Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BSC-2010-04

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Prozedurale Texturen in einem interaktiven Open-Source 3D-Modeller

Konni Hartmann

am Lehrstuhl für

Computergrafik und ökologische Informatik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

9. April 2010

Georg-August-Universität Göttingen
Zentrum für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel. +49 (5 51) 39-17 20 10
Fax +49 (5 51) 39-14 69 3
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 9. April 2010

Bachelorarbeit

Prozedurale Texturen in einem interaktiven Open-Source 3D-Modeller

Konni Hartmann

9. April 2010

Betreut durch Prof. Dr. Winfried Kurth,
Reinhard Hemmerling
Lehrstuhl für Computergrafik und ökologische Informatik
Georg-August-Universität Göttingen

Danksagung

Am Entstehungsprozess dieser Arbeit waren viele Personen direkt und indirekt beteiligt, denen ich hier meinen Dank aussprechen möchte.

An erster Stelle danke ich Prof. Dr. Winfried Kurth für seine freundliche Betreuung meiner Arbeit. Insbesondere die Geduld und Flexibilität, die Sie mir in den letzten Wochen entgegengebrachten, waren ausschlaggebend für die erfolgreiche Fertigstellung.

Ich danke Reinhard Hemmerling, der mit mir in langen Gesprächen und Emails Vorschläge und Idee zu meiner Bachelorarbeit ausgetauscht hat. Jederzeit war ich willkommen und konnte mir Fragen kompetent beantworten lassen.

Mein besonderer Dank gilt meiner Lebensgefährtin und Freundin Daniela Foese. Ohne ihre tatkräftige Unterstützung und ihre nicht endende Geduld hätte ich diese Arbeit niemals beenden können.

Ich danke auch meiner Mutter Freia Hartmann für die Zeit im Januar, in der Sie mich zum Schreiben motivierte, obwohl sie selbst mehr als genug zu tun hatte.

Desweiteren möchte ich mich bei meinen Kommilitonen, insbesondere André Arens und Fabian Glaser, für die schöne Zeit in Göttingen bedanken. Ich hoffe der Master-Studiengang wird genauso lustig und lehrreich.

Aufgabenstellung

GroIMP ist eine in Java programmierte Modellierungssoftware, welche am Lehrstuhl Grafische Systeme entwickelt wird. Zur Visualisierung eines damit erzeugten Modells existiert bereits eine einfache OpenGL-Darstellung. Diese soll erweitert werden. Die bisherige OpenGL-Darstellung visualisiert den aktuellen Szene-Graphen, indem für jeden (sichtbaren) Knoten ein entsprechendes Objekt gezeichnet wird. In der aktuellen Implementierung dieser Darstellung lassen sich aber nur einfache Farben oder Bilder als Texturen für Objekte verwenden. Optionen, wie in Echtzeit berechnete prozedurale Texturen funktionieren jedoch nicht. In GroIMP lassen sich für die Objekte auch prozedurale Materialien definieren. Diese sollen in ein GLSL-Programm (Fragment Shader) transformiert werden, welches dann für einen zu zeichnenden Bildpunkt des Objekts die Farbe berechnet. Dabei soll auch eine Beleuchtungsberechnung pro Pixel vorgenommen werden. Zur Beschleunigung des Zeichenvorgangs ist ein „deferred shading“ zu implementieren.

Inhaltsverzeichnis

Abkürzungsverzeichnis

BPS	Bilder pro Sekunde
CSG	Constructive-Solid-Geometry
FBO	Frame-Buffer-Object
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
GroIMP	Growth Grammar-related Interactive Modelling Platform
HDR	High Dynamic Range
JOGL	Java Bindings for OpenGL
LDR	Low Dynamic Range
OGL1	OpenGL 1.1 basierte Vorschau in GroIMP
OpenGL	Open Graphics Library
PCF	Percentage Closer Filtering
Proteus	Procedural Texture System
Twilight (PTR)	Twilight Pathtracer
Twilight (STDR)	Twilight Standard Raytracer
VBO	Vertex-Buffer-Object

1 Einleitung

1.1 Motivation

Diese Arbeit befasst sich mit der Erweiterung der 3D-Modellierungsplattform GroIMP um eine in OpenGL implementierte Ansicht, welche die in GroIMP vorkommenden prozeduralen Materialien detailgetreu darstellt.

GroIMP verfügt bereits über eine OpenGL Darstellung, allerdings beschränkt sich diese auf minimale Anforderungen und nutzt entsprechend nur grundlegende OpenGL 1.1 Methoden. Die Zielsetzung dieser Implementierung beruht nach wie vor darauf, dass sie ein weites Spektrum verschiedenster Systeme unterstützt und so auch auf älteren Computern nutzbar bleibt. Somit ist sie nicht in der Lage, die Fähigkeiten moderner Grafikkarten (kurz GPU, Graphics Processing Unit), wie sie seit einigen Jahren zu der Standardausrüstung von Heimrechnern zählen, effizient zu nutzen. Effizient bedeutet, eine raytracernahe Vorschau aller in GroIMP spezifizierter Parameter zu erlauben. Bei der Arbeit mit GroIMP spart eine direkte Vorschau Zeit, da bei sofortiger grafischer Rückmeldung Änderungen dargestellt werden können, welche bisher einen langen Rendervorgang benötigten. Unerwartete Effekte können so frühzeitig erkannt und behoben werden. Insbesondere Prozedurale Texturen werden bisher gar nicht oder nur unzureichend dargestellt. Das liegt daran, dass bis vor einigen Jahren die technischen Voraussetzungen programmierbarer Grafikkarten nicht gegeben waren. Mit der OpenGL Shading Language (GLSL) gibt es seit OpenGL 2.0 eine leicht zugängliche Möglichkeit der Programmierung von Grafikkarten. Das ermöglicht die von Grafikkarten beschleunigte Berechnung und Darstellung prozeduraler Texturen. Diese Entwicklung motiviert zusätzlich, die OpenGL Darstellung in GroIMP zu überarbeiten.

Zum Schluss bleiben noch weitere Techniken zu nennen, die mit der Einführung von OpenGL 1.1 noch nicht verfügbar waren.

Dies ist unter Anderem die Lichtberechnung pro Pixel, welche über GLSL-Programme realisiert werden kann.

Desweiteren erlaubt die OpenGL Erweiterung ARB_shadow die einheitlich Berechnung und Darstellung von Echtzeitschatten auf den Grafikkarten der Hersteller NVIDIA, ATI und Intel. Die Bildberechnung mit Fließkommazahlen erlaubt beispielsweise High Dynamic Range Rendering (HDR Rendering, Zeichnen mit hohem Dynamikumfang). Diese Techniken werden immer öfter in Anwendungen genutzt und heben somit das Verlangen vieler Nutzer nach visuell ansprechender Darstellung. Diesem Trend folgt auch die Implementierung dieser Arbeit, indem sie Methoden, die sonst eher im Bereich der Computerspiele verwendet werden, GroIMP zu Verfügung stellt.

1.2 Aufbau der Arbeit

Diese Arbeit beginnt in Kapitel zwei, indem der Ausgangspunkt der Arbeit erläutert wird. Es werden die technischen Voraussetzungen, welche GroIMP bietet, erklärt und deren Probleme, in Hinblick auf diese Arbeit, erläutert. Desweiteren werden die Grundlagen für die Nutzung von OpenGL und der OpenGL Shading Language gelegt, sowie einige nützliche Techniken angesprochen, welche in der Arbeit Verwendung finden.

Kapitel drei beschreibt die Lösungsidee zur Darstellung prozeduraler Texturen. Es wird die Generierung von GLSL-Codes zu in GroIMP definierten Shadern besprochen. Es wird außerdem

die Bilderzeugung durch einzelne Zeichenschritte vorgestellt. Um die Zielsetzung zu erreichen werden hierbei einige Modifikationen zu den in Kapitel zwei erläuterten Techniken präsentiert.

Kapitel vier erläutert die Implementierung des erarbeiteten Systems in der Programmiersprache Java. Zunächst wird die Architektur vorgestellt. Es werden Methoden zum Zwischenspeichern sowohl für Formen, als auch für Materialien vorgestellt und eine implementierungsspezifische Sicht auf die Generierung von GLSL-Shadern gezeigt. Es wird speziell auf die Nutzung von Texturen und Zeichenpuffern in OpenGL eingegangen, welche zur Realisierung der in Kapitel drei vorgestellten Bildkomposition nötig sind.

Kapitel fünf bewertet die Implementierung qualitativ sowie quantitativ und erläutert bestehende Probleme.

In Kapitel sechs wird die Arbeit zusammengefasst und bewertet. Desweiteren wird ein Ausblick auf mögliche Verbesserungen und Erweiterungen der vorgestellten Implementierung gegeben.

2 Grundlagen

In diesem Kapitel werden die Grundlagen für die Implementierung einer 3D-Ansicht für die Anwendung GroIMP gelegt. Zunächst soll ein Überblick über GroIMP gegeben werden. Für diese Arbeit entscheidend ist der Aufbau von Materialien, insbesondere deren prozedural berechnete Anteile. Zudem muss die Graphstruktur, über die die Szene beschrieben wird, beachtet werden. Es folgt eine kurze Erläuterung zum Phong-Lichtmodell. Die grafische Ausgabe wird mittels OpenGL realisiert, weshalb eine Einführung in ausgewählte Aspekte von OpenGL beschrieben wird. Der letzte Abschnitt präsentiert einige bekannte Techniken der interaktiven Bilderzeugung.

2.1 GroIMP

GroIMP (Growth Grammar-related Interactive Modelling Platform)[1] ist eine 3D-Modellierungsplattform, welche vom Lehrstuhl für ökologische Informatik an der Georg-August Universität Göttingen entwickelt wird. Neben der interaktiven 3D-Vorschau besitzt GroIMP einen internen Raytracer mit Namen Twilight. Dieser kann über verschiedene Techniken wie Raytracing [kurz: Twilight (STDR)], Pathtracing [Twilight (PTR)] oder Photon Mapping hochwertige Bilder erzeugen. Ziel dieser Arbeit ist es die Darstellung des Path- und Raytracers in Twilight möglichst nahe zu kommen.

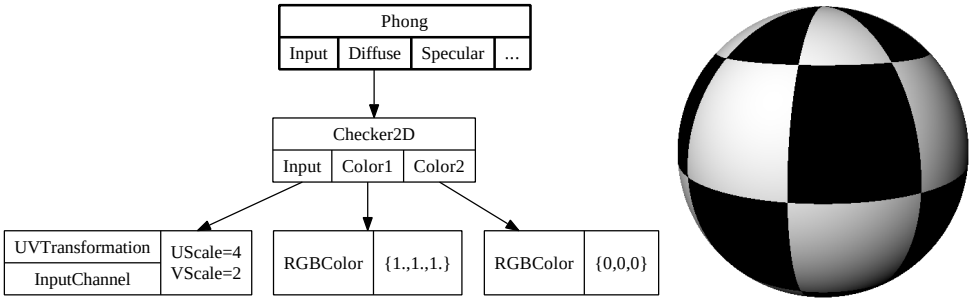
Im Folgenden werden die Bestandteile in GroIMP, welche für die Implementierung einer 3D-Darstellung wichtig, sind erläutert.

2.1.1 Material-Shader

Materialien in GroIMP sind in einer Baumstruktur angelegt. Hierbei bestimmt die Wurzel die Art des Materials. Diese können sein: RGBA, Lambert, Phong, SunSkyLight, SideSwitch und AlgorithmSwitch. GroIMP definiert noch weitere Materialtypen, die allerdings als Teil des GPU-Raytracers „Sunshine“ entworfen wurden und somit nicht in dieser Arbeit betrachtet werden.

Materialien, können Parameter besitzen, die durch austauschbare Funktionen, in GroIMP ChannelMap genannt, bestimmt werden. ChannelMaps sind beispielsweise feste Farbwerte (RGBColor), eine Textur (Image) oder durch eine prozedural berechnete Funktion. Die wichtigsten prozeduralen ChannelMaps sind Turbulence (pseudozufällige Muster, unter Anderem zur Darstellung von Wolken, Marmor oder Feuer), Checkerboard 2D (ein Schachbrettmuster) und Blend (farbkodierte Darstellung eines Zahlenbereichs). Diese sind für die interaktive Vorschau entscheidend, da sie bisher nicht korrekt angezeigt werden können. Diese sind Träger von Eigenschaften, die aus anderen ChannelMaps berechnet werden können. So entsteht durch das Verketten von ChannelMaps eine Baumstruktur mit beliebig komplexer Tiefe. Anhand des Beispiels in Abbildung 2.1 ist die Verknüpfung einzelner ChannelMaps zu erkennen.

[1] http://www.grogra.de (Stand: März 2010)



(a) Material Baum (b) Ausgabe

Abbildung 2.1: Beispiel „Checker2D“. Das Material ist eine Instanz des Phong-Materials mit einem Schachbrettmuster in der diffusen Komponente. In 2.1a ist der reduzierte Baum des Materials zu sehen. In 2.1b ist einer Kugel das Material zugewiesen. Die Kugel wurde mit Twilight (STDR) gezeichnet.

2.1.2 Szenegraph

GroIMP sichert die Szene in einem unidirektionalen Graphen. Der Graph bildet die Basis der in GroIMP eingebundenen Relationalen Wachstumsgrammatik XL. Über XL Befehle kann der Graph manipuliert werden. Hierbei können Knoten erzeugt, verschoben, entfernt oder Eigenschaften verändert werden. XL betrifft nur die Manipulation des Graphens und nicht die Darstellung, weshalb auf Erläuterungen verzichtet wird. Weitere Informationen sind der Dissertation von Ole Kniemeyer [18] zu entnehmen .

Die zweite Möglichkeit der Graphmanipulation bietet die interaktive Darstellung der Szene. In dieser können manuell neue Primitive eingefügt oder bestehende Objekte verändert werden. Durch direkte Interaktion können Objekte transformiert, Materialeigenschaften geändert oder objektspezifische Parameter angepasst werden. Änderungen an Objekten werden im zugrunde liegenden Graphen übernommen. Das bedeutet, fügt der Benutzer eine Kugel in der interaktiven Ansicht in die Szene ein, so wird im GroIMP-Graphen ein Knoten erzeugt, der Träger der Eigenschaften der Kugel ist.

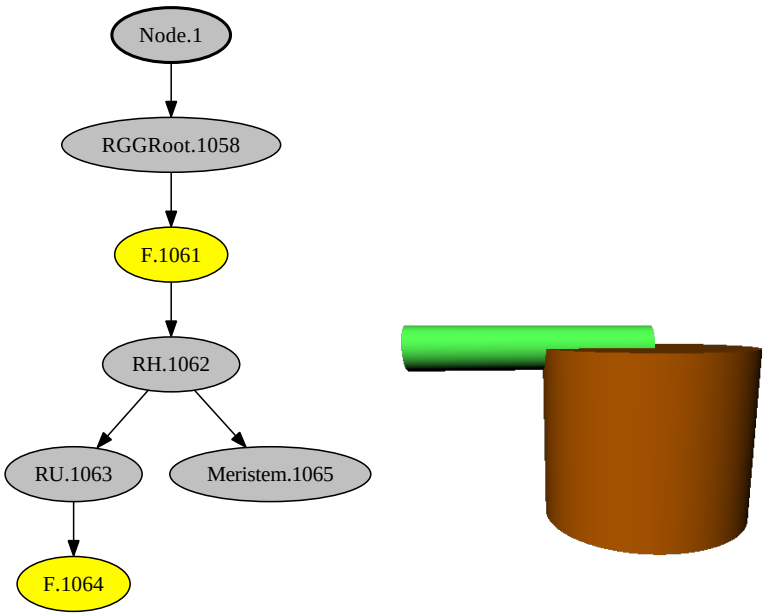
Die Knoten im Graphen können zeichenbare Objekte beinhalten, welche es darzustellen gilt. Hierzu wird zunächst der Szenegraph, eine Baumstruktur welche eine Sicht auf den GroIMP-Graphen darstellt, traversiert. Für jeden Knoten wird die Eigenschaft „Shape“ (zu deutsch „Form“) abgefragt und wenn möglich gezeichnet. Es gibt in GroIMP eine Vielzahl an nicht-zeichenbaren Knoten, welche Transformationen darstellen, Parameter für Kind-Knoten festlegen oder Träger anderer Informationen sind. Diese Knoten müssen beim Traversieren des Graphen zwar beachtet, aber nicht gezeichnet werden (vergleiche Beispiel in Abbildung 2.2).

2.1.3 Lichtquellen

GroIMP definiert vier Typen von Lichtquellen:

- Punktlichtquellen, die von einem Punkt in der Szene in alle Richtungen gleichstark strahlen
- Kegellichtquellen (spotlights), die mit festgelegtem Öffnungswinkel von einem Punkt aus gerichtet einen Teil der Szene beleuchten
- direktionale Lichtquellen, die gleichmäßig mit parallelen „Lichtstrahlen“ die gesamte Szene beleuchten
- Flächenlichtquellen, die als Grundstruktur ein zweidimensionales Parallelogramm besitzen

Flächenlichtquellen werden aufgrund ihrer komplexen Schatten durch Punktlichtquellen angenährt. Eine weitere oft genutzte Art von Lichtquellen ist die Klasse „SunSkyLight“. Diese ist gleichzeitig



(a) Szenegraph (b) Ausgabe

Abbildung 2.2: Beispiel eines Szenegraphen 2.2a in GroIMP, sowie die mit Twilight (STDR) gezeichnete Szene 2.2b. Nur die im Graphen gelb markierten Knoten werden gezeichnet. Das Beispiel zeigt die erste Ableitung des FTree Modells, welches Teil der Beispieldateien zu GroIMP ist.

Lichtquelle und Material. Als Material für einen Sky-Knoten im Szenegraphen bildet dieser das von Preetham beschriebene Modell der Tagesbeleuchtung [27] ab und wird zum Zeichnen eines Himmels genutzt. Als Lichtquelle dient die gesamte Himmels-Halbkugel, welche von „SunSkyLight“ definiert wird. Der in GroIMP vorhandene Standard Raytracer, Twilight (STDR), nähert diesen durch eine große Anzahl an Punktlichtquellen an. Twilight (PTR) verrechnet die Beleuchtung durch den Himmel direkt. Beide Methoden sind für eine interaktive Darstellung zu langsam und demnach ungeeignet.

2.2 Phong-Beleuchtungsmodell

Das primäre Beleuchtungsmodell in GroIMP ist das Phong-Beleuchtungsmodell welches 1975 von Bui-Tuong Phong [25] aufgestellt wurde. Hierbei wird die diffuse Reflektion über das Lambertsche Beleuchtungsmodell berechnet (mit diffusen Anteil k_d) . Das Phong Modell definiert zusätzlich die spekulare Reflektion k_S. Sei $\overrightarrow{L}$ der Richtungsvektor zu einer Lichtquelle von einem betrachteten Punkt. Sei $\overrightarrow{V}$ der Vektor vom Punkt zur Kamera, $\overrightarrow{N}$ die Normale der betrachteten Oberfläche sowie $\overrightarrow{R}$ der an der Oberfläche reflektierten Lichtvektor $\overrightarrow{L}$. Beschreibt weiterhin θ den Winkel zwischen der Normale und dem Lichtvektor und ϕ den Winkel zwischen $\overrightarrow{L}$ und $\overrightarrow{R}$. Somit lässt sich die Farbe eines beleuchteten Punktes, f_{Phong}, als

$$f_{Phong}\left(\overrightarrow{L},\overrightarrow{V}\right) = k_d \cdot \cos\theta + k_s \cdot \cos^{n_s}\phi = k_d \cdot \left(\overrightarrow{N}\cdot\overrightarrow{L}\right) + k_s \cdot \left(\overrightarrow{R}\cdot\overrightarrow{V}\right)^{n_s}$$

beschrieben. In Abbildung 2.3 ist der zweidimensionale Fall dargestellt. Wichtig ist, dass die diffuse Komponente nur von der Blickrichtung und der Normale abhängt, während bei der spekularen Reflektion auch der Richtungsvektor der Lichtquelle beiträgt.

In der 3D-Programmierung wird das Phong-Modell oft durch einfache Annäherungen beschrieben.

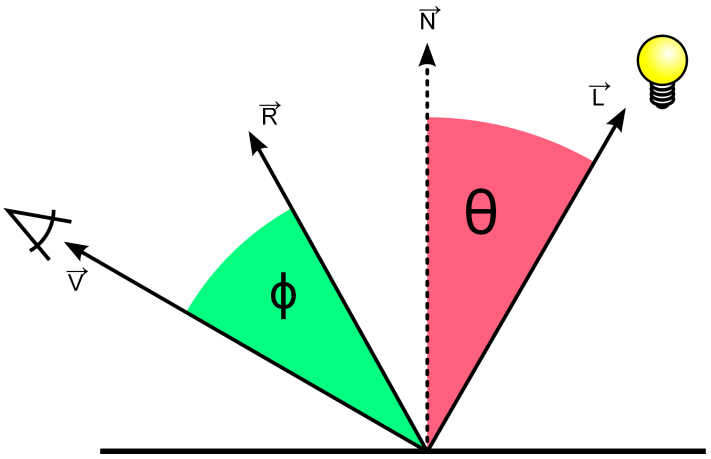
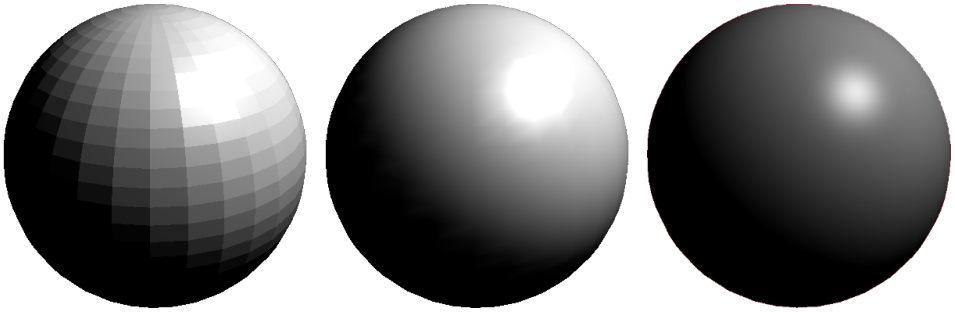


Abbildung 2.3: Skizze zur Beleuchtung nach dem Phong-Modell.



(a) (b) (c)

Abbildung 2.4: Verschiedene Beleuchtungsverfahren im Vergleich. 2.4a zeigt Flat Shading, 2.4b Gouraud Shading sowie 2.4c Phong Shading.

Flat Shading Im Unterschied zur Phong-Beleuchtung bestimmt Flat Shading nicht in jedem Punkt, sondern für jede Fläche einen festen Farbwert. Somit genügt die Kenntnis um die Normale der Fläche und die Position eines Punktes auf dieser, um die Fläche zu beleuchten. Diese Methode lässt sich entsprechend sehr schnell auswerten.

Gouraud Shading Henri Gouraud entwickelte bereits 1971 das nach ihm benannte Gouraud Shading [11]. Dieses wird inzwischen als eine schnelle Approximierung des Phong Lichtmodelles genutzt. Beim Gouraud Shading werden diffuse und spekulare Lichtbeiträge nur in jeder Ecke einer Fläche ausgewertet und über die gesamte Fläche linear interpoliert.

Abbildung 2.4 zeigt die genannten Verfahren im Vergleich. Sowohl Flat, als auch Gouraud Shading werden direkt von OpenGL unterstützt, während Phong Shading manuell programmiert werden muss.

2.3 OpenGL

Da GroIMP in Java implementiert ist und so auf einer Vielzahl unterschiedlicher Betriebssysteme nutzbar ist, muss die Darstellung über eine Schnittstelle erfolgen, die gleichermaßen verbreitet ist.

Die Schnittstelle OpenGL (Open Graphics Library) wurde ursprünglich von SGI (Silicon Graphics) entwickelt[2] mit dem Ziel eine plattformunabhängige Spezifikation zur Programmierung von

[2] http://www.sgi.com/products/software/opengl/ (Stand: April 2010)

2D- und 3D-Computergrafik zu schaffen. Die Spezifikation wurde seitdem vom OpenGL ARB (Architecture Review Board) [3], einem Konsortium mit stimmberechtigten Mitgliedern aus einer Reihe von Firmen wie Apple, AMD/ATI, Intel und NVIDIA. Seit Juli 2006 wird OpenGL von der Khronos Group, zu der auch das ARB gehört, weiterentwickelt.

Somit erfüllt OpenGL die Anforderung plattformunabhängig zu sein. DirectX als Alternative zu OpenGL ist nur unter Windows-Betriebssystemen verfügbar und scheidet so für die Darstellung dreidimensionaler Szenen in GroIMP aus. OpenGL wird in GroIMP über „JOGL“ (Java Bindings for OpenGL)[4], einer Sammlung von Java-Wrapperklassen, angesprochen. Bislang wird die Version 1.1.1a der Bibliothek genutzt, die alle benötigten Funktionen und Erweiterungen zu Verfügung stellt. Zum Zeitpunkt dieser Arbeit ist die aktuelle JOGL Version 2.0 noch im Beta-Stadium, weshalb weiterhin die Version 1.1.1a vom Mai 2008 genutzt wird.

Für die Implementierung bietet OpenGL in der Version 2.0 bereits eine standardisierte Schnittstelle zur Programmierung der Grafikkarte. Somit wird im weiteren die aktuelle Version 3.x, die substantielle Änderungen der API vornimmt, nicht betrachtet.

Es folgt ein kurzer Einblick in die genutzten OpenGL-Funktionen.

2.3.1 Beleuchtung in OpenGL

OpenGL nutzt zur Beleuchtung der Szene das Blinn-Phong-Modell [2] welches die selben Parameter wie das Phong-Modell besitzt, allerdings etwas schneller auszuwerten ist. Die Beleuchtung wird zusätzlich nach der Idee des Gouraud-Shadings (2.2) nur in jedem Eckpunkt der Geometrie berechnet und auf der eingeschlossenen Fläche interpoliert. Neben der spekularen und diffusen Reflektion wird das Lichtmodell durch weitere Terme ergänzt. Dies sind eine ambiente sowie eine emittierende Farbe.

Umgebungslicht (engl.: ambient light) ist eine nicht-physikalische Lichtquelle, die die gesamte Szene beleuchtet. Sie ist in jedem Punkt der Szene gleich stark und wird über die entsprechende Farbkomponente des Materials modifiziert. Die Idee besteht darin auch Teile der Szene darstellen zu können, die indirekt beleuchtet werden. Die emittierende Farbe gibt die Farbe und Stärke an, mit der ein bestimmtes Material selbst leuchtet. Sei k_e der emittierende Farbwert sowie $k_a := m_a \cdot l_a$ das Produkt aus ambienter Farbe des Materials m_A und der ambienten Helligkeit der Lichtquelle l_a, so lassen sich diese als Konstanten in die Gleichung hinzufügen:

$$f_{Light} = f_{Blinn-Phong}\left(\overrightarrow{L}, \overrightarrow{V}\right) + k_a + k_e$$

2.3.2 Fixed-Function Pipeline

OpenGL in der Version 2.0 nutzt zur Darstellung von Grafiken zwei verschiedene Pipelines. Die Fixed-Function Pipeline bietet viele Effekte wie Beleuchtung oder Nebel oder verschiedene Arten der Texturierung von Objekten über feste Schnittstellen an. Diese lässt sich grob aus der Abbildung 2.5 entnehmen.

OpenGL verarbeitet Gruppen von Vertizes, Punkte im Raum, denen Attribute zugewiesen werden können. Diese werden vom Programmierer genutzt um Geometrie zu beschreiben. Vertizes werden zunächst einzeln transformiert. Dann werden Vertizes, nach Angaben des Programmierers, zu Primitiven zusammengefasst. Diese sind meistens Dreiecke, können aber auch Vierecke, Punkte, Linien oder komplexere Körper darstellen (Beispielsweise Dreiecksfächer). Danach werden die Primitiven überprüft, ob sie gezeichnet werden müssen. Sollen sie gezeichnet werden, so werden die Primitiven im Rasterisierungsschritt in einzelne Fragmente zerlegt. Diese stellen konkrete

[3] http://www.opengl.org/about/arb/ (Stand: April 2010)
[4] http://kenai.com/projects/jogl/pages/Home (Stand: 07.03.2010)

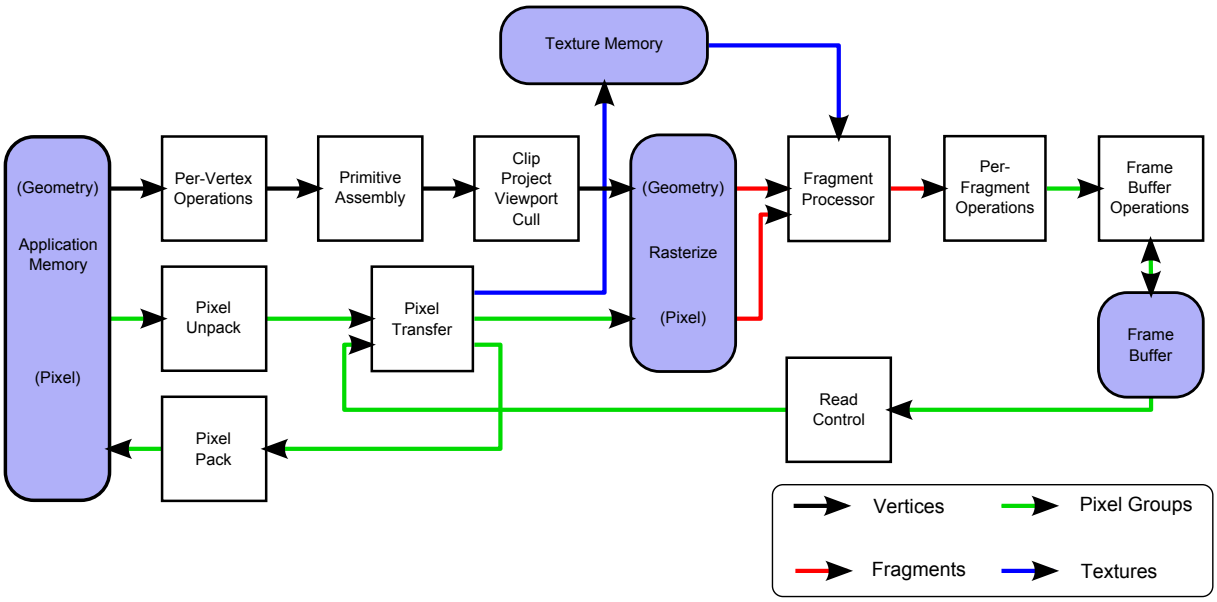


Abbildung 2.5: OpenGL Fixed-Function Pipeline. Struktur nach [28, Kapitel 1.6 Abbildung. 1.1]

Punkte auf der Oberfläche der Primitiven dar und sichern Informationen wie Position und Farbe. Insbesondere die Farbe eines Fragmentes wird in der Fixed-Function Pipeline durch die Farbe der Vertizes sowie genutzte Texturen bestimmt. Fragmente werden dann in einen Puffer gezeichnet und liegen dort als Pixel vor. Alternativ können Fragmente verworfen werden. Somit nehmen Fragmente mit den selben xy-Koordinaten deren Platz ein. In OpenGL besteht die Möglichkeit sowohl Pixel, als auch Vertex-Informationen in den Speicher der Anwendung zurückzulesen und dort zu verarbeiten.

Neben der Fixed-Function Pipeline existiert noch die Programmable Pipeline, die im Zusammenhang mit der OpenGL Shading Language erläutert wird.

2.3.3 Zeichenpuffer

OpenGL kennt mehrere Speicher, in denen Informationen zu gezeichneten Bildern gesichert werden. Einige können genutzt werden, um das Schreiben neuer Werte zu unterbinden. Die gängigsten sind:

Farbpuffer Dieser sichert Farbinformationen, welche in verschiedenen Formaten vorliegen können. Meist werden RGB-Farbwerte und ein optionaler Alpha-Wert gesichert. Die Genauigkeit kann vom Benutzer bestimmt werden. Farbpuffer werden meist genutzt um die zweidimensional Darstellung einer Szene zu speichern. Da OpenGL nicht festlegt, wie die enthaltenen Farben zu interpretieren sind, lassen sich auch andere Informationen in Farbpuffern sichern.

Tiefenpuffer Der Tiefenpuffer sichert zu jedem Pixel den Abstand zur Kamera auf der z-Achse. Der Tiefenpuffer wird genutzt um zu entscheiden, wann ein Fragment verworfen oder in Farb-, Tiefen- und Stencilpuffer geschrieben werden soll. Dieser Test wird Tiefentest genannt. OpenGL vergleicht den gesicherten z-Wert mit dem des aktuellen Fragmentes. Der Programmierer legt zuvor fest, bei welchem Vergleichsergebnis Änderungen an den aktiven Puffern vorgenommen werden.

Stencilpuffer Der Stencilpuffer, auch Schablonenpuffer genannt, sichert zu jedem Pixel eine Bitfolge, die durch Bitoperationen verändert werden kann. OpenGL erlaubt hierbei bekannte

Operationen, wie das Inkrementieren des Wertes oder binäre Operatoren. Operationen auf dem Stencilpuffer können an Bedingungen geknüpft werden. Zudem kann ein Stencil-Test, welcher ähnlich dem Tiefentest den Inhalt des Stencilpuffers mit einem Referenzwert vergleicht, genutzt werden, um das Zeichnen in bestimmte Bereiche eines Puffers zu untersagen.

2.3.4 Projektionen

OpenGL besitzt keine explizite Kamera, durch die die Szene betrachtet wird. Eine virtuelle Kamera sitzt im Zentrum des von OpenGL genutzten Koordinatensystems. Objekte in OpenGL werden über die homogenen Koordinaten ihrer Vertizes beschrieben. Um ein Objekt anzuzeigen die Vertizes so verschoben, rotiert und skaliert, dass sie vom Zentrum entlang der negativen z-Achse liegen. Diese Transformation wird in OpenGL mittels einer homogenen 4x4-Matrix beschrieben, welche auch WorldToView-Matrix genannt wird. Sie beschreibt die Transformation der homogenen Koordinaten aus dem globalen Koordinatensystem in das Sichtkoordinatensystem.

Das Sichtkoordinatensystem wird durch eine weitere Transformationen auf eine zweidimensionale Ebene, im folgenden Bildebene genannt, projiziert. Prinzipiell ist zwischen zwei verschiedenen Arten von Ansichten zu unterscheiden:

- Perspektivische Projektionen verkürzen Strecken mit zunehmender Entfernung zum Augpunkt. Sie geben ein vertrautes Bild der Szene, da die menschliche Wahrnehmung auch perspektivisch ist.
- Orthogonale Projektionen zeigen Objekte und Strecken abstanzinvariant an. Sie sind besonders beim präzisen Erstellen von Szenen nützlich.

Beide Arten von Projektionen lassen sich über eine 4x4-Matrix beschreiben, die in OpenGL entsprechend Projektionsmatrix genannt wird. Es werden weiterhin zwei parallel zur Sichtebene liegende Ebenen festgelegt: die „near-plane“ und die „far-plane“. Die „near-plane“ gibt den minimalen Abstand an, den ein Fragment von der Bildebene in z-Richtung haben muss, um gezeichnet zu werden. Die „far-plane“ beschreibt den maximalen Abstand zur Bildebene, in dem ein Fragment noch gezeichnet werden kann. Da Grafikkarten nur mit begrenzter Rechengenauigkeit arbeiten definiert das Verhältnis zwischen near- und far-plane zusammen mit der Auflösung des Tiefenpuffers die Auflösung in z-Richtung.

Nach der Projektion der Szene werden die Koordinaten noch normalisiert und auf die Auflösung des angezeigten Bildbereichs skaliert.

2.3.5 Texturen

Einen wichtigen Datentyp in OpenGL stellen Texturen da. Sie werden zum Sichern von Farbinformationen, Normalen, Schatten (Shadow Maps), Schattierungen (Light Maps) und auch für allgemeinen Informationen wie Zwischenergebnisse einer Rechnung genutzt. Texturen sichern Informationen in einem festen Format, welches vom Programmierer bestimmt werden kann. So gibt es unter anderem verschiedene Formate zum sichern von Farbinformationen und Formate zum speichern von Tiefenwerten. Eine Textur ist eine Sammlung von Bildpunkten, welche entsprechend Texel genannt werden.

Texturen werden über normalisierte Texturkoordinaten ausgelesen. Diese geben sowohl die S-, als auch die T-Koordinate im Bereich $[0, 1]$ an, beginnend in der unteren linken Ecke der Textur. So steht beispielsweise die Koordinate $(0.5, 0.5)$ für die Mitte der Textur. Texturen müssen in der Breite und Höhe in Pixeln jeweils eine Zweierpotenz sein. Bei der Nutzung von Texturen werden oft die Bezeichner U und V synonym für S und T genutzt, weshalb Texturkoordinaten auch abkürzend als UV-Koordinaten bezeichnet werden.

Neben den zweidimensionalen Standardtexturen (`GL_TEXTURE_2D`), existieren eine Reihe weiterer Texturtypen.

`GL_TEXTURE_RECT_ARB` Eine Spezialisierung der genannten `GL_TEXTURE_2D` Texturen stellen Textur-Rechtecke dar. Sie sind nicht an Größen in Form von Zweierpotenzen gebunden. Desweiteren werden sie nicht durch normalisierte Texturkoordinaten ausgelesen, sondern direkt in Pixelkoordinaten angesprochen. Die Textur überdeckt somit den Bereich $[0, Breite] \times [0, Höhe]$. Textur-Rechtecke sind sie gute Kandidaten um Zwischenschritte einzelner Zeichenvorgänge zu sichern, da der Darstellungsbereich in Form eines Fensters ebenso beliebige Breite und Höhe haben kann.

`GL_TEXTURE_CUBE` Textur-Würfel stellen eine einfache Möglichkeit dar, die gesamte Umgebung eines Punktes zu sichern. Ein Textur-Würfel besteht aus sechs gleichgroßen Texturen, welche Würfelförmig angeordnet sind. Zum Schreiben der Texturdaten können diese einzeln gebunden und beschrieben werden. Um aus einem Textur-Würfel zu lesen, wird ein Vektor angegeben. Dieser wird an den Ursprung des Würfels angelegt, der Schnittpunkt mit einer Seite berechnet und der Wert an der Position des Schnittpunktes ausgelesen. Somit eignen sich Textur-Würfel zur Sicherung von 360°-Ansichten. Sie lassen sich unter Anderem zur Darstellung von Reflektionen oder der Bestimmung des Schattenwurfes von Punktlichtquellen nutzen.

Da Textur-Würfel intern sechs separate Texturen verwalten treten meist ungewollte Nebeneffekte auf. Funktionen, die zu ihrer Berechnung die Umgebung eines Pixels miteinbeziehen erhalten am Rand einer Würfelseite keinen Zugriff auf die anliegende Seite. Dies Betrifft beispielsweise bilineare Filterung. So sind die Kanten bei niedrigen Auflösungen des Texturwürfels deutlich sichtbar.

`GL_TEXTURE_3D` Dieser Typ stellt eine dreidimensionale Textur dar, welche über normalisierten Texturkoordinaten S, T und R ausgelesen wird. 3D-Texturen lassen sich nutzen um Volumeninformationen zu sichern. Sie können beispielsweise Volumen-Nebel oder die Voxel-Darstellung eines 3D-Modells beinhalten. 3D-Texturen sind sehr speicherintensiv und werden daher nur selten in hohen Auflösungen genutzt.

2.3.6 OpenGL Shading Language

Prozedural berechnete Materialien werden in dieser Arbeit über die OpenGL Shading Language, im folgenden GLSL genannt, für die Grafikkarte beschrieben. GLSL-Programme werden in Textform an den Grafikkartentreiber übergeben und dort in ausführbaren Code kompiliert. Für die Arbeit werden GLSL-Programme der Version 1.1 geschrieben. Diese stammt aus 2004[5] und wird somit von einer Vielzahl an Grafikkarten unterstützt.

Ein GLSL-Programm besteht aus mindestens zwei Komponenten: dem Vertex-Shader und dem Fragment-Shader. Wird einer der beiden Shader nicht explizit angegeben, so ersetzt die Fixed-Function Pipeline das fehlende Programm. Vertex-Shader sind Code-Fragmente, welche auf der Grafikkarte für jeden Vertex ausgeführt werden. Ihre Hauptaufgabe besteht darin, die Koordinaten der Vertizes in das Sichtkoordinatensystem zu transformieren. Desweiteren können sie Werte berechnen, welche im zugehörigen Fragment-Shader weiter genutzt werden.

Fragment-Shader werden im Rasterizierungsprozess für jedes Fragment ausgeführt. In der Regel legen sie die Farbe und den Tiefenwert des zugehörigen Fragmentes fest, oder verwerfen dieses.

Optional kann auf neueren Grafikkarten ein Geometry-Shader ein GLSL-Programm ergänzen, welcher auf Basis vorhandener Vertizes neue Primitive generiert. Dieser ist nicht fester Bestandteil von OpenGL 2.0 und wird in dieser Arbeit nicht genutzt.

Vertex- und Fragment-Shader, zusammenfassend als Shader bezeichnet, ersetzen Teile der Fixed-Function Pipeline von OpenGL. Die Abbildung 2.6 zeigt die entsprechend erweiterte

[5] siehe Spezifikation http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.10.59.pdf (Stand: März 2010)

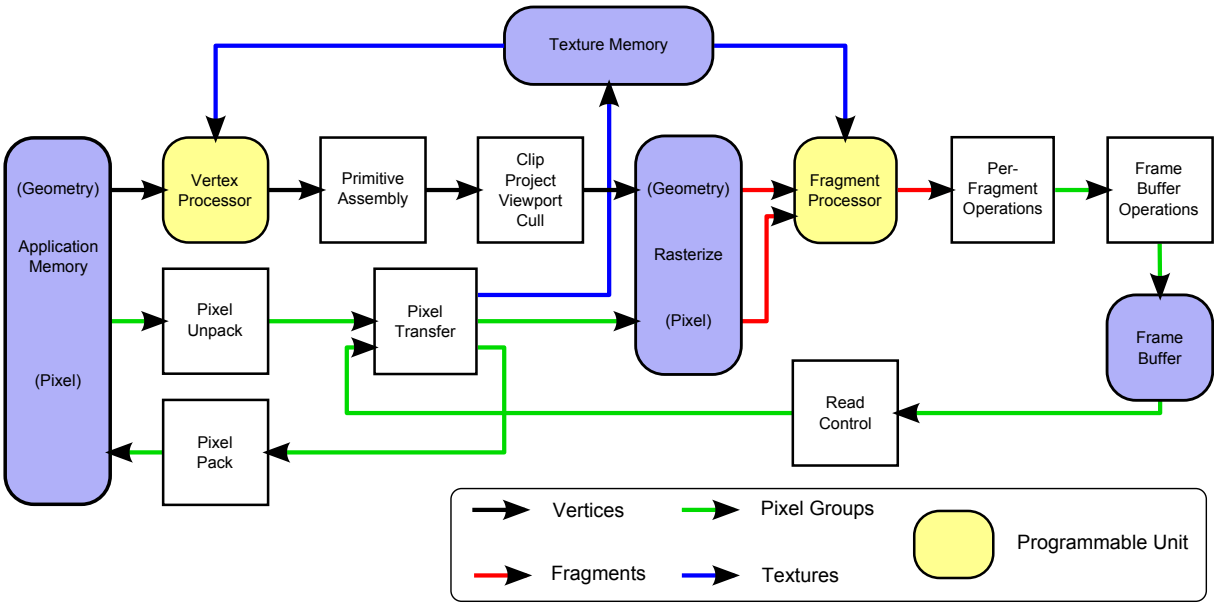


Abbildung 2.6: OpenGL Programmable Pipeline. Struktur nach [28, Kapitel 2.3 Abbildung 2.1]

Programmable Pipeline. Da Shader-Programme Funktionen der Fixed-Function Pipeline teilweise ersetzen, müssen diese bei Bedarf innerhalb der Shader nachgebildet werden. So muss, falls ein Fragment-Shader angegeben wurde dieser beispielsweise die Beleuchtung der Fragmente selbständig vornehmen.

Kommunikation zwischen CPU und Grafikkarte Um Informationen mit der Grafikkarte auszutauschen, gibt es in GLSL verschiedene Möglichkeiten. Zunächst können mit jedem Vertex Informationen wie Normale, Farbe oder Textur-Koordinaten sowie selbst definierte Attribute mitgegeben werden. Diese stehen Vertex-Shadern zur Verfügung und können durch diese verarbeitet und an Fragment-Programme übergeben werden.

Für kleinere Datenmengen, die nicht Vertex-abhängig sind, lassen sich Werte an ein aktives GLSL-Programm schicken. Hierfür definiert GLSL den Variablenmodifikator `uniform`. Uniform-Variablen können sowohl für Vertex- als auch für Fragment-Shader definiert werden. Um den Wert einer Uniform-Variable zu setzen muss das entsprechende Programm aktiv sein. Zuerst muss die Position der Variable erfragt werden. Danach kann über einen zweiten Befehl der Wert gesetzt werden.

Für große Datenmengen bietet sich die Nutzung einer Textur an, in der die Daten gesichert werden. Texturen können auch von Vertex- und Fragment-Shadern als Datenquelle genutzt werden, allerdings ist der Zugriff auf Texturen deutlich langsamer als der Zugriff auf Attribute oder Uniform-Variablen.

Um zwischen verschiedenen Zeichenschritten Informationen von einem GLSL-Programm an ein anderes zu übergeben, werden die benötigten Informationen vom Fragment-Shader berechnet und in eine Textur geschrieben. Diese kann zu einem späteren Zeitpunkt wieder als Eingabe weitere Berechnungen dienen. Neben Texturen werden von GLSL-Shader hauptsächlich verschiedene Puffer zur Ausgabe genutzt. Allerdings können diese nicht von Shader-Programmen gelesen werden.

2.3.7 Vollbild-Shader

Bei GLSL-Programmen ist zu beachten, dass sie zum Zeichnen von geometrischen Objekten genutzt werden und nur für die verarbeiteten Vertizes aktiv sind. Soll der Inhalt einer Textur

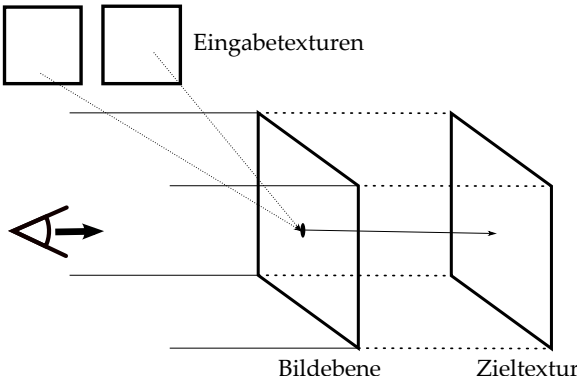


Abbildung 2.7: Vollbild Shader

verändert werden, so bietet sich das folgende Verfahren an:

Wie in Abbildung 2.7 schematisch dargestellt werden zunächst die Eingabetexturen und der Ausgabepuffer gebunden. Der Sichtbereich wird an die Größe des Ausgabepuffers angepasst und eine orthogonale Projektion gewählt. Sollen die Texel einer Eingabetextur modifiziert ausgegeben werden ist es wichtig, dass deren Texel 1:1 auf die Pixeln der Bildebene projiziert werden. Dann wird ein GLSL-Programm gebunden, welches die Informationen der Eingabetexturen verarbeitet. Es wird ein Rechteck gezeichnet, welches den Sichtbereich ausfüllt. Pro Fragment des Rechtecks legt der Fragment-Shader nun genau einen Pixel im Ausgabepuffer fest. Dabei kann aus den Quelltexturen an beliebiger Position gelesen werden.

2.3.8 Frame-Buffer-Objects in OpenGL

Um mit OpenGL Bilder zu zeichnen wird ein Bereich auf dem Bildschirm zur Verwendung mit OpenGL bestimmt. Hierbei werden Farbpuffer und bei Bedarf Tiefen- und Stencilpuffer gebunden. Soll nun aber in eine Textur gezeichnet werden, oder ein Bild welches größer als der auf dem Bildschirm angeforderte Bereich ist erzeugt werden, so muss dies unabhängig vom angeforderten Bildbereich geschehen.

Zum indirekten Zeichnen in Texturen und Zeichenpuffer existiert in OpenGL mit der Erweiterung `GL_EXT_framebuffer_object` ein seit der Version 2.0 standardisiertes Verfahren. Um in eine Textur zu rendern, wird ein Frame-Buffer-Object (kurz FBO) erzeugt. An dieses werden ein oder mehrere Zeichenpuffer oder Texturen gebunden. Ein FBO ist ein Objekt zum Sichern und Verwalten verschiedener Puffer.

Soll der z-Test genutzt werden, muss ein Tiefenpuffer an das FBO gebunden werden. Dieses muss eine Textur oder ein Zeichenpuffer sein. Die Mitnutzung des Bildschirm-Tiefenpuffers ist nicht möglich. Gleiches gilt für den Stencil-Test: auch hier muss ein eigener Stencilpuffer gebunden werden. Sollen Farbwerte gesichert werden, muss desweiteren mindestens ein Farbpuffer gebunden sein. Es besteht die Möglichkeit mehr als einen Farbpuffer zu binden, um in mehrere Texturen oder Zeichenpuffer gleichzeitig zu zeichnen.

Das Binden mehrerer Farbpuffer und -texturen unterliegt der Beschränkung, dass für jedes gebundene Objekt die Anzahl der Bits pro Pixel gleich sein muss. Desweiteren müssen die Dimensionen aller an den FBO gebundenen Zeichenziele übereinstimmen. Ein Problem ist, dass jede Kombination von Farb-, Tiefen- und Stencilpuffern unbegründet durch den Grafikkartentreiber abgelehnt werden kann. Somit sollten FBOs nur mit gängigen Pixelformaten genutzt werden.

Mit OpenGL 2.0 und der Erweiterung `GL_ARB_draw_buffers` ist es desweiteren möglich, im Fragment-Shader eines GLSL-Programms verschiedene Farbwerte in die unterschiedlichen Farb-Puffer eines FBOs zu zeichnen.

2.3.9 3D-Darstellung in GroIMP

GroIMP besitzt bereits, wie im Abschnitt 1.1 erläutert eine OpenGL-Darstellung, welche innerhalb dieser Arbeit vereinfacht mit OGL1 bezeichnet wird. Diese nutzt die feste Grafikpipeline (siehe Abschnitt 2.3.2) von OpenGL um Objekte darzustellen. Hierbei werden Lichtquellen und Objekte in OpenGL-Lichter und Polygone übersetzt. Für die in GroIMP definierbaren Shader werden Texturen erzeugt, welche eine Vorschau auf die tatsächlich prozedural berechneten Materialien bieten. Problematisch sind hierbei Volume-Shader, welche dreidimensionale Eingabedaten haben und somit nicht in einer zweidimensionale Textur gesichert werden können. Somit zeigt OGL1 eine korrekte Vorschau nur für eine geringe Teilmenge der GroIMP-Shader an. Die bisherige OpenGL Darstellung definiert für jedes zeichenbare Objekt in GroIMP eine entsprechende Methode, um dieses in OpenGL darzustellen. Diese werden in dieser Arbeit mitgenutzt.

Neben OGL1 existieren verschiedene Raytracer, die ein hochwertiges Bild der Szene erzeugen können. Insbesondere ist der interne Raytracer „Twilight“ zu nennen, dessen Ausgabe, insbesondere Twilight (STDR), als Referenz für diese Arbeit gilt.

2.4 Techniken

Hier werden einige gebräuchliche Techniken der Echtzeitdarstellung von Szenen auf der Grafikkarte vorgestellt. Dieser Abschnitt soll nur einen Überblick geben und ist keine erschöpfende Erläuterung. Die Techniken werden in angepasster Form helfen eine möglichst detaillierte Vorschau der Szene zu erzeugen.

2.4.1 Textur Ping-Pong

In vielen Renderverfahren ist es nötig, die Ausgabe eines vorherigen Zeichenschrittes in folgenden Zeichenschritten zu verändern. Dies geschieht meist in Form von Texturen. Wird beim Verändern des Texturinhaltes der zuvor enthaltene Wert benötigt, so muss diese Textur sowohl als Lese-, als auch als Schreibpuffer gebunden werden. Zu diesem Zustand sagt die Beschreibung der Frame-Buffer-Objects:

> „(44) What should happen if a texture that is currently bound to the context is also used as an image attached to the currently bound framebuffer? In other words, what happens if a texture is used as both a source for texturing and a destination for rendering?
>
> RESOLUTION: resolved, (b2) - results are undefined because the framepuffer is not ”framepuffer complete”.“ [6]

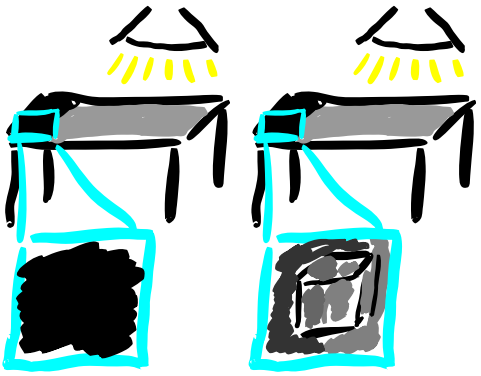
Die Lösung bietet Textur Ping-Ponging. Diese Technik ist weit verbreitet und zum Beispiel im „GPGPU::Basic Math Tutorial“[7] von Dominik Göddeke nachzulesen. Es werden zwei Texturen gleichzeitig genutzt. Jeweils eine als Eingabe und die andere als Ausgabe. Ist ein Zeichenschritt vollständig abgeschlossen, wechseln beide Texturen ihre Rollen. Wichtig ist hierbei, das jeder Zeichenschritt die gesamte Eingabetextur wieder ausschreiben muss, da sonst in folgenden Schritt Informationen fehlen.

2.4.2 High Dynamic Range Rendering

In der Bilderzeugung unterscheidet man zwischen HDR (high dynamic range, hoher Dynamikumfang) und LDR (low dynamic range, niedriger Dynamikumfang). Der Dynamikumfang, in der

[6] http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt (Stand: März 2010, Revision #120, zuletzt geändert am 22.04.2008)

[7] http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html (Stand: März 2010)



(a) LDR (b) HDR

Abbildung 2.8: Vergleich LDR mit HDR. Der vergrößerte Ausschnitt wurde im HDR-Bild seperat skaliert. Das Bild enthält bereits den Würfel. Im LDR-Bild ist diese Information nicht vorhanden, weshalb zur Darstellung des Würfels die Szene mit stärkerer Beleuchtung neu gezeichnet werden müsste.

Optik auch Kontrastumfang genannt, beschreibt das Verhältnis von der größten Leuchtdichte zu der Kleinsten. High Dynamic Range Rendering (HDR Rendering) bezeichnet ein Verfahren, welches Bildinformationen im gesamten Kontrastbereich sichert. Der Wertebereich beim Low Dynamic Range Rendering (LDR Rendering) ist für jeden Farbkanal auf den statischen Bereich $[0,1]$ eingeschränkt. Weitere Informationen bietet beispielsweise das Buch von Rainhard et al. [31]

HDR Rendering eignet sich insbesondere zum Zeichnen von Szenen, deren Kontrastbereich nicht vorab bekannt ist. Um ein HDR-Bild auf dem Bildschirm darzustellen, muss dieses jedoch auf das Farbspektrum des Bildschirmes reduziert werden. Verfahren, die dies bewirken werden Tonemapping-Verfahren genannt.

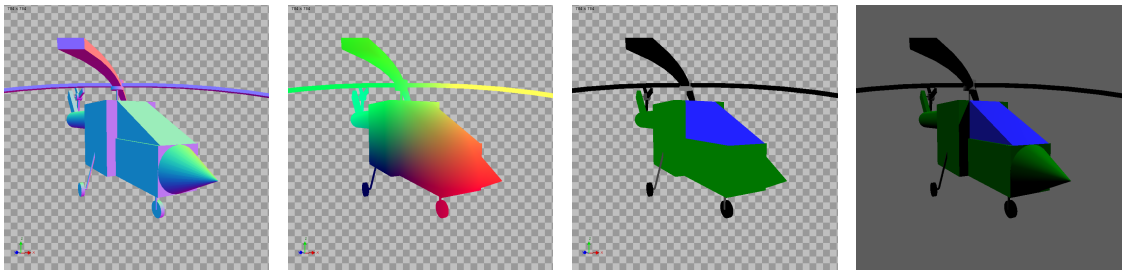
Ein simpler Tonemapping-Operator ist die lineare Skalierung. Hierbei wird der dynamische Wertebereich des HDR-Bilder $W = [w_{min}, w_{max}]$ ermittelt und durch die folgende lineare Abbildung skaliert: $f(x \in W) = \frac{x - w_{min}}{w_{max} - w_{min}}$. Für einen Farbraum kann hierbei beispielsweise die wahrgenommene Helligkeit (Luminanz) oder das Maximum beziehungsweise Minimum über die Farbwerte des Bildes, wie in GroIMP genutzt, verwendet werden. Es gibt eine Vielzahl weiterer Tonemapping-Operatoren, wie der kontrasterhaltende Operator von Larson et al. [20] oder der von Reinhard und Devlin entwickelte, schnell auszuwertende Operator [30], die meist in der Darstellung von HDR-Fotografien Verwendung finden[8].

Abbildung 2.8 zeigt eine Beispielszene sowohl als LDR-, als auch als HDR-Version. Die in der Vergrößerung des HDR-Bildes sichtbaren Details sind Teil des Bildes, werden aber erst bei passendem Tonemapping erkennbar.

2.4.3 Deferred Shading

Deferred Shading, zu deutsch „verzögertes Beleuchten“, bezeichnet eine Reihe von Verfahren, welche auf der Idee basieren in einem vorbereitenden Zeichenschritt alle später benötigten Parameter zu bestimmen und zu sichern (siehe Abbildung 2.9). Somit bleibt folgenden Zeichenschritten erspart, die benötigten Eigenschaften mehrfach auszuwerten. Dies bietet insbesondere bei viel genutzten Funktionswerten zeitaufwendiger Berechnungen einen Geschwindigkeitsvorteil. Die zugrunde liegende Technik wurde von Deering et al. [7] beschrieben, auch wenn der Name „Deferred Shading“ erst später geprägt wurde. Sie findet oft in der Darstellung von Szenen mit einer hohen Anzahl an Lichtquellen Anwendung, da sie erlaubt, die Berechnung von Materialeigenschaften

[8] Einen Vergleich verschiedener Tonemapping-Operatoren wurde beispielsweise von Martin Čadík vorgenommen. http://www.cgg.cvut.cz/members/cadikm/tmo/ (Stand: März 2010)



(a) Normalen-Textur (b) Positions-Textur (c) Diffuse-Textur (d) beleuchtete Szene

Abbildung 2.9: „Deferred Shading“ zur Beleuchtung einer Szene. Die Szene „Hubschrauber“ stammt aus der Beispielbibliothek von GroIMP.

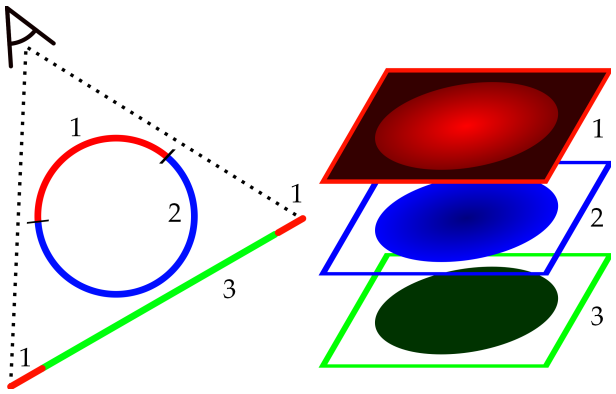


Abbildung 2.10: Zweidimensionale Beispielszene zu Depth Peeling. Die einzelnen z-Ebenen sind farbkodiert in Rot, Blau und Grün eingefärbt und werden in dieser Reihenfolge vom genannten Algorithmus extrahiert.

von der Berechnung der Beleuchtung zu trennen. Die Beleuchtung kann dann pro Lichtquelle in einem kleinen lokal begrenzten Bereich um die Lichtquelle ausgewertet werden. Dieses bietet insbesondere in der Grafikkartenprogrammierung einen weiteren Vorteil: Mittels Deferred Shading können Programme, die Materialeigenschaften bestimmen, getrennt von denen die eine Oberfläche beleuchten, geschrieben und kompiliert werden.

2.4.4 Depth Peeling

Depth Peeling, zu deutsch etwa „Tiefen-Schälung“, ist eine von Everitt [9] vorgestellte Technik zur Zerlegung des Bildes in einzelne Tiefenebene. Die Zerlegung wird Anhand der Abbildung 2.10 verdeutlicht. Die Idee besteht darin, die Entfernung vom Beobachter zu jedem gezeichneten berechneten Fragment zu sichern. Diese werden dann als im folgenden Zeichenschritt als zusätzlichen z-Test genutzt, um bereits extrahierte Fragmente nicht weiter zu betrachten. Eine einfache Realisierung zeigt der Pseudocode in Algorithmus 2.1.

Diese Zerlegung kann zur Darstellung verschiedener Effekte genutzt werden: um Tiefenunschärfe darzustellen, Schatten halbtransparenter Objekte zu zeichnen oder, wie in dieser Arbeit genutzt, szenen- und geometrieunabhängig transparente Objekte zu rendern. Depth Peeling ist ein Multi-Pass-Verfahren, das bedeutet es, wird mehrfach die selbe Szene gezeichnet, um das gewünschte Ergebnis zu erzielen.

2.4.5 Under Blending

Um zwei Farbwerte miteinander zu kombinieren lassen sich die von OpenGL definierten Blending-Funktionen nutzen. Hierbei soll einem Bild im Speicher der Grafikkarte, dem Zielbild, Werte eines

Algorithmus 2.1 Pseudocode zu Depth Peeling

```
1: Setze Tiefentest auf LESS
2: Initialisiere Tiefenpuffer a und b
3: loop
4:    Binde a als aktuellen Tiefenpuffer und b als Vergleichstextur
5:    Rendere Szene
6:    Lade Vergleichswert z_old aus Textur b
7:    z_cur ← z-Wert des aktuellen Fragmentes
8:    if z_old weiter entfernt als z_cur then
9:       Verwerfe aktuelles Fragment
10:   else
11:      Zeichne Fragment
12:   end if
13:   Vertausche die Rollen von a und b
14: end loop
```

anderen Bildes, dem Quellbild, hinzugefügt werden. Beim Zeichnen transparenter Fragmente wird oft Alpha Blending genutzt. Hierbei wird angenommen, dass das Zielbild unter dem Quellbild liegt. Ist c_{dst} der Farbwert eines Punktes im Zielbild, c_{src} der Farbwert des selben Punktes im Quellbild, sowie $a_{src} \in [0,1]$ die Durchlässigkeit der Quelle im Punkt. Dann lässt sich der kombinierte Farbwert c'_{dst}, dem Ergebnis des Blendings von Ziel- und Quellbild folgendermaßen berechnen:

$$c'_{dst} = c_{src} \cdot a_{src} + c_{dst} \cdot (1 - a_{src})$$

Wie von Bavoil und Myers [1, S. 6f] beschrieben lässt sich diese Funktion für den Fall, dass das Quellbild unter dem Zielbild liegt umstellen, wenn zusätzlich die Durchlässigkeit des Zielbildes a_{dst} bekannt ist:

$$\begin{aligned} c'_{dst} &= c_{src} \cdot a_{src} \cdot a_{dst} + c_{dst} \\ a'_{dst} &= (1 - a_{src}) \cdot a_{dst} \end{aligned}$$

a'_{dst} beschreibt die Durchlässigkeit des kombinierten Bildes. Diese kann für weitere Bildkompositionen gesichert werden. Die Formeln lassen sich auch getrennt auf die einzelnen Farbkanäle eines Bildpunktes anwenden. So lässt sich für jeden Farbkanal die Durchlässigkeit seperat festlegen.

2.4.6 Darstellung von Schatten

Zur Darstellung einer dreidimensionalen Szene die Lichtquellen beinhaltet, helfen Schatten die räumliche Tiefe besser zu erkennen. In der Berechnung und Darstellung von Schatten wird grundsätzlich drei Kategorien von Algorithmen unterschieden. Solche, die auf der Idee der Shadow Volumes (deutsch: Schattenkörper) basieren, solche die Shadow Maps (Schattentexturen) nutzen und solche, die über Raytracing Schatten generieren.

Für Shadow Volumes, zuerst von Franklin Crow 1977 in [6] vorgeschlagen, wird für jedes schattenwerfende Objekte in der Szene ein Schattenkörper erzeugt und nachträglich als Schatten in die Szene eingefügt. Der Schattenkörper entspricht der Silhouette des Objekts aus Sicht der Lichtquelle. Diese wird von der Lichtquelle weg extrudiert. Eine schnelle, robuste Implementierung des Verfahrens beschreiben beispielsweise McGuire et al. [23]. Für Formen, wie Würfel oder Kugeln, ist die Generierung der Schattenkörper einfach und effizient zu berechnen. Für

beliebige, auf Polygonen basierte Modelle besteht die Möglichkeit über einen Geometrie-Shader den passenden Schattenkörper automatisch zu generieren, allerdings wird die Kenntnis der Nachbarschaftsbeziehung der Polygone vorausgesetzt. Diese Methode unter Anderem von Stich et al. [34] beschrieben.

Shadow Maps, zum ersten mal publiziert 1978 von Lance Williams [35], beschreiben Verfahren, in denen für jede Lichtquelle ein "Schattenbild" der Szene aus Sicht der Lichtquelle berechnet. Dieses enthält in Bildpunkten jeweils den Abstand zur Lichtquelle. Um über eine Shadow Map zu bestimmen, ob ein Punkt im Schatten liegt, werden die Koordinaten des betrachteten Punktes in das Koordinatensystem der Lichtquelle umgerechnet. Nun wird bestimmt, auf welchem Texel der Shadow Map der Punkt liegt. Die gesicherte Entfernung aus der Shadow Map wird dann mit der Entfernung des Punktes zur Lichtquelle verglichen. Ist der betrachtete Punkt weiter als der Referenzwert von der Lichtquelle entfernt, so befindet sich ein Hindernis zwischen Lichtquelle und Punkt. Dieser liegt entsprechend im Schatten. Nach der Idee der ursprüngliche Shadow Map Technik existieren viele Umsetzungen, die deutlich hochauflösendere Schatten erzeugen. So besteht die Möglichkeit in Abhängigkeit zur Kamera mehrere Shadow Maps zu verwenden. Für Bereiche nahe der Kamera werden große Schattentexturen erzeugt, während für weiter entfernte Bereiche kleine Texturen genügen. Die von Dimitrov entwickelten Cascaded Shadow Maps [8] bedienen sich dieser Technik. Eine andere Möglichkeit bieten Projektionsmatrizen, die dem Bereich nahe der Kamera einen größeren Ausschnitt der Schattentextur zuweisen. Perspective Shadow Maps von Stamminger und Drettakis [33] arbeiten nach dieser Methode. Desweiteren können Filtermethoden, wie „Percentage Closer Filtering“ [29] (kurz PCF), Aliasing reduziert werden.

Raytracing, zu deutsch Strahlenverfolgung, wird hauptsächlich in der nicht-interaktiven Darstellung genutzt. Hierbei wird direkt getestet, ob zwischen einem betrachteten Punkt und den Lichtquellen der Szene Hindernisse vorhanden sind. Die Qualität liegt zwar deutlich über der Qualität der anderen beiden Verfahren, allerdings benötigt sie auch ein vielfaches mehr Rechenzeit und eignet sich demnach nicht für eine interaktive Vorschau.

3 Entwicklung der eigenen Lösung

Das Ziel der Arbeit ist es die in GroIMP vorkommenden prozeduralen Materialien detailgetreu über GLSL-Code abzubilden. Desweiteren soll die Darstellung der Materialien eine gute Vorschau auf das von Twilight produzierte Ergebnis bieten.

Dies geschieht in drei Schritten. Zunächst müssen die GroIMP-Materialien, die in der Szene genutzt werden, in GLSL-Programme übersetzt werden. Die Materialerzeugung wird in Abschnitt 3.1 erläutert. Als nächstes müssen, entsprechend der in GroIMP vorkommenden geometrischen Formen, Körper in OpenGL gezeichnet werden, welche Träger der Materialien sind. Hierfür baut die Arbeit auf die in GroIMP schon verfügbare OpenGL-Darstellung auf und nutzt deren Zeichenoperationen, weshalb auf diesen Schritt nicht näher eingegangen wird. Zuletzt muss die aus Form und Material gewonnene Information in ein Bild übersetzt werden. Dies geschieht in einer Reihe von Zeichenschritten, die in Abschnitt 3.2 erarbeitet werden.

3.1 Generierung der GLSL-Material-Programme

3.1.1 Vorüberlegung

Zunächst werden die von GroIMP erzeugbaren Materialien als mehrdimensionale Funktionen betrachtet. Ein Material f bildet dann eine Menge von Eingabeparametern A auf eine Menge von Funktionswerten B ab.

Die Abbildung f wird genutzt, um in einem Punkt auf der Oberfläche eines Objektes die Materialeigenschaften (B) auszuwerten. Die Menge der Eingabeparameter (A) ergibt sich aus dem Punkt. Sie ist eine beliebige Kombination der folgenden Parameter

- Die Position des Punktes im lokalen Koordinatensystem des Objektes
- Die Position des Punktes im globalen Weltkoordinatensystem
- Die Texturkoordinate im Punkt
- Die Normale der Oberfläche im Punkt im globalen Weltkoordinatensystem
- Eine Tangente und die zugehörige Binormale im Punkt

Materialien lassen nach der Dimension der Argumentmenge in vier Kategorien einteilen:

1. $\dim(A) = 0$
2. $\dim(A) \leq 2$ (Texturkoordinaten)
3. $\dim(A) = 3$ (Normale oder Position)
4. $\dim(A) > 3$ (Kombinationen aus lokaler Position, globaler Position, Normale, Texturkoordinaten und Anderen)

Die erste Kategorie erzeugt konstante Funktionswerte. Diese ließen sich im voraus berechnen und direkt als Attribute der zu zeichnenden Vertizes angeben. Demnach müssen solche Materialien nicht in GLSL-Code übersetzt werden. Dies trifft beispielsweise auf die mit RGBAShader erzeugbaren Materialien zu.

Ist A ein- oder zweidimensional, so ließen sich die Werte in Texturen vorberechnen. Dies wird in der bereits bestehenden Vorschau genutzt um eine näherungsweise korrekte Darstellung der Funktionswerte von f zu erhalten. Dies ist nur möglich, wenn die Eingabeparameter aus einem endlichen Wertebereich stammen. Um Aliasing gering zu halten muss die Größe der Textur in Abhängigkeit zum Wertebereich der Eingabe angepasst werden. Eine Kugel, die einen großen Teil der Szene bedeckt benötigt höher auflösende Texturen als ein Objekt das nur in wenigen Pixel des Bildes zu sehen ist. Ein Material kann genutzt werden, um die Oberfläche verschiedener Objekte zu beschreiben. Um nicht für jedes Objekt einzeln eine passende Textur zu erzeugen, könnte eine maximale, benötigte Auflösung über alle Objekte bestimmt werden. Für eine große Zahl verschiedener Materialien benötigen die erzeugten Texturen einen Großteil des Speichers der Grafikkarten. So belegen bereits 256 Texturen mit einer Auflösung von 512×512 Pixeln bei 4 Byte pro Pixel 256 MB Grafikkartenspeicher.

Ist A dreidimensional, so kann die Überlegung zu zweidimensionalem A auf Volumentexturen (siehe `GL_TEXTURE_3D` in Abschnitt 2.3.5) erweitert werden. Hierbei muss beachtet werden, dass bei Volumentexturen nur sehr geringe Auflösungen praktikabel sind. Bereits bei einer Auflösung von $128 \times 128 \times 128$ und 4 Byte pro Pixel belegt eine solche Textur 8 MB Grafikkartenspeicher.

Für $\dim(A) > 3$ bieten Grafikkarten bisher keine passenden Texturformate an. Die theoretische Texturgröße würde desweiteren den Speicher gängiger Grafikkarten (zum aktuellen Zeitpunkt 1024 MB) weit übersteigen.

Materialien durch Texturen zu approximieren ist demnach nur in Ausnahmefällen möglich. Die Shadersprache GLSL beherrscht alle Befehle, die zur Berechnung der GroIMP-Materialien nötig sind[9]. Die Vorschau soll eine möglichst exakte Wiedergabe der Materialien beinhalten. Es liegt also nahe die Darstellung von Materialien der von GroIMP nachzuempfinden. Hierfür werden die Materialien in GLSL reimplementiert.

3.1.2 Material

Zunächst werden GroIMP-Shader auf der Material-Ebene betrachtet. Hier unterscheidet GroIMP zwischen RGBA, Lambert- und Phong-Materialien. RGBA sowie Lambert-Materialien stellen eine Untermenge der mit Phong erzeugbaren Shader dar. Demnach genügt es, nur die Klasse Phong zu betrachten. Es existieren weitere Materialien wie SunSkyLight, Side- und AlgorithmSwitch. Diese werden in der Implementierung gesondert behandelt.

Phong-Materialien besitzen folgende, konfigurierbare Parameter:

- Channel Input (Primär eine Transformation der Normale. Beispielsweise für Bump-Mapping genutzt)
- Diffuse colour (diffuse Reflektivität)
- Specular colour (spekulare Reflektivität)
- Shininess (spekulare Exponent in der Phong-Gleichung)
- Transparency (Lichtdurchlässigkeit)
- Diffuse Transparency
- Transparency Shininess
- Ambiente colour (Grundfarbe in der das Material ohne Beleuchtung erscheint)
- Emissive colour (Ausgestrahltes Licht)
- Interior (Brechzahl)

[9] GLSL kann WHILE-Programme simulieren, weshalb GLSL Turing-vollständig ist.

Zunächst ist zu entscheiden, welche dieser Information in den GLSL-Shader kompiliert werden und welche Daten zur Laufzeit dem Shader übergeben werden. Sollen Parameter über Uniform-Variablen übergeben werden, so muss dies möglicherweise mehrfach geschehen. Zudem müssen die Werte der entsprechenden Parameter in GroIMP auf Änderungen überprüft werden. Dies muss entweder über den GroIMP-Material-Baum oder über einen Cache passieren.

Statische Shader, in denen genutzte Parameter fest einkompiliert werden, sind demnach etwas schneller. Insbesondere kann der GLSL-Compiler besser optimieren, wenn die Werte der genutzten Parameter bekannt sind. Eine nennenswerte Ausnahme bilden die RGBA-Shader, die in GroIMP oft Verwendung finden, wenn eine grobe, farbliche Unterscheidung der Objekte genügt. Um nicht für jeden Farbton ein eigenes GLSL-Programme zu erzeugen sollten Uniform-Variablen genutzt werden, um die GLSL-Übersetzung des RGBA-Materials nur einmal kompilieren zu müssen.

Die genannten Parameter (mit Ausnahme von Interior) sind vom Nutzer wählbare Funktionen, die in GroIMP ChannelMaps genannt werden. Um einen statischen Shader für das Phong-Material zu erzeugen müssen also die verwendeten ChannelMaps in statische GLSL-Codefragmente übersetzt werden.

3.1.3 ChannelMap

Aus Sicht des eines Materials ist eine ChannelMap ist eine Funktion, die die schon bekannte Menge der Eingabeparameter (A) unter Angabe eines Kanals (Channel) nach $\mathbb{R}$ abbildet. Der Kanal ist als eine Auswahl verschiedener Abbildungen von A zu verstehen. Kanäle in GroIMP sind:

- Globale Position: PX, PY, PZ
- Globale Normale: NX, NY, NZ
- Lokale Position: X,Y,Z
- Texturkoordinate: U,V,W
- Partielle Ableitung der globalen Position in U-Richtung: DPXDU, DPYDU, DPZDU
- Partielle Ableitung der globalen Position in V-Richtung: DPXDV, DPYDV, DPZDV
- Farbe: R, G, B, A
- Brechzahl: IOR

Neben den Parametern aus A kann eine ChannelMap intern weitere ChannelMaps zur Berechnung des Wertes eines Kanals nutzen.

Um die Eingabeparameter weiter zu formalisieren, lassen sich diese auch als eine ChannelMap interpretieren. Im folgenden bezeichnet DefaultChannelMap eine ChannelMap, die zu einem erfragten Kanal den entsprechenden Eingabeparameter des Materials zurück gibt. Wird beispielsweise der Kanal X angefragt, so ist der Rückgabewert die X-Koordinate des betrachteten Punktes im lokalen Koordinatensystem des Objektes.

Eine ChannelMap bildet demnach eine Eingabe, in Form einer Menge von ChannelMaps, sowie einen Kanal auf eine realwertige Zahl ab.

Da ChannelMaps austauschbare Komponenten im Materialbaum (siehe Abschnitt 2.1.1) sind liegt es nahe, die Generierung einzelner Programm-Fragmente auf den ChannelMaps nachempfundene Klassen aufzuteilen. Hierzu muss zunächst für jede GroIMP-ChannelMap eine Übersetzerklasse geschrieben werden. Diese sollte für eine gegebene GroIMP-ChannelMap, der DefaultChannelMap und einem Kanal ein statisches Codefragment generieren.

Das Codefragment selbst muss entsprechend ein Term sein, der den benötigten Wert berechnet. Die Baumstruktur, in der ChannelMaps implizit miteinander verknüpft sind legt es nahe, Codefragmente rekursiv zu erzeugen. Wird der GLSL-Code zu einer ChannelMap benötigt, lässt diese

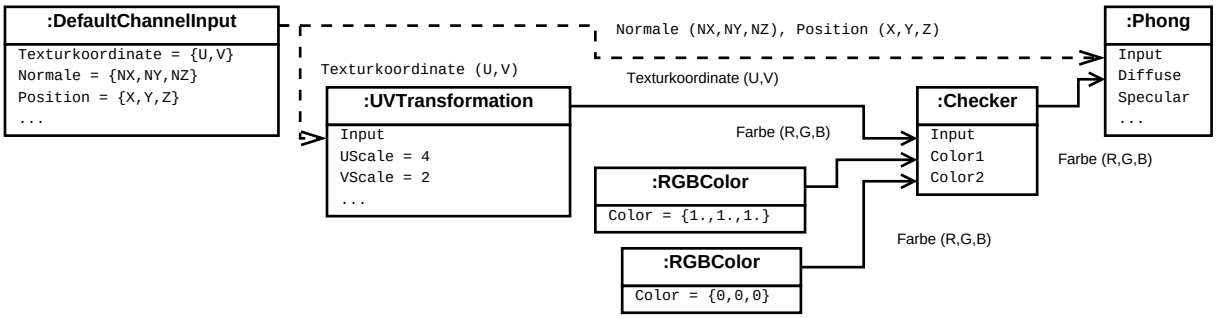


Abbildung 3.1: Materialbaum zum Beispiel Checker2D. Die einzelnen Rechtecke sind Instanzen der ChannelMaps bzw des Materials. Die Kanten geben Kanäle an, für die Codefragmente erzeugt werden.

zunächst für alle ihr untergeordneten ChannelMaps Code erzeugen. Diese Fragmente werden dann in einem Term kombiniert und beschreiben eine Übersetzung der ursprünglichen ChannelMap.

Ist der Material-Baum vollständig abgelaufen, so liegt GLSL-Code für alle Kanäle der mit dem Material verknüpften ChannelMaps. Diese können genutzt werden um ein vollständiges GLSL-Programm zu erzeugen. Hierzu werden neben den ChannelMaps auch die GroIMP-Materialien in GLSL-Code übersetzt. Sie stellen die Wurzel des Baumes da und fassen somit alle erzeugten Codeblöcke zusammen.

3.1.4 Datenkanäle (Channel)

Oft ist geht die Berechnung eines Kanals mit der anderer einher. So wird beispielsweise bei der Anfrage nach dem U-Kanal einer UV-Transformation implizit bereits U und V bestimmt. Wird dann nach dem V-Kanal gefragt, so würde ein zweites Codefragment erzeugt werden, dass eine bereits durchgeführte Rechnung wiederholt.

Um beim Erzeugen von Codeblöcken nicht jeden angefragten Kanal einzeln generieren zu müssen werden folgende Annahmen über die Kanäle gemacht:

- Wird eine Texturkoordinate erfragt, werden sowohl U-, als auch V-Koordinate benötigt.
- Wird eine Position erfragt, werden alle drei Koordinaten benötigt.
- Wird die Ableitung der Position in U- oder V-Richtung erfragt, so werden alle drei Koordinaten benötigt.
- Wird eine Normale erfragt, werden alle drei Elemente des Normalenvektors benötigt.
- Wird ein Farbwert erfragt, wird der von der ChannelMap verarbeitete Typ benötigt (ob Farb- oder Grauwert wird von der ChannelMap selbst bestimmt).

Somit genügt die Anfrage des Kanals, welcher die aktuelle x-Koordinate enthält, um als Rückgabe einen Term zu erhalten, der sowohl X,Y als auch den Z-Kanal der ChannelMap umfasst. Somit können Codefragmente nicht nur Realwertige Ergebnisse berechnen sondern auch Vektoren mit zwei oder mehr Einträgen.

Definiert ein ChannelMap-Objekt keine eigenen Informationen zu einem Kanal, wird die Anfrage an das DefaultChannelInput-Objekt delegiert. Desweiteren ist das Standardverhalten der ChannelMaps die das Feld „Channel input“ besitzen, wenn sie den Kanal nicht selbst implementieren, den Wert des „Channel input“-Objekts zurückzugeben.

3.1.5 Beispiel: Checker2D

Die beschriebene Idee wird nun am Beispiel des vorherigen Kapitels (Abbildung 2.1) veranschaulicht. Der zugehörige Materialbaum ist in Abbildung 3.1 zu sehen.

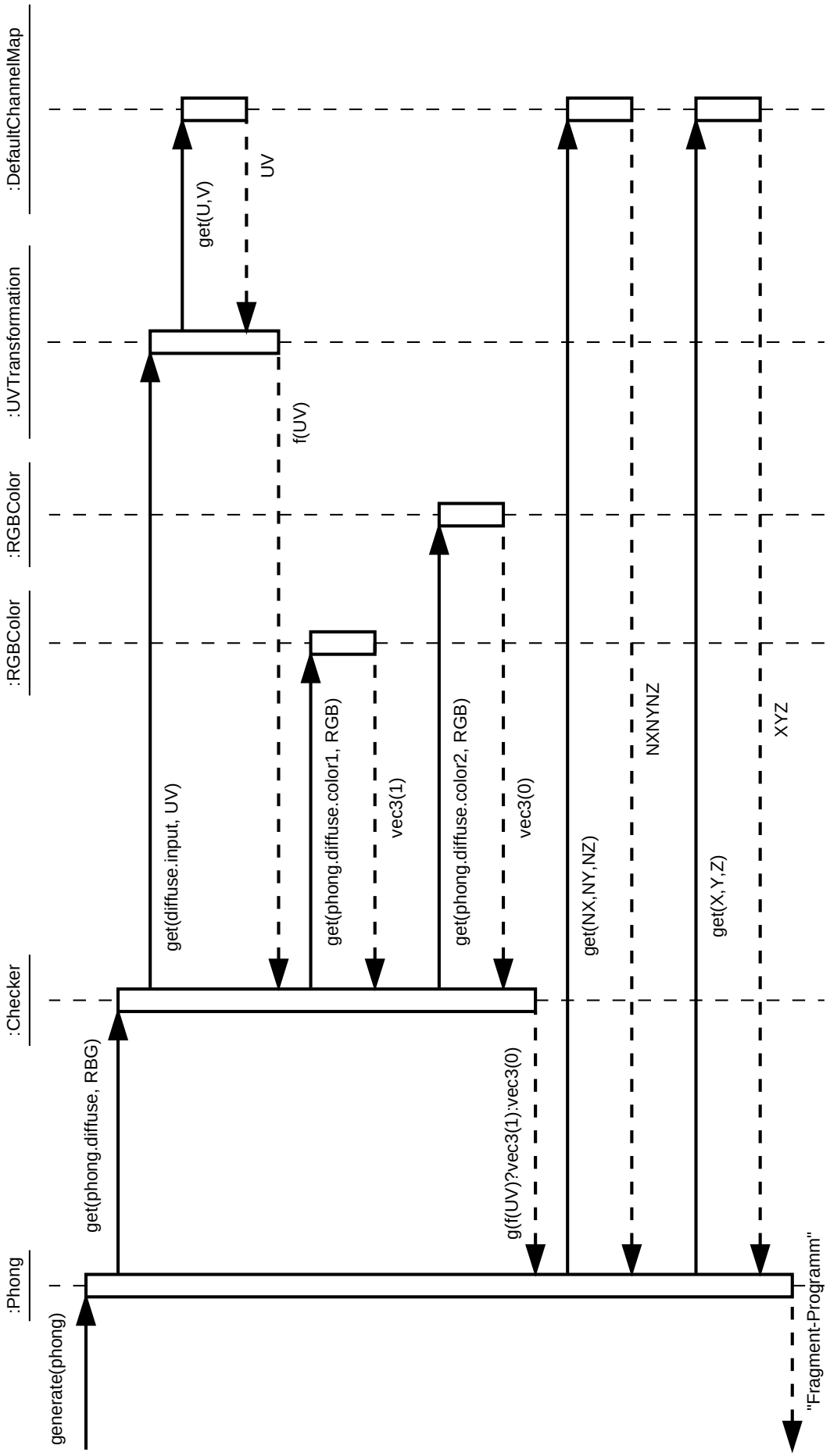


Abbildung 3.2: Sequenzdiagramm der Codeerzeugung des Beispiels „Checker 2D“.

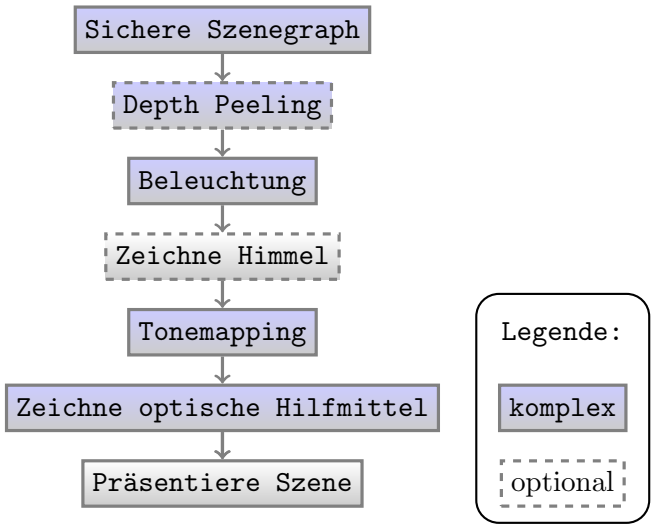


Abbildung 3.3: Übersicht der Bildkomposition. Komplexe Knoten stellen Kombinationen aus mehreren Zeichenschritten dar, welche in den entsprechenden Abschnitten erläutert werden. Optionale Knoten können Übersprungen werden. Unter welchen Bedingungen dies passiert ist aus den jeweiligen Abschnitten zu entnehmen.

Seien `f(...)` die in „UVTransformation“ angegebene Skalierung einer UV-Koordinate und `g(...)` eine boolesche Funktion, die genau dann wahr ist, wenn die transformierte UV-Koordinate in einem Feld der Farbe Color1 liegt. Die Erzeugung des GLSL-Codes lässt sich dann, entsprechend dem Sequenzdiagramm der Abbildung 3.2, durchführen. Hierbei wird wie erarbeitet von jeder ChannelMap für einen erfragten Kanal ein Codesegment erzeugt. Dieses wird dann weiterverarbeitet, bis das Phong-Material alle Informationen besitzt um ein GLSL-Fragment-Programm zu erzeugen.

Da das Modell der Codegenerierung direkt an GroIMP angelehnt ist, können prinzipiell alle Materialien abgebildet werden.

3.2 Bildkomposition

Materialien alleine genügen nicht um ein Bild der Szene zu erzeugen. Sie bieten die Basis für das Deferred Shading und sind demnach nur Eingabeparameter für weitere Zeichenschritte, welche zusammen das fertige Bild erzeugen. Hierzu werden die in Abschnitt 2.4 vorgestellten Techniken benötigt. Die Kombination ergibt die in Abbildung 3.3 gezeigte Abfolge von Zeichenschritten, welche im Folgenden erarbeitet werden sollen.

Im Groben orientiert sich der Aufbau des Bilder an der Depth Peeling-Technik und erzeugt das Bild von „Vorne nach Hinten“. Jeder Zeichenschritt erzeugt seine Ausgabe in einer Textur, die von den folgenden Schritten als Eingabe genutzt werden kann.

3.2.1 Zwischenspeichern der Szene

Zunächst wird überprüft, ob sich der Szenengraph seit dem letzten Zeichenvorgang geändert hat. Ist dies der Fall, werden alle gesicherten zeichenbaren Objekte verworfen. Materialien können weiterhin gesichert werden, da diese durch eine eindeutige Referenz identifiziert werden können. Durch vollständiges Ablaufen des Graphens wird der Cache erneut gefüllt. Es müssen gegebenenfalls weitere GLSL-Materialien erzeugt werden.

Da für Multipass-Verfahren wie Depth Peeling in jedem Schritt die selben Eingabedaten benötigt werden, müssen zu Beginn des Zeichenvorganges alle Eigenschaften, welche durch externe Einflüsse

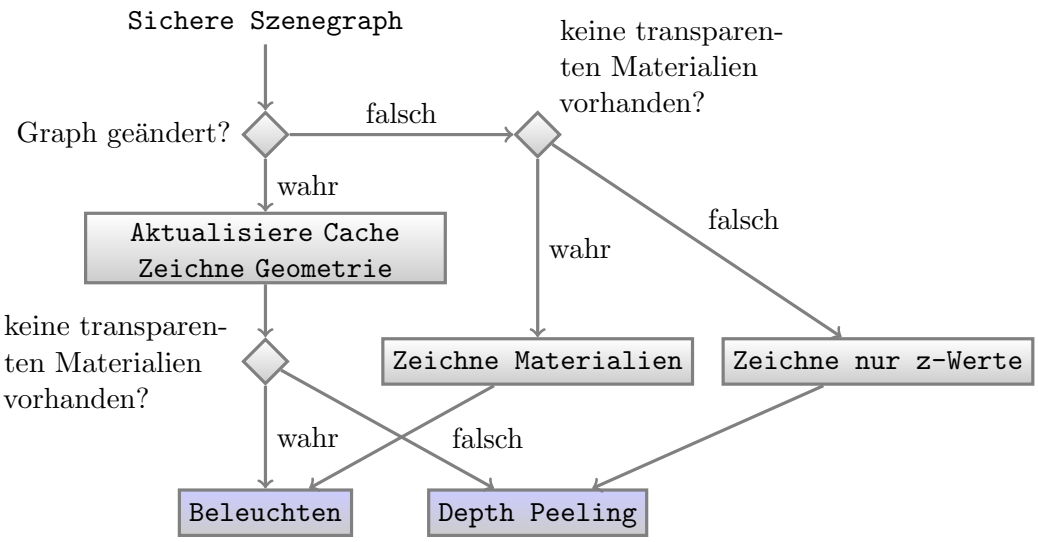


Abbildung 3.4: Zeichenschritte, die während des Cachen der Szene anfallen.

verändert werden können, gesichert werden. Dies betrifft insbesondere die Kameraposition und die Objekte der Szene. Entsprechende Caching-Ansätze für Objekte und deren Materialien werden im nächsten Kapitel erläutert. Der grobe Ablauf ist der Abbildung 3.4 zu entnehmen.

Um in späteren Zeichenschritten die gefundenen Objekte besser klassifizieren zu können, werden diese zunächst grob unterteilt.

Solide Objekte werden hierbei sofort gezeichnet, um den Tiefenpuffer für die weiteren Renderschritte zu füllen. Steht bereits fest, dass nur solide Materialien in der Szene vorkommen, so kann ein Zeichenschritt eingespart werden. Hierzu werden während des Füllens des Tiefenpuffers direkt alle Materialeigenschaften ausgeschrieben, wie in Abschnitt 3.2.3 noch erläutert wird.

3.2.2 Depth Peeling

Um lichtdurchlässige Objekte zeichnen zu können, müssen diese der Entfernung zur Kamera nach absteigend oder aufsteigend sortiert sein. Je nach Sortierung kann dann mittels der in Abschnitt 2.4.5 beschriebenen Gleichungen das Bild zusammengesetzt werden. Ein korrektes Ergebnis kann allerdings nur erzielt werden, wenn nicht nur Objekte oder deren Vertizes sondern die Fragmente der Objekte geordnet werden.

Das Sortieren der Fragmente wird durch die Nutzung von Deferred Shading noch weiter erschwert. Nach dem Deferred Shading existieren pro Pixel der Bildebene nur noch die Werte eines Fragments. Ist dieses transparent, so fehlen zur korrekten Darstellung alle Fragmente die von diesem verdeckt werden.

Eine Möglichkeit besteht darin, die Szene zunächst mittels Deferred Shading zu rendern und anschließend halbtransparente Objekte über ein direktes Beleuchtungssystem zu berechnen und nachträglich einzufügen, wie es in vielen Spielen üblich ist:

> „The single largest drawback of deferred shading is its inability to handle alpha-blended geometry. Alpha blending is not supported partly because of hardware limitations, but it is also fundamentally not supported by the technique itself as long as we limit ourselves to keeping track of the material attributes of only the nearest pixel. In Tabula Rasa, we solve this the same way everyone to date has: we render translucent geometry using our forward renderer after our deferred renderer has finished rendering the opaque geometry.“ [19, Kapitel 19.8.1, Seite 450]

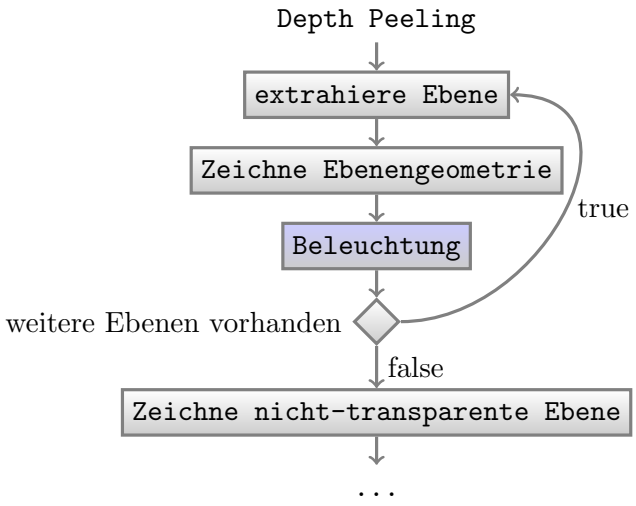


Abbildung 3.5: Zeichenschritte, die während der Beleuchtung der Szene anfallen.

Da die geforderte Implementierung nicht den strengen Geschwindigkeitsanforderungen der Grafikengine eines 3D-Spiels unterliegt, lässt sich ein einheitliches Bildergebnis wie folgt erzielen. Aus der Erläuterung von Hargreaves zum Deferred Shading [16] geht hervor, dass Depth Peeling wie vorgestellt von Everitt [9] genutzt werden kann, um halbtransparente Materialien nicht gesondert betrachten zu müssen. Für jede Tiefenebene, welche über Depth Peeling extrahiert wird, kann Deferred Shading genutzt werden. Somit wird die Berechnung aller Materialeigenschaften für jedes verwendete Fragment nur einmal durchgeführt.

Depth Peeling selbst gibt es in vielen Varianten. Multi-Layer-Verfahren, wie „Dual Depth Peeling“ [1] oder „Depth Peeling via Bucket Sort“ [22] bieten zwar hohe Geschwindigkeiten, können aber nur eine geringe Anzahl an Parametern verarbeiten, meist den Farbwert (RGB) und einen Alphawert (A). Somit lassen sich über mehrere Renderziele mehrere Ebenen gleichzeitig berechnen.

Da eine Designentscheidung lautet die Hardwareanforderungen bei OpenGL 2.0-fähigen Grafikkarten anzusiedeln, sind mehr als vier gleichzeitig aktive Renderziele nicht aktzeptabel. Deferred Shading erzeugt allerdings eine große Anzahl an Ausgabeparametern, insgesamt sechzehn. Somit werden für eine Ebene bereits vier Renderziele benötigt.

Eine mögliche Lösung ist, zunächst nur die z-Ebenen zu bestimmen und pro Ebene in einem nachträglichen Zeichenschritt die benötigten Parameter auszuschreiben. Allerdings gibt es einen weiteren Nachteil bei Multi-Layer-Verfahren. So besteht das Problem, dass die Fragmente der Ebenen meist ungeordnet extrahiert werden. Sie müssen dann nachträglich sortiert werden. Es ist somit nicht möglich das Depth Peeling zu unterbrechen und mit Teilergebnissen die Szene zu zeichnen. Es bleibt daher nur die Möglichkeit eines Single-Layer-Verfahrens, welches die Berechnung komplexer Shader reduziert.

Es bietet sich hierbei eine auf GLSL basierte Variante des von Everitt [9] vorgestellten Depth Peelings an. Hierbei werden einzelne Ebenen von „Vorne nach Hinten“ extrahiert. Über das in Abschnitt 2.4.5 erläuterte Under Blending können dann extrahierte Ebenen inkrementell verarbeitet werden. Beim Erreichen eines vom Nutzer festgelegten Schwellenwertes bricht das Verfahren, wie von Everitt beschrieben ab:

> „The nature of the transparency computation is that samples further back have diminished effect, so truncation is a reasonable (and efficient) form of approximation.“[10]

Für jede Depth Peeling Ebene, sowie die Ebene der soliden Objekte wird das Zeichnen der Geometrie und die Beleuchtung gleichermaßen durchgeführt:

[10] Cass Everitt[9, p. 9]

3.2.3 Zeichnen der Geometrie

Um eine gegebene Ebene anzuzeigen wird nach der Idee des Deferred Shadings zunächst für jeden Pixel eine Reihe an Parametern mittels der bereits generierten GLSL-Materialien erzeugt, in Texturen gesichert und dann in weiteren Schritten verarbeitet:

Deferred Shading Das Phong-Material, wie es in GroIMP vorkommt legt die folgenden, schon bekannten Parameter fest:

Diffuse colour, Specular colour, Shininess, Transparency, Diffuse Transparency, Transparency Shininess, Ambiente colour, Emissive colour, sowie Interior

Da nur das Phong-Lichtmodell zur Beleuchtung der Szene genutzt wird, lassen sich die Parameter aller anderen Materialien auf die bekannten Parameter verteilen.Lichtbrechung soll in der Vorschau nicht angezeigt werden, weshalb der Parameter Interior ignoriert werden kann. Somit sind 8 Werte durch das Phong-Material bestimmt. Diese können je nach ChannelMap aus bis zu 3 Komponenten bestehen, weshalb bis zu 24 Werte zu sichern sind. Dazu kommen die für die Lichtberechnung nötigen Angaben: Position und Normale des Fragments. Wie von Harris und Hargreaves [16] beschrieben, können Position und Normale komprimiert gespeichert werden: Liegt die Normale als Vektor (nx, ny, nz) vor, kann aus nx und ny über $nz = \sqrt{1 - nx^2 - ny^2}$ die z-Koordinate rekonstruiert werden. Hierbei geht das Vorzeichen des z-Wertes verloren. Dieses muss zusätzlich gesichert werden, da Aufgrund perspektivischer Projektionen auch negative z-Werte entstehen können. Dies geht beispielsweise aus der Präsentation von Mark Lee [21] hervor. Da das Vorzeichen von nz somit zusätzlichen Speicher benötigt und nicht problemlos gesichert werden kann, muss ein anderes Verfahren genutzt werden.

Wie von Aras Pranckevičius [26] analysiert, eignet sich unter Anderem die flächentreue Azimutalprojektion um eine Normale zu kodieren . So ergibt sich für die Normale (nx, ny, nz) die Kodierungsvorschrift

$$(Nx, Ny) := \sqrt{\frac{2}{1 - nz}} \cdot (nx, ny)$$

mit der Umkehrfunktion

$$(nx, ny, nz) = \left(\sqrt{1 - \frac{Nx^2 + Ny^2}{4}} \cdot (Nx, Ny), -1 + \frac{Nx^2 + Ny^2}{2} \right)$$

Das genannte Verfahren ist für $nz = 1$ nicht definiert, weshalb dieser Fall durch entsprechende Skalierung der Normale verhindert werden muss. Pranckevičius nutzt hierfür ein leicht verändertes Verfahren, indem für die Kodierung:

$$(Nx, Ny) := (nx, ny) \,/\, \sqrt{8 \cdot nz + 8} + 0.5$$

gerechnet wird. Für die Dekodierung wird

$$\begin{aligned} (NX, NY) &:= 4 \cdot (Nx, Ny) - 2 \\ (nx, ny, nz) &= \left(\sqrt{1 - \frac{NX^2 + NY^2}{4}} \cdot (NX, NY), -1 + \frac{NX^2 + NY^2}{2} \right) \end{aligned}$$

bestimmt.

Für die Position genügt es, nur die z-Koordinate zu sichern, da über die bekannte Projektion der Kamera die x- und y-Koordinate bestimmt werden können. Da für das Depth Peeling bereits die

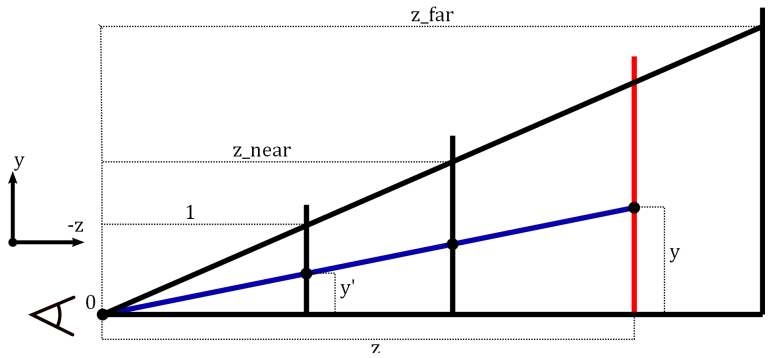


Abbildung 3.6: Die z=-1 Ebene im Sichtbereich.

z-Koordinaten der Fragmente in Texturen gesichert wird, lässt sich ein Eintrag in den Deferred Shading Texturen einsparen. Nachdem der z-Wert ($\tilde{z}$) aus der passenden Textur ausgelesen wurde, muss dieser zunächst wieder in einen linearen Wertebereich transformiert werden. Hierzu wird der homogene Vektor $(0, 0, \tilde{z}, 1)$ mit der inversen Projektionsmatrix multipliziert. Nun wird der Vektor durch seinen vierten Eintrag, die w-Komponente, geteilt. In der z-Komponente des Ergebnisses steht nun die z-Koordinate des Fragments im Sichtkoordinatensystem. Diese wird im Folgenden vereinfacht mit z bezeichnet. Um aus diesem Wert die x-, und y-Komponente zu bestimmen wird projektionsabhängig vorgegangen:

Für perspektivische Projektionen werden die xy-Komponenten der vier Schnittpunkte einer Ebene mit Entfernung 1 zur Kamera mit der Pyramide des Sichtbereiches bestimmt. Bei der Lichtberechnung wird je Lichtquelle ein Vollbild-Rechteck (siehe 2.3.7) gezeichnet. Den vier Vertizes des Rechtecks können nun die berechneten Schnitt-Koordinaten als Texturkoordinaten der vier Ecken mitgegeben werden. OpenGL interpoliert die Koordinaten linear auf dem Rechteck, weshalb zu jedem Fragment die xy-Komponenten für eine Entfernung von 1 zur Verfügung stehen. Das zweidimensionalen Beispiel der Abbildung 3.6 zeigt, wie die Koordinaten zu rekonstruieren sind. Sei der Schnittpunkt der Linie, die durch das aktuelle Fragment mit der $z = -1$ Ebene als $(x', y', -1)$ gegeben, gilt nach dem Strahlensatz für die y-Koordinate: $\frac{y'}{1} = \frac{y}{z} \leftrightarrow y = z \cdot y'$. Für die x-Koordinate folgt äquivalent: $x = z \cdot x'$. Da der rekonstruierte z-Wert negativ ist, lässt sich die ursprüngliche Koordinate ($\overrightarrow{pos}$) folgendermaßen bestimmen: $\overrightarrow{pos} = (-s, -t, 1) \cdot z$. Bei orthogonalen Projektionsmatrizen werden in der Texturkoordinate direkt die Ausmaße der Projektion gesichert, weshalb die ursprüngliche Koordinate über $\overrightarrow{pos} = (s, t, z)$ berechnet werden kann.

Somit bleiben nach Sichern von Normale und Position bei vier Renderzielen 14 weitere freie Felder, in denen 24 Materialeigenschaften gesichert werden müssen. Um dies zu erreichen, ist es nötig, entweder die Zahl der Parameter zu reduzieren, oder diese zu komprimieren.

Zunächst ist anzumerken, dass der Parameter Ambient colour keinen ersichtlichen Effekt in GroIMP erzielt. Es existiert eine unfertige Klasse mit der Bezeichnung AmbientLight, welche den Parameter Ambient colour nutzen könnte. Diese ist jedoch auskommentiert. Nach dem Lichtmodell von Phong (2.2) ist der spekulare Exponent für den gesamten Farbbereich konstant, weshalb angenommen wird, dass für die Vorschau jeweils ein Wert für die Parameter Shininess und Transparency Shininess genügt.

Demnach sind noch 17 Werte in 14 Felder zu kodieren. Es wird hier ein Verfahren um zwei Byte Werte in einen 16-Bit-Float Feld zu packen verwendet, welches im folgenden Kapitel näher beschrieben wird.

Nach dem Deferred Shading sind sämtliche geometrischen Informationen, wie auch alle notwendigen Materialeigenschaften in Texturen gesichert. Um diese weiter zu verarbeiten, werden im Folgenden Vollbild-Shader (siehe 2.3.7) genutzt. Diese erhalten neben den Texturen des Deferred

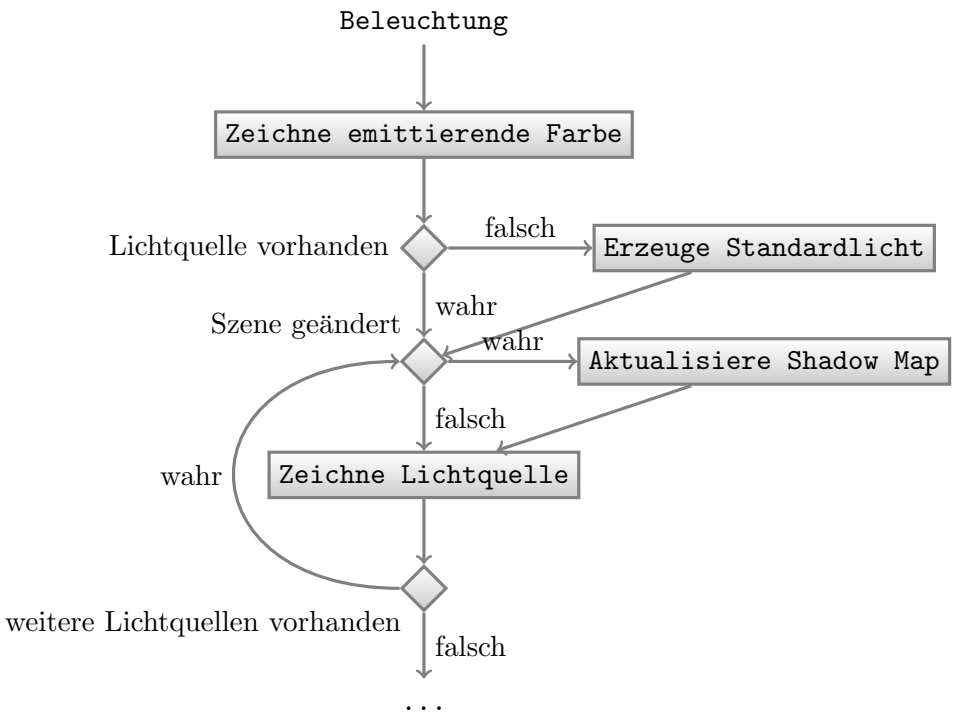


Abbildung 3.7: Zeichenschritte, die während der Beleuchtung der Szene anfallen.

Shadings zusätzliche Eingabedaten und legen das Ergebnis ihrer Rechnung abermals in Texturen ab.

3.2.4 Beleuchtung

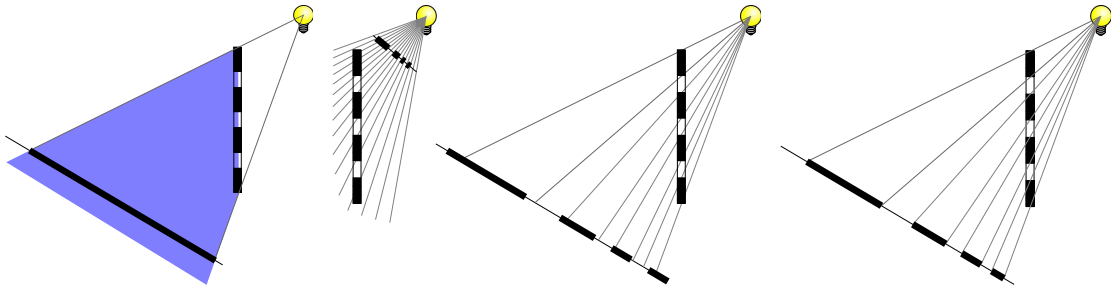
Vorbereitung zur Beleuchtung Für jede Ebene wird der Ziel-Alphawert (a_{dst}) nach der für das Under Blending (2.4.5) bestimmten Formel modifiziert. Die einzelnen Lichtquellen die im Folgenden gezeichnet werden sind linear unabhängig und können durch Skalieren der Beiträge im Sinne des Under Blendings als eine Ebene betrachtet werden.

Um die Beleuchtung der Szene zu realisieren, müssen alle Lichtquellen in GLSL-Programmen reimplementiert werden. Hierbei ist zu unterscheiden, dass für die Darstellung mit und ohne Schatten unterschiedliche GLSL-Programme notwendig sind, um die Anzahl der Code-Verzweigungen zu reduzieren. Lichtquellen verarbeiten Informationen der durch Deferred Shading erzeugten Texturen. Entsprechend der Übersicht in Abbildung 3.7 wird für jede Lichtquelle der Szene folgendermaßen vorgegangen.

Licht- und Schattenerzeugung einer Lichtquelle In diesem Schritt werden zu einer gegebenen Lichtquelle zunächst Schatteninformationen erzeugt.

Die Darstellung von Schatten kann nicht durch Schattenvolumen realisiert werden. In vielen Szenen werden Texturen mit einem Alpha-Kanal genutzt um Beispielsweise Blätter oder Nadeln eines Baumes anzuzeigen. Diese werden auf einfache geometrische Körper, wie ein Rechteck, projiziert. Abbildung 3.8 skizziert den Fall eines teiltransparenten Materials für die diskutierten Verfahren im Zweidimensionalen. Da Schattenvolumen nur auf der Geometrie basieren, erzeugen sie nur von dieser einen Schatten. Raytracing-Verfahren sind bisher noch zu zeitaufwendig, demnach fällt die Wahl auf Shadow Maps.

Neben den Standard Shadow Maps nach Williams [35] gibt es eine Reihe von szenenabhängigen Verfahren. Für Parallele Lichtquellen eignen sich beispielsweise „Parallel-Split-Shadow-Maps“ [38].



(a) (b) (c)

Abbildung 3.8: Schattenberechnung verschiedener Verfahren. Der gelbe Punkt zeigt eine Punktlichtquelle. Diese beleuchtet eine Fläche, deren Material abwechselnd solide und transparent ist. Der Schatten der Fläche ist auf der Ebene in der unteren linken Ecke des Bilder zu sehen. 3.8a zeigt die Technik Shadow Volumes, mit dem Schattenkörper in blau hinterlegt. Die materialbedingte Lichtdurchlässigkeit der Fläche kann von Shadow Volumes nicht abgebildet werden. 3.8b zeigt die Schatten der Shadow Map Methode. Links wird die Schattentextur erzeugt, rechts wird diese in die Szene projiziert. Aufgrund der begrenzten Auflösung einer Shadow Map sind Abweichungen zu den exakten Schatten sichtbar. 3.8c zeigt die korrekten Schatten, welche mittels einer Strahlverfolgungsmethode erzeugt wurden.

Diese sind meist deutlich aufwendiger zu berechnen als Standard Shadow Maps, da die Szene analysiert werden muss. Desweiteren existieren projektionsabhängige Shadow Map Verfahren wie „Perspective Shadow Maps“ von Stamminger und Drettakis [33], welche die Auflösung einer Shadow Map durch eine an die Kameraposition und -projektion angepasste Transformation nahe des Betrachters erhöhen. Solche Verfahren erzeugen bei selbem Speicherverbrauch weniger Bildartefakte als Standard Shadow Maps, müssen aber bei jeder Änderung der Szene oder der Kamera neu berechnet werden. Bei einer hohen Zahl von Lichtquellen, wie sie durch Deferred Shading unterstützt werden, kann die Darstellung nicht mehr interaktiv erfolgen. Es fehlen Verfahren zur Beschleunigung des Zeichnens der Szene, weshalb statische Shadow Maps zu bevorzugen sind. Somit bleiben Standard Shadow Maps, da diese nur bei Änderungen in der Szene neu gezeichnet werden müssen.

Die Erzeugung passender Shadow Maps unterscheidet sich je nach Typ der Lichtquelle leicht voneinander:

Parallele Lichtquellen Diese beleuchten jeden Punkt des Raumes mit gleicher Intensität. Zum Generieren einer Shadow Map ist demnach das Wissen über die Ausmaße der Szene nötig. Um eine ungefähre Aussage über die Szenenausmaße zu treffen, wird während der Traversierung des Graphens zu jedem zeichenbaren Objekt eine Hüllkugel bestimmt. Über diese lässt sich eine weitere Hüllkugel bestimmen, welche die gesamte Szene und somit alle schattenwerfenden Objekte beinhaltet. Der Radius der Hüllkugel kann genutzt werden um die Breite, Höhe und Tiefe einer orthogonalen Projektion festzulegen, welche zum Erzeugen der Shadow Map nötig ist. Es ist anzumerken, dass beispielsweise Ebenen keine beschränkte Ausdehnung haben und demnach beim Erzeugen von Shadow Maps nicht beachtet werden sollten.

Punktlichtquellen Für die Umsetzung von Schatten für omnidirektionale Lichtquellen eignen sie die schon beschriebenen Textur-Würfel. Die Idee besteht darin die Szene für jede der sechs Seiten des Würfels mit einer perspektivischen Projektionsmatrix zu zeichnen, die einen Blickwinkel von genau 90° definiert. Zu beachten ist, dass die Koordinaten zum Auslesen des „Schattenwürfels“ dem Richtungsvektor von der Lichtquelle zu der zu testenden Bildposition entspricht. Um aus den sechs möglichen Transformationen, die sich aus den einzelnen Seiten des Würfels ergeben die Passende auszuwählen, wird bestimmt, welche Seite vom Richtungsvektor geschnitten wird. Diese Transformationen sind allerdings, da sie orthonormal zueinander stehen, nur Permutationen der selben Matrix. Durch einen Größenvergleich der Komponenten des besagten Vektors lässt sich schnell die benötigte Permutation finden

und der Schattentest auswerten. Um Wurzelberechnungen einzusparen, ließe sich in den Texturen des Würfels die Entfernung zur Lichtquelle im Sichtkoordinatensystem speichern. Eine entsprechende Implementierung liefert das Verfahren von Gerasimov [10]. So müsste für einen Entfernungsvergleich die Strecke zwischen Fragment und Lichtquelle nicht in das Koordinatensystem der Lichtquelle transformiert werden. Dies hat allerdings den Nachteil, dass für die Generierung des Schattenwürfels für viele Materialien eigene GLSL-Programme erzeugt werden müssten.

Kegellichtquellen Der Schatten dieses Lichtquellentyps kann bei einem Öffnungswinkel von weniger als 180° durch eine einzige Shadow Map dargestellt werden. Für einen größeren Winkel können entweder Projektion wie für „Dual Paraboloid Shadow Mapping“ von Brabec et al. [3] beschrieben genutzt werden oder das für Punktlichtquellen definierte Verfahren. „Schattenwürfel“ weisen deutlich weniger Verzerrung, weshalb sie für diese Arbeit vorzuziehen sind.

Schatten bei transparenten Materialien Für lichtdurchlässige Materialien lässt sich, ähnlich dem von OpenGL bereitgestellten Alpha-Test, eine Grenze festlegen, ab der ein Fragment einen Schatten wirft. Gibt der Mittelwert über die Transparenz eines Fragmentes an, das dieses zu mehr als 50% Lichtdurchlässig ist, so wird es nicht in die Shadow Map geschrieben. Ein besseres Ergebnis ließe sich über Depth Peeling erzielen, indem mehrere Schattenebenen extrahiert werden und diese mit einem Lichtdurchlässigkeitsfaktor getrennt für RGB gesichert werden. Dieser Ansatz ist sehr Rechenintensiv und wurde dementsprechend nicht nicht weiter verfolgt.

Nachdem die Shadow Maps angelegt wurden kann die Szene beleuchtet werden. Hierbei wird für jede Lichtquelle zunächst aus den Deferred Shading Texturen die Szene rekonstruiert. Danach wird getestet, ob das aktuelle Fragment im Schatten liegt. Liegt es nicht im Schatten, so wird die Phong-Beleuchtungsgleichung aus Abschnitt 2.2 angewandt. Das Ergebnis wird zum Ergebnis der vorherigen Beleuchtungsschritte in einer HDR-Textur addiert.

3.2.5 Zeichnen des Himmels

Nachdem alle Objekte der Szene von „Vorne nach Hinten“ sortiert gezeichnet wurden, muss das restliche Bild mit einem Hintergrund versehen werden. In GroIMP kann dies entweder ein Himmel, definiert durch eine SkyNode-Objekt oder ein Standardmuster sein. Der Himmel wird am besten durch eine Kugel mit unendlichem Radius beschrieben, welche die Szene umgibt. Da für den Himmel eine Leuchtstärke angegeben wird und dieser selbst nicht durch andere Lichtquellen beleuchtet werden kann, entfällt in diesem Schritt das Deferred Shading. Für den Himmel wird ein GLSL-Programm erzeugt, welches diesen in entsprechender Intensität direkt per Under Blending in die Szene einfügt.

3.2.6 Nachbearbeitung des Bildes

Nachdem das Bild durch einzelne Beleuchtungsschritte in ein HDR Format geschrieben wurde, muss dieses durch einen Tonemapping-Schritt in ein anzeigbares Format reduziert werden:

Tonemapping GroIMP nutzt einen linearen Tonemapping Operator. Sind die Pixel des HDR Bildes als Farbtupel(R, G, B) in der Menge P enthalten, so berechnet sich der Farbwert eines Punktes (c_{out}) durch den ursprünglichen Farbwert (c_{in}) geteilt durch das Maximum aller Farbkanäle über alle Punkte im Bild.

$$c_{out} = \frac{c_{in}}{\max_{col \in P} (col.r, col.g, col.b)}$$

Ein alternativer Tonemapping Operator ist der von Reinhard und Devlin [30] vorgestellte Operator, welcher sowohl globale, als auch lokale Bildinformationen nutzt. Hierbei lassen sich Details, welche beim linearen Mapping verschwinden würden besser darstellen. Die Wahl des Tonemapping Operators beeinflusst das dargestellte Bild deutlich, weshalb dem Nutzer überlassen werden sollte, welches Verfahren er bevorzugt.

Um Tonemapping vorzunehmen, müssen zum Zeitpunkt des Tonemappings einige globale Werte über das Eingabebild vorliegen. Nach Green und Cebenoyan [12, S. 13ff] lassen sich diese effizient über eine Kette von Shadern extrahieren, welche das Eingabebild zunächst auf ein Teilbild mit den relevanten Parametern reduzieren. In weiteren Schritten wird jeweils eine Gruppe von Pixeln auf einen Pixel reduziert, welcher die benötigten globalen Parameter für die Pixelgruppe speichert. Dieser Vorgang wird wiederholt, bis die Parameter für das gesamte Bild in einem Pixel vorliegen, welcher zur weiteren Bearbeitung durch einen Tonemapping-Shader ausgelesen werden kann.

Zeichnen von Hilfsmitteln Tonemapping verändert die Farbwerte eines Bildes. Es existieren verschiedene, visuell unterstützende Hilfsmittel, wie der Lichtkegel einer Kegellichtquelle oder ein Gitternetz, welches die Orientierung erleichtert, die in der interaktiven Ansicht gezeichnet werden müssen. Diese Objekte haben eine feste Farbe in der sie dargestellt werden und dürfen deshalb nicht als Teil des Tonemappings verändert werden. Um zu verhindern, dass sie zum Tonemapping beitragen werden in den vorherigen Schritten Fragmente, deren Farbe durch genannte Werkzeuge bestimmt wird zunächst mit einer neutralen Farbe (Schwarz) gezeichnet. Sie werden nachträglich in die Szene eingefügt. Mittels Under Blending kann Hilfsgeometrie hinter lichtdurchlässigen Objekten korrekt eingefügt werden.

Hilflinien: Da die Tiefenwerte der Hilfsobjekte bereits in den Tiefenpuffer geschrieben wurden, lassen sie sich leicht wieder in die Szene einfügen. Der Tiefentest wird auf `GL_EQUAL` gesetzt und die Objekte werden erneut gezeichnet.

Standard-Hintergrund: Definiert die gegebene Szene keinen Himmel oder ist das Zeichnen des Himmels deaktiviert vereinfacht ein grau gekachelter Hintergrund die Wahrnehmung transparenter Objekte. So können sehr dunkle helle Objekte gut gesehen werden. Diese Ebene stellt die hinterste zeichenbare Ebene da.

Nachdem alle Hilfsmittel gezeichnet wurden kann das fertige LDR-Bild in den primären Zeichenpuffer kopiert und angezeigt werden. Aus der bereits vorhandenen OpenGL Implementierung werden nun noch einige weitere Hilfsmittel über die Szene gezeichnet, wie eine Vorschau auf die Lage der Raumachsen, Hilfsobjekte zum Rotieren, Skalieren und Verschieben von Objekten oder die Auflösung der Darstellung.

4 Implementierung

Im folgenden Kapitel soll ein Überblick über die Implementierung der entwickelten Lösung gegeben werden. Im ersten Abschnitt wird die Architektur vorgestellt. Danach wird ein kurzer Einblick in wichtige Pakete und Klassen der Java-Implementierung gegeben. Im nächsten Abschnitt wird die implementierte Caching-Strategie für den Szenegraphen erläutert. Diese ist notwendig, da das Traversieren des Szenegraphens sehr zeitaufwendig ist. Es folgt eine Erläuterung zur Verwendung der Deferred Shading Texturen sowie Erklärungen zur Generierung von GLSL-Programmen zu gegebenen Materialien. Das Kapitel schließt mit einer Depth Peeling Implementierung sowie Erläuterungen zu der Verwendung des Stencilpuffers.

4.1 Architektur

Zunächst soll ein grober Überblick über die Architektur der Implementierung gegeben werden. Der folgende Abschnitt zeigt ein stark vereinfachtes UML-Diagramm. Da oft GroIMP-interne Objekte in OpenGL Objekte übersetzt werden mussten, zeigt der zweite Abschnitt ein für diese Aufgabe entwickeltes Entwurfsmuster.

4.1.1 Paket-Diagramm

Abbildung 4.1 zeigt eine Übersicht der Klassen und Pakete. Eine detaillierte Beschreibung ist der javadoc-Dokumentation zu entnehmen.

Im folgenden sollen ausgewählte Aspekte der API paketorientiert erläutert werden.

glsl In diesem Paket befinden sich die Klassen, welche die Kommunikation mit GroIMP regeln. GLSLDisplay ist hierbei die Klasse, die die von GroIMP benötigten Schnittstellen implementiert und die Methode `render(...)` zur Verfügung stellt, welche der Einstiegspunkt der Bilderzeugung ist. GLSLDisplay stellt desweiteren einige Funktionen bereit, welche es erlauben Einfluss auf die von OGL1 bereitgestellten Zeichenmethoden zu nehmen. Die bestehende OpenGL Implementierung ist hierbei in der Klasse GLDisplay enthalten, von der GLSLDisplay die benötigten Zeichenmethoden erbt.

Weitere wichtige Klassen in diesem Paket sind OpenGLState, welche als Kommunikationsinterface zwischen OpenGL und vielen anderen Klassen dient. Hier wird beispielsweise sichergestellt, dass ein bereits gebundenes GLSL-Programm nicht erneut gebunden wird. Desweiteren kapselt OpenGLState global genutzte Objekte wie die einzelnen FBOs in Form einer Instanz der Klasse GLSLFBOManager. Zudem bietet es kleinere nützliche Funktionen, wie die Messung von Zeitintervallen oder das Überschlagen des genutzten Grafikkartenspeichers an. GLSLUpdateCacheVisitor wird instanziiert um den Szenegraphen von GroIMP abzulaufen und relevante Knoten zu cachen.

glsl::utility Hier werden Klassen definiert, welche einen erleichterten Zugang zu den in OpenGL vorhandenen Objekten darstellen. So werden FBOs, Texturen, Renderpuffer, Shader und andere Objekte in eigene Klassen gekapselt, welche Funktionen wie `delete`, `create` und `resize` bereitstellen. Desweiteren wird eine Datenklasse Drawable definiert, die alle benötigten Informationen

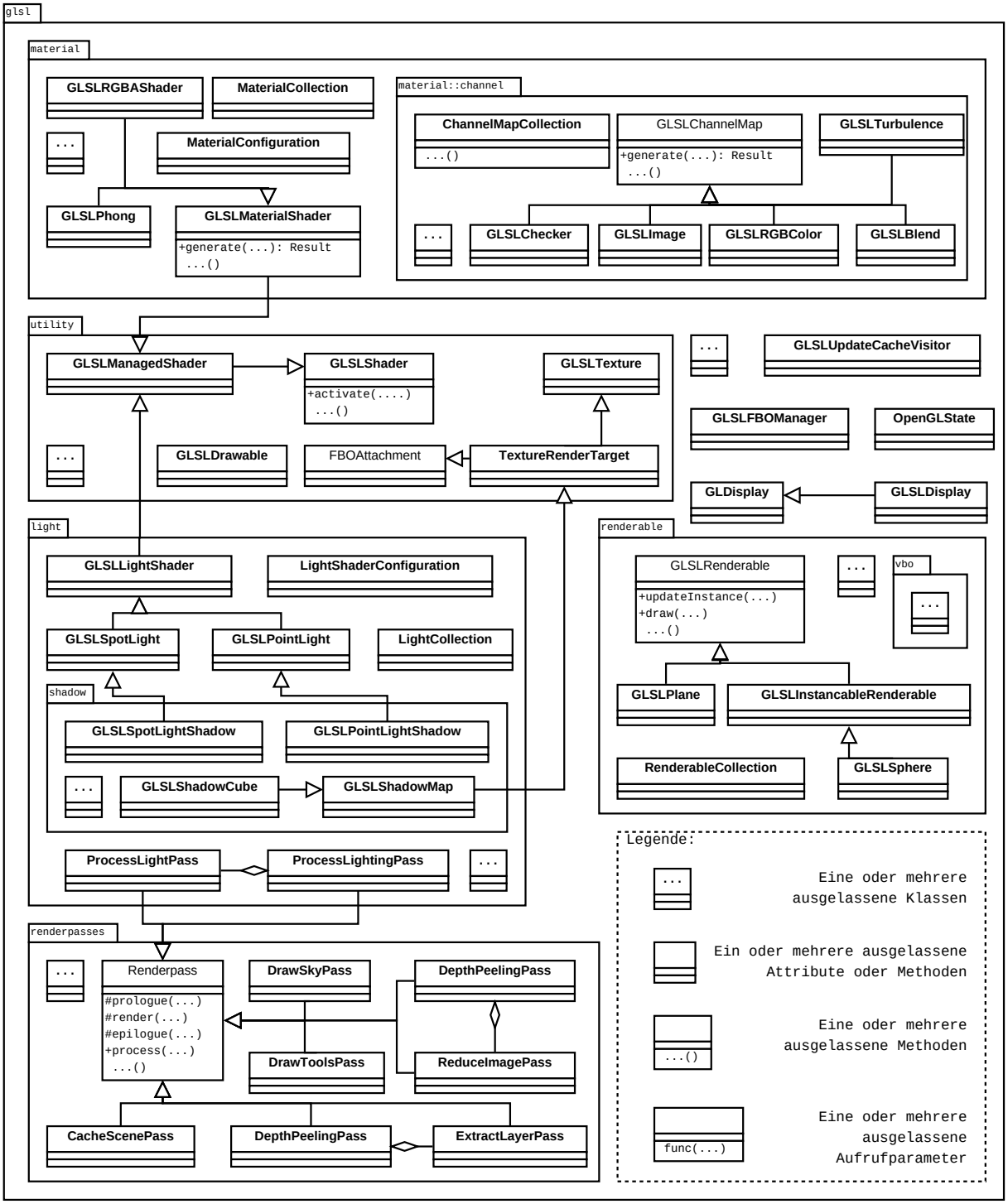


Abbildung 4.1: Architektur von Proteus

eines zeichenbaren GroIMP-Objektes kapselt. Hervorzuheben ist die Klasse CachedShaderCollection, die GLSL-Programme wie Material- und Licht-Shader bei Bedarf erzeugt und automatisch entfernt, sollten diese nicht genutzt werden.

glsl::material In diesem Paket sind die Klassen der Material-Shader gebündelt. Zusätzlich werden verschiedene materialspezifische Konfigurationen definiert (diese werden in Abschnitt 4.2.2 erläutert). MaterialCollection entspricht hierbei der Prototypensammlung aus dem nächsten Abschnitt (4.1.2).

glsl::material::channel Dieses Paket bündelt GLSL-spezifische Übersetzungen der in GroIMP vorhandenen ChannelMap-Klassen.

glsl::renderable Hier befinden sich Übersetzungen der zeichenbaren Objekte aus GroIMP. Diese werden zum Zwischenspeichern von geometrischen Formen benötigt. RenderableCollection wird genutzt, um zu einer vorhandenen Shape eine neue Instanz zu erzeugen, sowie diese mit Informationen aus dem zugehörigen GroIMP-Objekt zu initialisieren. Desweiteren können je nach Typ der GroIMP-Form alternative Zeichenmethoden implementiert werden (siehe beispielsweise Abschnitt 4.5.1).

glsl::renderable::vbo In diesem Paket werden Übersetzungen der geometrischen Körper in Vertex-Buffer-Objects[11] (kurz VBOs) gesammelt. Bisher sind nur Würfel implementiert. Demnach dient dieses Paket als Platzhalter für GroIMP-Objekte, welche in Zukunft durch VBOs realisiert werden sollen. Demnach ist der VBOManager nur eine Dummy-Klasse, welche zum Erfragen eines VBOs zur Darstellung einer Box genutzt werden kann.

glsl::light In diesem Paket werden die Übersetzungen zu verschiedenen Lichtquellen gebündelt. Bisher existieren nur Lichtquellen, welche Phong-Beleuchtung umsetzen. Desweiteren werden die für die Beleuchtung nötigen Renderschritte hier definiert.

glsl::light::shadow Zur Darstellung von Schatten werden hier benötigte Überladungen der in glsl::light definierten Lichtquellen angegeben. Damit kann die Berechnung von Schatten entweder pro Lichtquelle oder global deaktiviert werden, um die Zeichengeschwindigkeit zu erhöhen. Desweiteren werden eigene Klassen zur Verwaltung von Shadow Maps definiert.

glsl::renderpass Dieses Paket beinhaltet die Zeichenschritte, Renderpasses genannt, die zur Erzeugung des Bildes nötig sind. Die Basis bildet hier die gleichnamige Klasse Renderpass, welche die abstrakten Methoden `prologue`, `render` und `epilogue` bereitstellt. Diese werden von der öffentlichen Methode `process` nacheinander aufgerufen. Einzelne Renderpasses können Weitere bündeln und so eine Zeichenhierarchie bilden. Die in 3.3 vorgestelle Bildkomposition entsprechend der einzelnen Elemente zerlegt und in Implementierungen von Renderpass realisiert. Für spätere Modifikationen können so einzelne Elemente wiederverwendet werden. In `prologue` setzen Renderpässe alle benötigten Parameter, wie das Binden bestimmter FBOs und Texturen. In der Methode `render` werden tatsächliche Zeichoperationen ausgeführt, zum Beispiel das Zeichnen eines Himmels. Die Methode `epilogue` dient als Sammelfunktion zum Wiederherstellen des Standard OpenGL Zustands. Hierbei ist zu beachten, dass aus Geschwindigkeitsgründen Renderpässe nicht gezwungen sind diesen Zustand wiederherzustellen. So werden Texturmatrizen und gebundene Texturen erst beim Beenden des letzten Renderschrittes zurückgesetzt.

[11] http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt (Stand: April 2010)

4.1.2 Entwurfsmuster zur Übersetzung von GroIMP-Objekten nach OpenGL

Da primär eine Übersetzung von GroIMP bzw Java nach OpenGL bzw. GLSL stattfindet hat sich folgendes Muster für Übersetzerklassen, welche nur die Funktionalität anderer Klassen abbilden, bewährt:

- die Klasse besitzt eine `instanceFor()` Methode, welche das Klassenobjekt der Quellklasse angiebt.
- die Klasse besitzt eine `createInstance()` Methode, welche eine Instanz der Klasse erzeugt und zurückgibt. Hat die Klasse keinen inneren Zustand, so kann die Instanz sich selbst zurückgeben (`return this`). Hierfür könnte prinzipell auch der clone-Befehl genutzt werden, allerdings soll ausgedrückt werden, dass der innere Zustand nicht kopiert wird.
- Eine HashMap mit Instanzen der Übersetzerklassen stellt diese dem Programm zur Verfügung. Als Schlüssel wird die Rückgabe der `instanceFor()` Methode genutzt. Hierfür wird jeweils eine eigene Klasse mit der Namensendung „Collection“ definiert.
- Wird nun eine Übersetzung benötigt, kann bei der entsprechenden „Collection“ über die Klasse des Quellobjekts ein Übersetzerobjekt angefragt werden. Der Befehl `getInstance()` erzeugt dann eine neue Instanz, die für die Übersetzung genutzt werden kann.

Dieses Schema folgt der Idee des Prototyp-Entwurfsmusters[12] und wird genutzt um Prototypen für die folgenden Objekte zu verwalten:

GroIMP Materialien, ChannelMaps, Lichtquellen, Shadow Maps sowie die zeichenbaren Formen

4.2 Materialien

In diesem Abschnitt wird die Implementierung der Übersetzung von GroIMP-Materialien in GLSL-Code erläutert. Es wurden zunächst alle benötigten Material- und ChannelMap-Klassen von GroIMP nach dem Entwurfmuster aus Abschnitt 4.1.2 reimplementiert. Somit kann effizient zu jeder ChannelMap eine Instanz der passenden Übersetzerklasse erhalten werden. Wie in der Entwicklung der eigenen Lösung (Abschnitt 3.1) gezeigt müssen kanalabhängige Übersetzungen vorgenommen werden. Jede der Übersetzerklassen implementiert die Methode `generate`, die zu einem gegebenen Kanal (Channel), einer DefaultChannelMap und der zu übersetzenden ChannelMap ein passendes GLSL-Code-Segment generiert. Um die Erläuterung der Erzeugung von Code-Segmenten zu vereinfachen muss zunächst das Ausgabeformat erklärt werden. Danach werden Konfigurationen von Shadern sowie die Nutzung von Variablen erläutert. Der Vorgang der Code-Generierung wird schließlich am Beispiel des Schachbrett-Materials gezeigt. Zuletzt wird kurz der Cache für GLSL-Programme beschrieben. Dieser ist nötig, um nicht bei jeder Änderungen alle Materialien neu kompilieren zu müssen.

4.2.1 Zusammenfügen von Code-Segmenten

Bei der Generierung des GLSL-Quellcodes erzeugen einzelne ChannelMaps Code-Segmente in Form von Strings. Diese beschreiben einen GLSL Term, der einen Wert berechnen, desses Datentyp von der erzeugenden Klasse abhängt. Demnach muss bei der Aneinanderreihung dieser Terme, auf die Datentypen geachtet werden. So kann eine Funktion, welche erwartet mit `float`-Werten zu rechnen, nicht mit Parametern vom Typ `vec3` aufgerufen werden. Um die Konvertierung von Rückgabewerten einzelner Shader zu koordinieren, erzeugt jede implementierte ChannelMap-Übersetzung als Rückgabewert der `generate`-Methode ein Result-Objekt. Das Result-Objekt ist ein Tupel, dessen erste Komponente der GLSL-String ist. Die zweite Komponente enthält, einen

[12] http://home.earthlink.net/~huston2/dp/prototype.html (Stand: April 2010)

Tabelle 4.1: Konvertierungsschema zu Result-Objekten. Die Zeile gibt das Quellformat einer Variable `v` *an. Die Spalte zeigt Zielformat, in welches* `v` *übersetzt werden soll.*

	`float`	`vec2`	`vec3`	`vec4`
`float`	-	`vec2(v)`	`vec3(v)`	`vec4(vec3(v),1.)`
`vec2`	`dot(v, vec2(0.5))`	-	`vec3(v, 0.)`	`vec4(v, 0., 1.)`
`vec3`	`dot(v, vec3(1./3.))`	`v.xy`	-	`vec4(v, 1.)`
`vec4`	`dot(v, vec4(0.25))`	`v.xy`	`v.xyz`	-

int-Wert, welcher den Rückgabetyp definiert. Da der GLSL-Datentyp des Strings ohne Parsen bestimmt werden kann, codiert der Rückgabetyp einen der folgenden GLSL-Datentypen: `float`, `vec2`, `vec3`, `vec4`, `boolean`. Die Result-Klasse definiert desweiteren Konvertierungsvorschriften zwischen `float`, `vec2`, `vec3` und `vec4` um den Rückgabewert einer ChannelMap in ein passendes Format umzuwandeln. Diese folgen dem Schema aus Tabelle 4.1, das sich während der Übersetzung der ChannelMaps bewehrt hat. Wird eine andere Konvertierung bevorzugt, so muss diese manuell im erzeugten GLSL-Code vorgenommen werden.

4.2.2 Konfigurationen

Sowohl die GLSL-Shader, die zur Materialbestimmung verwendet werden, als auch solche, die Lichtquellen beschreiben treten in unterschiedlichen Variationen auf. So unterscheidet sich beispielsweise der GLSL-Code von Materialien, die gleichzeitg zum Zeichnen des Himmels genutzt werden von denen, die die Oberfläche eines Objektes darstellen. Der Unterschied liegt in der Generierung der Texturkoordinaten, sowie in der Ausgabe der Bildinformationen. Um dies zu unterstützen werden Konfigurationsklassen genutzt. Diese erzeugen aus den Code-Fragmenten eines Material- oder Licht-Shaders ein fertiges GLSL-Programm.

Konfigurationen werden weiterhin genutzt, um innerhalb eines GLSL-Shaders lokale und globale Variablen zu verwalten. Sie erlauben außerdem Funktionen für den zu generierenden Shader zu registrieren, welche dann innerhalb der Berechnungen aufgerufen werden können. Die zentrale Verwaltung ist notwendig um zu verhindern, dass unterschiedliche Code-Fragmente Variablen mit der gleichen Bezeichnung nutzen. Für einfache GLSL-Programme wie beispielsweise das Tonemapping werden keine Konfigurationen benötigt, da das Programm im Ganzen vorliegt und nicht aus Segmenten zusammengesetzt wird.

Lokale Variablen Um eine Variable zum Speichern von Zwischenergebnissen einer Rechnung anzulegen wird über die Konfiguration eine temporäre Variable angefordert. Der Konfiguration wird hierbei direkt der zu sichernde Wert und Typ mitgeteilt. Die Konfiguration erzeugt einen eindeutigen Namen für die angeforderte Variable über eine fortlaufende Nummer. Es wird ein Eintrag in der Konfiguration vorgenommen, der den Wert der angeforderten Variable zur späteren Code-Generierung sichert. Die Methode besitzt den eindeutigen Namen als Rückgabewert. Dieser kann dann für weitere Rechnungen genutzt werden.

Globale Variablen Das selbe Prinzip wird für Uniform-Variablen und Texturen genutzt. Für Uniform Variablen wird ein neuer Bezeichner erzeugt und zurückgegeben. Beim Aktivieren des Shaders muss gegebenenfalls der Variable ein Wert zugewiesen werden. Uniform-Variablen sind aufgrund des statischen Codes der generiert wird nur auf Material-Ebene einsetzbar. Hierzu muss das Material die Uniform-Bezeichner selbst sichern und bei Aktivierung des GLSL-Programmes mit passenden Werten initialisieren. Die Konfiguration stellt hier nur die Eindeutigkeit des Bezeichners sicher.

Bei Texturen ist der Wert kein GLSL-Code sondern ein Referenz-Objekt über das eine passende OpenGL-Textur erzeugt werden kann. Hier wird das Texturmanagement von OGL1 mitgenutzt. Dort werden OpenGL-Texturen in einer HashMap gesichert und durch ein Java-Bild-Objekt referenziert. Die Konfiguration selbst erzeugt wieder einen eindeutigen Namen und gibt diesen an die aufrufende Methode zurück. Das Binden der passenden Texturen sobald das GLSL-Programm aktiviert wird übernimmt die Konfiguration automatisch.

Methoden Um Methoden zu registrieren wird ein anderer Ansatz verfolgt. Da verschiedene ChannelMaps durchaus die selbe Methode nutzen können ist es unnötig für jede ChannelMap diese Methode neu zu definieren. Ein Beispiel ist die `noise`-Methode, die pseudozufällige Werte erzeugt. Beim Registrieren einer Methode wird eine eindeutige Signatur und der Methodenkörper übergeben. Sollte bereits eine Methode mit gleicher Signatur vorhanden sein, so wird diese überschrieben. Dadurch sind ChannelMaps für eindeutige Bezeichnungen verantwortlich.

Die Übersetung der Material-Shader aus GroIMP stellt demnach die statische Basis für jeden Shader dar. Hier werden die Code-Fragmente einzelner ChannelMaps nach dem Vorbild eines GroIMP Materials erzeugt. Diese werden dann durch eine Konfiguration zu einem vollständigen Fragment-Programm zusammengefügt. Desweiteren sichert die Konfiguration instanzspezifischen Daten eines Shaders. Dies sind beispielsweise die Texturen, die der Shader für seine Berechnungen benötigt.

4.2.3 Beispiel: Checker

Anhand des schon bekannten Beispiels aus Abbildung 2.1 soll die Erzeugung eines fertigen GLSL-Programms erläutert werden. Im folgenden wird die ChannelMap „Checker“ im Detail erläutert. Die ChannelMaps RGBColor und UVTransformation sind simple statische Übersetzungen, weshalb nur die Ergebnisse der Code-Erzeugung gezeigt werden.

ChannelMap Checker Der Quellcode der Klasse ist in Listing 4.1 zu sehen.

Um die Anfrage eines Kanals der Checker-Klasse zu übersetzen wird zunächst die UV-Koordinate des aktuellen Fragmentes benötigt. Hierzu wird der Kanal Channel.U aus dem ChannelInput-Element der zugrundeliegenden ChannelMap übersetzt. Dies geschieht in den Zeilen 8-11. Die Methode `generateResultWithChannelDefault` sucht hierbei zunächst für die angegebene ChannelMap (`ch.getInputChannel()`) die passende Übersetzerklasse. Ist dies nicht möglich, so wird der Kanal aus der DefaultChannelMap entnommen. Der Rückgabestring wird in das vec2-Format umgewandelt, falls er dieses noch nicht erfüllt. Die Rückgabe von UVTransformation könnte im Beispiel ein Tupel der Form

```
("vec2(4.*UV.s, 2.*UV.t)",VEC2)
```

sein. Hierbei sind die Texturkoordinaten des Fragmentes in der Variable „UV“ gesichert. Das Rückgabeergebnis wird in einer Variable gesichert, um nicht öfter als nötig berechnet werden zu müssen. Generell gilt: Der Rückgabestring ist eine komplexe Funktion, deren Ergebnis bei mehrfacher Nutzung in einer temporären Variable gesichert werden sollte. ChannelMaps sollten ihre eigenen Rückgabewerte nicht zwischenspeichern.

In den Zeilen 13-16 werden die beiden Resultat-Strings der zugeordneten Farben erfragt. Die Methode `generateResult` prüft zunächst ob die angegebene ChannelMap nicht null ist. Ist dies der Fall, so wird die angegebene Standard-ChannelMap, der letzte Parameter, für die Übersetzung genutzt. Zu der ChannelMap wird dann eine passende Übersetzerklasse gesucht, an die der angefragte Kanal übergeben wird. Das Resultat der Anfrage wird an die aufrufende Klasse durchgereicht. Im Beispiel sind das respektive `("vec3(1.)",VEC3)` und `("vec3(0.)",VEC3).`

Listing 4.1: Übersetzerklasse: GLSLChecker

```
1  public class GLSLChecker extends GLSLChannelMapNode {
2      @Override
3      public Result generate(ChannelMap channelMap, MaterialConfiguration config,
4              GLSLChannelMap defaultChannelMap, int channel) {
5          assert channelMap instanceof Checker;
6          Checker ch = (Checker) channelMap;
7
8          String UV = config.registerNewTmpVar(Result.ET_VEC2,
9                  generateResultWithChannelDefault(ch.getInput(),
10                         config, defaultChannelMap, Channel.U).convert(Result.ET_VEC2));
11
12         Result colo1 = generateResult((ChannelMap) Checker.color1$FIELD
13                 .getObject(ch), config, defaultChannelMap, channel, ch.getColor1());
14         Result colo2 = generateResult((ChannelMap) Checker.color2$FIELD
15                 .getObject(ch), config, defaultChannelMap, channel, ch.getColor2());
16
17         int rstType = getMaxResultType(colo1, colo2);
18
19         String boolvar = "mod(floor(2.0*(" + UV + ").s) + floor(2.0*(" + UV
20                 + ").t ),2.0) == 1.0";
21
22         return new Result("(" + boolvar + " ? " + colo1.convert(rstType)
23                 + " : " + colo2.convert(rstType) + ")", rstType);
24     }
25
26     @Override
27     public Class instanceFor() {
28         return Checker.class;
29     }
30 }
```

Die Zeile 18 bestimmt über die Methode `getMaxResultType` den kleinsten Datentyp, der die Informationen beider Farben sichern kann.

Im letzten Schritt, den Zeilen 20-24, wird der GLSL-Code, welcher den Shader repräsentiert erzeugt und an die aufrufende Funktion zusammen mit dem passenden Rückgabe-Typ gesendet. Hierbei werden beide Farbresultate in den zuvor bestimmten Datentypen umgewandelt.

Sind die transformierten UV-Koordinaten in einer Variablen mit Namen UV gesichert, so könnte ein mögliches Ergebnis vom Typ `vec3` der folgende String sein:

```
mod(floor(2.*tmp0.s) + floor(2.*tmp0.t), 2.) == 1.)?vec3(1.):vec3(0.)
```

Es wird die bei der Generierung eine temporäre Variable angelegt, die das Ergebnis der UVTransformation ChannelMap sichert.

```
vec2 tmp0 = vec2(4.*UV.s, 2.*UV.t);
```

Das Phong Material erzeugt nun Code für alle möglichen Parameter wie beispielsweise die diffuse und spekulare Farbe. In der Konfiguration wird dann das fertige Programm erzeugt. Generiert diese nun ein Material das zum Setzen der Deferred Shading Texturen genutzt werden soll, so wird das erzeugte GLSL-Programm wie in Listing 4.2 aussehen.

4.2.4 Material-Cache

Um nicht bei jedem Neuzeichnen der Szene die genutzten Material-Shader neu generieren zu müssen, werden diese zwischengespeichert.

Dieser Zwischenspeicher ist als HashMap angelegt. GLSL-Programme werden unter einem Schlüssel abgelegt, welcher aus der zugehörigen Konfiguration und dem zugrundeliegenden GroIMP-Shader erzeugt wird. Hierbei werden Shader, welche für Lichtquellen genutzt werden zusammen mit solchen, die Materialien darstellen in der gleichen HashMap gesichert. Wird ein neuer GLSL-Shader während eines Zeichenschrittes erzeugt, werden, nachdem der Zeichenvorgang beendet wurde alle ungenutzten Shader entfernt. Bisher wird für den GroIMP-Shader zur Schlüsselgenerierung nur die Java-interne `hashCode()` Methode der Klasse Object genutzt, welche typischerweise die interne Objekt-Adresse verwendet[13]. Somit werden für zwei unterschiedliche GroIMP-Shader, welche die selben Attribute besitzen, zwei gleiche GLSL-Programme erzeugt und unter unterschiedlichen Schlüsseln abgelegt. Dies geschieht, da die Erzeugung eines „tiefen" Schlüssels, basierend auf den Attributen des Objektes, von der Tiefe des Shaderbaumes abhängt. Um die benötigte Zeit zum Nachschlagen in der HashMap minimal zu halten ist der zusätzliche Speicheraufwand zu vernachlässigen.

4.3 Deferred Shading aus Implementierungssicht

Ein großer Vorteil des Deferred Shadings in der Berechnung von Szenen mit einer großen Zahl an Lichtquellen besteht darin, dass diese oft nur in einem lokalen Bereich Einfluss auf die Beleuchtung nehmen. So können wie von [16] beschrieben Hüllkörper für Lichtquellen genutzt werden, um die Zahl der Fragmente zu reduzieren, die von einer Lichtquelle beleuchtet werden müssen. Zu beachten ist, dass HDR-Rendering diese Optimierungen, wie sie beim Deferred Shading gängig sind erschwert. Es lässt sich kein Entfernung angeben, ab der eine Lichtquelle zu vernachlässigen ist. Deshalb können Punktlichtquellen nicht durch eine Kugel begrenzt werden. Für Scheinwerferlichtquellen ließe sich ein Kegel mit unendlicher Länge als Hüllkörper nutzen. Da allerdings die Lichtberechnung nur einen unwesentlichen Teil der gesamten Renderzeit ausmacht (siehe Messungen in Kapitel 5.2) wurde auf diese Optimierung verzichtet.

[13] http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Object.html#hashCode() (Stand: März 2010)

Listing 4.2: Checker2D in GLSL

```
1  #version 110
2  #extension GL_ARB_draw_buffers : enable
3
4  varying vec2 uv;
5  varying vec3 normal;
6  varying float depth;
7  varying vec4 pos;
8  varying vec3 n_pos;
9  varying vec3 g_pos;
10
11 // Die folgenden beiden Methoden werden noch besprochen //
12 float packToFloat(vec2 val) {
13     val = clamp(val, 0.0, 1.0);
14     val.x = floor(val.x * 127.5) * 0.125;
15     vec2 tmp = vec2(floor(val.x), fract(val.x));
16     return (tmp.y + ( floor(val.y * 127.5) * 0.0009765625) + (1.0)) * exp2(tmp.x);
17 }
18
19 vec2 encodeNormal(vec3 normal){
20     float f = sqrt(8.0*normal.z+8.0);
21     return normal.xy / f + 0.5;
22 }
23
24 void main() {
25     vec2 tmp0 = vec2(4.0 * (uv).s + 0.0 * (uv).t + 0.0,
26                      0.0 * (uv).s + 2.0 * (uv).t + 0.0);
27
28     vec3 n_normal = normalize(normal);
29     if (!gl_FrontFacing) n_normal *= -1.0;
30     vec3 emissive = vec3(0.0);
31     vec3 diff_transp = vec3(0.0);
32
33     gl_FragData[0] = vec4(encodeNormal(n_normal), 4.0, 0.0);
34     gl_FragData[1] = vec4((mod(floor(2.0*(tmp0).s) +
35         floor(2.0*(tmp0).t),2.0) == 1.0 ?
36         vec3(1.0,1.0,1.0) : vec3 (0.0,0.0,0.0)), packToFloat(emissive.rg));
37     gl_FragData[2] = vec4(vec3(0.5), packToFloat(vec2(emissive.b, diff_transp.r)));
38     gl_FragData[3] = vec4(vec3(0.0), packToFloat(diff_transp.gb));
39 }
```

Wie bereits erarbeitet müssen neben Position und Normale noch 17 Werte in den verbleibenden 14 Feldern der Deferred Shading Texturen gesichert werden. Hierfür wird das folgende Verfahren genutzt.

4.3.1 „Packing“ von Byte-Werten in den Datentyp HalfFloat

Um den Speicherplatz und die benötigte Bandbreite für die Berechnung der Deferred Shading Texturen zu reduzieren, werden Texturen vom Typ *GL_RGBA16F_ARB* genutzt. Diese beinhalten vier Werte vom Typ *GL_HALF_FLOAT_ARB*. Die Informationen sind aus der Spezifikation der Erweiterung `ARB_half_float_pixel` [4] entnommen.

Im Folgenden soll ein Verfahren beschrieben werden, um zwei Werte in einem Wert des Typs HalfFloat zu sichern und diese wieder auszulesen. Hierfür ist es wichtig zu wissen, wie Zahlen in HalfFloat dargestellt werden.

Der HalfFloat-Datentyp ist 2 Byte lang. Er orientiert sich am gängigen Float-Datentyp[14], stellt allerdings nur 5 Bit für den Exponenten sowie 10 Bit für die Mantisse zu Verfügung. Desweiteren wird 1 Bit als Vorzeichenbit genutzt. In der normalisierten Darstellung errechnet sich der Wert einer HalfFloat-Zahl N als

$$(-1)^S \cdot 2^{(E-15)} \cdot \left(1 + M/2^{10}\right)$$

mit

$$\begin{aligned} S &:= \lfloor (N\%65536)/32768 \rfloor \\ E &:= \lfloor (N\%32768)/1024 \rfloor \\ M &:= N\%1024 \end{aligned}$$

wobei N als Integerwert interpretiert wird und „%“ für den in der Informatik gängigen modulo-Operator steht. Zu beachten sind die im Standard definierten Sonderfälle. Hierzu schreibt die ARB_EXTENSION:

> „4. Should the special representations NaN, INF, and denormal be supported? RESOLVED: Implementation dependent. The spec reflects that Nan and INF produce unspecified results. Denormalized numbers can be treated as a value of 0.“ [15]

Somit muss verhindert werden, einen der angesprochenen Sonderfälle durch Rechenoperation zu erreichen. Die 16 Bit lassen sich auf der CPU leicht über Bitoperatoren direkt auslesen und beschreiben, aber unter GLSL in der Version 1.1 werden Bitoperationen nicht unterstützt.

Ein einfaches Verfahren um zwei 5-Bit-Werte (in_1 und m_2) zu sichern, ist es diese durch entsprechende Multiplikationen in die 10 Bits der Mantisse zu schreiben und durch Addition des Wertes 1.0f die normalisierte Darstellung zu erzwingen: $out = in_1 \cdot 2^{-5} + in_2 \cdot 2^{-10} + 1$. Somit bleiben allerdings die 5 Bit des Exponenten ungenutzt.

Eine mögliche Lösung ist es, einen der Werte vollständig in der Mantisse zu kodieren und den anderen auf Exponent und die übrigen Bit der Mantisse aufzuteilen:

Kodieren Die GPU stellt in GLSL-Programmen Byte-Werte auf einen Zahlenbereich von $[0, 1]$ normalisiert dar, weshalb die tatsächliche Kodierung zwei Zahlen des Datentypes Float kodiert. Beide Zahlen werden um das niederwertigste Bit reduziert. Nun werden die 7 Bit der zweiten Zahl in den hinteren 7 Bit der Mantisse gesichert. Die erste Zahl wird in Ihre höherwertigen 4 Bit ($high(in_1)$) und die niederwertigen 3 Bit ($low(in_2)$) zerlegt. Die 3 Bit werden durch Addition in den übrigen 3 Bit der Mantisse abgelegt. Um den Exponenten zu sichern, wird zu der Mantisse

[14] nach IEEE 754-1985 nachzulesen in [24]
[15] OpenGL ARB-Erweiterung `ARB_half_float_pixel`[4, Issues 4]

Listing 4.3: Komprimieren von zwei Byte-Werten in einen HalfFloat

```
1  float packToFloat( vec2 inVal ) {
2    inVal.x = floor( inVal.x * 128. ) * 0.0078125 * 16.;
3    vec2 tmp = vec2( floor( inVal.x ), fract( inVal.x ) );
4    return ( tmp.y + ( inVal.y * 0.125 ) + 1.0 ) * exp2 ( tmp.x );
5  }
6
7  vec2 unpackFromFloat( float inVal ) {
8    float exp = floor( log2( inVal ) );
9    inVal = ( inVal * ( 1.0 / exp2( exp ) ) - 1.0 ) * 8.0;
10   return vec2(
11     floor( inVal ) * 0.0078125 + exp * 0.0625 ,
12     fract( inVal )
13   );
14 }
```

Tabelle 4.2: Aufteilung der Phong-Parameter für das Deferred Shading.

Textur	R	G	B	A	
1	Normal.xy		Shininess	TranspShininess	
2	Diff.rgb			Em.rg	
3	Spec.rgb			Em.b	DiffTransp.r
4	Transp.rgb			DiffTransp.gb	

1.0f addiert und das Ergebnis mit $2^{high(in_1)}$ multipliziert. Für $in_1 = 1 \vee in_2 = 1$ wird die kodierte Zahl einen der genannten Sonderfälle erreichen. Um dies zu verhindern, werden die Wertebereiche der beiden Zahlen leicht verschoben: $\tilde{in}_1 := in_1 * 255/256$ $\tilde{in}_2 := in_2 * 255/256$. Dies hat nur minimale Auswirkung auf die Genauigkeit der Rechnung.

Dekodieren Um die Komponenten eines gepackten HalfFloats (in) zu dekodieren ist ohne Bitoperatoren die Berechnung eines Logarithmus zur Basis 2 nötig. Entpackt wird nach folgendem Schema: Bestimme Exponenten $exp := \lfloor \log_2(in) \rfloor$ und den zuvor genutzten Multiplikator $mul := 2^{-exp}$. Nun kann die eingegebene HalfFloat-Zahl auf die Mantisse reduziert werden: $\tilde{in} := (in * 2^{-exp} - 1) * 8$. Die Multiplikation mit 8 schiebt die niederwertigen 3 Bit von $\tilde{in}_1$ vor das Komma. Somit erhalte beide Zahlen als $\tilde{out}_1 := \lfloor \tilde{in} \rfloor / 128 + exp/16$ $\tilde{out}_2 := \text{fract}(\tilde{in})$. Aus diesen können nun die Werte für out_1 und out_2 rekonstruiert werden, indem diese mit $\frac{256}{255}$ multipliziert werden.

Algorithmus 4.3 zeigt die Implementierung des Kodierungs- und Dekodierungsschemas in GLSL-Code.

4.3.2 Verteilung der Parameter auf Texturen

Somit können die Parameter wie in der Tabelle 4.2 auf vier 16-Bit-Float-Texturen aufgeteilt. Hierbei werden die eher selten genutzten Parameter „Emissive Color“ und „Diffuse Transparency“ über das zuvor beschriebene Verfahren komprimiert.

4.4 Zwischenspeicher für zeichenbare Objekte

Der in GroIMP implementierter Szenegraph beinhaltet viele zeichenirrelevante Knoten. Somit ist es Hilfreich den Graphen einmal vollständig zu traversieren und alle relevanten Objekte in einer dafür ausgelegten Datenstruktur zu sichern. Hierbei müssen alle zum Zeichnen benötigten Informationen mitgespeichert werden.

Der so erzeugte Cache muss synchron zum Szenegraphen gehalten werden. Eine Möglichkeit besteht darin, Änderungen an Objekten ohne erneute Traversierung des Graphens durch direkte Änderungen an den gecachten Objekten umzusetzen. GroIMP besitzt hierfür ein Eventsystem, das bei Änderungen passende Nachrichten erzeugt. Problematisch sind mit XL beschriebene Modelle. In diesen können zeichenbare Objekte während der Graphtraversierung instanziiert werden. Diese Objekte besitzen keine eigenen Knoten im GroIMP-Graphen und können somit nicht eindeutig referenziert werden. Das vorhandene Eventsystem gibt hier nur unzureichende Informationen, so dass der Graph neu traversiert werden muss. Inkrementelle Änderungen an einem bestehenden Cache sind also nicht durchführbar.

In der Implementierung wird sich der folgenden Lösung bedient. Bei Änderungen am Szenegraphen identifiziert durch eine fortlaufend, bei Änderungen inkrementierte Zahl, wird der aktuelle Cache verworfen. Dann wird der gesamte Szenegraph neu traversiert. Wird ein zeichenbares Objekte gefunden, identifiziert durch den Parameter Shape, welcher die Schnittstelle Renderable implementiert, so wird ein Abbild erzeugt, das alle relevanten Informationen sichert:

- Weltkoordinaten des Objektes
- GroIMP-Material
- GLSL-Material
- zeichenbare Form (entweder eine Referenz oder eine Kopie bei instanziierten Formen)
- Hüllgeometrie

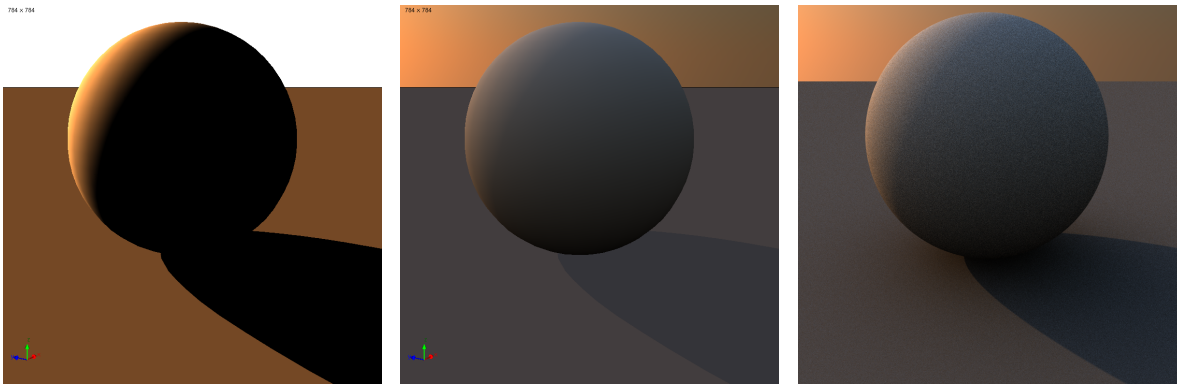
Die gesicherten Objekte werden in unterschiedlichen Vektoren abgelegt. Diese sind sortiert nach soliden Materialien, transparenten Materialien, Hilflinien und Tools sowie Objekte die einen Schriftzug beinhalten. Ändert sich der Graph nicht, so existieren alle Informationen, um die Szene zu zeichnen. Nach dem Sichern des Szenegraphens werden die Elemente der soliden und transparenten Objekte nach dem Index ihrer GLSL-Materialien sortiert, um die Zahl der Shaderwechsel zu minimieren.

4.5 Ausnahmen in der Darstellung zeichenbarer Objekte

Neben den in OGL1 implementierten Zeichenmethoden, die zu jedem zeichenbaren Objekt eine Liste von Vertizes an die Grafikkarte senden, gibt es auch solche Objekte, die nur schlecht über Polygone gezeichnet werden können. Für die folgende Typen wurde deren implizite Darstellung der Geometrie genutzt um diese genauer zeichnen zu können.

4.5.1 Ebenen

Für Ebenen wäre es theoretisch möglich ein sehr großes Quadrat aufzuspannen. Grafikkarten interpolieren aber bei sehr weit entfernten Vertizes, aufgrund ihrer begrenzten Rechengenauigkeit, Parameter wie Textur-Koordinaten und Entfernung im Tiefenpuffer nur sehr ungenau. Hier wird auf eine implizite Darstellung der Ebenen zurückgegriffen. Einem GLSL-Programm wird ein Punkt der Ebene sowie die Normale übergeben. In einem Vollbild-Shader wird für jedes Fragment ein virtueller Schnittpunkt zwischen der Ebene und einem Strahl der im Augpunkt beginnt und durch die Bildkoordinaten geht berechnet. Aus diesem werden benötigte Parameter wie die Position und die UV-Koordinaten bestimmt.



(a) Direkte Beleuchtung der Sonne (b) Direkte Beleuchtung der Sonne, sowie des Himmels (c) Twilight (PTR) ohne diffuse Reflektion

Abbildung 4.2: Beleuchtung einer Kugel mittels SunSkyLight

4.5.2 Sky

Für Knoten im Graphen, die den Himmel repräsentieren wird der gleiche Ansatz verfolgt. Für eine runde Wölbung der Kuppel würde eine sehr hohe Anzahl an Polygonen benötigt werden. Alternativ lässt sich die Programmable Pipeline nutzen. In einem Vollbild-Shader wird für jedes zu zeichnende Fragment der Schnittpunkt der Sichtlinie mit einer unendlich ausgedehnten Kugel, die die Szene umgibt, berechnet. Für den Schnittpunkt werden dann die getroffene Position auf der Kugel, sowie dazu passende UV-Koordinaten berechnet.

Beide Körper müssen durch ein GLSL-Programm simuliert werden, dass gleichzeitig die Parameter der zugehörigen Materialen bestimmt. Dies wird über eigene Konfigurationen (siehe 4.2.2) für Ebenen und SkyNodes erreicht.

4.6 Beleuchtung durch SunSkyLight

Die Implementierung von Punkt-, Kegel- und direktionalen Lichtquellen lässt sich unproblematisch aus der Literatur entnehmen. So erläutert beispielsweise [28] die Programmierung Phong-basierter Lichtquellen für alle drei Typen. Interessanter ist jedoch der Typ SunSkyLight.

Als Lichtquelle wird SunSkyLight zunächst durch eine direktionale Lichtquelle simuliert, welche aus Richtung der Sonne auf die Szene scheint. Für die Farbe wird der Farbwert des SunSky-Shader im Punkt der Sonne berechnet. Insbesondere die Farben in der Szene werden stark verfälscht, da beispielsweise die Blautöne des Himmels nicht zu der Beleuchtung beisteuern. Der Unterschied wird in Abbildung 4.2 deutlich.

Um die Beleuchtung des Himmels besser darstellen zu können, wird eine nicht-physikalische Approximation der Beleuchtung über Textur-Würfel realisiert. Zunächst wird der Himmel direkt in einen Texturwürfel gezeichnet. Dieser dient als Referenz für perfekte spekulare Reflektion, also für Materialien mit sehr großen spekularen Exponenten. Aus dem ersten Würfel wird ein weiterer Würfel erzeugt, welcher die rein diffuse Reflektion beschreibt. Hierzu wird für jeden Pixel zunächst die Blickrichtung auf den Pixel aus dem Würfelinneren bestimmt. Wie in Abbildung 4.3 gezeigt, wird dann die diffuse Reflektion der Umgebung auf einer Oberfläche deren Normale parallel zur Blickrichtung steht berechnet. Um die Szene mittels der beiden Texturwürfel zu beleuchten wird für jedes Fragment zunächst die Blickrichtung in Weltkoordinaten an der Oberfläche gespiegelt. Über diesen Vektor lässt sich nun der spekular reflektierte Anteil der Umgebung aus dem ersten Würfel gewinnen. Für den diffusen Anteil wird die Oberflächennormale

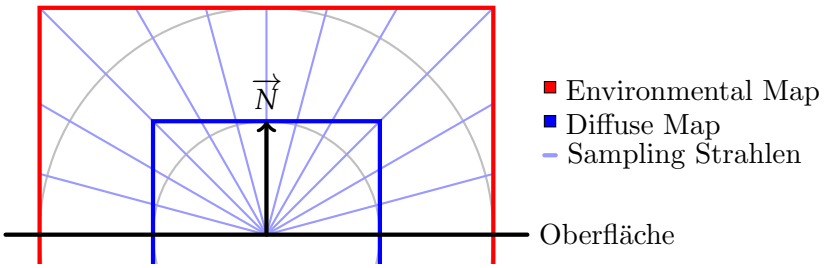


Abbildung 4.3: Berechnen des diffusen Farbanteiles

des Fragmentes in Weltkoordinaten umgerechnet und zum nachschlagen der diffusen Reflektion im zweiten Texturwürfel genutzt.

Dieses Verfahren ist auch nur eine sehr einfache Annäherung an bildbasierte Beleuchtung. Die Referenzwürfel werden für die gesamte Szene nur einmal vorberechnet und beim Beleuchten jedes Objektes genutzt. Somit werden insbesondere Hindernisse, die den Himmel verdecken nicht in die Beleuchtung mit einbezogen. Bereits 1986 erläuterte Ned Greene in [13] die beschriebene Idee anhand von virtuellen Texturwürfeln. Das Anzeigen der rein spekularen Reflektion wird auch häufig als „Cube Environmental Mapping" bezeichnet.

4.7 Depth Peeling aus Implementierungssicht

In diesem Abschnit wird eine Depth Peeling Variante beschrieben, die zusammen mit Deferred Shading genutzt werden kann. Im Extraktionsschritt des Depth Peelings werden zunächst nur die Tiefeninformation der Ebene geschrieben und in einen Renderpuffer gesichert. Die Ebene wird beleuchtet, in den HDR-Puffer geschrieben (mittels Underblending) und dann wird der Inhalt des Tiefenpuffers in eine Tiefentextur kopiert. Somit kann dieser als Eingabe für den nächsten Extraktionsschritt dienen. Dies muss geschehen, da zum Zeitpunkt der Implementierung auf den vorhandenen Testsystemen der Stencil-Puffer nur im Zusammenhang mit einem Tiefenpuffer genutzt werden kann. Wird eine Textur direkt als Tiefenpuffer gebunden, so steht der Stencil-Puffer nicht zu Verfügung. Theoretisch existiert das Renderpufferformat `GL_STENCIL_INDEX8_EXT`. Dieses wird aber auf keiner der getesteten Grafikkarten[16] in Kombination mit Tiefentexturen unterstützt. Der Kopiervorgang kann über den OpenGL-Befehl `glCopyPixels` direkt auf der Grafikkarte durchgeführt werden.

Der in Algorithmus 4.1 beschriebene Pseudocode zeigt, dass während des Depth Peelings nur potentiell transparente Objekte gezeichnet werden. Objekte deren Material als solide klassifiziert wurde werden in einem vorbereitenden Schritt genutzt, um eine "hintere" Grenze für das Depth Peeling festzuhalten. Diese Grenze wird zusätzlich überprüft. Für den ersten Extraktionsschritt existiert keine nähere Ebene, weshalb das auslesen und vergleichen der Textur b' gespart werden kann.

Um das Depth Peeling zu beenden, falls keine weitere Ebene extrahiert werden kann, bedient sich das beschriebene Verfahren der in OpenGL verfügbaren "Occlusion Queries"[17]. Diese können eingesetzt werden, um die Zahl der in einem Zeitraum geschriebenen Fragmente zu überprüfen. Die Anfrage wird für den Extraktionsschritt formuliert. Die Beleuchtung der Szene gibt OpenGL genug Zeit um das Ergebnis in den Hauptspeicher zu laden. Zu Beginn des nächsten Extraktionsschrittes kann dann die Zahl der zuvor extrahierten Pixel ausgelesen werden. Wird festgestellt, dass diese einen festgelegten Schwellenwert unterschreitet, so werden die folgenden Extraktionsschritte aufgrund der Extraktionsrichtung maximal genausoviele Fragmente schreiben. Somit lässt sich über den Schwellenwert ein frühzeitig Abbruch des Depth Peelings erreichen.

[16] NVIDIA GeForce 7600, NVIDIA GeForce 8600M GT, ATI Radeon HD 3760 X2, Intel GMA 4500MHD
[17] http://oss.sgi.com/projects/ogl-sample/registry/ARB/occlusion_query.txt (Stand März 2010)

Algorithmus 4.1 Pseudocode des genutzten Depth Peeling Verfahrens

```
1: Setze Tiefentest auf LESS
2: Initialisiere Tiefenpuffer a und b
3: Initialisiere Tiefentexturen a' und b'
4: Rendere z-Werte der soliden Geometrie in Puffer a
5: Kopiere Inhalt von a nach a'
6: Binde b als aktuellen Tiefenpuffer
7: loop
8:    Lösche Inhalt von b
9:    Zeichne transparente Objekte mit z-Test auf LESS
10:   for all Fragment ∈ Szene do
11:      Lade Vergleichswert z_front aus Textur b'
12:      Lade Vergleichswert z_back aus Textur a'
13:      z_cur ← z-Wert des aktuellen Fragmentes
14:      if z_front < z_old < z_back then
15:         Zeichne Fragment in den Tiefenpuffer
16:      else
17:         Verwerfe aktuelles Fragment
18:      end if
19:   end for
20:   Zeichne mit z-Test auf EQUAL die Szene erneut, fülle diesmal Deferred Shading Texturen
21:   Beleuchte die so extrahierte Ebene
22:   if kein Fragment gezeichnet then
23:      return
24:   end if
25:   Kopiere Inhalt von b nach b'
26: end loop
```

4.8 Extrahieren der Tonemapping-Parameter

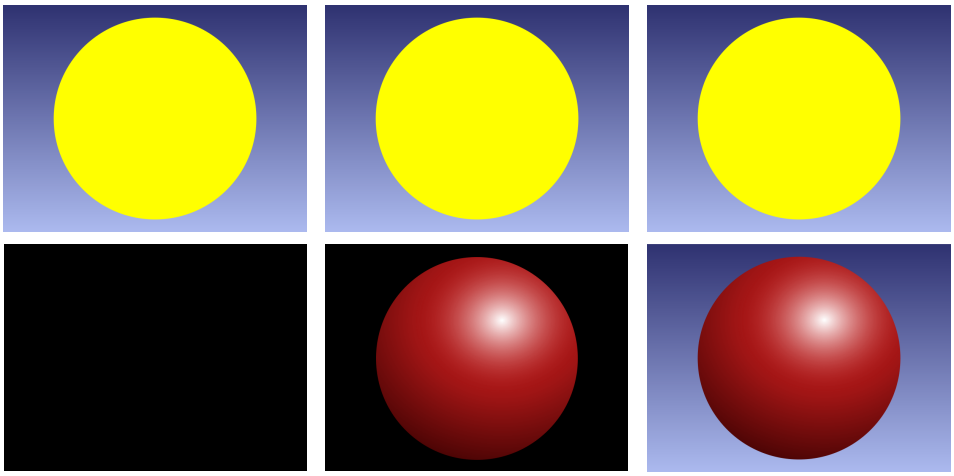
Um die, für das Tonemapping benötigten, globalen Bildparameter zu erhalten, dauert das Zurücklesen des Bildes in den Arbeitsspeicher zu lange. Eine effiziente Methode zur Ermittlung der Parameter wurde bereits im Abschnitt 3.2.6 erläutert. Nach dem Tonemapping wird der Inhalt der Deferred Shading Texturen nicht mehr benötigt, weshalb die beschriebene Reduktion in diesen Texturen stattfinden kann. Da auch hier wieder Textur-Ping-Pong betrieberen werden muss, sind zwei Texturen verfügbar, in denen beliebige Werte gesichert werden können. Diese sind die maximale Farbintensität je Farbkanal, die maximale Luminanz, die mittlere Luminanz, sowie die mittlere Farbintensität je Kanal.

Für die bisher Implementierten Tonemapping Operatoren genügen diese Parameter. Sollten weitere Parameter benötigt werden, so kann, in Abhängigkeit vom gewählten Tonemapping Verfahren, der Reduktionsschritt angepasst werden, um diese zu extrahieren.

4.9 Stencil Setup

In der Implementierung wird der Stencilpuffer der GPU genutzt, um die Nutzung aufwendiger GLSL-Programme auf relevante Bildbereiche zu beschränken. Hierbei werden folgende Identifikationsbits genutzt:

Bit0: Ist dieses gesetzt, so stellt der Bildpunkt eine zu beleuchtende Oberfläche dar. Desweiteren kennzeichnet dieses Bit das Vorhandensein eines soliden Objektes.



(a) (b) (c)

Abbildung 4.4: Konflikt zwischen Textur-Ping-Pong und Stencil-Test. Vor der Beleuchtung der Kugel (4.4a) *ist die Zieltextur schwarz. Nachdem die Kugel beleuchtet wurde ist diese in die Zieltextur geschrieben* (4.4b). *Der blaue Hintergrund des Bildes wurde nicht beleuchtet und demnach nicht in die Zieltextur kopiert. Das nachträgliche Kopieren stellt sicher, dass die aktuelle Textur den gesamten Inhalt des Bildes enthält* (4.4c).

Bit1: Ist dieses gesetzt, so stellt der Bildpunkt die Oberfläche eines Körpers dar, der erst nach dem Tonemapping in die Szene eingefügt wird. Diese werden nicht beleuchtet, sondern in einem festen Farbton gezeichnet.

Sind beide Bits nicht gesetzt, so ist der Bereich des Bildes leer und muss im entsprechenden Zeichenschritt durch einen Himmel oder einen anderen Hintergrund aufgefüllt werden. Durch den Stencil-Tests wird sichergestellt, dass nur Fragmente, die Objekte der Szene beschreiben beleuchtet werden. Für das Beleuchten einzelner z-Ebenen (siehe Abschnitt 4.7) ist hierbei, wie im Abschnitt 2.4.1 beschrieben, zu beachten, dass bei Ping-Pong Techniken dennoch der gesamte Texturinhalt von der Quelltextur in die Zieltextur übertragen werden muss. Hierzu wird der Stencil-Test so eingestellt, dass alle zuvor unveränderten Fragmente in einem zweiten Zeichenschritt in die Zieltextur kopiert werden. Der Vorgang ist in Abbildung 4.4 illustriert. Von dem genannten Effekt sind alle Zeichenschritte betroffen, die nur einen Teil der HDR-Texturen modifizieren, wie die Beleuchtung und das Zeichnen der Hilflinien. Beim Tonemapping wird das gesamte Bild verändert, weshalb das nachträgliche Kopieren des Bildes entfällt.

5 Bewertung

Dieses Kapitel vergleicht die in dieser Arbeit entstandene Implementierung, im Folgenden Proteus (Procedural Texture System) genannt, mit der vorhandenen OpenGL Implementierung (OGL1) sowie dem Standard Raytracer von Twilight. Es werden zunächst Bildergebnisse der einzelnen Darstellungen gegenübergestellt. Dann werden quantitative Messungen wie Geschwindigkeit und Speicherverbrauch durchgeführt. Im Anschluss werden bestehende Probleme der aktuellen Implementierung diskutiert.

Testsysteme Für die Messungen wurden die Systeme aus der Tabelle 5.1 genutzt. Proteus wurde hierbei auf System A entwickelt.

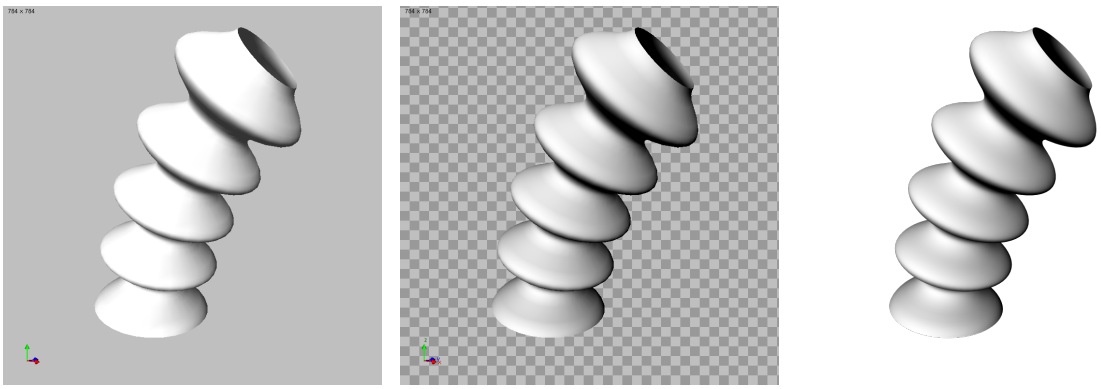
Tabelle 5.1: Testsysteme

Bauteil	System A	System B
CPU	2.4 GHz Intel Core 2 Duo	3.16 GHz Intel Core 2 Duo
Grafikkarte	GeForce 8600M GT	ATI Radeon HD 3870 X2
RAM	4 GB RAM	4 GB RAM
OS	OSX 10.6	Windows Vista Home Premium (32-Bit)

5.1 Bildvergleiche

Zunächst sollen anhand einiger Bildvergleiche die Ergebnisse dieser Arbeit präsentiert werden. Wenn nicht anders genannt beziehen sich die Bilder und Messergebnisse auf den Standard Raytracer von Twilight.

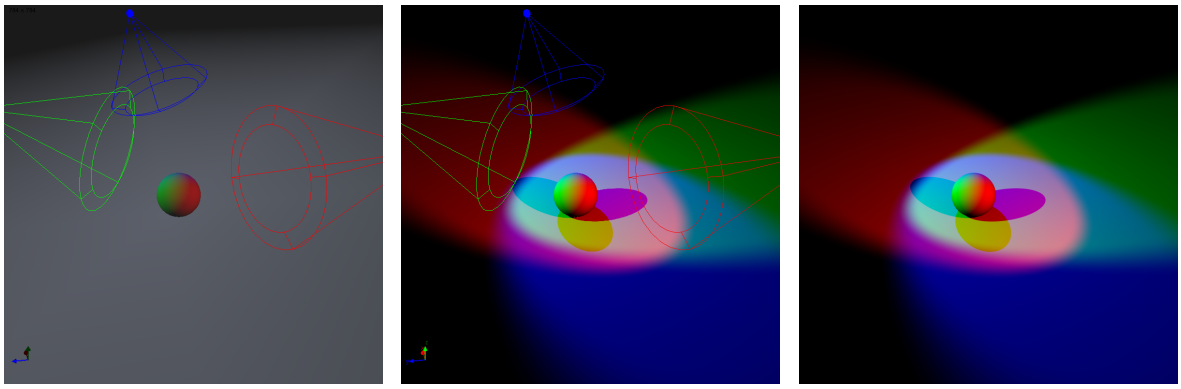
RGBA-Material (Abbildung 5.1) Das von Proteus erzeugte Bild kann aufgrund der Beleuchtung pro Pixel eine deutlich genauere Vorschau der Szene geben. Kombiniert mit der HDR-Bildberechnung erlaubt diese eine exakte Vorschau auf das Bildergebnis von Twilight.



(a) OGL1 (b) Proteus (c) Twilight

Abbildung 5.1: Darstellung einer Oberfläche aus der Beispielszene „NURBS Demo“.

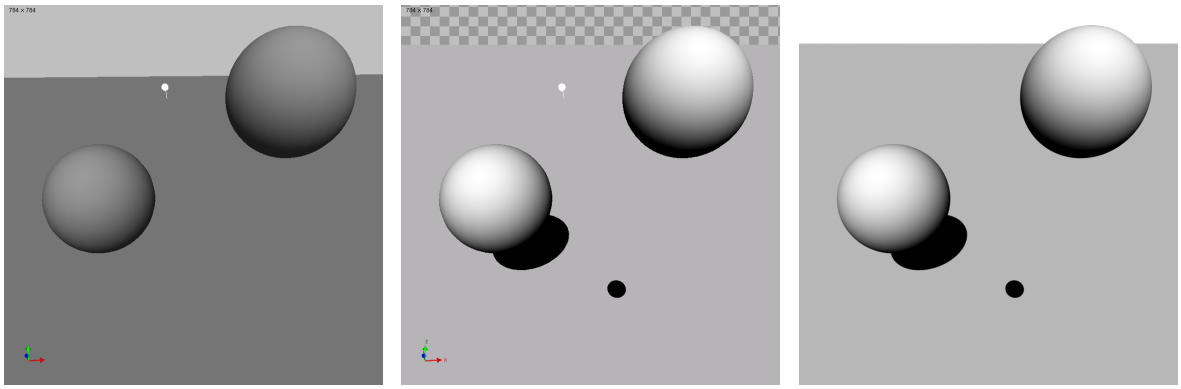
Beleuchtete Szene (Abbildung 5.2) Die Beleuchtung in Proteus spiegelt sichtbar genauer die Ausleuchtung der Szene wieder. So werden, wie in der Abbildung zu sehen, insbesondere Kegellichtquellen exakt wiedergegeben. In dieser Szene stellt OGL1 die Ebene mit einer geringen Zahl an Vertizes dar, weshalb das genutzte Gouraud Shading Beleuchtungsbeiträge der drei Lichtquellen auf der Ebene auf einen Grauton reduziert.



(a) OGL1 (b) Proteus (c) Twilight

Abbildung 5.2: Szene mit drei Scheinwerferlichtquellen.

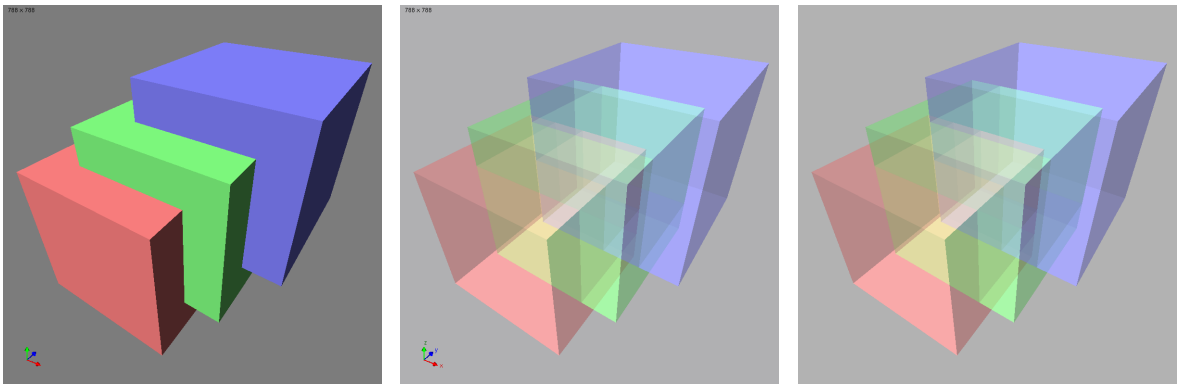
Schatten (Abbildung 5.3) Schatten erlauben eine bessere Wahrnehmung der räumlichen Verhältnisse innerhalb der Szene. Diese werden in Proteus nur unter optimalen Bedingungen annähend pixelgenau und bieten demnach nur eine schnelle Approximierung. Bei direktionalen Lichtquellen in einer Szene, deren Objekte eng beieinander liegen, sind die dargestellten Schatten eine gute Orientierungshilfe, wie die Abbildung zeigt.



(a) OGL1 (b) Proteus (c) Twilight

Abbildung 5.3: Durch eine direktionale Lichtquelle beleuchtete Szene.

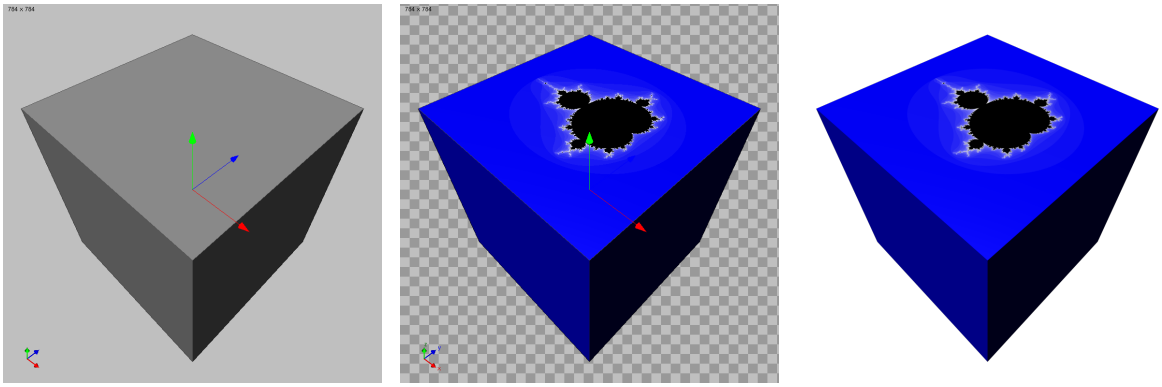
Transparenz (Abbildung 5.3) Über das implementierte Depth Peeling lassen sich transparente Materialien in vielen Fällen ohne optische Unterschiede zu Twilight in die Szene integrieren. Schatten von Objekten mit transparenten Materialien können allerdings nicht berechnet werden.



(a) OGL1 (b) Proteus (c) Twilight

Abbildung 5.4: Drei verschachtelte, lichtdurchlässige Würfel.

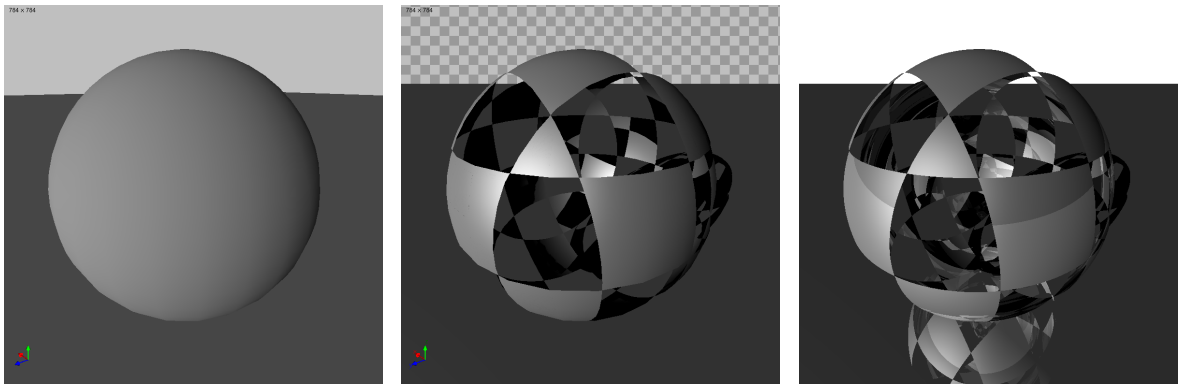
Prozedurale Texturen I (Abbildung 5.5) Die Hauptaufgabe der Arbeit bestand darin, prozedural berechnete Materialien in Echtzeit darzustellen. Proteus ist im Stande neben Oberflächentexturen, wie sie in OGL1 angezeigt werden, auch Volumentexturen zu berechnen und korrekt darzustellen, wie in der Abbildung zu sehen ist. Da das ChannelMap-System von GroIMP in GLSL-Code übersetzt wurde, kann eine detaillierte Reproduktion der GroIMP-Materialien angezeigt werden.



(a) OGL1 (b) Proteus (c) Twilight

Abbildung 5.5: Prozedural berechnete Darstellung der Mandelbrotmenge. In der Szene wird die ChannelMap Blend, mit der ChannelMap Mandel als Eingangskanal in der diffusen Farbkomponente genutzt. Die Texturkoordinaten werden über eine weitere ChannelMap (UVTransformation) so verändert, dass die Mandelbrotmenge mittig auf dem Würfel zu sehen ist.

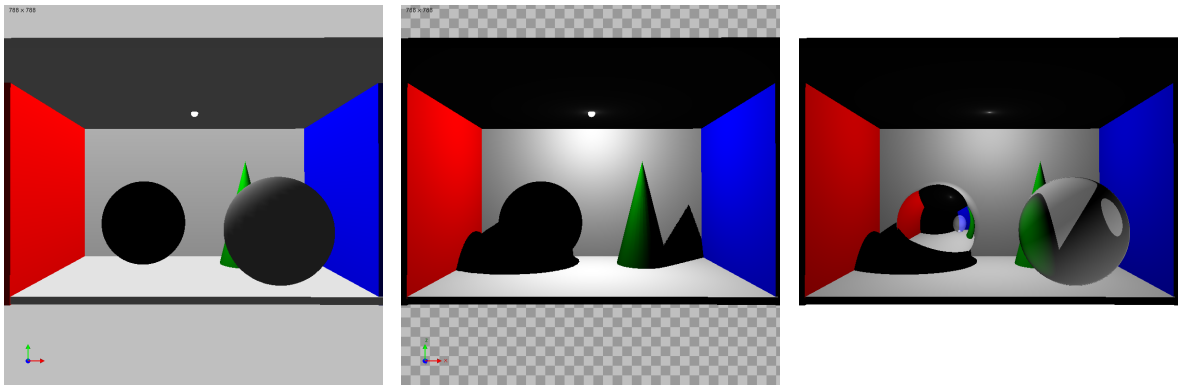
Prozedurale Texturen II (Abbildung 5.6) Neben der diffusen und spekularen Farbkomponente können auch andere Parameter prozedural berechnete Einträge haben. So zeigt die Abbildung eine halbtransparente Kugel, für deren Material die Lichtdurchlässigkeit über ein Schachbrettmuster bestimmt wird. Proteus zeichnet sowohl das Material als auch den entstehenden Schatten korrekt. Da die Fläche entweder vollständig transparent oder vollständig solide ist, erzielt der genutzte Alpha-Test das gewünschte Ergebnis. Die von Twilight dargestellten Reflektionen ließen sich nur für eine geringe Zahl an Objekten berechnen, weshalb Proteus diesen Effekt nicht umsetzt.



(a) OGL1 (b) Proteus (c) Twilight

Abbildung 5.6: Checkboard3D in der transparenten Komponente.

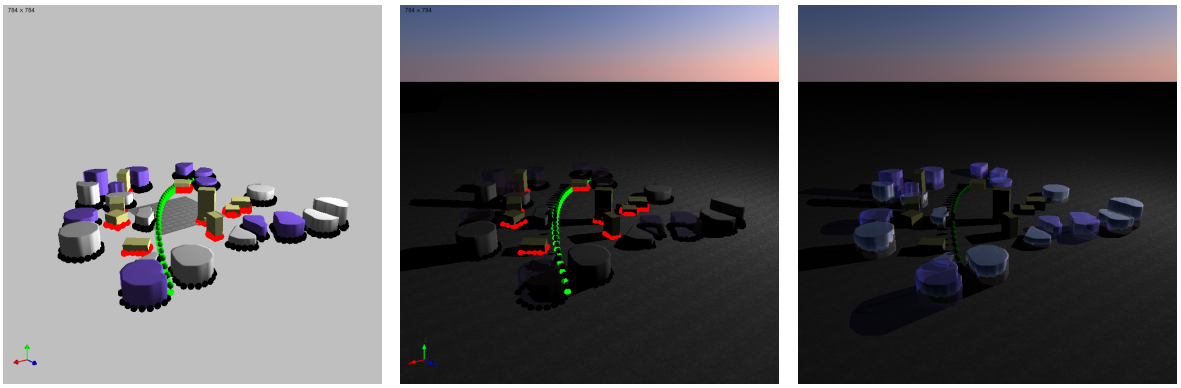
Reflektion und Brechung (Abbildung 5.7) Neben spekulare Reflektion kann Proteus Effekte wie die Lichtbrechung nicht darstellen. Die vollständig transparente Kugel rechts im Bild wird von Proteus nicht angezeigt. Für eine Vorschau könnte es hier wünschenswert sein, den Nutzer auf die vorhandene Kugel hinzuweisen. Die perfekt spiegelnde Kugel links im Bild wird wird entsprechend ihrer anderen Materialeigenschaften vollständig schwarz angezeigt.



(a) OGL1 (b) Proteus (c) Twilight

Abbildung 5.7: Beispielszene „Cornell Box".

Komplexe Szene (Abbildung 5.8) Die Beispieldatei "City" zeigt deutlich die unterschiedliche Darstellung in OGL1 und Proteus. Aber auch das von Twilight erzeugte Bild zeigt eine starke Abweichung von der in Proteus gezeichneten Szene. Es fehlen die Reflektionen der umgebenden Objekte und des Himmels. Zudem werden die Schatten transparente Objekte nur teilweise angezeigt. In dieser Szene ist die Beleuchtung des SunSkyLights deaktiviert und eine einzelne Punktlichtquelle außerhalb des Sichtbereiches leuchtet die Szene aus. Theoretisch könnte Proteus dennoch, wie in Abschnitt 4.6 erläutert, die „Environmental Map" berechnen und so die Reflektion des Himmels in die Szene einfügen.



(a) OGL1 (b) Proteus (c) Twilight

Abbildung 5.8: Beispielszene „City". Die Szene wird ausschließlich über das SunSkyLight-Material des Himmels beleuchtet.

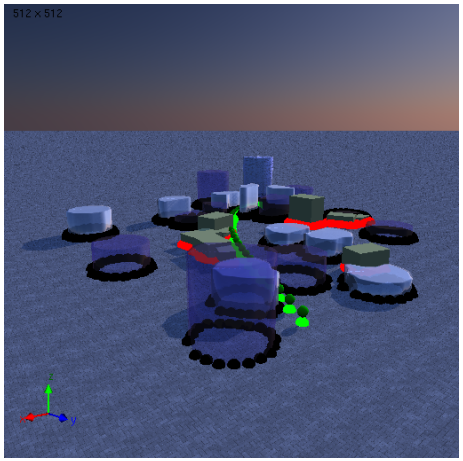
5.2 Messungen

In diesem Abschnitt sollen quantitativ die Darstellungen OGL1 und Proteus vergleichen werden. Der Schwerpunkt ist hierbei auf Speicher- sowie Zeitaufwand der Zeichenmethoden gelegt. Für die Geschwindigkeitsmessungen wurde eine Bildauflösung von 512x512 Pixeln genutzt. Es wurde jeweils über eine Messreihe gemittelt, bei der die Anzahl der Messungen in den jeweiligen Abschnitten aufgeführt wird. Da die Grafikpipeline von OpenGL in den meisten Implementierungen Befehle nicht direkt ausführt, sondern zunächst in eine Warteschlange einreiht, wurde vor jeder Messung die Methode `glFinish` genutzt. So wird OpenGL angewiesen alle verbleibenden Operationen durchzuführen. Die Ausführung wird fortgesetzt, sobald alle OpenGL-Befehle ausgeführt wurden. Eine direkt folgende Messung sollte demnach einen guten Eindruck über die tatsächliche Dauer der Befehle geben. Die Zeit selbst wurde Mittels des Java-Befehls `System.nanotime()` gemessen. Dieser Befehl gibt die Zeit in Nanosekunden seit dem Systemstart zurück. Die Messgenauigkeit liegt auf dem genutzten System bei einer Mikrosekunden. Es wurde sowohl vor, als auch nach dem Zeichnen der Szene die Zeit gestoppt und die Differenz als Messwert genommen. Um keine unerwünschten Effekte durch die Java oder jogl zu erhalten wurden vor der Messung jeweils 5 Zeichenvorgänge durchgeführt die nicht in die Wertung eingeflossen sind.

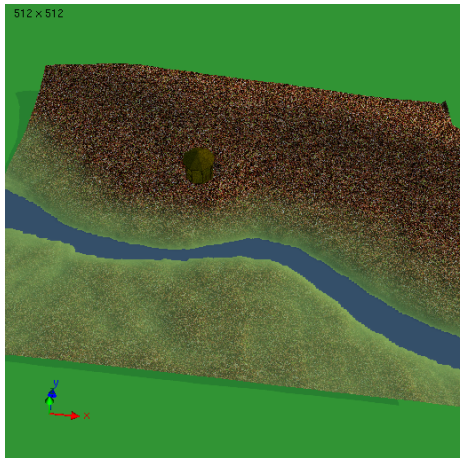
Für die Messungen wurden die Szenen der Abbildung 5.9 genutzt.

Zeichengeschwindigkeitsvergleiche Im ersten Vergleich sollen die OpenGL-Darstellung und Proteus in statischen und in dynamischen Szenen verglichen werden. Hierbei wird die Dauer eines Zeichenvorganges in Millisekunden über eine Messreihe von 1000 Messungen gemittelt. Es wird außerdem die minimale sowie die maximale Zeichendauer innerhalb der Messreihe angegeben. Eine statische Szene meint eine Szene, in der sich zwischen zwei gezeichneten Bildern der Szenegraph nicht ändert. Entsprechend sind dynamische Szene solche, in der durch fortlaufende Änderungen des Szenegraphens eine Animation erzeugt wird. Die letzte Spalte der Tabelle gibt eine Hochrechnung des Mittelwerts auf Bilder pro Sekunde (kurz BPS) an. Dieser Wert liegt deutlich über den tatsächlich erreichbaren Zeiten, da in GroIMP nur bei Bedarf das Bild aktualisiert wird. Desweiteren wird bei dynamischen Szenen zwischen zwei Zeichenvorgängen der GroIMP-Graph verändert, was einen zusätzlichen Zeitaufwand bedeutet, der in diese Messung nicht einfließt.

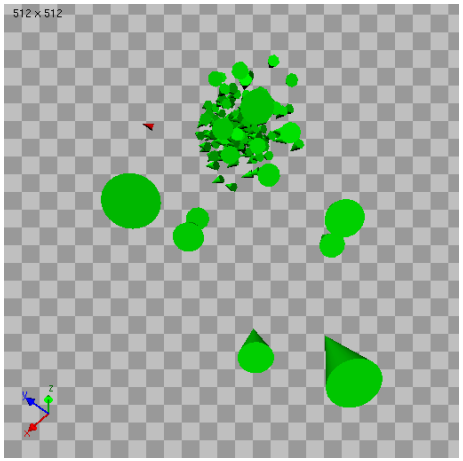
Wie in der Tabelle 5.2 zu sehen ist in Szenen mit wenigen Objekten OGL1 deutlich schneller. Dies lässt sich so erklären, dass Proteus neben der Darstellung einen deutlich höheren Verwaltungsaufwand besitzt. So wird beispielsweise auch in einfachen Szenen ohne explizite Lichtquellen



(a) City, beleuchtet durch SunSkyLight

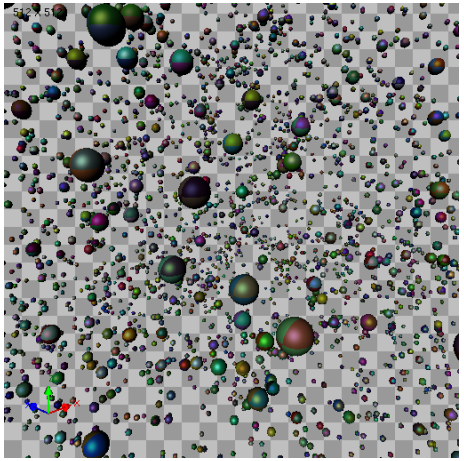


(b) FSPM

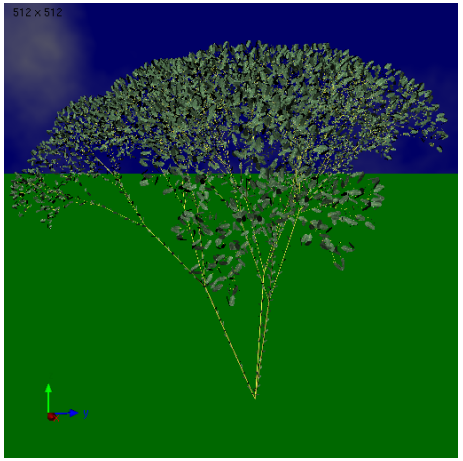


(c) Boids

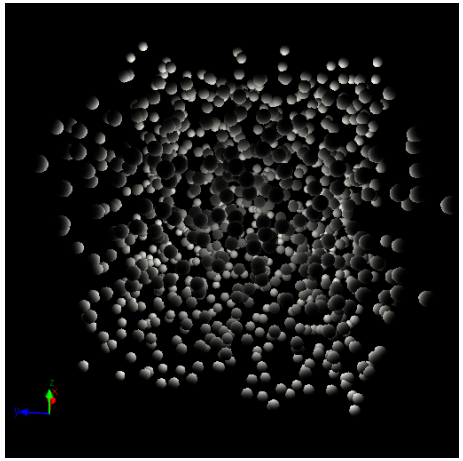
(d) Hubschrauber



(e) 4000 Kugeln



(f) Bush



(g) Licht

Abbildung 5.9: Übersicht der für die Messungen genutzten Beispielszenen.

Tabelle 5.2: Geschwindigkeitsmessungen. A und B stehen jeweils für die Systeme A und B. Knoten meint die Zahl der Knoten, die beim Traversieren des Szenegraphens betreten werden. Rend. gibt die Zahl der gezeichneten Objekte wieder. Boids meint die statische Boids-Szene, während Boids (dyn) die animierte Szene bezeichnet.

Szene		Ansicht	Mittel		Minimum		Maximum		BPS [in s^{-1}]	
Knoten	Rend.	—	A	B	A	B	A	B	A	B
City		OGL1	63,4	11,59	61,98	10,47	88,54	15,15	15,77	86,32
2573	1265	Proteus	68,24	38,13	66,62	34,66	548,21	45,25	14,66	26,23
FSPM		OGL1	8,41	6,15	6,19	4,77	38,3	10,56	118,01	162,66
479	129	Proteus	267,12	21,49	262,11	19,45	345,99	47,46	3,74	46,54
Boids		OGL1	2,79	2,81	2,48	2,54	23,05	9,96	358,42	355,53
307	151	Proteus	7,81	4,28	7,54	3,76	18,09	7,65	128,04	233,41
Boids (dyn)		OGL1	2,64	2,65	2,4	2,45	6,45	14,24	378,79	377,28
307	151	Proteus	7,92	4,46	7,29	3,64	24,99	13,52	126,26	224,13
Hubschrauber		OGL1	2,43	1,66	1,46	1,32	27,97	3,41	411,52	601,01
223	45	Proteus	18,76	5,29	14,48	3,83	36,43	7,93	53,3	188,99

Tabelle 5.3: Geschwindigkeitsmessungen. Jeder Test bestehend aus zwei umrahmten Zeilen beschreibt in der ersten Spalte die Zahl der Knoten im Graphen (oben) sowie die Zahl der gezeichneten Objekte (unten).

Knoten	Ansicht	Mittel		Minimum		Maximum		BPS [in s^{-1}]	
Rend.	—	A	B	A	B	A	B	A	B
51	OGL1	1,45	1,28	1,19	1,11	16,54	3,19	689,66	782,45
8	Proteus	8,76	2,95	7,67	2,78	25,57	6,71	114,16	339,06
183	OGL1	2,06	1,91	1,76	1,7	14,15	5,8	485,44	522,21
26	Proteus	14,32	3,04	13,12	2,71	31,7	7,5	69,83	328,76
603	OGL1	4,02	3,51	3,71	3,21	16,12	5,48	248,76	285,23
86	Proteus	15,46	4,02	14,42	3,54	27,74	8,01	64,68	248,62
1899	OGL1	9,18	8,85	8,58	8,36	28,75	31,51	108,92	113,03
272	Proteus	22,04	8,95	21,7	8,4	32,03	12,81	45,37	111,74
5847	OGL1	26,94	24,69	26,36	24,24	52,58	34,44	37,12	40,5
842	Proteus	33,69	19,21	33,19	18,76	53,22	21,88	26,69	52,05
17787	OGL1	71,57	73,8	70,74	72,33	80,81	86,2	13,97	13,55
2570	Proteus	42,16	51,6	41,66	50,78	56,83	66,46	23,72	19,38
53763	OGL1	206,84	222,54	204,98	219,18	216,98	241,02	4,83	4,49
7784	Proteus	109,91	151,83	108,99	147,81	124,56	545,29	9,1	6,59
161943	OGL1	619,25	673,81	611,94	664,07	759,04	1049,37	1,61	1,48
23474	Proteus	313,9	445,5	311,49	439,28	326,63	551,15	3,19	2,24

HDR Rendering betrieben. Der Vergleich in bewegten Szenen zeigt, dass OGL1 kaum von Änderungen am Graphen beeinflusst wird. Für Proteus ist ein leichter Anstieg der Zeichenzeit zu erkennen. Dies ist insbesondere auf die genutzten Caching-Strategie zurückzuführen, die in einer dynamischen Szene den vollständigen Szenegraphen abläuft. Dabei muss der Cache jedes mal erneut gefüllt werden. Die Szene FSPM nutzt ein komplexes Material für den Boden, welches aus mehreren überlagerten Texturen und Noise-Funktionen besteht. Ältere Grafikkarten, wie sie im System A verbaut sind, können Alternativen im GLSL-Code nicht effizient berechnen, weshalb die Geschwindigkeit deutlich langsamer als erwartet ausfällt.

Stress Test 1) „Bush“ Die nächste Messung basiert auf der Szene „Bush“ der Beispieldateien von GroIMP. In der Szene wird ein Busch mit einer großen Zahl an Blättern gezeichnet. Die Blätter werden über texturierte Parallelogramme gezeichnet. Den Stamm bilden mehrere Zylinder, denen ein RGBA-Material zugeordnet ist. In jeder Ableitung des Modells verdreifacht sich die Zahl der Knoten im Graphen.

Da die Szene statisch ist und eine große Zahl nicht-zeichenbarer Knoten im Graphen enthält, zeigen sich die Vorteile der genutzten Caching-Methode. Die in der Szene genutzten Materialien sind sehr einfach gehalten, weshalb hier ideale Voraussetzungen für Proteus geschaffen sind. Desweiteren ist gut zu erkennen, dass das Rendern mit OpenGL unter OSX deutlich schneller als unter Windows durchgeführt wird.

Stress Test 2) „4000 Kugeln“ In diesem Test werden OGL1 und Proteus auf einer Szene getestet, in der zunehmend die Zahl der zeichenbaren Objekte fest auf 4000 eingestellt ist. Diesmal ist der Graph bewusst klein gehalten, so dass nur zeichenbare Knoten existieren. Desweiteren werden viele verschiedene Materialien vom Typ Phong genutzt. Die Zahl der Unterschiedlichen Materialien wurde kontinuierlich erhöht. Es wurden jeweils Variationen des bereits bekannten Schachbrett-Materials erzeugt.

Tabelle 5.4: Geschwindigkeitsmessungen

#Material	Ansicht	Mittel		Minimum		Maximum		BPS [in s^{-1}]	
—	—	A	B	A	B	A	B	A	B
1	OGL1	87,09	45,28	85,25	42,91	121,28	55,54	11,48	22,09
	Proteus	55,79	18,52	55,06	17,1	67,93	207,2	17,92	54
10	OGL1	87,39	57,82	85,82	52,03	93,22	898,28	11,44	17,29
	Proteus	80,69	47,07	79,4	43,19	103,57	822,87	12,39	21,25
40	OGL1	88,85	61,86	87,22	51,13	97,2	955,95	11,26	16,16
	Proteus	81,43	51,15	80,35	48,53	107,56	570,51	12,28	189,55
80	OGL1	89,29	—	87,65	—	102,97	—	11,2	—
	Proteus	83,1	51,09	81,82	49,26	101,58	192,85	12,03	19,57
120	Proteus	85,48	60,61	83,94	58,46	107,01	171,14	11,7	16,5
1000	Proteus	114,92	—	113,92	—	125,83	—	8,7	—
2000	Proteus	162,24	—	160,61	—	178,63	—	6,16	—

Tabelle 5.5: Geschwindigkeitsmessungen der Szene „Licht“. Die Spalte #Lichter gibt an, wie viele Punktlichtquellen in der Szene berechnet werden. Für jede Lichtquelle wird außerdem genau eine Kugel gezeichnet.

#Lichter	Mittel		Minimum		Maximum		BPS [in s^{-1}]	
—	A	B	A	B	A	B	A	B
1	7,57	2,56	6,54	2,39	14,14	5,2	132,06	389,96
2	8,72	3,05	7,60	2,86	32,92	6,21	114,65	327,57
4	10,91	4	9,95	3,8	28,93	7,21	91,64	249,78
8	15,66	5,87	14,57	5,66	31,34	9,92	63,87	170,26
16	25,32	9,48	24,07	9,23	48,64	15,33	39,5	105,53
32	44,72	18,01	43,45	17,75	66,55	27,77	22,36	55,52
64	83,26	34,68	82,08	34,34	117,68	51,77	12,01	28,83
128	160,60	66,4	159,30	65,46	203,62	69,32	6,23	15,06
256	315,57	130,57	313,86	130,12	398,38	132,42	3,17	7,66
512	615,67	258,15	613,72	257,4	773,6	261,74	1,62	3,87
1024	1209,67	513,78	1205,96	512,75	1528,84	518,81	0,83	1,95

Insgesamt ist zu erkennen, das beide Darstellungen eine vergleichbare Zeichenzeit aufweisen. Ab etwa 100 verschiedenen Materialien hat OGL1 bereits den gesamten Texturspeicher von System A belegt, weshalb weitere Messungen nicht durchgeführt werden konnten. Bei System B war die Messung mit 80 verschiedenen Materialien unter OGL1 bereits nicht mehr möglich. Bei 1000 Materialien hat System B auch Probleme mit der Darstellung. (Die Berechnung dauert etwa 2 Sekunden pro Bild). Entsprechend der Messungen ist Proteus insbesondere für Szenen mit nicht-trivialen Materialien geeignet.

Stress Test 3) „Licht“ Die Szene „Licht“ erzeugt Punktlichtquellen an zufälligen Positionen. Für jede Lichtquelle wird der Schatten deaktiviert und eine Kugel gezeichnet, die diese umgibt. Die Szene eignet sich gut um die Geschwindigkeit des Deferred Shadings zu testen. Wieder wurde über je 1000 Messungen gemittelt. Die Zahl der Lichtquellen wurde nach jeder Messung verdoppelt. Damit Aufgrund des Stencil-Tests nicht nur Teile des Bildes beleuchtet werden wurde in den Hintergrund eine schwarze Ebene gesetzt. OGL1 unterstützt maximal acht Lichtquellen, weshalb auf einen Vergleich verzichtet wurde.

Zwar unterstützt Proteus prinzipiell eine beliebige Anzahl an Lichtquellen, allerdings reduzieren diese in großer Zahl die Zeichengeschwindigkeit beträchtlich. Für LDR-Beleuchtung müssen

Punktlichtquellen nur lokal ausgewertet werden. Bei der HDR-Beleuchtung ist dies nicht möglich, weshalb für jeden Pixel der Bildebene der Beitrag aller Lichtquellen berechnet werden muss. Es ist dennoch anzumerken, dass der Raytracer von Twilight auf dem System A für die selbe Szene mit 1024 Lichtquellen etwa 21,09 Sekunden rechnet[18].

Vergleich der Rechenzeit einzelner Zeichenschritte Es soll nun weiter aufgeschlüsselt werden, wie viel Zeit in den einzelnen Renderschritten der Bildkomposition von Proteus gebraucht wird. Um die Messungen möglichst genau zu halten wurde vor jeder Messung der Befehl `glFinish` ausgeführt. Demnach fällt die Summe der gemessenen Zeiten etwas höher als in den vorigen Tests aus. Es wurde über insgesamt 1000 Bilder gemittelt. Die Zeiten sind abermals in Millisekunden angegeben. „Zeichnen“ beinhaltet das Aktualisieren des Caches sowie das erste Zeichnen der soliden Geometrie. „Depth Peeling“ umfasst sowohl das Extrahieren aller Ebenen, als auch das füllen der Deferred Shading Texturen. „Beleuchtung“ beinhaltet sowohl das Erzeugen der Shadow Maps, als auch die Ausleuchtung der Szene. „Himmel“ fast die Zeichenschritte zum Anzeigen eines Himmels sowie zur Darstellung des Hintergrundes zusammen. „Tonemapping“ bezieht sich auf das Extrahieren der benötigten, globalen Bildinformationen, sowie das Tonemapping selbst. „Tools“ meint alle anderen Zeichenoperationen, die nach dem Tonemapping Objekte zeichnen. „Anzeigen“ misst die Dauer zum Kopieren des Bildes in den Farbpuffer der Darstellung. Für die Bush Szene wurde die Iteration mit 7784 zeichenbaren Objekten genutzt. In der 4000 Kugeln Szene wurden 80 verschiedene Materialien eingesetzt. Für den Licht-Test wurden 128 Lichtquellen verschiedener Farbe verwendet.

In dieser Messung zeigt sich ebenfalls, dass die Zeichenschritte, in denen Geometrie gezeichnet werden muss den Hauptteil der Zeit ausmacht. Der erste Zeichenschritt, indem die Szene in die Deferred Shading - Texturen geschrieben wird dauert rund 90% der Zeit. Für die „City“-Szene wird während des Depth Peelings die Szene etwa 4 mal neu gezeichnet. In der Hubschrauber-Szene wird in jedem Bild der Schatten einer Punktlichtquelle neu berechnet, was bisher mit dem sechsmaligen Zeichnen der Szene gleichzusetzen ist.

Speicherbedarf Insgesamt werden für die in Abschnitt 3.2 genannte Renderpipeline die folgenden Texturen, welche in der Auflösung des Bildbereiches vorliegen benötigt:

- 4x RGBA_16F für Deferred Shading
- 1x DEPTH_24_STENCIL_8_EXT geteilt von Deferred Shading und HDR Rendering
- 2x DEPTH_COMPONENT24 Textur als Eingabetextur für das Depth Peeling
- 1x DEPTH_24_STENCIL_8_EXT als Ausgabepuffer für das Depth Peeling
- 2x RGBA_16F für das Beleuchten und zeichnen des HDR Bildes
- 1x RGBA8 Textur für die Sicherung der ALPHA-WERTE

Demnach ergibt sich ein Gesamtbedarf von 68 Byte pro Pixel. Die folgende Tabelle zeigt exemplarisch den Speicherbedarf für verschiedene Auflösungen:

Auflösung	Speicherbedarf
512x512	17 MB
1024x768	51 MB
1024x1024	68 MB
2048x2048	272 MB

[18] gemittelt über 5 Messungen. System B benötigte etwa 26 Sekunden, was auf das verwendete Betriebssystem zurückzuführen ist.

Tabelle 5.6: Vergleich einzelner Abschnitte der Bildkomposition

Szene	Zeichnen		Depth P.		Beleuchtung		Himmel		Tonem.		Tools		Anzeigen	
System	A	B	A	B	A	B	A	B	A	B	A	B	A	B
Bush (7784)	106,48	142,82	—	—	1,33	0,68	3,78	1,49	2,38	0,74	0,38	0,16	0,8	0,30
City	19,31	5,24	5,77	2,61	2,15	2,22	0,77	0,24	2,25	0,55	2,34	0,25	0,78	0,2
Boids <statisch>	4,36	2,81	—	—	0,84	0,32	1,29	0,42	2,25	0,75	0,39	0,11	0,79	0,29
Boid <dynamisch>	4,52	2,98	—	—	0,73	0,26	1,31	0,43	2,21	0,72	0,33	0,1	0,78	0,28
Hubschrauber	3,31	2,06	—	—	4,16	3,35	8,92	3,33	2,43	0,7	0,32	0,13	0,8	0,28
4000 Kugeln (80)	88,52	52,65	—	—	1,37	0,64	1,32	0,43	2,29	0,76	0,38	0,16	0,78	0,29
FSPM	261,13	16,75	—	—	2,14	2,44	0,39	0,14	2,1	0,55	0,4	0,13	0,7	0,2
Licht (128)	5,74	1,4	—	—	1,82	0,61	0,5	0,13	2,49	0,54	0,34	0,13	0,83	0,19

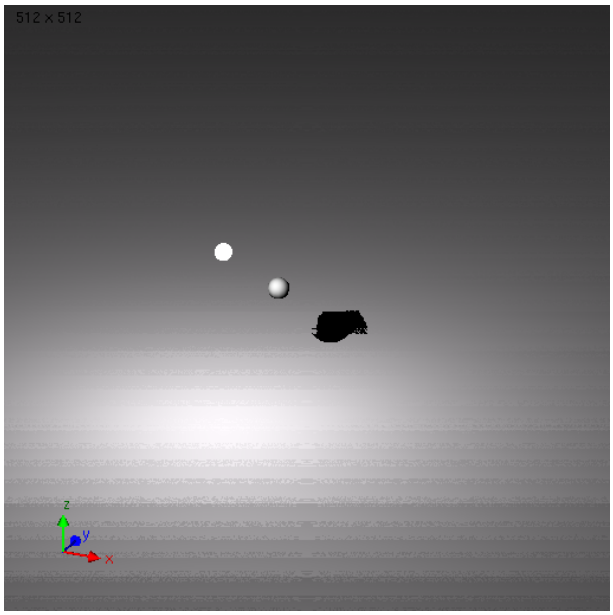


Abbildung 5.10: Aliasing-Fehler in der Rekonstruktion der Sichkoordinaten. Sowohl die diffuse Beleuchtung als auch der erzeugte Schatten weisen Streifen und ein leichtes Rauschen auf.

Zu dem gezeigten Speicherbedarf kommt noch, je nach Szene, der benötigte Platz für Shadow Maps, sowie herkömmliche Texturen. Außerdem können Grafikkarten aus Geschwindigtkeitsgründen zusätzlichen Speicherplatz belegen, der hier nicht einberechnet werden kann. Somit ist insbesondere beim Zeichnen mit hoher Auflösung eine Grafikkarte mit genügend Speicher nötig.

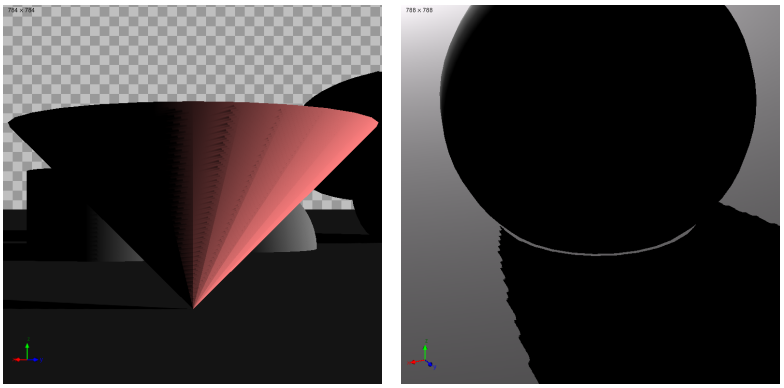
5.3 Bestehende Probleme

Dieser Abschnitt diskutiert bekannte Probleme der Implementierung und schildert kurz einige Ideen zu deren Lösung, welche aus verschiedenen Gründen nicht in die fertige Implementierung eingeflossen sind.

Rechengenauigkeit Ein Hauptproblem stellt die Rechengenauigkeit der GPU da. Bei zunehmender Entfernung zu den betrachteten Objekten verringert sich die Auflösung des Tiefenpuffers, da dieser sie Werte nicht linear speichert. Hierbei ist auch der Abstand zwischen near- und far-plane entscheidend. Je weiter die far-plane entfernt ist, desto geringer fällt die relative z-Auflösung aus. Dies äußert sich in Bildfehlern, wie sie in Abbildung 5.10 zu sehen sind.

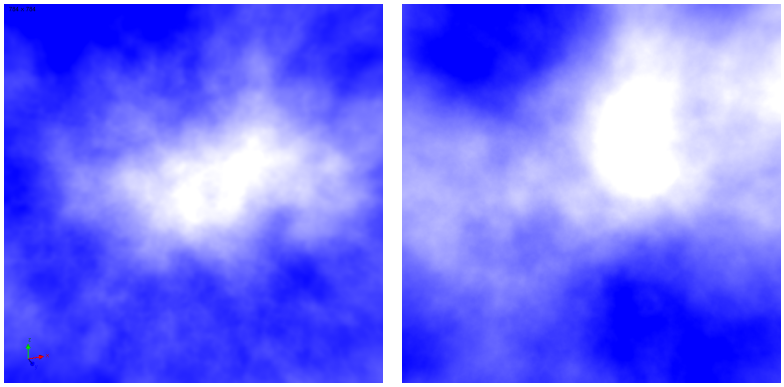
Eine Möglichkeit besteht darin Herstellerspezifische Erweiterungen zu nutzen, um einen 32-Bit Tiefenpuffer zu erzeugen. Hierbei geht allerdings möglicherweise die Unterstützung für den Stencilpuffer verloren. Eine weitere Möglichkeit bietet die manuelle Kodierung des z-Wertes. Hierbei werden weitere freie Einträge in den Texturen des Deferred Shading benötigt, die nicht verfügbar sind. Somit kann eine Lösung nur durch Anheben der Hardwareanforderungen realisiert werden. Es steht noch aus, ob ein linearisierter Tiefenpuffer genügend Genauigkeit bietet um das Problem zu beheben.

Schatten Shadow Maps zeigen aufgrund der endlichen Auflösung von Texturen deutliche Bildartefakte. Desweiteren leiden sie unter der Rechengenauigkeit der Grafikkarte. So können Fragmente nicht ohne Genauigkeitsverlust von Sichtkoordinaten in das Koordinatensystem der Lichtquelle umgerechnet werden. Aus beiden Problemen entsteht ein Effekt der als “shadow acne” bezeichnet wird. Um diesen zu verhindern werden die Entfernung in der Schattentextur um einen



(a) „shadow acne“ (b) Losgelöste Schatten

Abbildung 5.11: Probleme bei der Darstellung von Schatten in Proteus



(a) Proteus (b) Twilight

Abbildung 5.12: Bild auf einen Himmel, dessen „Wolken“ durch ein eine noise-Funktion erzeugt werden. Beide Bilder zeigen den selben Ausschnitt..

geringen Wert erhöht. Hierbei entsteht ein weiterer Effekt, bei dem der Schatten eines Objektes nicht mehr direkt an diesem beginnt. Beide Probleme sind in der Abbildung 5.11 zu sehen.

Um diese Effekte zu reduzieren muss die effektive Auflösung der Shadow Maps erhöht werden. Dies kann durch verbesserte Shadow Mapping Verfahren erreicht werden, die allerdings aus denen in Abschnitt 3.2.4 genannten Geschwindigkeitsgründen noch keine Verwendung finden. Eine weitere Möglichkeit besteht darin die Shadow Map zu Filtern. So kann beispielsweise PCF mit „Receiver Plane Depth Bias“ [17] kombiniert „shadow acne“ reduzieren.

Noise-Shader GroIMP verwendet eine interne Noise Funktion, welche aus Geschwindigkeitsgründen nicht exakt in GLSL-Code wiedergegeben werden kann. Die Wahl der Noise-Funktion für Proteus viel auf Simplex-Noise nach der frei verfügbaren Implementierungen von [14]. Diese weicht entsprechend von der in Twilight implementierten Funktion ab. Somit erzeugen beide Darstellungen für prozeduralen Texturen, welche die Noise-Funktion nutzen unterschiedliche Bilder, wie in Abbildung 5.12 zu sehen ist. Eine Übersetzung der in GroIMP genutzten Noise-Funktion für die GPU ist möglich. Ob diese durch geeignete Optimierung ähnliche Geschwindigkeiten wie Simplex-Noise erreichen kann muss noch überprüft werden.

6 Zusammenfassung und Ausblick

Es soll nun eine Zusammenfassung der Ergebnisse dieser Arbeit gegeben werden. Darauf Aufbauend schließt diese Arbeit mit einem Ausblick, der Anregungen zu Verbesserungen und Erweiterungen der Darstellung enthält.

6.1 Zusammenfassung

Dieses Kapitel fasst die bisherigen Ergebnisse der Bachelorarbeit zusammen. Die Aufgabenstellung umfasst eine Reimplementierung der in GroIMP verfügbaren Material-Shader in GLSL-Code sowie deren effiziente Darstellung mittels eines “deferred shadings”. Dieses Ziel wurde erreicht.

Um das ChannelMap-System von GroIMP in GLSL-Code zu übersetzen wurde ein Rahmenwerk in Java implementiert, welches die schnelle Ergänzung und Weiterentwicklung einzelner ChannelMaps erlaubt. Das vorgestellte System erzeugt zu beliebig komplexen GroIMP-Materialien ein passendes GLSL-Programm, beschränkt durch die Befehlsobergrenze der Grafikkarte. Zur Speicherung der Parameter des in GroIMP definierten Phong-Materials wurde für das “deferred shading” eine Methode zum Kodieren und Dekodieren von Daten entwickelt.

Es wurden spezielle Optimierungen für statische Szenen vorgenommen, wie das Zwischenspeichern des Szenegraphens. Als Referenz wurde der Standard-Strahlenverfolger von Twilight genutzt, sowie ergänzend der für Twilight implementierte Pathtracer betrachtet.

Bisher bestehende Anzeigefehler ließen sich auf die Rechengenauigkeit der verwendeten Datentypen zurückführen. Durch Änderungen an der Speichermethodik lassen sich diese Effekte reduzieren. In den getesteten Beispielszenen traten die erwähnten Genauigkeitsfehler erst bei großer Entfernung zur Szene auf.

Die Implementierung wurde sowohl auf NVIDIA- als auch auf ATI-Grafikkarten getestet, muss sich jedoch noch bewehren. Die Vielzahl an Treibern, Grafikkarten-Modellen und Betriebssystemen setzen Tests voraus, welche durch die zu Verfügung stehenden Testsysteme nur zu einem kleinen Teil abgedeckt werden. Die Darstellung durchläuft abschließende Tests und wird in den nächsten Versionen von GroIMP optional neben der Drahtgitter-Ansicht sowie der bestehenden OpenGL-Implementierung verfügbar sein.

Die beschriebene Implementierung Proteus ist auf einfache Erweiterung ausgelegt und kann leicht mit einer Vielzahl an Funktionen und Effekten ergänzt werden. Die Implementierung ist insbesondere für statische Szenen geeignet und steht in diesen der OpenGL 1.1 Implementierung in Nichts nach. Für dynamische Szenen mit vielen Objekten müssen sowohl der Cache als auch benötigte Shadow Maps neu erstellt werden, weshalb hier Proteus weniger Leistung als OGL1 erbringt. Durch Deaktivieren der Schatten lässt sich der Zeichenvorgang leicht beschleunigen. Ebenso sind Szenen mit einer geringen Anzahl an Objekten mit OGL1 wesentlich schneller zu berechnen, da Proteus einen größeren Verwaltungsaufwand für Objekte hat.

Proteus dient Vorschau auf erstellte Materialien und Lichtkompositionen. Hierbei werden auch Effekte wie Transparenz und Schatten angezeigt. Komplexere Lichteffekte wie Reflektion und Brechung werden nicht dargestellt.

6.2 Ausblick

Innerhalb der GLSL-Code-Generierung kann durch Maßnahmen wie dem Entrollen von Schleifen oder genauerer Ausnutzung paralleler Berechnungen die Rechenzeit der generierten Shader reduziert werden. Insbesondere Shader, welche die Noise-Funktion nutzen können stark optimiert werden. Hier wird bisher Simplex-Noise entwickelt von Ken Perlin nach den Erläuterungen und frei verfügbaren Implementierungen von [14] genutzt. Bei ausgiebiger Nutzung der Noise-Funktion ist das Nachschlagen in der Permutationstextur der Flaschenhals. Für eine GLSL basierte Implementierung könnte eine Aufteilung in Arrays oder 4x4 Matrizen das langsame Lesen aus Texturen ersetzen. Zum Zeitpunkt der Implementierung gab es allerdings Probleme mit Arrays in GLSL 1.20 auf OSX-Systemen[19], weshalb diese Idee zu einem späteren Zeitpunkt weiterverfolgt wird.

Weitere Maßnahmen zur Verbesserung der visuellen Qualität und der Geschwindigkeit sind im Folgenden ihrer Kategorie entsprechend aufgelistet:

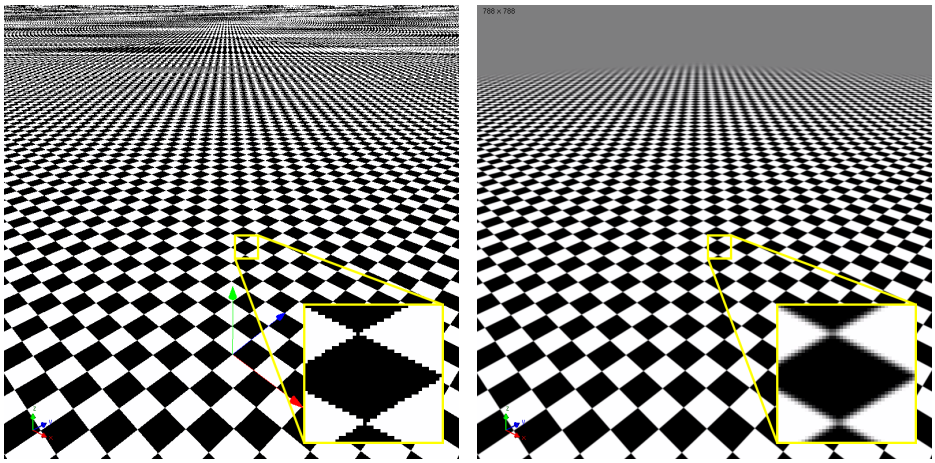
Edge Antialiasing Hardwarebeschleunigte Antialiasingverfahren, wie etwa Multi-Sampling, sind für Renderer, welche auf Deferred Shading aufbauen nicht anwendbar. Um Aliasing Artefakte an den Rändern von Objekten zu reduzieren lässt sich ein „Edge Bluring“ implementieren. Zunächst wird ermittelt, welche Fragmente auf einer Kante liegen. Dies kann, wie in [19, Kapitel 19.5.2] beschrieben, über die Normale und den z-Wert der umliegenden Fragmente geschehen. Liegt das Fragment auf einer Kante, so wird ein Unschärfefilter auf dieses angewandt. Problematisch ist der Zeitpunkt der Anwendung, da für das genannte Verfahren die umgebenden Pixel einer Kante bekannt sein müssen. Dies ist mit dem bisher genutzten Depth Peeling, welches die Szene von Vorne nach Hinten konstruiert nicht kombinierbar. Änderungen am Depth Peeling können „Edge Bluring“ auch für transparente Ebenen erlauben.

Shader Antialiasing Beim Einsatz prozeduraler Texturen sind herkömmliche Filtermethoden wie MipMaps nicht anwendbar. Um Aliasing zu reduzieren müssen shaderspezifische Filterlösungen genutzt werden. Aliasing entsteht wenn innerhalb eines Fragmentes Details vorkommen, welche kleiner als das Fragment selbst sind. Beim Sampeln wird nur ein Funktionswert genommen, anstatt die Funktion über die ganze Pixelgröße zu approximieren. Theoretisch kann dieser Effekt verhindert werden, wenn die Funktion, welche die Pixelfarbe beschreibt über die Pixelfläche integriert wird. Dies ist analytisch nur für bestimmte Shader möglich und insbesondere in den in GroIMP verketteten Shadern nicht praktikabel. Eine einfache Annäherung bietet das Ersetzen des Pixelwertes durch einen bekannten Durchschnittswert. Dieser ist bei einigen Shadern leichter zu bestimmen als das Integral und kann oft auch approximiert werden. Die numerische Auswertung der Integrale könnte auch helfen, allerdings werden die Implementierten GLSL-Shader dann sehr Aufwendig, da für ein korrektes Ergebnis die Werte in jeder ChannelMap einzeln integriert werden müssten.

Eine weitere Form von Aliasing entsteht bei scharfen Übergängen innerhalb eines Shaders. Diese entstehen meist durch die Nutzung der `step`-Funktion oder anderer, nicht glatter Funktionen. In diesen Fällen kann das Ersetzen durch eine glatte Funktion das entstehende Bild weicher erscheinen lassen. Beide Techniken wurden exemplarisch für die ChannelMap: Checker implementiert (siehe Abbildung 6.1). Durch Analyse der Shader können weitere Antialiasing-Verfahren implementiert werden.

Beschleunigung der Zeichenroutinen Bisher erfolgt die Darstellung primitiver geometrischer Objekte, ausgenommen der Kugel, über keinerlei Beschleunigungsstrukturen. OpenGL bietet hier

[19] Fehlernummer #5183554 in der Apple Fehler Datanbank. Siehe auch http://openradar.appspot.com/6121615 (Stand: April 2010)



(a) ohne Antialiasing (b) mit Antialiasing

Abbildung 6.1: Shader Antialiasing

Vertex-Buffer-Objects und Displaylisten, welche in der Lage sind Vertexlisten und Indizierungen direkt auf der Grafikkarte zu sichern. Da Objekte der Szene pro Zeichenschritt wiederholt gezeichnet werden müssen, ließe sich durch entsprechende Anpassungen die Darstellung merklich beschleunigen. Schnellere Zeichenvorgänge erlauben unter anderem aufwendigere Shadow Map Techniken.

In Proteus werden keine keine Methode um Objekte, welche nicht sichtbar sind beim Zeichnen zu überspringen genutzt. (Solche Verfahren werden „Culling" genannt). Die im Shape-Cache hinterlegten Volumeninformationen lassen sich nutzen, um ein einfaches Frustum Culling, wie in [5] beschrieben, zu implementieren.

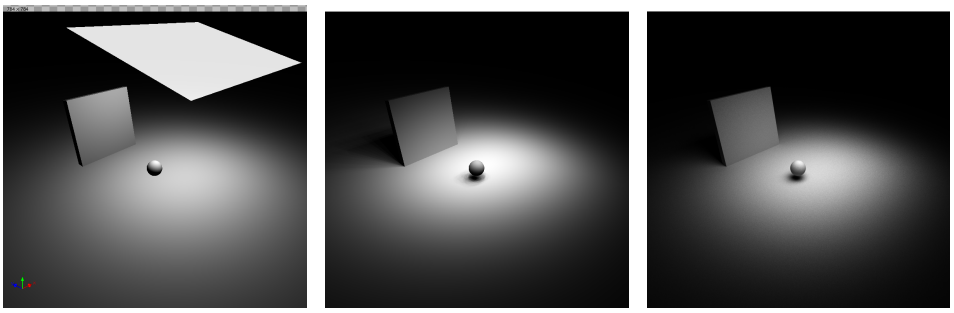
Licht und Schatten Primär lassen sich in der Darstellung von Schatten noch viele Verbesserungen vornehmen. So könnten beispielsweise „Cascaded Shadow Maps" [8], wie sie in interaktiven Anwendungen wie Spielen eingesetzt werden, die Qualität der angezeigten Schatten deutlich erhöhen. Bei dieser Methode können Schattentexturen allerdings nicht mehr vorberechnet werden, da diese Abhängig von der Kameraposition bestimmt werden. Zunächst muss der Zeichenvorgang beschleunigt werden, bevor Methoden dieser Art genutzt werden können.

Ein Lichtquellentyp, welcher bisher nicht korrekt berechnet wird, ist die Flächenlichtquelle. Beleuchtung durch ein Flächenlicht lässt sich über GLSL-Programme implementieren. An einer Übersetzung der in [32] vorgestellten Methode in GLSL-Code wird gearbeitet. Bisher wurde nur die diffuse Beleuchtung umgesetzt, wie in Abbildung 6.2 zu sehen. Problematisch ist die Implementierung von Schatten. Eine Möglichkeit bestünde in der Annäherung durch Scheinwerferlichtquellen. Eine darauf basierende Technik, welche Entlang von Kanten einer Flächenlichtquelle sampelt stellen [37] vor.

Weitere Verbesserungsmöglichkeiten Bisher wurde die Implementierung für das HDR-Rendering konzipiert. Für LDR-Rendering lassen sich einige Optimierungen vornehmen. Insbesondere der Aufwand der Lichtberechnung ließe sich durch Light Volumes reduzieren.

Wie im vorherigen Kapitel in Abschnitt 5.2 gezeigt ist Deferred Shading wie für Proteus implementiert sehr speicherintensiv. Für sehr hohe Auflösung kann das Zerlegen des Bildes in einzelne Kacheln, die Nacheinander berechnet werden den Speicheraufwand auf Kosten der Zeichenzeit deutlich vermindern.

Im Bereich der optischen Effekte existieren eine Reihe von Algorithmen, die vollständig im Bildraum operieren und so gut mit Deferred Shading zusammenarbeiten. Ein Beispiel wäre



(a) Proteus (b) Twilight (STDR) (c) Twilight (PTR)

Abbildung 6.2: Flächenlichtquellen in GroIMP. In Bild 6.2a wurde die Flächenlichtquelle mit dargestellt. Der Standard-Raytracer von Twilight nähert Flächenlichter über mehrere Scheinwerferlichtquellen an. Der Pathtracer erzeugt ein realistisches Bild, benötigt allerdings eine hohe Zahl an Strahlen.

„Interactive image-space refraction of nearby geometry“ von Wyman [36]. Desweiteren können insbesondere in statischen Szenen Reflektionen sehr gut vorberechnet und angezeigt werden.

GroIMP das Erzeugen von komplexer Geometrie mittels Constructiv-Solid-Geometry (kurz CSG). Das Zeichnen von CSG-Bäumen lassen sich mittels geeigneter Multi-Pass-Verfahren im Bildbraum realisieren, wie der “Blister” Algorithmus von Hable [15] zeigt.

Die unzureichende OpenGL Unterstützung der Intel Grafikkarten lässt bisher nicht zu, dass Proteus auf entsprechenden GPUs getestet werden kann. Eine im Funktionsumfang reduzierte Version der Darstellung könnte auch auf diesen lauffähig sein.

Literaturverzeichnis

[1] Bavoil, Louis ; Myers, Kevin: *Order Independant Transparency with Dual Depth Peeling.* NVIDIA OpenGL SDK. http://developer.download.nvidia.com/SDK/10/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf. Version: Februar 2008, Abruf: 05.04.2010

[2] Blinn, James F.: Models of light reflection for computer synthesized pictures. In: *SIGGRAPH Comput. Graph.* 11 (1977), Nr. 2, S. 192–198. http://dx.doi.org/10.1145/965141.563893. – DOI 10.1145/965141.563893. – ISSN 0097–8930

[3] Brabec, Stefan ; Annen, Thomas ; Seidel, Hans-peter: Shadow Mapping for Hemispherical and Omnidirectional Light Sources. In: *In Proc. of Computer Graphics International*, 2002, S. 397–408

[4] Brown, Pat ; Leech, Jon ; Mace, Rob ; Paul, Brian: *ARB_half_float_pixel.* ARB, Oktober 2004. http://www.opengl.org/registry/specs/ARB/half_float_pixel.txt, Abruf: 05.04.2010

[5] Clark, James H.: Hierarchical geometric models for visible surface algorithms. In: *Commun. ACM* 19 (1976), Nr. 10, S. 547–554. http://dx.doi.org/10.1145/360349.360354. – DOI 10.1145/360349.360354. – ISSN 0001–0782

[6] Crow, Franklin C.: Shadow algorithms for computer graphics. In: *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques.* New York, NY, USA : ACM Press, 1977. – ISSN 0097–8930, S. 242–248

[7] Deering, Michael ; Winner, Stephanie ; Schediwy, Bic ; Duffy, Chris ; Hunt, Neil: The triangle processor and normal vector shader: a VLSI system for high performance graphics. In: *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques.* New York, NY, USA : ACM, 1988. – ISBN 0–89791–275–6, S. 21–30

[8] Dimitrov, Rouslan: *Cascaded Shadow Maps.* http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf. Version: August 2007, Abruf: 05.04.2010

[9] Everitt, Cass: *Interactive Order-Independent Transparency.* http://developer.nvidia.com/attach/6545. Version: Mai 2001, Abruf: 05.04.2010

[10] Gerasimov, Philipp S.: Omnidirectional Shadow Mapping. In: Fernando, Randima (Hrsg.): *GPU Gems.* Pearson Higher Education, 2004. – ISBN 0321228324, Buch 12

[11] Gouraud, H.: Continuous Shading of Curved Surfaces. In: *IEEE Trans. Comput.* 20 (1971), Nr. 6, S. 623–629. http://dx.doi.org/10.1109/T-C.1971.223313. – DOI 10.1109/T–C.1971.223313. – ISSN 0018–9340

[12] Green, Simon ; Cebenoyan, Cem: *High Dynamic Range Rendering on the GeForce 6800.* http://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_HDR.pdf. Version: 2004, Abruf: 05.04.2010

[13] Greene, Ned: Environment mapping and other applications of world projections. In: *IEEE Comput. Graph. Appl.* 6 (1986), Nr. 11, S. 21–29. http://dx.doi.org/10.1109/MCG.1986.276658. – DOI 10.1109/MCG.1986.276658. – ISSN 0272–1716

[14] Gustavson, Stefan: Simplex Noise demystified / Linköping University, Sweden. Version: 2005. http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf. 2005. – Forschungsbericht

[15] Hable, John ; Rossignac, Jarek: Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes. In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*. New York, NY, USA : ACM, 2005, S. 1024–1031

[16] Harris, Mark ; Hargreaves, Shawn: *6800 Leagues under the sea - Deferred Shading*. http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf. Version: Juli 2004, Abruf: 05.04.2010

[17] Isidoro, J. R.: *Shadow Mapping: GPU-based Tips and Techniques*. Conference Session. Version: März 2006. http://developer.amd.com/media/gpu_assets/Isidoro-ShadowMapping.pdf, Abruf: 08.04.2010

[18] Kniemeyer, Ole: *Design and implementation of a graph grammar based language for functional-structural plant modelling*, LS Praktische Informatik/ Graphische Systeme, Dissertation, Februar 2009

[19] Koonce, Rusty: Deferred Shading in Tabula Rasa. In: Nguyen, Hubert (Hrsg.) ; iXBT.com (Veranst.): *GPU Gems 3*. Addison-Wesley Professional, August 2007. – ISBN 0321515269, Buch 19

[20] Larson, Gregory W. ; Rushmeier, Holly ; Piatko, Christine: A Visibility Matching Tone Reproduction Operator for High Dynamic Range Scenes. In: *IEEE Transactions on Visualization and Computer Graphics* 3 (1997), Nr. 4, S. 291–306. http://dx.doi.org/10.1109/2945.646233. – DOI 10.1109/2945.646233. – ISSN 1077–2626

[21] Lee, Mark: Pre-lighting in Resistance 2 / Insomniac Games, Inc. Version: April 2009. http://www.insomniacgames.com/research_dev/articles/2009/1500923. 2009. – Forschungsbericht

[22] Liu, Fang ; Huang, Meng-Cheng ; Liu, Xue-Hui ; Wu, En-Hua: Efficient depth peeling via bucket sort. In: *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*. New York, NY, USA : ACM, 2009. – ISBN 978–1–60558–603–8, S. 51–57

[23] McGuire, M. ; Hugues, J. F. ; Egan, K. T. ; Kilgard, M. ; Everitt, C.: Fast, Practical and Robust Shadows / Brown University. 2003. – Forschungsbericht

[24] P754, IEEE T.: *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, August 1985. – 18 S.

[25] Phong, Bui T.: Illumination for computer generated pictures. In: *Commun. ACM* 18 (1975), Nr. 6, S. 311–317. http://dx.doi.org/10.1145/360825.360839. – DOI 10.1145/360825.360839. – ISSN 0001–0782

[26] Pranckevičius, Aras: *Compact Normal Storage for small G-Buffers*. http://aras-p.info/texts/CompactNormalStorage.html. Version: August 2009, Abruf: 05.04.2010

[27] Preetham, A. J. ; Shirley, Peter ; Smits, Brian: A practical analytic model for daylight. In: *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1999. – ISBN 0–201–48560–5, S. 91–100

[28] Randi J., Rost: *OpenGL Shading Language*. Addison Wesley Professional, 2006

[29] Reeves, William T. ; Salesin, David H. ; Cook, Robert L.: Rendering antialiased shadows with depth maps. In: *SIGGRAPH Comput. Graph.* 21 (1987), Nr. 4, S. 283–291. http://dx.doi.org/10.1145/37402.37435. – DOI 10.1145/37402.37435. – ISSN 0097–8930

[30] Reinhard, Erik ; Devlin, Kate: Dynamic Range Reduction Inspired by Photoreceptor Physiology. In: *IEEE Transactions on Visualization and Computer Graphics* 11 (2005), Nr. 1, S. 13–24. http://dx.doi.org/10.1109/TVCG.2005.9. – DOI 10.1109/TVCG.2005.9. – ISSN 1077–2626

[31] Reinhard, Erik ; Ward, Greg ; Pattanaik, Sumanta ; Debevec, Paul: *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting (The Morgan Kaufmann Series in Computer Graphics)*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2005. – ISBN 0125852630

[32] Snyder, John M.: Area Light Sources for Real-Time Graphics / Microsoft Research; Advanced Technology Division; Microsoft Corporation. Version: 1996. http://research.microsoft.com/en-us/um/people/johnsny/papers/arealights.pdf, Abruf: 05.04.2010. 1996. – Forschungsbericht

[33] Stamminger, Marc ; Drettakis, George: Perspective shadow maps. In: *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM, 2002. – ISBN 1–58113–521–1, S. 557–562

[34] Stich, Martin ; Wächter, Carsten ; Keller, Alexander: Efficient and Robust Shadow Volumes Using Hierarchical Occlusion Culling and Geometry Shaders. In: Nguyen, Hubert (Hrsg.) ; iXBT.com (Veranst.): *GPU Gems 3*. Addison-Wesley Professional, August 2007. – ISBN 0321515269, Buch 11

[35] Williams, Lance: Casting curved shadows on curved surfaces. In: *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM, 1978, S. 270–274

[36] Wyman, Chris: Interactive image-space refraction of nearby geometry. In: *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–201–1, S. 205–211

[37] Ying, Zhengming ; Tang, Min ; Dong, Jinxiang: Soft Shadow Maps for Area Light by Area Approximation. In: *PG '02: Proceedings of the 10th Pacific Conference on Computer Graphics and Applications*. Washington, DC, USA : IEEE Computer Society, 2002. – ISBN 0–7695–1784–6, S. 442

[38] Zhang, Fan ; Sun, Hanqiu ; Xu, Leilei ; Lun, Lee K.: Parallel-split shadow maps for large-scale virtual environments. In: *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–324–7, S. 311–318