ zu erhalten läuft man wie schon erwähnt den Baum ab. In diesem Beispiel wurden die Schlüsselwörter wie „assign“, „var_ref“ und „expression“ schon von Mesa festgelegt. Da wie zuvor erwähnt die max-Funktion nicht dargestellt wird, da diese „inlined“ ist, bekommt man hierfür direkt ein Ergebnis, welches in diesem Fall eine Konstante darstellt. Dadurch ergibt sich wieder der 3-Adress-Code. Durch den von Mesa gegebenen Befehl „assign“ werden drei Werte gemeinsam dem Ergebnis zugewiesen. Der Wert aus der max-Funktion „constant bool(1)“, die Variable „litColor“ und die „expression“, welche sich wiederum aus einem 3-Adress-Code zusammensetzt. Ein solcher Baum kann je nachdem wie verschachtelt seine Aufrufe sind unbestimmt tief werden. Durch Optimierungen die der Compiler während der Laufzeit vornimmt, werden diese Bäume aber möglichst kleingehalten. Dies geschieht durch algebraische Optimierungen wie $x * 1 = x$ oder wiedererkannte Operationsmuster der Codegenerierung wie $vec1 * vec2 + vec3 == mad(vec1, vec2, vec3)$.

3.2.2 LIR: Lower intermediate Code

Um den LIR aus einem HIR zu bekommen, wird auf dem HIR ein Lowering ausgeführt. Dieses Lowering hat zu Folge, dass zum Beispiel auch die Matrixoperationen nochmal auf Punktmultiplikationen aufgeschlüsselt werden. Dies ist nötig um den verschiedenen Treibern, die Mesa unterstützt, auch volle Funktionsfähigkeit zu gewährleisten. Jeder dieser Treiber hat andere Voraussetzungen an die LIR. So braucht zum Beispiel ein Intel-Chipsatz aus der 915-Generation [IN14] alle Funktionen „inlined“, alle Loops „unrolled“, alle If-Bedingungen für eine bessere Lesbarkeit abgeflacht, keine Variablen Arrayzugriffe und die Matrixoperationen aufgeschlüsselt. Unter „unrolled“ versteht man das Beschleunigen der Laufzeit eines Programmes, durch das Auseinanderfächern einiger Schleifen. So werden diese entweder komplett aufgelöst und jeder Durchlauf einzeln nacheinander ausgeführt, oder durch eine gleichwertige Schleife ersetzt, die zwar größer ist

aber dafür weniger Durchläufe braucht. Das Mesa backend bei Software-Rendering benötigt unter anderem die Matrix und Strukturzuordnung aufgeschlüsselt, unterstützt aber wiederum Funktionsaufrufe und die dynamische Sprungvorhersage, welche für die intelligente Pipelineverteilung nötig sind. Aus diesen Gründen kann es kein allgemeines Lowering für alle Treiber geben und die Treiber selber müssen auf dem Higher Intermediate Code das Lowering, das sie benötigen, durchführen [FR14].

3.3 Linken des Shaders

In Mesa existieren für die Kompilierung der Shader zwei wichtige Link-Befehle. Der erste Namens „link_shaders“ verlinkt den kompilierten Shader mit einem zweitem Shader (z.b. Fragment) und weist den aus dem IR-Code gewonnen Attibutes und Uniforms Indizes und Speicherplätze zu. Jedoch ohne spezielle Werte, sodass man die Variablen später über eine Hashtable ansprechen und belegen kann. Dieser Link-Befehl wird benötigt um die Variablen in der Hashtable verteilen zu können und den Speicher für die später zugewiesenen Attibutes und Uniforms frei zu halten. Alle Uniforms werden einem mesa-internem Konstrukt dem „gl_program“ zugewiesen und mit 0 initialisiert. Da diese Arbeit allerdings nur den Vertex Shader voraussetzt, kann dieser Link Befehl nicht komplett auf diese Problemstellung übertragen werden. Es ist nötig manuell alle Uniforms zu übermitteln und alle wichtigen Parameter für den IR-Code des Shaders in das Konstrukt von Mesa zu übernehmen, wenn kein Fragment Shader betrachtet werden soll. Die zweite Möglichkeit besteht darin, die vorhandene Funktion von Mesa umzuschreiben, sodass nur der Vertex Shader bearbeitet wird. Allerdings kann dies zu späteren Problemen führen sobald überprüft wird, ob zwei Shader miteinander verlinkt wurden und der Compiler auf noch nicht initialisierte Speicherbereiche zugreift [KH13e].

Der zweite Link-Befehl konvertiert den IR-Code selbst in die von Mesa gegebene „gl_program“ Struktur und heißt „_mesa_ir_link_shader“. Zudem führt er verschiedene treiberspezifische Optimierungen wie auch Lowerings auf dem IR-Code aus. Diese Optimierungen und Lowerings führen zu der von Mesa vorgegeben Struktur, die „gl_program“ voraussetzt. Durch dieses Linken erhält man ein gefülltes Konstrukt auf diesem weitere Funktionen aufbauen. In Abbildung 7 ist ein kleiner Ausschnitt des „gl_program“ Konstrukts zu sehen. Hier wird während dem Link-Vorgang, der IR-Code dem Struct zugewiesen, Optionen übernommen, die mesa-interne Parameterliste für Uniforms erstellt und vieles mehr. Alle Instruktionen von Mesa werden dabei in Programminstruktionen konvertiert um später im allgemeinen Kontext nutzbar zu werden.

```
.
.
.
GLuint Id;
GLubyte *String;  /**< Null-terminated program text */
GLint RefCount;
GLenum Target;    /**< GL_VERTEX/FRAGMENT_PROGRAM_ARB */
GLenum Format;    /**< String encoding format */

struct prog_instruction *Instructions;

GLbitfield64 InputsRead;     /**< Bitmask of which input regs are read */
GLbitfield64 OutputsWritten; /**< Bitmask of which output regs are written */
GLbitfield SystemValuesRead;   /**< Bitmask of SYSTEM_VALUE_x inputs used */
GLbitfield InputFlags[MAX_PROGRAM_INPUTS];   /**< PROG_PARAM_BIT_x flags */
GLbitfield OutputFlags[MAX_PROGRAM_OUTPUTS]; /**< PROG_PARAM_BIT_x flags */
GLbitfield TexturesUsed[MAX_COMBINED_TEXTURE_IMAGE_UNITS];  /**< TEXTURE_x_BIT bitmask */
GLbitfield SamplersUsed;   /**< Bitfield of which samplers are used */
GLbitfield ShadowSamplers; /**< Texture units used for shadow sampling. */


/** Named parameters, constants, etc. from program text */
struct gl_program_parameter_list *Parameters;
.
.
.
```

Abbildung 7, Ausschnitt aus dem „gl_program“ Konstrukt

3.4 Attributes & Uniforms

Die Attribute [KH13d] und Uniforms [KH13a] des Shaders werden in Mesa erst kurz vor der Berechnung des Shaderprogram belegt. Im vorigen Schritt, dem Linken, wurde eine Hashtable erstellt, welche allen aktiven Uniforms Speicherplatz reserviert. Mesa versucht dabei so effizient wie möglich zu sein, indem alle inaktiven Uniforms, also diejenigen die nicht den Ausgangswert des Shaders verändern, keinen Speicherplatz bekommen und somit nicht genutzt werden können. Alle aktiven Uniforms, die den Ausgangswert des Shaders beeinflussen, werden in die Hashtable übernommen und bekommen eine Position zugewiesen. Durch diese Position lassen sich später den Uniform Variablen des Shaders Werte zuweisen. Um die konkreten Werte der Uniforms zu speichern nutzt Mesa die in „gl_program“ integrierte Parameterliste. Um die konkreten Werte den Positionen zuzuweisen ruft Mesa die Funktion „_mesa_uniform“ auf. Mit Hilfe dieser Funktion werden die Werte an die vorher in der Hashtable definierten Position abgelegt [KH13a]. Um die Position der einzelnen Uniforms zu erhalten, ruft man den Befehl „_mesa_get_uniform_location“ auf, die als Rückabewert den die durch den Parser bestimmte Position für eine Uniform Variable liefert.

Die Attribut Variablen wurden durch das Linken mit einem Index versehen, durch den man sie ansprechen kann. Um den Attributen einen Wert zuzuweisen werden häufig Bufferobjekte verwendet. Dabei werden durch die Funktion „_mesa_VertexAttribPointer“ bestimmte Attributwerte in die Attributvariablen geladen. Dies ist durch den Index möglich, den jeder durch das Linken des Programms erhalten hat. Die vorher festgelegte Position wird durch den Index ermittelt und der Funktion, die den Attributen Werte zuweist, übergeben. Wenn der Wert an die Variable gebunden ist, wird durch „_mesa_EnableVertexAttribArrayARB“ ein Array von generischen Vertex Attributen [KH13b] aktiviert und die vorher bestimmten Werte übernommen. So wird der Shader vor dem Berechnen noch mit dem Pointer des Arrays verknüpft, dass die Werte für die Vertex Attribute übernommen werden können [KH13b, KH13c, KH13d].

3.5 LLVM Bausteine

Um den Aufbau und die Vorgehensweise von LLVM besser zu verstehen, wird in diesem Abschnitt der Aufbau einer LLVM-Funktion für den LLVM Compiler untersucht. Dabei wird die Funktion Schritt für Schritt aufgebaut und in LLVM-Form gebracht. Da in den letzten Abschnitten dieses Kapitels schon erklärt wurde, wie Mesa den Shader vorbereitet um von einem LLVM Compiler bearbeitet werden zu können, wird hier nur auf den LLVM Aufbau eingegangen. Dabei baut Mesa die LLVM-Funktionen während der Laufzeit auf, wie in den nächsten Abschnitten beschrieben wird.

3.5.1 LLVM Funktionen, Basisblöcke und der Instruction Builder

Die LLVM Basisblöcke sind der Hauptbestandteil der LLVM Funktion. Jede Funktion basiert auf solchen Blöcken und wird aus ihnen gebildet. Ein solcher Basisblock ist dabei ein Container, der in sich sequentiell bearbeitet wird. In LLVM werden diese Container als Werte übergeben, da sie von anderen Objekten referenziert werden können. Außerdem „leben“ sie in einem LLVM Kontext, der für die Zuordnung aller Objekte verantwortlich ist. Über diesen ist es möglich alle Objekte die in einem Kontext existieren anzusprechen. Durch den Befehl „Create“, ist es möglich einen neuen Block innerhalb eines bereits existierendem Kontextes zu erzeugen. In Mesa wird für jeden Shader eine main-Funktion erzeugt, welche einen Block zugewiesen bekommt. Der durch „Create“ erzeugte Block ist allerdings noch leer und muss noch gefüllt werden. Die gewünschte Funktion muss mittels LLVM- Befehlen erzeugt werden. Diesen werden Typen übergeben, die beschreiben wie Rückgabewerte und Übergabewerte aussehen und bilden somit den Funktionskopf. Unter anderen sind dies Typen wie Integer oder Float, welchen noch die Bitlängen mitgegeben werden und die Funktion beschreiben. Anschließend muss der Block der Funktion über „LLVMAppendBasicBlock“ zugewiesen werden. Die Funktion besitzt nun einen Funktionskopf und einen Block in dem Befehle bearbeitet werden können. Diese werden in LLVM als Anweisungen in einem Block realisiert. Mit Hilfe eines sogenannten „Instruction Builders“ ist es möglich Anweisungen in einem Block zu erzeugen. LLVM bietet eine Vielzahl an vordefinierten Anweisungen, wie Addieren oder Multiplizieren an, welche nun in den Block angehängt werden können. Dabei repräsentiert der „Instruction Builder“ einen Pointer, der in den Block zeigt und innerhalb Funktionen und Anweisungen anhängen kann [UN13].

Durch diese Hilfsmittel ist es möglich in LLVM eine Funktion zu realisieren. Mesa baut eine solche Funktion mit oben beschriebenen Mitteln für jeden Shader auf. Sind mehrere Shader miteinander verknüpft so wird eine main-Funktion für beide geschrieben, die in sich sequentiell bearbeitet wird. Dabei wird der IR-Code des Shaders in die jeweiligen vordefinierten Funktionen von LLVM übersetzt und einzeln in den Block angehängt. Dies geschieht in Mesa bei der Erzeugung des Konstrukts „Variant“. Dort wird die Funktion mit zugehörigen Blöcken und Anweisungen als LLVM Funktion aus dem übergebenem geparstem Shader erzeugt und im mesa-internen „Variant“ Konstrukt gespeichert.

3.5.2 LLVM Passes und PassManager

Der LLVM-Pass ist ein wichtiger Teile des LLVM Compilers. Er kümmert sich um alle Transformationen und Optimierungen und bildet das analytische Ergebnis für diese Transformationen der Funktionen aus dem letzten Abschnitt. Zusätzlich zum gerade erläuterten Pass gibt es einen „PassManager“ (PM). Dieser kümmert sich um die Verwaltung der analytischen Ergebnisse, sowie die Speicherverwaltung und die dazugehörigen Passes. LLVM bietet verschiedene Typen von „PassManagern“ an. Je nach Bedarf muss ein Modul PM, Funktion PM, Loop PM, Basisblock PM oder CallGraph PM gewählt werden. Wurde der „PassManager“ bestimmt, weist man ihn zum Beispiel der zuvor erstellten Funktion zu. Mit „LLVMRunFunctionPassManager“ bietet LLVM die Möglichkeit, den der Funktion zugehörige PM zu starten, der dann alle definierten LLVM-Optimierungen in der Funktion vornimmt [LL14a].

Mesa nutzt diese Technik auch um den erzeugten LLVM-Code des Shaders zu optimieren. Deswegen wird ein „FunctionPassManager“ erstellt und auf die LLVM Shader Funktion angewendet. Dies geschieht ebenfalls bei der Erzeugung des „Variants“. Nach der Erstellung der Funktionen und das erzeugen der Anweisung des dazugehörigen Blockes wird der PM erstellt und auf der Funktion aufgerufen. Der „PassManager“ gibt dabei eine Eins zurück, wenn er die bearbeitete Funktion verändert hat, eine Null wenn nichts verändert oder optimiert werden musste.

3.5.3 LLVM ExecutionEngine und der Just-in-Time-Compiler

Es gibt nun viele Möglichkeiten den vorbereiteten LLVM IR Code weiter zu verarbeiten oder zu nutzen. So gibt es die Möglichkeit ihn weiter zu optimieren und ihm noch mehr „Passes“, die den Code weiter optimieren sollen, hinzuzufügen, den Code als Assembly Datei auszuführen oder ihn „Just-in-Time“ (JIT) kompilieren. Da Mesa die letzte Möglichkeit nutzt ist es sinnvoll diese Art der Weiterverarbeitung genauer zu betrachten. Um den IR Code JIT zu kompilieren wird eine sogenannte „ExecutionEngine“ benötigt. Diese abstrakte „ExecutionEngine“ stellt entweder den JIT Compiler oder den LLVM Interpret dar. Dabei wählt LLVM automatisch den JIT Compiler, falls einer auf dem Rechner vorhanden ist. Die einfachste Möglichkeit diesen JIT Compiler zu nutzen ist über die Funktion „getPointerToFunction“ gegeben. Durch diese Funktion weißt man dem JIT Compiler eine Funktion zu, die er nun durchläuft und dabei alles kompiliert. Das gilt auch für geschachtelte Funktionen die in einem Block enthalten sein könnten. Auch externe Funktionen, wie zum Beispiel mathematischer Art werden „Just-in-Time“ geladen und implementiert. Damit wird das Kompilieren erleichtert, da nicht jede Funktion einzeln mit der „ExecutionEngine“ ausgeführt werden muss, sondern nur die Top-Level Funktion [LL14b].

Mesa macht sich das auch bei der „Variant“ Erzeugung zunutze. Nachdem alle „Passes“ aus dem letztem Abschnitt definiert wurden und durch den PM ausgeführt worden sind, wird eine „ExecutionEngine“ erzeugt. Diese wird der Funktion zugeordnet. Da diese Funktion die main-Funktion des shaders darstellt, ist diese gleichzeitig die Top-Level

Funktion des Shaders und alle anderen Funktionen die in diesem Block enthalten sind, werden mitkompiliert. Der „Variant“ von Mesa hält die „ExecutionEngine“ als Funktionspointer in seinem Konstrukt unter dem Namen „jit_func“ und kann von dort dann ausgeführt werden. Durch Ausführen dieser Funktion erhält Mesa die „gl_Position“ des Shaders.

4 Konzepte & Ergebnisse

In diesem Kapitel werden die Konzepte, die benötigt werden um die „gl_Position“ eines Vertex Shaders zu berechnen, genauer betrachtet. Anhand eines Beispielshaders wird untersucht, welche Vorgehensweisen von Mesa sinnvoll sind und welche bei der dynamischen Ausführung von Positionstransformationen unnötig sind. Um für die weiteren Abschnitte ein gemeinsames Verständnis des Shaders zu bekommen wird in jedem Teil, wenn die Sprache von einem Vertex Shader ist, der für diese Arbeit gewählte Vertex Shader aus Abbildung 8, angenommen. Die vorgestellten Konzepte basieren alle auf Mesa, welches eine Möglichkeit der Positionstransformation bietet. So werden sowohl der Lexer und Parser wie auch der Kompilierungsvorgang, das Linken und der LLVM Compiler von Mesa untersucht und die Konzepte erläutert.

```
static const char vertex_shader[] =
"attribute vec3 position;\n"
"attribute vec3 normal;\n"
"\n"
"uniform mat4 ModelViewProjectionMatrix;\n"
"uniform mat4 NormalMatrix;\n"
"uniform vec4 LightSourcePosition;\n"
"uniform vec4 MaterialColor;\n"
"\n"
"varying vec4 Color;\n"
"\n"
"void main(void)\n"
"{\n"
"    // Transform the normal to eye coordinates\n"
"    vec3 N = normalize(vec3(NormalMatrix * vec4(normal, 1.0)));\n"
"\n"
"    // The LightSourcePosition is actually its direction for directional light\n"
"    vec3 L = normalize(LightSourcePosition.xyz);\n"
"\n"
"    // Multiply the diffuse value by the vertex color (which is fixed in this case)\n"
"    // to get the actual color that we will use to draw this vertex with\n"
"    float diffuse = max(dot(N, L), 0.0);\n"
"    Color = diffuse * MaterialColor;\n"
"\n"
"    // Transform the position to clip coordinates\n"
"    gl_Position = ModelViewProjectionMatrix * vec4(position, 1.0);\n"
"}";
```

Abbildung 8, Input Vertex Shader String

4.1 Der Lexer & Parser

Die Grundlagen über den Lexer und Parser wurden im vorigen Kapitel besprochen. In diesem Abschnitt werden beide Konzepte auf den Nutzen für die dynamische Ausführung von Positionstransformationen untersucht und anhand eines Shaderbeispiels ausgeführt. Es soll vermittelt werden, wie der Lexer und das Parsen eines Vertex Shaders arbeitet und welche Teile des in Mesa implementierten Lexer und Parser für einen Vertex Shader vernachlässigt werden können.

4.1.1 Der Lexer

Das Parsen des Shader Input Strings ist der erste Schritt der für das Konzept für die Berechnung der Position ausgeführt wird. Dieser wird benötigt um den Shader Input String zu scannen. Dafür wird der aus dem im vorigen Kapitel beschriebene Parser von Flex genutzt. Dabei wird ein Scanner oder auch Lexer genannt automatisch initialisiert, indem Speicher für ihn allokiert wird und ihm Globale Werte, welche einen noch nicht benutzten Scanner darstellen, zugewiesen werden. Danach ist der Lexer bereit, den String nach Mustern zu untersuchen. Dabei wird der String in einen Bufferarray geladen, der um zwei Felder größer als der String selbst ist. Dies verhindert das Überlaufen des Lexers in den nicht allokierten Speicherbereich. Ein solcher Lexer ist die Grundlage für den folgenden Parsevorgang. Er wird verwendet um über den String zu scannen und Muster zu finden.

Um einen Lexer für das Programm zu nutzen, wird der von Mesa vordefinierte genommen. Dieser Lexer wird beim Starten des Parsevorgangs in Mesa durch den Aufruf von „_mesa_glsl_parse“ automatisch erzeugt und für diese Arbeit unverändert genutzt. Ohne den Lexer wäre es nicht möglich den von Flex gestellten Parser zu nutzen und den Shader zu bearbeiten. Somit wurde für dieses Projekt der von Mesa gegebene Lexer implementiert um den Scanner für die Tokenerzeugung, die im letzten Kapitel vorgestellt wurde, zu nutzen.

4.1.2 Der Parsevorgang

Durch den im letzten Abschnitt erzeugtem Lexer ist es möglich den String zu Parsen. Dies ist nötig um den Shader String in die gewünscht AST Form zu bekommen, welcher der nächste Schritt im Programm ist. Mit Hilfe des ASTs ist es möglich einen IR zu erstellen, der danach in LLVM übersetzt werden kann. Daher ist die Erzeugung des Abstract Syntax Tree der logische nächste Schritt um den Shader in Intermediate Code zu übersetzen. Der Scanner läuft über den String und untersucht ihn auf bekannte Muster um den AST zu erzeugen. Dabei erzeugt er auch eine Symboltabelle, welche alle möglichen Datentypen wie zum Beispiel die in GLSL definierten „ARB_texture_cube_arrays“, die im Shader benötigt werden könnten, bereithält. Wenn der Lexer ein Muster erkannt hat, speichert er das gefundene Muster als „ast_node“, „ast_function“, „parameter“ oder entsprechendes in einer „ast_expression“, „ast_function“ oder andere, die den Typ des gefunden Musters beschreiben, um später daraus einen kompletten Abstract Syntax Tree zu erzeugen. Hier werden die Knoten schon unterschieden indem bei „identifier“, „int_constants“, „uint_constants“, usw. eine Fallunterscheidung gemacht wird. Dies ist allerdings nur eine kleine Auswahl von Möglichkeiten, die der Parser unterscheidet.

Da es enorm viele Fälle zu betrachten gäbe, ist es sinnvoll den schon vordefinierten Parser von Mesa für den für diese Arbeit ausgewählten Vertex Shader zu nutzen. Dieser deckt bis zu 340 verschiedene Fälle ab. Da Mesa OpenGL ES 2.0 bietet würden fehlende Fälle durch die Mesa Entwicklung abgedeckt werden. Der Abstract Syntax Tree kann durch den Aufruf von „_mesa_glsl_parse“ erzeugt werden. Daraus ergibt sich im ersten Schritt des Programms, der in Abbildung 9 abgebildete erstellte AST für den Vertex Shader als Textform

wie auf Abbildung 6a. Die Textform entsteht durch das Ablaufen der Knoten des Baumes wie im Abschnitt 3.2.1 gezeigt wurde und nur diese werden auch ausgegeben. Jeder der durch ein Semikolon getrennter Teil stellt dabei einen Knoten in 3-Adress-Code da.

```
attribute vec3 position ; attribute vec3 normal ; uniform mat4 ModelViewProjectionMatrix ;
uniform mat4 NormalMatrix ; uniform vec4 LightSourcePosition ;
uniform vec4 MaterialColor ; varying vec4 Color ;
void  main (void ){
vec3 N = normalize ( vec3 ( NormalMatrix * vec4 ( normal , 1.000000 ) ) ) ;
vec3 L = normalize ( LightSourcePosition . xyz ) ;
float diffuse = max ( dot ( N , L ) , 0.000000 ) ;
Color = diffuse * MaterialColor ;
gl_Position = ModelViewProjectionMatrix * vec4 ( position , 1.000000 ) ; }
```

Abbildung 9, Textform des Vertex Shader Abstract Syntax Tree

Aus übersichtlichen Gründen wurden einige Zeilenumbrüche eingebaut, die der AST normalerweise nicht besitzt. So gäbe es nur nach „void main(void){„ ein Zeilenumbruch und der Rest wird jeweils in eine Zeile geschrieben. Hier werden die Unterschiede zum Ausgangs-Vertex Shader noch nicht deutlich. Jedoch erkennt man, dass die Formatierung der Shaders geändert wurde und die Kommentare entfernt. Zudem wurden alle Zeilen als Baum gespeichert und befinden sich nun als Liste, die eine Baumstruktur darstellt, im Speicher. Die Knoten des Baumes enthalten nun auch Zusatzwissen, wie „attribute“, „uniform“ oder ob der Knoten eine Funktion darstellt.

Diese Informationen und der Abstract Syntax Tree selber sind wichtige Bausteine um den Intermediate Code für LLVM zu erzeugen. Durchaus wäre es möglich für diesen Schritt auch einen anderen Parser zu wählen. So bietet zum Beispiel die Eclipse-Comunity mit „AST View“ ein Plug-In an mit dem man eine beliebige Java-Quelle untersuchen kann und daraus ein Abstract Syntax Tree erstellt wird. Weitere Möglichkeiten bieten unter anderen auch „Coco/R“ [JO11] für C#, C++ und Java oder „Pyparsing“ [PY13] für Python. Da Mesa allerdings zusätzlich den LLVM Compiler bietet, ist es am sinnvollsten auch den von Mesa bereitgestellten Parser zu nutzen. Durch die von Mesa vordefinierten Muster, die Flex erkennen soll, wird der Shader schon in die richtige Form für den Intermediate Code gebracht. Dieser wird benötigt um auf Maschinenebene den Code auszuführen.

4.2 Erzeugen des Intermediate Codes

Der Shader wurde durch die letzten Schritte in die Form des Abstract Syntax Tree gebracht. Um ausgehend von dieser Form den Shader weiter bearbeiten zu können wird der Intermediate Code gebraucht. Mit Hilfe des IR-Codes ist es möglich den Shader auf Maschinenebene mit LLVM auszuführen und die Position des Vertex zu berechnen. In diesem Abschnitt geht es um das Erzeugen des Intermediate Codes aus dem zuvor erzeugtem Abstract Syntax Tree. Im letzten Kapitel wurde schon kurz erklärt wofür der Higher Intermediate und der Lower Intermediate Code gebraucht wird. Nun wird in den folgenden

Abschnitten geklärt, wie diese aus dem AST entstehen und geklärt ob man zur Positionsberechnung des Vertex Shaders auch beide benötigt.

Auch hier bietet sich an, den schon in Mesa implementierten Teil wiederzuverwenden. Dieser kümmert sich um die Erzeugung des Intermediate Codes und kann durch schon vordefinierte Optimierungen, wie das Beseitigen von nicht genutztem Code, daraus den Lower Intermediate Code, erzeugen. Daher wird für die Erstellung des IR-Codes in den nächsten Schritten ein Teil von Mesa verwendet.

4.2.1 Vom AST zum HIR

Bevor das eigentliche Umwandeln des Abstract Syntax Tree's zum Higher Intermediate Code stattfindet müssen einige Variablen initialisiert werden. Durch den Aufruf von „_mesa_ast_to_hir“ wird die Initialisierung der Variablen gestartet. Dies geschieht über einen „builtin_variable_generator“, welcher die Funktionen „generate_constants“, „generate_uniforms“ und „generate_varyings“ zum generieren der Constants, Uniforms und Varyings bereitstellt. Dadurch werden den gerade erzeugten Variablen GLSL spezifische Typen für Matrizen oder Vektoren zugeordnet, um bei der Konvertierung des Shaders dessen Variablen einen spezifischen Typ zuzuordnen. Dabei werden auch einige mesa-spezifische und OpenGL ES versionsabhängige Variablen erzeugt, die später zum Kompilieren des Shaders benötigt werden. Sind die Variablen erzeugt, wird die Liste, die den Abstract Syntax Tree darstellt, durchlaufen. Durch Ablaufen der Abstract Syntax Tree Knoten wird jeder Knoten in eine high-level Intermediate Darstellung konvertiert. Dazu wird die Funktion „hir()“ der „ast_node“ genutzt. Dies ist durch die vorige Sortierung der einzelnen, erkannten Muster im AST möglich. Jeder Knoten wird nun einem Typ und einer Variable, die durch den „buitlin_variable_generator“ erzeugt wurden, zugeordnet. Nach der Konvertierung der einzelnen Knoten des Baumes und der Variablen, die im ersten Schritt generiert wurden, ist es möglich über eine „exec_list“ alle IR relevanten Codestücke zu erhalten. Für den Vertex Shader ergibt sich dabei der in Abbildung 10a und 10b erzeugte Higher Intermediate Code.

```
(function normalize
  (signature vec3
    (parameters
      (declare (in ) vec3 arg0)
    )
    (
    ))

)

(function dot
  (signature float
    (parameters
      (declare (in ) vec3 arg0)
      (declare (in ) vec3 arg1)
    )
    (
    ))

)

(function max
  (signature float
    (parameters
      (declare (in ) float arg0)
      (declare (in ) float arg1)
    )
    (
    ))

)
```

Abbildung 10a, Vertex Shader Higher Intermediate Code Funktionen

Der in Abbildung 10a und 10b erzeugte Code ist um einiges größer als der eigentliche Vertex Shader. Dies liegt unter anderem an den durch den „builtin_variable_generator“ generierten Variablen. An dem hier erzeugtem HIR Code sieht man, dass alle Varyings, Uniforms und andere Datentypen als 3-Adress-Code dargestellt sind. Dieser baut sich aus dem „declare“-Befehl, dem Mesa-Typ wie zum Beispiel „shader_out“ und dem Datentyp wie „float“, „vec4“ oder anderen auf. Die „main“-Funktion stellt den größten Teil des HIR dar. Hier werden temporäre Variablen genutzt um die Zwischenergebnisse zu speichern und die 3-Adress-Code Darstellung bei zuhalten. Funktionen wie „normalize“, „dot“ und „max“ aus Abbildung 10a sind hier leer, da Mesa diese während der Laufzeit extern lädt und diese direkt als LLVM Code implementiert werden. Mesa besitzt zum laden dieser Funktionen einen eigene Sammlung an IR-Funktionen, die solche Standard Funktionen darstellen. Darunter zählen auch wie in Abbildung 10a zu sehen ist, „normalize“, „dot“ und „max“. Diese werden nur bei Gebrauch, durch einen JIT-Compiler in den Code geladen um Ausführungszeit und unnötige Implementierungszeit aller dieser Standard Funktionen zu vermeiden.

Durch diese Sammlung an Funktionen wird viel Arbeit erspart, indem man nicht jede einzelne dieser Standardfunktionen selber implementieren muss. Daher wird für diesen Schritt Mesa für die Erzeugung von IR Code verwendet. Durch die kurze Darstellung der Standardfunktionen wird dem Programm Ladezeit und Speicherkapazität erspart.

```
(
(declare (shader_out ) float gl_PointSize)
(declare (shader_out ) vec4 gl_Position)
(declare (uniform ) (array vec4 55) gl_CurrentAttribFragMESA)
(declare (uniform ) (array vec4 33) gl_CurrentAttribVertMESA)
(declare (uniform ) gl_DepthRangeParameters gl_DepthRange)
(declare () int gl_MaxVaryingVectors)
(declare () int gl_MaxFragmentUniformVectors)
(declare () int gl_MaxVertexUniformVectors)
(declare () int gl_MaxDrawBuffers)
(declare () int gl_MaxTextureImageUnits)
(declare () int gl_MaxCombinedTextureImageUnits)
(declare () int gl_MaxVertexTextureImageUnits)
(declare () int gl_MaxVertexAttribs)
(declare (shader_in ) vec3 position)
(declare (shader_in ) vec3 normal)
(declare (uniform ) mat4 ModelViewProjectionMatrix)
(declare (uniform ) mat4 NormalMatrix)
(declare (uniform ) vec4 LightSourcePosition)
(declare (uniform ) vec4 MaterialColor)
(declare (shader_out ) vec4 color)
(function main
  (signature void
    (parameters
    )
    (
      (declare () float diffuse)
      (declare () vec3 L)
      (declare () vec3 N)
      (declare (temporary ) vec4 vec_ctor)
      (assign  (w) (var_ref vec_ctor)  (constant float (1.000000)) )
      (assign  (xyz) (var_ref vec_ctor)  (swiz xyz (var_ref normal) ))
      (declare (temporary ) vec3 vec_ctor@2)
      (assign  (xyz) (var_ref vec_ctor@2)  (swiz xyz (expression vec4 * (var_ref NormalMatrix) (var_ref vec_ctor) ) ))
      (declare (temporary ) vec3 normalize_retval)
      (call normalize (var_ref normalize_retval)  ((var_ref vec_ctor@2) ))

      (declare (temporary ) vec3 assignment_tmp)
      (assign  (xyz) (var_ref assignment_tmp)  (var_ref normalize_retval) )
      (assign  (xyz) (var_ref N)  (var_ref assignment_tmp) )
      (declare (temporary ) vec3 normalize_retval@3)
      (call normalize (var_ref normalize_retval@3)  ((swiz xyz (var_ref LightSourcePosition) )))

      (declare (temporary ) vec3 assignment_tmp@4)
      (assign  (xyz) (var_ref assignment_tmp@4)  (var_ref normalize_retval@3) )
      (assign  (xyz) (var_ref L)  (var_ref assignment_tmp@4) )
      (declare (temporary ) float dot_retval)
      (call dot (var_ref dot_retval)  ((var_ref N) (var_ref L) ))

      (declare (temporary ) float max_retval)
      (call max (var_ref max_retval)  ((var_ref dot_retval) (constant float (0.000000)) ))

      (declare (temporary ) float assignment_tmp@5)
      (assign  (x) (var_ref assignment_tmp@5)  (var_ref max_retval) )
      (assign  (x) (var_ref diffuse)  (var_ref assignment_tmp@5) )
      (declare (temporary ) vec4 assignment_tmp@6)
      (assign  (xyzw) (var_ref assignment_tmp@6)  (expression vec4 * (var_ref diffuse) (var_ref MaterialColor) ) )
      (assign  (xyzw) (var_ref color)  (var_ref assignment_tmp@6) )
      (declare (temporary ) vec4 vec_ctor@7)
      (assign  (w) (var_ref vec_ctor@7)  (constant float (1.000000)) )
      (assign  (xyz) (var_ref vec_ctor@7)  (swiz xyz (var_ref position) ))
      (declare (temporary ) vec4 assignment_tmp@8)
      (assign  (xyzw) (var_ref assignment_tmp@8)  (expression vec4 * (var_ref ModelViewProjectionMatrix) (var_ref vec_ctor@7) ) )
      (assign  (xyzw) (var_ref gl_Position)  (var_ref assignment_tmp@8) )
    ))
)
```

Abbildung 10b, Vertex Shader Higher Intermediate Code

4.2.2 Vom HIR zum LIR

Im letzten Abschnitt wurde der Higher Intermediate Code durch den Abstract Syntax Tree erzeugt. Das Ziel ist es jedoch durch weitere Optimierungen des Codes einen Lower Intermediate Code zu erhalten. Dieser wird speziell für die jeweils verwendeten Treiber angepasst und hat somit keine allgemeine Darstellung. Es ergibt sich aber der Vorteil eines kürzeren Intermediate Codes, den man danach weiter verarbeiten kann. Solche Optimierungen sind unter anderen das Beseitigen von „totem“ Code oder das Aufsplittern der Struktur. Das bedeutet, dass Codeteile oder Funktionen die nicht genutzt werden, entfernt werden. Durch derartige Optimierungen des Codes, erhält man den LIR. Diese werden in Phase 2 im Optimizer aus Abbildung 3 ausgeführt. Für den Vertex Shader ergibt sich somit aus dem Higher Intermediate Code der auf Abbildung 11 abgebildete Lower Intermediate Code.

```
(
(declare (shader_out ) float gl_PointSize)
(declare (shader_out ) vec4 gl_Position)
(declare (uniform ) (array vec4 55) gl_CurrentAttribFragMESA)
(declare (uniform ) (array vec4 33) gl_CurrentAttribVertMESA)
(declare (uniform ) gl_DepthRangeParameters gl_DepthRange)
(declare () int gl_MaxVaryingVectors)
(declare () int gl_MaxFragmentUniformVectors)
(declare () int gl_MaxVertexUniformVectors)
(declare () int gl_MaxDrawBuffers)
(declare () int gl_MaxTextureImageUnits)
(declare () int gl_MaxCombinedTextureImageUnits)
(declare () int gl_MaxVertexTextureImageUnits)
(declare () int gl_MaxVertexAttribs)
(declare (shader_in ) vec3 position)
(declare (shader_in ) vec3 normal)
(declare (uniform ) mat4 ModelviewProjectionMatrix)
(declare (uniform ) mat4 NormalMatrix)
(declare (uniform ) vec4 LightSourcePosition)
(declare (uniform ) vec4 MaterialColor)
(declare (shader_out ) vec4 Color)
(function main
  (signature void
    (parameters
    )
    (
      (declare () vec3 N)
      (declare (temporary ) vec4 vec_ctor)
      (assign  (w) (var_ref vec_ctor)  (constant float (1.000000)) )
      (assign  (xyz) (var_ref vec_ctor)  (var_ref normal) )
      (declare (temporary ) vec3 normalize_retval)
      (call normalize (var_ref normalize_retval)  ((swiz xyz (expression vec4 * (var_ref NormalMatrix) (var_ref vec_ctor) ) )))

      (assign  (xyz) (var_ref N)  (var_ref normalize_retval) )
      (declare (temporary ) vec3 normalize_retval@9)
      (call normalize (var_ref normalize_retval@9)  ((swiz xyz (var_ref LightSourcePosition) )))


      (declare (temporary ) float dot_retval)
      (call dot (var_ref dot_retval)  ((var_ref N) (var_ref normalize_retval@9) ))


      (declare (temporary ) float max_retval)
      (call max (var_ref max_retval)  ((var_ref dot_retval) (constant float (0.000000)) ))

      (assign  (xyzw) (var_ref Color)  (expression vec4 * (var_ref max_retval) (var_ref MaterialColor) ) )
      (declare (temporary ) vec4 vec_ctor@10)
      (assign  (w) (var_ref vec_ctor@10)  (constant float (1.000000)) )
      (assign  (xyz) (var_ref vec_ctor@10)  (var_ref position) )
      (assign  (xyzw) (var_ref gl_Position)  (expression vec4 * (var_ref ModelviewProjectionMatrix) (var_ref vec_ctor@10) ) )
    ))
)
```

Abbildung 11, Vertex Shader Lower Intermediate Code

Durch den Vergleich des LIR mit dem zuvor erzeugten HIR erkennt man, dass der Declare Teil des Shaders gleich geblieben ist und die Optimierungen sich ausschließlich auf die „main“-Funktion auswirken. Durch das Entfernen, der nicht gebrauchten Funktionen, und den Codestücken, die keine Funktion hatten, wurde die „main“-Funktion gekürzt. Dadurch verkürzt sich in diesem Beispiel der Code des Shaders um ca. 53% im Vergleich zum HIR. Die Funktionen „normalize“, „dot“ und „max“ sind hier immer noch vorhanden und sind nicht durch die Optimierungen betroffen.

Diese Optimierungen, die den Lower Intermediate Code ausmachen, sind nicht zwingend notwendig. Allerdings verkürzen sie in den meisten Fällen den Code und senken damit später die Laufzeit des Programmes. Zudem ist es sinnvoll den Code als LIR auszudrücken, da einige Treiber nur so den Shader weiter in die jeweils gewünschte Form bringen können. Mesa bietet hier auch eine Lösung. Der von Mesa gegeben Optimizer kann genutzt werden um einen LIR aus einem HIR zu erhalten. Durch diesen Optimizer erhält man für den Vertex Shader, den in Abbildung 11 dargestellten, LIR.

4.3 Der Linker

Um die Kompilierung des Codes zu erreichen, muss man den Shader noch in eine für den LLVM Compiler lesbare Form bringen. Ausgehend von einem Intermediate Code, ist dies

über das sogenannte „Linken“ des Shaders möglich. Das „Linken“ ermöglicht den Variablen und Funktionen des IR-Codes in eine für das Programm besser nutzbare Form zu konvertieren. Diese Form kann dann im weiteren Verlauf dem LLVM Compiler übergeben werden, um aus dem IR-Code, LLVM-Code zu erstellen. Um das „Linken“ des Shaders zu erreichen verwendet man die zwei konkreten „Linker“ Aufrufe aus dem letzten Kapitel.

In den vorigen Abschnitten wurde der Shader in eine allgemeinere Form, den IR Code gebracht. Diese alleine reicht allerdings noch nicht aus um ihn zu kompilieren und die gewünschte Ergebnisse zu erzielen. Eine spezielle Weiterverarbeitung und Gliederung des Shaders besteht darin ihn zu „linken“. „Linken“ bedeutet dabei den Shader zuerst mit, wenn vorhanden, weiteren Shader zu verknüpfen und diese verknüpften Shader danach in ein Konstrukt zu laden. In diesem Konstrukt befinden sich außerdem viele Zusatzinformationen über die Umgebung. Da in dieser Arbeit nur von einem Vertex Shader ausgegangen wird, wird auf den ersten Schritt, das Verknüpfen von mehreren Shadern, nicht weiter eingegangen. Der zweite Teil indem der Shader in ein Konstrukt, den „gl_shader_program“, eingliedert wird, muss genauer Untersucht werden. Einer der wichtigsten Funktionen der „linker“-Methode ist das Reservieren von Speicher und das Indexen für die „Attributes“, „Uniforms“ und „Varyings“, die aus dem Shader gewonnen werden. Dies geschieht für Attribute in vier Phasen [RO10]:

- Alle bisher reservierten Positionen der Vertex Shader Inputs werden ungültig gemacht. Dies ist notwendig um sicher zu gehen, dass keine Altlasten in den neuen Shader übernommen werden. Altlasten treten dabei auf, wenn verschiedene Shader während der Laufzeit geladen werden und die Hashtables mit den vorigen Variablen noch nicht erneuert wurden.

- Die Positionen der durch die Nutzer definierten Inputs werden durch die Methode „glBindVertexAttribLocation“ und die durch den Nutzer definierte Outputs durch „glBindFragDataLocation“ zugewiesen.

- Diejenigen Attribute, die bisher keine Position zugewiesen bekommen haben, werden absteigend nach ihren benötigten Slots sortiert. Slots sind dabei Funktionen, die auf Signale reagieren können. Bei Verknüpfung von Slot und Signal werden die Slots, die mit den Signalen verknüpft sind ausgeführt, wenn das entsprechende Signal angelegt wird. Wenn keine spezielle Userzuweisung der Positionen für Attribute erfolgt, so werden automatisch aufsteigende Indizes an die Attribute des Input Shaders verteilt. Die Attribute werden im IR Code durch das Beiwort „shader_in“ erkannt.

- Jedem Input ohne Position wird an eine Position gebunden. Dabei bekommen Attribute wie schon erwähnt spezielle Indizes und Uniforms werden in eine für sie spezielle Hashtable eingeordnet. Dies ermöglicht einen erleichterten manuellen Zugriff auf die Variablen. Diese werden dann im späteren Verlauf des Programmes mit Hilfe der Positionen aus den Hashtable mit dem Wert des zu berechnenden Vertex belegt.

Nachdem jedes Attribut, ob vom Nutzer definiert oder neu zugewiesen, eine Position im Speicher bekommen hat, werden wieder die im letzten Abschnitt angesprochenen

Optimierungen auf dem Code vorgenommen. Diese Optimierungen verändern den in dieser Arbeit genutzten Vertex Shader nicht, da die Optimierungen nur auf mehrere verknüpfte Shader Auswirkungen hätte.

Der nächste Schritt des Linkers ist es mit Hilfe von „_mesa_ir_link_shader“, den im Shader enthaltenen „Uniforms“, Positionen zuzuweisen. Dies geschieht über die Methode „link_assign_uniform_locations. Hier werden zuerst alle genutzten „Uniforms“ mit einem Index versehen. Alle Uniforms, die im Shader noch keine Initialisierung hatten, werden mit dem Wert 0 belegt. Das beinhaltet zum Beispiel auch Sampler „Uniforms“. Nun wird der größte „Uniform“-Array bestimmt um zu garantieren, dass genug Platz für alle Arrayindizes ist. Der Lower Intermediate Code des Shaders wird nun auf „Uniforms“ durchsucht und wenn gefunden, im „UniformBlock“ unter dessen Name mit eindeutiger Position gespeichert. Die „Varyings“ werden an dieser Stelle noch nicht an spezielle Positionen gebunden, da diese erst im späterem Verlauf gebunden werden. Das „Linken“ ist praktisch um sicher zu stellen, dass für jede „Attribute“- und jede „Uniform“-Variable auch ihr Standort im Speicher reserviert wird. Allerdings gibt es auch im Lower Intermediate Code noch Variablen, die nicht für die Berechnung der Position benötigt werden. Um diesen unnötigen Speicherverbrauch zu reduzieren wird eine weitere Optimierung benötigt. Diese müsste die diejenigen Variablen aussortieren, welche nicht für die Positionsberechnung benötigt werden und nur für die zusätzlich vom userspezifizierten Variablen Speicherplatz reservieren. Allerdings ist diese Aussortierung nur möglich, wenn keine speziellen Treibereigenschaften erfüllt werden müssen. Ansonsten wäre es nötig die Variablen des IR-Codes noch auf spezielle Treibereigenschaften zu untersuchen und diejenigen Variablen, die für einen gewünschten Treiber benötigt werden, zu behalten.

Am Anfang des Abschnittes wurde erwähnt, dass das „Linken“ des Shaders auch das binden des „gl_shader_programs“ beinhaltet. Bisher wurde nur das Erstellen der Variablen betrachtet und das Binden an das „gl_shader_program“ noch nicht angesprochen. Das „gl_shader_program“ ist ein mesa-internes Konstrukt, dass zur Sammlung aller Informationen über verschiedene Shader dient. Dadurch ist es später einfacher auf verschiedene Informationen des jeweiligen Shaders zuzugreifen. Hier findet man unter anderem, Informationen über die Attribute, deren Positionen in einer Hashtable gespeichert sind, die Uniform Blöcke welche die Positionen der Uniforms gespeichert haben, aber auch den oder die Shader selbst. Viele Felder, die das Konstrukt mit sich bringt, wie Beispielsweise das über den Geometry Shader sind dabei für den Vertex Shader nicht von Nutzen und können daher ignoriert oder gelöscht werden. Insgesamt ist es dennoch eine sinnvolle Sache alle Informationen über einen oder mehrere Shader und deren Variablen unter einem Konstrukt zu speichern, um einen übersichtlichen und schnellen Zugriff auf alle wichtigen Werte zu bekommen, welche Mesa liefert. Außerdem kann der LLVM Compiler mit Hilfe der Informationen aus diesem Konstrukt einen LLVM-kompilierungsfähigen Code aus dem IR Code erstellen, weshalb auch dieses mesa-interne Konstrukt zur Weiterverarbeitung für diese Arbeit gewählt wurde.

4.4 LLVM Compiler

In den letzten Abschnitten des vierten Kapitels wurde der Input String des Vertex Shaders über mehrere Verfahren, wie dem Parsen und dem Linken, in ein Intermediate Code überführt, der in einem Konstrukt gegliedert wurde. Dieser bietet nun die Grundlage, die einzelnen Programmteile dem LLVM [CO14] Compiler zu übergeben und auf Maschinenebene auszuführen. Durch diese Schritte wird es möglich aus dem IR-Code den LLVM-Code zu erzeugen und ihn endgültig auszuführen. Dabei liefert das Programm die Position für einen speziellen Vertex, welches das Ziel dieser Arbeit ist.

Für den Letzten Teil des Programmes wurden die für diese Arbeit wichtigen Programmteile von Mesa herausgesucht, die für das Erzeugen des LLVM-Codes zuständig sind. Mesa bietet mit seinen Konstrukten wie der „Gallivm“, dem „draw_context“ sowie auch mit dem „Variant“, gute Möglichkeiten der LLVM Gliederung an. Deshalb wurde dieses Konzept auch für dieses Programm verwendet und nur in wenigen Punkten, die in den nächsten Abschnitten erläutert werden, angepasst.

4.4.1 Die Gallivm

Da bisher noch keine LLVM-spezifischen Objekte angelegt wurden, ist es notwendig diese im ersten Schritt für das Programm zu erzeugen. Dies ist durch ein Konstrukt das alle benötigten LLVM Objekte gliedert möglich. Wichtige LLVM-spezifische Objekte sind dabei die „ExecutionEngine“, der „PassManager“ und der „InstructionBuilder“. Diese drei Objekte bilden die Grundlage um LLVM-Code zu erzeugen und auszuführen. Zusätzlich wird wie in Abbildung 12 dargestellt noch das „modul“, welches eine weitere Gliederung in LLVM darstellt, der „provider“, welcher den „PassManager“ unterstützt, das „target“, welcher zu den „Opaque“-Typen in LLVM gehört, welche alle Arten von Daten referenzieren können und der „context“, welcher alle globalen LLVM Daten verwaltet, erstellt. Da Mesa all diese Grundbausteine von LLVM schon in der „Gallivm“ kombiniert, wird sie in dieser Arbeit verwendet um alle wichtigen LLVM-spezifischen Objekte zu erzeugen.

Die Gallivm aus Abbildung 12 ist wie auch das „gl_shader_program“ ein von Mesa definiertes Konstrukt. Der Unterschied zum „gl_shader_Program“ liegt darin, dass die Gallivm Felder vom LLVM-Typ besitzt. Solche Felder sind dabei zum Beispiel der Context, auf dem LLVM arbeitet oder das zu jedem Shader erzeugte Modul, indem sich LLVM dann während des Kompilierens bewegt. Bei der Erzeugung des Konstrukts wird außerdem eine „ExecutionEngine“ und ein „Builder“ angelegt, welche im Kapitel 3.5 ausführlich behandelt wurden. Diese sind zum Ausführen des LLVM Codes des Shaders notwendig.

```
struct gallivm_state
{
   LLVMModuleRef module;
   LLVMExecutionEngineRef engine;
   LLVMModuleProviderRef provider;
   LLVMTargetDataRef target;
   LLVMPassManagerRef passmgr;
   LLVMContextRef context;
   LLVMBuilderRef builder;
};
```

Abbildung 12, Aufbau einer „Gallivm“

Da in Mesa die „Gallivm“ genutzt wird, um die benötigten Basisobjekte des LLVM Codes für die Ausführung des Shaders zu erzeugen, kann darauf nicht verzichtet werden. Durch die „Gallivm“ werden alle wichtigen Objekte des LLVM Compilers erzeugt, die nun zur Nutzung bereit stehen.

4.4.2 Der „draw_context“

Im letzten Schritt wurden alle notwendigen LLVM Objekte erzeugt, welche allerdings noch nicht initialisiert oder einem Shader zugewiesen wurden. Um die „Gallivm“ eindeutig einem Shader zuordnen zu können, wird diese in diesem Schritt einem weiterem Konstrukt zugeordnet. Durch diese Zuordnung verknüpft man die LLVM Objekte mit einem speziellem LLVM Kontext, der wie zuvor angesprochen die Globalen Daten von LLVM verwaltet. Zudem müssen Pointer von LLVM auf den Zielshader, der bearbeitet werden muss gesetzt werden und etliche LLVM Typen während der Laufzeit noch initialisiert werden. Um dies zu realisieren bietet Mesa den „draw_context“ an, welcher die soeben genannten Aufgaben übernimmt.

Der „draw_context“ wird ausschließlich von Mesa genutzt. Der „draw_context“ ist ein privater Kontext für das Zeichenmodul. Dieser Kontext beinhaltet alle möglichen Informationen über Treibereigenschaften bis zum Geometrie Shader und ist somit größtenteils für den Vertex Shader uninteressant. Allerdings gibt es einen Punkt der auch bei der Erzeugung des Kontextes wichtig ist und man diesen genauer betrachten sollte. Bei der Initialisierung des „draw_context“ wird die sogenannte „draw_llvm“, welcher in Abbildung 13 dargestellt ist, erzeugt. Bei der Erstellung dieser „draw_llvm“ werden dabei wichtige Objekte erzeugt, die für das spätere Kompilieren des Shaders von Nutzen sind. Durch das Einrichten eines „draw_contexts“ wird die zuvor erzeugte Gallivm initialisiert und mit einem JIT Compiler gelinkt. Dies geschieht über den LLVM Befehl „LLVMInitializeNativeTarget“ und bewirkt, dass das Ziel später richtig mit den JIT Compiler verlinkt wird.

```
struct draw_llvm {
   struct draw_context *draw;

   struct draw_jit_context jit_context;

   struct gallivm_state *gallivm;

   struct draw_llvm_variant_list_item vs_variants_list;
   int nr_variants;

   /* LLVM JIT builder types */
   LLVMTypeRef context_ptr_type;
   LLVMTypeRef buffer_ptr_type;
   LLVMTypeRef vb_ptr_type;
   LLVMTypeRef vertex_header_ptr_type;
};
```

Abbildung 13, Aufbau des „draw_llvm“

Ohne diesen Schritt würde später der Shader nicht durch den JIT ausführbar sein und ist somit notwendig. Allerdings benötigt man dazu nicht den kompletten „draw_context“ und kann manuell den „draw_llvm“ durch Aufrufen von „draw_llvm_create“, erzeugen. Dies erspart nicht nur Arbeit, sondern hält auch überflüssigen, durch den Kontext belegten, Speicherplatz frei. Nicht alle Felder des „draw_llvm“ Konstrukts werden direkt beim Erzeugen initialisiert. So ist sind zum Beispiel Felder wie „jit_context“ und alle LLVM JIT Builder Typen noch nicht belegt. Diese werden im späteren Verlauf aber durch die Erzeugung eines „Variants“ belegt.

4.4.3 Der Variant

Da alle benötigten Objekte von LLVM nun erzeugt und initialisiert wurden, ist es möglich den LLVM Code aus dem IR-Code zu erzeugen und die Position des Vertex zu errechnen. Dazu wird wie in Kapitel 3.5 beschrieben einen „Instruction Builder“ und eine „ExecutionEngine“ benötigt. Da diese in den letzten Schritten erzeugt wurden, wird nun durch das zuvor erzeugte Konstrukt, indem der IR-Shader liegt, ein LLVM-ausführbarer Code erzeugt. Dies wird durch die Erzeugung einer LLVM-Funktion realisiert, die durch den „Instruction Builder“ erzeugt wird. Mit Hilfe des „PassManager“ werden noch nötige LLVM-interne Optimierungen auf dem erstellen LLVM-Code durchgeführt. Die „ExecutionEngine“ führt den Code dann aus und übergibt dem Programm einen „Function Pointer“ mit dem man die berechnete Position für den Vertex erhält. Um diesen Vorgang zu realisieren bietet Mesa die Erzeugung des „Variants“ an. Dieser führt oben genannte Schritte durch und speichert den „Function Pointer“ in der „jit_func“ ab.

Die Erzeugung des „Variants“ stellt einen wichtigen Punkt beim Kompilieren des Shaders dar. Hier wird der LLVM-generierte Code für einen Vertex Shader erstellt. Hierbei benötigt der Variant das zuvor angesprochenen, bisher noch nicht vollständige, „draw_llvm“ Konstrukt. Der Variant erzeugt einen LLVM-Typ „Vertex_header“, der Informationen über die Arraygröße und des Arraytyps des Shaders enthält. Die Hauptarbeit des „Variants“ liegt aber bei der Erzeugung des JIT Kontextes, der Erstellung des LLVM-generiertem Codes für

Funktionen und dem zuordnen der Variablen aus dem IR Code des Shaders. Dies alles wird durch die Methode „draw_llvm_generate“ realisiert. Das Erzeugen des LLVM-generiertem Codes aus dem IR Codes des Shaders läuft in folgenden Schritten ab:

- Die Gallivm und deren Informationen werden geladen und für den Bearbeitungszeitraum in temporären Variablen gespeichert. Dabei werden viele LLVM Objekte neu erstellt, die während der Laufzeit zum Kompilieren benötigt werden. Darunter zählen zum Beispiel: „func_type“, „context_ptr“, LLVM-Konstanten und „arg_types“, welche später Zwischenergebnisse der Berechnungen bereithalten.

- „JIT-Builder-Types“ werden erstellt. Dies bedeutet, dass für die Kompilierung benötigte LLVM Typen für verschiedene Strukturen noch erzeugt werden müssen. An dieser Stelle werden die restlichen Felder des „draw_llvm“ Konstrukts aus Abbildung 13, mit den verschiedenen Typen von LLVM, gefüllt. In Abbildung 14 ist ein Beispiel dieser Typen für den Kontext abgebildet.

- Die Funktions-Typen, die im letzten Schritt erzeugt wurden werden nun als LLVMFunctionType zusammengefasst.

- Eine neue LLVM-Funktion wird mit Hilfe der Funktions Typen erstellt und in das zuvor erzeugte Modul der Gallivm hinzugefügt. Diese Funktion wird gleichzeitig in das Variant gespeichert um später auch manuell Zugriff darauf zu haben. Bisher hat die Funktion aber noch keinen Funktionsblock und besteht nur aus dem „Header“.

- Nachdem alle Vorbereitungen getroffen wurden wird der „Function-body“ erstellt. Dabei erzeugt der Variant einen LLVMBlock im Kontext der Gallivm für die zuvor erzeugte Funktion der Funktions Typen.

- Der aus der Gallivm gewonnene Builder wird auf den Block gesetzt. Der Pointer des Builders zeigt nun auf den Funktionsblock, der die oben erstellte Funktion beschreibt.

- Der Variant erstellt durch LLVM Funktionen und durch die Informationen, die der Gallivm entnommen werden können, eine Methode für Schleifen. Diese definiert zum einem den Beginn einer Schleife und zum anderem durch einen andere Methode die End-Bedingung.

- In dem vorherigen erzeugten Block sind nun alle Schleifen und Funktionen des Shaders. Diese wurden mit Hilfe des IR Codes als LLVM Anweisung in den Block eingefügt und müssen noch ausgeführt werden.

- Die Ausführung des Blockes übernimmt ein Pass-Manager, der über den kompletten LLVM Code läuft und dabei die erzeugten LLVM Code Zeilen ausführt. Der Pass Manager wird auch der Gallivm entnommen und führt alle Funktionen, die als Vorlagen in der vorgegeben Funktion enthalten sind aus.

- Der Variant lädt den Pointer zum Globalen Code, der im letztem Schritt ausgeführt wurde in eine void-Funktion, die nun als JIT-Funktion berechnet werden kann und

übergibt sie dem Variant. Von dort aus ist es nun möglich, den Shader zu berechnen und die „gl_Position“ für ihn zu bekommen.

```
static LLVMTypeRef
create_jit_context_type(struct gallivm_state *gallivm,
                        LLVMTypeRef texture_type, const char *struct_name)
{
   LLVMTargetDataRef target = gallivm->target;
   LLVMTypeRef float_type = LLVMFloatTypeInContext(gallivm->context);
   LLVMTypeRef elem_types[5];
   LLVMTypeRef context_type;

   elem_types[0] = LLVMPointerType(float_type, 0); /* vs_constants */
   elem_types[1] = LLVMPointerType(float_type, 0); /* gs_constants */
   elem_types[2] = LLVMPointerType(LLVMArrayType(LLVMArrayType(float_type, 4),
                                                 DRAW_TOTAL_CLIP_PLANES), 0);
   elem_types[3] = LLVMPointerType(float_type, 0); /* viewport */
   elem_types[4] = LLVMArrayType(texture_type,
                                 PIPE_MAX_VERTEX_SAMPLERS); /* textures */
```

Abbildung 14, Ausschnitt der Typerzeugung

Im Struct „draw_llvm_variant“, der in Abbildung 15 zu sehen ist, ist die für den Vertex Shader interessante JIT-Funktion „draw_jit_vert_func“ nun mit dem soeben erstellendem LLVM Program durch einen „Function Pointer“ verknüpft. Zusätzlich lädt Mesa die in Abschnitt 4.2.1 Abbildung 10a, dargestellten Funktionen nun aus seiner Sammlung von LLVM-Funktionen in den ausführbaren LLVM-Code. Damit spart man sich bis zur Ausführung die Mitführung aller benötigten Standardfunktionen. Durch Ausführen dieser Funktion wird die „gl_Position“ des Shaders berechnet.

```
struct draw_llvm_variant
{
   LLVMValueRef function;
   LLVMValueRef function_elts;
   draw_jit_vert_func jit_func;
   draw_jit_vert_func_elts jit_func_elts;

   struct llvm_vertex_shader *shader;

   struct draw_llvm *llvm;
   struct draw_llvm_variant_list_item list_item_global;
   struct draw_llvm_variant_list_item list_item_local;

   /* key is variable-sized, must be last */
   struct draw_llvm_variant_key key;
};
```

Abbildung 15, Aufbau des „Variants“

Bisher wurden jedoch noch keine Attribute oder Uniforms dem Code übergeben. Deshalb ist es nicht möglich nun für den Vertex Shader einen Wert zu errechnen. Mesa bietet natürlich auch hier eine Lösung an, die jedoch nicht in der Zeit, die für diese Arbeit geplant war, gefunden worden ist. Sollte die Übertragung der Werte in die Uniforms und die Attributes implementiert werden, müsste man noch eine Möglichkeit finden, den zu Anfangs

produzierten IR-Code für den Shader mit einem der Mesa internen Konstrukte zu binden. Durch die vielen, für den Vertex Shader überflüssigen Konstrukte die Mesa nutzt, konnte in dieser Arbeit nicht das Richtige gefunden werden, in welches der IR-Code weitergeben werden müsste. Somit verweilt der IR-Code des Shaders noch in „gl_program“ und wird nicht korrekt mit den LLVM Funktionen und Mesa Structs verknüpft. Durch Implementieren dieser beiden Punkte wäre es möglich die Berechnung der „gl_Position“ durch ausführen der JIT-Funktion, manuell zu erhalten.

4.5 Das Programm

Um die einzelnen Schritte des Programmes nochmal nachvollziehen zu können, wird in diesem Abschnitt eine kurze Zusammenfassung gegeben, in welcher Reihenfolge die einzelnen Komponenten aufgerufen werden. Dabei werden die einzelnen Aufrufe den in den anderen Abschnitten erklärten Teilen zugeordnet.

```
//Init LLVM:

struct pipe_context *llvm_pipe = llvmpipe_create_context(NULL);
assert(!llvm_pipe);
struct gallivm_state *llvm_gallivm = gallivm_create();
assert(!llvm_gallivm);
struct draw_context *llvm_ctx = draw_create_context(llvm_pipe, true, llvm_gallivm);
assert(!llvm_ctx);

const struct draw_llvm_variant_key *llvm_key = draw_llvm_make_variant_key(llvm_ctx->llvm, "store");
struct draw_llvm_variant *variant = draw_llvm_create_variant (llvm_ctx->llvm, 2, llvm_key);

//END init LLVM
```

Abbildung 16, Initialisierung der LLVM Komponenten

Für das hier erstelle Konzept wurde zuerst die Erzeugung der LLVM Komponenten vorgenommen, so wird die „Gallvim“ aus Abschnitt 4.4.1 und der „Variant“ aus Abschnitt 4.4.3 mit allen dazugehörigen Konstrukten und Objekte erstellt. Das Variant wird an dieser Stelle allerdings noch nicht mit dem IR-Code des Shaders initialisiert, da bisher der Shader Input Code noch nicht geparst wurde. Danach folgen einige Initialisierungen der mesa-internen Konstrukte die verwendet wurden. So muss der Kontext, das „gl_program“ und der „gl_shader“ zuerst erstellt und initialisiert werden bevor der Lexer über den Shader Parsen kann. Auf Abbildung 17 werden diese Initialisierungen zusammenfassend als Aufrufe dargestellt. Der Shader der dem „gl_shader“ Konstrukt übergeben wird, ist in „ver_sh“ hinterlegt. Dieser ist derjenige der in Abbildung 8 dargestellt wurde.

```
initialize_context(ctx, (glsl_es) ? API_OPENGLES2 : API_OPENGL_COMPAT);

struct gl_shader_program *whole_program;

whole_program = rzalloc (NULL, struct gl_shader_program);
assert(whole_program != NULL);
whole_program->InfoLog = ralloc_strdup(whole_program, "");

_mesa_init_shader_program(ctx, whole_program);

 whole_program->Shaders =
     reralloc(whole_program, whole_program->Shaders,
     struct gl_shader *, whole_program->NumShaders + 1);

 assert(whole_program->Shaders != NULL);

 struct gl_shader *shader = rzalloc(whole_program, gl_shader);

 whole_program->Shaders[whole_program->NumShaders] = shader;

 whole_program->NumShaders++;

 shader->Type = GL_VERTEX_SHADER;
 shader->Source = ver_sh;
```

Abbildung 17, Initialisierung und Erzeugung aller benötigten Konstrukte

Im nächsten Schritt wird, wie in Abschnitt 4.1.1, der Lexer erzeugt und der Shader im Konstrukt „gl_shader“ geparst. Dies geschieht im Programm über den Aufruf von „compile_shader“. In Abbildung 18 wird die Funktion „compile_shader“ dargestellt, die dann _mesa_glsl_compile_shader“ wie beschrieben aufruft und den Shader in den LIR überführt.

```
void
compile_shader(struct gl_context *ctx, struct gl_shader *shader)
{
   struct _mesa_glsl_parse_state *state =
      new(shader) _mesa_glsl_parse_state(ctx, shader->Type, shader);

   _mesa_glsl_compile_shader(ctx, shader, dump_ast, dump_hir);

   /* Print out the resulting IR */
   if (!state->error && dump_lir) {
       printf("LIR: \n");
      _mesa_print_ir(shader->ir, state);
   }

   return;
}
```

Abbildung 18, Die Funktion „compile_shader“ für das Parsen des Shaders

Nachdem das Programm den Shader mit Hilfe des Lexers und Flex geparst hat und in den Lower Intermediate Code überführt hat, wird der Shader wie in Abschnitt 4.3 „gelinkt“. Da wie beschrieben zwei „Link“ Befehle existieren werden beide nacheinander aufgerufen. In Abbildung 19 sind beide Aufrufe „link_shaders“ und „_mesa_ir_link_shader“ zu sehen. Zur Sicherheit wurde hier noch überprüft, ob der Shader korrekt gelinkt wurde. Dies ist nötig um Sicherzugehen, dass kein Fragment Shader mit dem Vertex Shader verknüpft wurde.

```
if ((status == EXIT_SUCCESS) && do_link)  {
   printf("Alle Kommentare aus linker.cpp");
   link_shaders(ctx, whole_program);
   printf("GL_SHADER wurde gelinkt, Linkung zu MESA folgt: \n\n");
   _mesa_ir_link_shader(ctx, whole_program);
   printf("Alles erfolgreich gelinkt. gl_program kann mit whole_program genutzt werden! \n\n");
   status = (whole_program->LinkStatus) ? EXIT_SUCCESS : EXIT_FAILURE;

   if (strlen(whole_program->InfoLog) > 0)
  printf("Info log for linking:\n%s\n", whole_program->InfoLog);
}
```

Abbildung 19, „Link“ Befehle für den geparsten Shader

Das Linken des Shaders hat wie in den vorigen Abschnitten die Uniforms und die Attributes des Shaders in einer Hastable gespeichert. Diese Uniform können nun mit Werten über ihre „Location“ belegt werden. Über das zuvor erstelle Konstrukt „gl_program“, das durch den „Link“ Befehl nun mit dem Shader verknüpft wurde, und der Funktion „_mesa_get_uniform_location“ ist es möglich den Uniforms des Shaders durch „_mesa_uniform“ nun Werte zuzuweisen. In Abbildung 20 wurden die „locations“ der Uniforms für den Shader geladen und mit Werten verknüpft. Wie schon im vorherigen Abschnitt beschrieben ist es leider zeitlich nicht gelungen, den IR-Code des Shaders mit dem Variant zu verknüpfen. Deswegen gibt es bisher keine Funktion, die den Variant ausführt und den „Function Pointer“ für die „gl_Position“ zurückgibt.

```
GLint location0 = _mesa_get_uniform_location(ctx, whole_program, "ModelViewProjectionMatrix");
GLint location1 = _mesa_get_uniform_location(ctx, whole_program, "NormalMatrix");
GLint location2 = _mesa_get_uniform_location(ctx, whole_program, "LightSourcePosition");
GLint location3 = _mesa_get_uniform_location(ctx, whole_program, "MaterialColor");

const void *mvp = model_view_projection;
const void *nm = normal_matrix;
const void *lsp = LightSourcePosition;
const void *mc = materialColor;

_mesa_uniform(ctx, whole_program, location0, 25, mvp, GL_FLOAT);
_mesa_uniform(ctx, whole_program, location1, 25, nm, GL_FLOAT);
_mesa_uniform(ctx, whole_program, location2, 25, lsp, GL_FLOAT);
_mesa_uniform(ctx, whole_program, location3, 25, mc, GL_FLOAT);
```

Abbildung 20, Belegung der Uniforms des Shaders aus „gl_program“

5 Zusammenfassung und Ausblick

Der Schwerpunkt dieser Arbeit stützt sich auf Teile von Mesa3D, einem Open-Source Projekt zum Rendern von 3D Anwendungen. Dabei wird im dritten Kapitel gezeigt, durch welche Ansätze es Mesa schafft, einen Shader zu nutzen um Positionstransformationen auszuführen. Mit Hilfe von diesem Wissen wurde ein Konzept erstellt, das manuell vor der Ausführung von Rendering-Befehlen, die Position eines Vertex berechnet. Dieses hatte das Ziel die Position mit Hilfe eines Vertex Shaders schon vor der Ausführung der kompletten Pipeline zu berechnen. Im vierten Kapitel wird mit einem Beispiel vorgestellt, wie ein Shader bearbeitet werden muss, um ausgehend von seinem Input Strings durch LLVM verarbeitet werden zu können. Allerdings konnte das Programm nicht vollendet werden. Die Masse der C-Files die Mesa liefert und die Verschachtelungen der internen Funktionen von Mesa machten es in der Zeit unmöglich die richtigen Funktionsaufrufe zu finden. Das größte Problem wurde aber durch die internen Mesa3D Strukturen gegeben. Da Mesa mit vielen verschiedenen verschachtelten Strukturen arbeitet ist es oft sehr schwer alle von Mesa gegebenen Felder des Konstrukts mit Werten zu belegen. In den meisten Fällen war es entweder nicht möglich diese zu belegen oder nicht nötig, da dieses Projekt nur den Vertex Shader betrachtet. So konnte es auch vorkommen, dass man bis zu zehn Konstrukte ineinander geschachtelt vorfand und es nicht möglich war, herauszufinden welcher nun am Ende wirklich übergeben wurde. Zusätzlich arbeitet Mesa in seinen Methoden und Konstrukten sehr eng verknüpft mit dem Fragment Shader zusammen, welcher oft der Berechnung oder der Optimierung des Vertex Shaders im Weg stand.

Sollten diese gerade angesprochenen Probleme noch behoben werden, hätte man die Möglichkeit mit diesem Programm dynamisch während der Laufzeit die Position des Vertex, mit Hilfe des Vertex Shaders zu berechnen. Dadurch wäre es in Zukunft möglich, diese Implementierung zu verbessern, indem nur diejenigen Teile des Shaders ausgeführt werden, welche auch später für die Positionsberechnung verwendet werden. Durch diese Emulation der Positionstransformation würde man durch weitere Verfahren abschätzen können, wie die Laufzeit von bestimmten Rendering Befehlen sein würden. Dies würde den Vorteil bringen, dass man für Grafikanwendungen die Zeitkritisch laufen schon im Voraus die Berechnung verwerfen kann. Auch für andere eingebettete Systeme würde diese Abschätzung Vorteile bringen, indem auf Geräten die, zu lange für die Berechnungen des Rendering Befehls brauchen würden, schon von dem Rendern abbricht oder den Nutzer darüber informiert.

Abschließend würde ich sagen, dass Mesa zwar eine funktionsfähige Implementierung von OpenGL ES 2.0 darstellt, aber nicht in jedem Fall die richtige Wahl ist um alleinig die Position eines Vertex zu berechnen. In vielen Fällen, wäre es sinnvoller gewesen, die Mesa internen Konstrukte zu meiden und in dem einen oder anderen Fall selbst etwas zu implementieren. Leider reichte für die weiteren Implementierungen die Zeit nicht aus, sodass es nun offen bleibt, wie der IR-Code des Shaders in den Variant gelangt um hier eine LLVM-ausführbare Version des Shaders zu erhalten.

6 Literaturverzeichnis

[SO14a] Sourceforge.net (2014): The Mesa 3D Graphics Library. http://www.mesa3d.org

[KH14a] Khronos Group (2014): OpenGL ES – The Standard for Embedded Accelerated 3D Graphics. http://www.khronos.org/opengles/

[KH14b] Khronos Group (2014): OpenGL ES 2_X – The Standard for Embedded Accelerated 3D Graphics. http://www.khronos.org/opengles/2_X/

[IN14] Intel GmbH (2014): Intel 915GM Express-Notebookchipsatz http://www.intel.de /content/www/de/de/chipsets/mainstream-chipsets/mobile- chipset-915gm.html

[SI09] Simpson, Robert J.: The OpenGL ES Shading Language. Khronos Group, 2009

[CO14] Computer Science Department, University of Illinois at Urbana-Champaign (2014): The LLVM Compiler Infrastructure Project. http://llvm.org/

[BLW08] Brüning, Bernhard-Andreas; Latoschik, Marc Erich; Wachsmuth, Ipke: Interaktives Motion Capturing zur Echtzeitanimation virtueller Agenten. AG Wissensbasierte Systeme Bielefeld, International Media Informatics Berlin, 2008

[SA04] Segal, Marc; Akeley Kurt: The OpenGL Graphics System: A Specification. Silicon Graphics, Inc., 2004

[PO06] POP, Sebastian: The SSA Representation Framework: Semantics, Analyses and GCC Implementation. Mines Paris, 2006

[BM94] Brandis, Marc M; Mössenböck, Hanspeter: Single-Pass Generation of Static Single-Assignment Form for Structured Languages. ACM Transactions on Programming Languages and Systems, 1994

[CH14] Chris Lattner (2014): The Architecture of Open Source Applications: LLVM. http://www.aosabook.org/en/llvm.html

[SO08] Sourceforge.net (2008): Lexical Analysis With Flex, for Flex 2.5.37. http://flex.sourceforge.net/manual/

[JO03] Jones, Joel: Abstract Syntax Tree Implementation Idioms. Department of Computer Science, University of Alabama, 2003

[FR14] Freedesktop.org (2014): mesa/mesa – The Mesa 3D Graphics Library. http://cgit.freedesktop.org/mesa/mesa/tree/src/glsl/README

[LO14] Lopes, Crista: Analysis of Prog. Lang. Program Analysis. Lopes, Crista, 2014

[NA13] Nandivada, V. Krishna: CS3300 – Language Translators. IIT Madras, 2013 http://www.cse.iitm.ac.in/~krishna/cs3300/lecture5.pdf

[KH13a] Khronos Group (2013): Uniform (GLSL – OpenGL.org) http://www.opengl.org/wiki/Uniform_%28GLSL%29

[KH13b] Khronos Group (2013): GLAPI/glVertexAttribPointer – OpenGL.org. https://www.opengl.org/wiki/GLAPI/glVertexAttribPointer

[KH13c] Khronos Group (2013): GLAPI/glGetAttribLocation – OpenGL.org. https://www.opengl.org/wiki/GLAPI/glGetAttribLocation

[KH13d] Khronos Group (2013): Vertex Attribute – OpenGL.org. https://www.opengl.org/wiki/Vertex_Attribute

[KH13e] Khronos Group (2013): Vertex Attribute – OpenGL.org. http://www.khronos.org/opengles/sdk/docs/man/xhtml/glLinkProgram.xml

[JO11] Johannes Kepler Universität Linz, JKU (2011): The Compiler Generator Coco/R. http://www.ssw.uni-linz.ac.at/coco/

[PY13] Pyparsing.wikispace.com (2013): Pyparsing – home. http://pyparsing.wikispaces.com/

[SO14b] Sourcefoge.net (2014): Simple Examples – Lexical Analysis With Flex, for Flex 2.5.37. http://flex.sourceforge.net/manual/Simple-Examples.html#Simple-Examples

[SC13] Schnitzer, Stephan: Dynamische Ausführung von Positionstranformationen mittels OpenLG ES 2.0 Shaderprogrammen, Ausschreibung. Universität Stuttgart, 2013

[WT06] Wirth, N.; Tucker: Programming Languages, 2nd edition. The McGraw-Hill Companies, 2006

[AR13] Artale, Alessandro: Formal Languages and Compilers, Lecture X, Intermediate Code Generation. Faculty of Computer Science, Free University of Bolzano, 2013/14

[FS14] Free Software Foundation, Inc. (2014): Tokenization – The C Preprocessor. http://gcc.gnu.org/onlinedocs/cpp/Tokenization.html

[WE97] Welty, Christopher A.: Augmenting Abstract Syntax Trees for Program Understanding. IEEE, Vassar College, Computer Science Dept., 1997

[UN13] University of Illinois at Urbana-Champaign (20130): LLVM: LLVM-C interface to LVM. http://llvm.org/docs/doxygen/html/group__LLVMC.html

[LL14a] LLVM Compiler Infrastruktur (2014): Writing an LLVM Pass – LLVM 3.4 documentation. http://llvm.org/docs/WritingAnLLVMPass.html

[LL14b] LLVM Compiler Infrastruktur (2014): 4. Kaleidoscope: Adding JIT and Optimizer Support – LLVM 3.4 documentation. http://llvm.org/docs/tutorial/LangImpl4.html

[LL14c] LLVM Compiler Infrastruktur (2014): „clang“ C Language Family Fronted for LLVM. http://clang.llvm.org/

[LL14d] LLVM Compiler Infrastruktur (2014): DragonEgg – Using LLVM as a GCC backend. http://dragonegg.llvm.org/

[LL14e] LLVM Compiler Infrastruktur (2014): LLDB Homepage – The LLDB Debugger. http://lldb.llvm.org/

[RO10] Romanick, Ian: GLSL linker implementation, Zeile 1282 - 1296. Intel Corporation, 2010

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Unterschrift:

Stuttgart, 27.02.2014

Declaration

I hereby declare that the work presented in this thesis is entirely my own.
I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.
Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.
The electronic copy is consistent with all submitted copies.

Signature:

Stuttgart, 27.02.2014