Das effiziente Raytracen von Dreiecksnetzen auf Mehrkernprozessoren, GPUs und FPGAs mittels Geometrischer Algebra

Efficient Raytracing of Triangles on Multicore-CPUs, GPUs and FPGAs with Geometric Algebra

Master-Thesis von Michael Burger

November 2011

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Eingebettete Systeme
und ihre Anwendungen

Das effiziente Raytracen von Dreiecksnetzen auf Mehrkernprozessoren, GPUs und FPGAs mittels Geometrischer Algebra
Efficient Raytracing of Triangles on Multicore-CPUs, GPUs and FPGAs with Geometric Algebra

Vorgelegte Master-Thesis von Michael Burger

1. Gutachten: Prof. Dr. Andreas Koch
2. Gutachten: Dr. Dietmar Hildenbrand

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 16. November 2011

(M. Burger)

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben. Insbesondere danke ich:

- Prof. Andreas Koch und Dr. Dietmar Hildenbrand für die Betreuung der Arbeit.
- Ganz besonders Thorsten Wink und Sascha Mühlbach für technische Unterstützung bzgl. der FPGAs und der entsprechenden Softwareumgebung. Außerdem Florian Stock, Peter Müller und Patrick Charrier für Hilfestellungen bei SVN und Gaalop.
- Crispin Deul für viele fachliche Diskussionen zum Thema der Arbeit und verwandten Themen, sowie für seine kritischen und hilfreichen Kommentare. Die nicht fachlichen, aber dafür sehr unterhaltsamen ICQ-Gespräche seien aber ebenfalls erwähnt.
- Besonderen Dank natürlich meinen Eltern, ohne deren Unterstützung die Arbeit nie fertig gestellt worden wäre.
- Allen, die mich während der Implementierung ermutigt haben, wenn es mal nicht so schnell vorwärts ging, wie ich es mir gewünscht hätte oder sich ein weiteres Problem aufgetan hat. In loser Reihenfolge: Babs, Angi, Sarah, Martina, Simone und Katrin.
- Erich Schmitt und seiner Firma Eggs Media für das kostengünstige Ausdrucken der Arbeit.

Inhaltsverzeichnis

1 Einführung

1.1 Einleitung

Das auch heute in der Graphischen Datenverarbeitung (GDV) am weitesten verbreitete Bilderzeugungsverfahren ist es Objekte als Dreiecksnetze zu modellieren und mit Hilfe fest definierter Methoden, die DirectX und OpenGL zur Verfügung stellen, durch die Grafikkarte berechnen zu lassen. Dieses Verfahren nennt sich *Rasterisierung* und wird von der Hardware direkt unterstützt, wodurch eine hohe Geschwindigkeit erreicht wird. In modernen Computerspielen kommt diese Technik fast ausschließlich zum Einsatz. Hierbei sind mehrere Millionen Dreiecke pro Szene mittlerweile die Regel und die aktuelle Grafikhardware ist in der Lage derartige Szenen mit mehr als 30 Bildern pro Sekunde (fps) darzustellen.

Dennoch existiert bereits seit den sechziger Jahren des 20. Jahrhunderts mit dem *Raytracingverfahren* (dt. *Strahlenverfolgung*) eine zweite Variante der Bilderzeugung aus dreidimensionalen Szenenbeschreibungen. Im Vergleich zum erstgenannten Ansatz bietet Raytracing einige Vorteile, so dass es bis heute als alternativer Weg bestehen kann und weiterhin auf diesem Gebiet geforscht wird. Bei der Rasterisierung sind Entwickler gezwungen Reflexionen und Schattierungen durch bestimmte Verfahren anzunähern oder zu imitieren. Beispielsweise werden Reflexionen durch das Aufkleben von zu spiegelnden Objekten als Texturen umgesetzt. Raytracing besitzt die Möglichkeiten implizit Spiegelungen und Schattenwurf physikalisch korrekt zu berechnen und darzustellen. Ebenso verhält es sich mit den *Level-of-Detail(LOD)*-Berechnungen [LWC+02]. Momentan sind die Entwickler von 3D-Engines gezwungen, die Anzahl der Dreiecke, aus denen ein Objekt modelliert ist, dynamisch in Abhängigkeit von der Entfernung zum Betrachter zu reduzieren bzw. zu erhöhen. Dadurch sollen Over- und Undersampling vermieden werden. Beim Raytracing wird diese Schwierigkeit automatisch umgangen. Auch steigen bei Nutzung der Strahlenverfolgung die Modellierungsmöglichkeiten für die Szene, da keine Einschränkung auf Dreiecke (bzw. allgemeine Polygone) vorliegt. Auch ist es möglich allgemeine geometrische Objekte wie Quadriken [LB06] oder durch algebraische Funktionen beschriebene Oberflächen [Rei07] zum Einsatz zu bringen.

Dafür besitzt das Raytracing einen entscheidenden Nachteil in der Geschwindigkeit. Es ist momentan nicht möglich vergleichbar große Szenen bei gleicher Auflösung innerhalb derselben Zeit zu rendern. Dieses Problem wird auf verschiedene Weise angegangen. Während klassische Raytracer ausschließlich auf Standard-CPUs hin konzipiert sind, werden neuere Varianten für alternative Architekturen implementiert. Im Vordergrund stehen hierbei die Grafikkarten. Deren Aufbau wurde in den letzten Jahren in Hinsicht auf die Programmierbarkeit deutlich flexibler gestaltet. Diese Entwicklung begann mit der Einführung von T&L (DirectX7 /1999), führte über die Pixel-/Vertexshader (DirectX8 / 2000) und den Geometryshader (DirectX10 / 2008) bis hin zu den heute, mit Hilfe von OpenCL [Khr11] fast frei programmierbaren, 3D-Beschleunigern (DirectX11). Allerdings ist auch spezielle Raytracing-Hardware, entweder auf rekonfigurierbaren Bausteinen (FPGAs) oder als Chip, ein Thema der Forschung.

Eng verbunden hiermit ist das Gebiet der parallelen Programmierung, die immer mehr Verwendung findet. Nachdem die Grafikkartenarchitektur schon lange auf parallele Berechnungen ausgelegt ist, sind auch die Hersteller von Prozessoren diesem Trend gefolgt. Nachdem mit Hyper-Threading bereits 2002 zwei virtuelle Kerne auf einem Prozessor realisiert wurden, bestehen moderne Consumer-CPUs aus bis zu sechs physikalischen und mittels Hyper-Threading bis zu zwölf virtuellen Kernen[1]. Somit entstanden auch neue Sprachen, Paradigmen und Konventionen für das parallele Programmieren[MSM04].

Neben der Wahl der Hardwareplattform gibt es aber auch am eigentlichen Raytracing-Algorithmus Weiterentwicklungen um das Geschwindigkeitsdefizit zu reduzieren. Hierzu zählen räumliche Datenstrukturen oder effektivere Wege verschiedene 3D-Objektrepräsentationen auf dem zweidimensionalen Bildschirm abzubilden. Relativ unberührt bleibt dagegen die Lineare Algebra (LA), die nach wie vor die mathematische Grundlage nahezu aller Raytracingverfahren bildet. Auch in weiteren Anwendungsgebieten wie der Robotik stellt die LA den de-facto-Standard dar. In den letzten Jahren kam in diesen Gebieten allerdings die *Geometrischen Algebra (GA)* auf, welche bekannte mathematische Konzepte wie Rigid Body Motions, Komplexe Zahlen und Quaternionen in ihrem einheitlichen Rahmen zusammenführt[DFM07b]. Ihr Nachteil ist die Anzahl der Berechnungen und somit auch die langsamere Verarbeitungszeit im Vergleich zur LA. Um diesen zu kompensieren wird versucht, die Anzahl der Berechnungen durch Optimierungen zu reduzieren. Außerdem werden bestimmte Algorithmen direkt auf spezielle Architekturen hin optimiert.

1.2 Ziele dieser Arbeit

Da die Themenstellung dieser Arbeit im Verlauf der Ausarbeitung einige Male etwas variierte, verfolgen die in ihrem Rahmen stattfindenden Untersuchungen verschiedene Ziele bzw. Schwerpunkte.

Grundidee der Arbeit ist das Erstellen eines Raytracers mit Algorithmen in Konformer Geometrischer Algebra, welche durch das Tool Gaalop [Hil11] optimiert werden. Dieser Raytracer wird mit den bekannten Verfahren der Linearen Algebra verglichen, um so Stärken und Schwächen des Ansatzes in Geometrischer Algebra herausstellen zu können. Auch wird geprüft an welchen Stellen der automatisch generierte Code manuell weiter verbessert werden kann und ob es möglich ist derartige Optimierungsstrategien in einen Compiler zu integrieren.

Im Verlauf der Bearbeitung kristallisiert sich heraus, dass Geometrische Algebra (GA) auf Grund ihrer Struktur rein theoretisch Vorteile auf Hardware besitzt, welche die ihr innewohnende Parallelität ausnutzen kann, weshalb die Implementierung auf aktueller Grafikhardware mittels OpenCL [Khr11] erfolgt.

Ebenfalls ist es für das Fachgebeit GRIS interessant, dass es möglich ist, den Geometrische Algebra Raytracer der Uni Amsterdam, der mittels Gaigen und Gaigen2 optimiert wurde, in Bezug auf Geschwindigkeit mit einem Gaalop-Raytracer zu vergleichen. Daher wird zum einen der Amsterdamer Raytracer auf aktueller Hardware getestet und zum anderen ein Vergleich der Performance durchgeführt.

[1] Der Pentium 4 HT führte 2002 das Hyperthreading ein. Die Intel i7 Serien 97X, 98X und 99X für den Sockel 1366, welche 2010 auf den Markt kamen, bieten sechs HT-fähige Kerne

Eine weitere Frage, die beantwortet werden soll, ist, wie sich die Parallelität der GA-Berechnungen in einer Raytracer-Applikation auf einem FPGA umsetzen und ausnutzen lässt. Hierzu wird ein Raycaster in Verilog geschrieben und auf verschiedenen FPGAs getestet.

1.3 Verwandte Arbeiten

Generelle Vergleiche zwischen bestehenden Lösungen in Linearer Algebra und neuen Algorithmen in Geometrischer Algebra existieren von verschiedenen Autoren. So untersuchen Hildenbrand, Bayro-Corrochano und Zamora [HBCZE05] die Realisierung von Algorithmen der Robotik mit Konformer Geometrischer Algebra und kommen zu dem Schluss, dass sich die untersuchten Algorithmen direkt und intuitiv mit deren Methoden formulieren lassen. Derartige Algorithmen werden von Hildenbrand, Dorst et al. zum einen mittels des Library Generators Gaigen2 [Fon06] und zum anderen mit Hilfe von Maple [Map11] optimiert und im Rahmen des Avalon-Projektes in Software realisiert. Weitere Bereiche für welche die Nützlichkeit von GA untersucht wird, sind die Themengebiete der Color Edge Detection [MW06], die Simulation von Deformationen [BKHS09] oder das Raytracen von Punktwolken [Deu10].

Die wohl erste Arbeit, die sich mit der Berechnung von GA auf einem FPGA beschäftigt, stammt von Perwass et al. [PGS03] und wurde im Rahmen einer Diplomarbeit [Geb03] entwickelt. Auf einem FPGA wird ein Modul zur Berechnung des Geometrischen Produktes für Algebren zwischen 2D und 8D realisiert und bezüglich Effizienz mit Gaigen [FBD01] sowie CLU[2] verglichen. Silvia Franchini et al. [FGS$^+$09] [GSS$^+$05] versuchen die Parallelität der Berechnungen in GA in einer FPGA-Implementierung auf einem Xilinx Virtex2000E [3] nutzbar zu machen, die sie *CliffoSor* nennen. Der von ihnen entworfene GA-Koprozessor unterstützt diverse Produkte der 4D homogenen Algebra, sowie Differenzen und Summen. Sie vergleichen die Leistungsfähigkeit ihrer FPGA-Implementierung Cliffosor mit dem GA- Library-Generator Gaigen [FD03] von Daniel Fontijne und Leo Dorst. Für den Geschwindigkeitsvergleich wird unter anderem auch ein kleiner Raytracer eingesetzt, der die anfallenden Berechnungen entweder an die Gaigen-Bibliothek oder den FPGA weiterreicht. Der eigentliche Raytracing-Algorithmus läuft bei Franchinis Test auf der CPU des Host-Computers ab. Die Ausführungszeit ist auf dem Virtex2000E deutlich höher, wobei 98% der Gesamtlaufzeit aus Kommunikation von FPGA und Host bestehen. Generell benötigt die FPGA-Variante deutlich weniger clock cycles, so dass bei gleichem Takt eine normale CPU geschlagen wird. Allerdings enthält das mit dem CliffoSor generierte Bild aufgrund von Rechenungenauigkeiten Artefakte. Anzumerken ist, dass Gaigen mittlerweile durch das schnellere Gaigen2 [Fon06] abgelöst worden ist. Hildenbrand et al. untersuchen außerdem die Leistungsfähigkeit einer FPGA-Implementierung eines Inversen Kinematikmodells für einen Roboterarm [HLSK08] und erreichen einen 100fachen Geschwindigkeitsvorteil im Vergleich zu einer softwarebasierten GA-Lösung.

Auch der Bereich des Raytracings in LA auf einem FPGA ist Thema von Untersuchungen. Hier sind zum einen die Arbeiten von Joshua Fender und Jonathan Rose zu nennen, die eine vollständige Raytracing-Einheit auf einem Transmogrifier-3a Board [Uni04] mit vier Xilinx Virtex2000E FPGAs implementieren [Fen03]. Der Raytracer unterstützt eine räumliche Datenstruktur und ar-

[2] Das Projekt ist eingestellt. Es war unter `www.perwass.de/cbup/clu.html` zu finden und wurde aber teilweise in CluCalc integriert.

[3] Technische Details zu diesem Baustein kann der interessierte Leser auf [Xil02] finden.

beitet mit einem eigenen Datenformat, welches nur 64bit Speicherplatz für die Dreiecksposition benötigt. Er erreicht laut den Ergebnissen im Paper einen bis zu 20-fachen Geschwindigkeitsvorteil im Vergleich zum frei verfügbaren POVray-Raytracer [4], der auf einem Pentium 4 mit 2 GHz ausgeführt wird. Ein zweites Projekt zur Untersuchung von Raytracing in Hardware stellt der *SaarCor* dar [SWS02], der in der Arbeitsgruppe um Philipp Slusallek an der Universität Saarbrücken entworfen wurde. Der SaarCor wurde zur frei programmierbaren *Ray Processing Unit* (RPU) weiterentwickelt [WSS05] [SWW+04], das Projekt dann aber 2006 beendet.

[4] Neuere Versionen des Raytracers können kostenfrei unter [BC] heruntergeladen werden.

2 Geometrische Algebra

Dieser Abschnitt soll sich in Kurzform mit der Konformen Geometrischen Algebra kurz KGA oder Conformal Geometric Algebra (CGA) auseinander setzen. Es handelt sich hierbei nicht um eine generelle Einführung in dieses Themengebiet, sondern es werden nur die Punkte behandelt, welche für das Verständnis der folgenden Arbeit vonnöten sind. Für detailliertere Informationen über das Thema wird auf [DFM07a] und [PH03] verwiesen. Es soll bereits an dieser Stelle auf Aspekte hingewiesen werden, die für eine effiziente Implementierung berücksichtigt werden müssen bzw. eine solche ermöglichen.

Geometrische Algebra basiert auf den Grundlagen von Hermann Grassmann (1809-1877) und William Kingdon Clifford (1845-1879) [DFM07b, S. xxxiii f], welche von David Hestenes (*1933) zu einem einheitlichen System zusammengefügt und unter dem Namen *Clifford Algebra* als Alternative zur klassischen Linearen Algebra formuliert [HS87] wurde, die in den Bereichen wie Graphische Datenverarbeitung (GDV) und Robotik momentan das dominierende Element darstellt. Als Vorteile der KGA im Vergleich zur LA werden vor allem genannt:

- **Einheitliches System**: Während Programmierer momentan bei der Arbeit mit GDV mit unterschiedlichen Systemen und Aspekten wie Matrizenrechnung, Quaternionen oder Komplexen Zahlen konfrontiert werden, steht mit der Konforme Geometrische Algebra (KGA) ein alle diese Punkte umfassender Rahmen zur Verfügung, der nur einmal erlernt werden muss [Hil06, S. 14]. Besonders nützlich ist die Tatsache, dass sich die euklidischen Transformationen (Rotationen, Translationen, Reflexionen) im Rahmen der KGA direkt als *Versor*-Objekte darstellen lassen, welche sich bei Berechnungen genauso verhalten wie andere Objekte, z.B. Punkte oder Kugeln. Versoren werden in dieser Arbeit allerdings nicht benutzt, da die Dreiecksnetze bereits beim Einlesen mittels 4x4 Matrizen transformiert werden. Weiterführende Informationen zu Versoren finden sich in [DFM07a, S. 364 - 370].
- **Intuitivität**: Anstatt abstrakt denken zu müssen, wie sich z.B. eine Kugel und ein Strahl in der Darstellung als Matrize und Vektor schneiden lassen, kann dieser geometrische Sachverhalt in der KGA direkt abgebildet werden, was sehr intuitiv ist [DFM07a, S. 1-4]. Weitere Beispiele für intuitive Ausdrucksmöglichkeiten innerhalb von KGA werden im Verlauf der Arbeit sichtbar.
- **Kompaktheit**: Dadurch, dass sich geometrische Objekte und deren Beziehungen zueinander sehr kompakt beschreiben lassen, werden die in KGA formulierten Algorithmen kürzer und übersichtlicher. Dies erleichtert zum einen die Entwicklung selbst und zum anderen auch Fehlersuche und Wartung des Codes. Als Demonstration hierfür soll der Raycaster im Anhang A dieser Arbeit dienen, der in den wenigen Zeilen Code, aus denen er besteht, eine vollständige Funktionalität bietet. Er stellt ebenfalls ein kleines Tutorial für GA dar.

Im Verlauf dieser Arbeit wird auf diese Vorteile verwiesen, wenn sie sich bei der Implementierung zeigen bzw. auch dort, wo sie nicht vollständig genutzt werden können oder zu Gunsten der Effizienz aufgegeben werden müssen.

Christian Perwass unterscheidet in einer Einführung zur KGA zwischen *Clifford Algebra* und *Geometrischer Algebra*, da der Fokus der ersteren auf den algebraischen Eigenschaften, derjenige der zweiten dagegen auf den geometrischen Interpretationen liege [PH03, S. 1]. Diese Unterscheidung wird für diese Arbeit übernommen, deren Schwerpunkt auf der Bilderzeugung und somit auf der Geometrischen Algebra liegt. Details der zugrunde liegenden Algebra werden nur dann behandelt, wenn sie für das Verständnis der Implementierung nötig sind.

2.1 Basisvektoren

Während Objekte der 3D Linearen Algebra auf ihre drei Basisvektoren x, y und z bezogen sind, besteht die 5D Konforme Geometrische Algebra aus den fünf Basisvektoren e_1, e_2, e_3, e_0 und e_∞. e_1, e_2, e_3 kann man sich vereinfacht als Analogon zu den drei Richtungen eines dreidimensionalen Koordinatensystems vorstellen. e_0 stellt den eindeutigen Ursprung des Systems dar. e_∞ ist ein Punkt im Unendlichen, der allen Geraden und Ebenen gemeinsam ist und von euklidischen Transformationen nicht verändert wird [DFM07b, S. 356f]. Die Objekte der KGA werden mittels der Kombinationen dieser fünf Einheiten gebildet. Zu ihrer Kombination wird das Äußere Produkt (OP) benötigt, welches im nächsten Abschnitt vorgestellt wird.

2.2 Produkte der Geometrischen Algebra

Die wichtigste Operation für die KGA ist das *Geometrische Produkt*, welches durch den Operanden $*$ repräsentiert wird. Aus ihm lassen sich die Spezialfälle *Inneres Produkt* und *Äußeres Produkt* ableiten. Das Äußere Produkt OP wird durch den Operator $\wedge$ symbolisiert und besitzt die Eigenschaften Anti-Symmetrie, Linearität und Assoziativität. Geometrisch gibt das OP den Unterraum wider, der von den Räumen seiner beiden Operanden aufgespannt wird. Hieraus ergibt sich als weitere wichtige Eigenschaft, dass das OP eines Operanden mit sich selbst 0 ergibt. Um den Zusammenhang von Geometrie und Algebra ersichtlich zu machen, soll dieser Punkt kurz algebraisch bewiesen werden. Auf Grund der Antisymmetrie-Eigenschaft des OP gilt für zwei Operanden der KGA $(a \wedge b) = -(b \wedge a)$ und somit für einen einzelnen Operanden a: $(a \wedge a) = -(a \wedge a)$. Wie leicht zu erkennen ist, erfüllt nur die 0 diese Bedingung. Das Innere Produkt (IP), welches durch den Operanden . darstellt wird, dient als Metrik der KGA und bietet somit die Möglichkeit den Abstand zwischen manchen Objekten bestimmen zu können. Es lässt sich zeigen, dass der euklidische Abstand d zweier konformer Punkte P_1 und P_2 in folgendem Zusammenhang mit dem Inneren Produkt steht:

$$d = -2(P_1.P_2)$$

Für zwei Geraden oder zwei Ebenen ist das IP ein Indikator auf den Schnittwinkel der beiden Objekte. Ist es 1, so stehen die beiden Objekte senkrecht aufeinander, so dass zum Beispiel das IP von zwei der drei Basisvektoren e_1, e_2, e_3 stets 1 ist. Es ist somit eine Verallgemeinerung des Skalarproduktes der LA. Die beiden speziellen Punkte e_0 und e_∞ besitzen die Eigenschaften $e_\infty.e_\infty = 0$, $e_0.e_0 = 0$ und $e_\infty.e_0 = -1$.

2.3 Blades

Als *Blades* bezeichnet man die Kombinationen von Basisvektoren mittels des OPs. Für die oben genannten fünf Basisvektoren der KGA existieren 2^5 mögliche Kombinationen, also 32 Blades. Die Anzahl der verwendeten Basisvektoren definiert den *Grad* eines Blades. Die Basisvektoren selbst sind daher Blades mit dem Grad 1, ein Blade mit Grad 0 stellt einen Skalar dar. Das Grad-5-Blade wird *Pseudoskalar* genannt und besitzt die Eigenschaft, dass sein OP mit allen anderen möglichen Blades 0 ergibt.

2.4 Vektoren und Multivektoren

Die Objekte der GA werden mittels sogenannter *Vektoren* bzw. *Multivektoren* beschrieben. Diese bestehen aus gewichteten Summen einzelner Blades. Für die vorliegende Arbeit wird die Terminologie aus [DFM07a, S. 45ff] übernommen: Kommen in der Summe lediglich Blades des gleichen Grads k vor, so wird dieses Konstrukt als *k-Vektor* bezeichnet. Treten dagegen in der Summe Blades mit verschiedenem Grad auf, so handelt es sich um einen *Multivektor*. Darstellen lassen sich als primitive Objekte, welche beim Raytracing auch benötigt werden:

- **Punkt**: $P = \mathbf{x} + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0$, wobei es sich bei $\mathbf{x}$ um einen dreidimensionalen Vektor der Form $\mathbf{x} = x_0 * e_1 + y_0 * e_2 + z_0 * e_3$ handelt.
- **Kugel**: $S = P - \frac{1}{2}\mathbf{r}^2 e_\infty$, wobei P einen konformen Punkt im Sinne der obigen Definition darstellt und r der Radius der beschriebenen Kugel ist. Beim Betrachten der Vektoren für Kugel und Punkt fällt auf, dass es sich in der KGA bei Punkten um Kugeln mit dem Radius 0 handelt, was der geometrischen Intuition entspricht. In der LA dagegen lässt sich dieser Zusammenhang nicht herstellen.
- **Ebene**: $\pi = \mathbf{n} + d * e_\infty$, wobei $\mathbf{n}$ eine 3D Richtung repräsentiert, die durch $\mathbf{n} = n_x * e_1 + n_y * e_2 + n_z * e_3$ festgelegt ist. Der Skalar d ist der Abstand der Ebene zum Ursprung des Koordinatensystems.
- **Gerade**: $R = *(P_1 \wedge P_2 \wedge e_\infty)$, hierbei ist $*$ der *Dualitätsoperator*, der weiter unten aufgegriffen wird. P_1 und P_2 sind konforme Punkte; e_∞ der Basisvektor.
- **Punktpaar**: $PP = *(R \wedge S)$, wobei R eine Gerade und S eine Kugel ist. Ein Punktpaar entsteht, wenn ein Strahl eine Kugel schneidet und kennzeichnet seine Ein- und Austrittsstelle.

Bei Punkt, Kugel und Ebene handelt es sich somit um 1-Vektoren, bei Punktpaar und Gerade dagegen um 2-Vektoren.

Davon zu unterscheiden ist die Definition des *Nullvektors (null vector)*. Nullvektoren sind Vektoren, deren Inneres Produkt mit sich selbst Null ergibt [DFM07b, S. 587]. Punkte und Kugeln sind Beispiele für Nullvektoren.

Für Details zu weiteren Objekten und ihren Eigenschaften sei auf [Hil06, S. 19-21] verwiesen.

Eine weitere in den Raytracern ausgenutzte Eigenschaft der KGA ist die Tatsache, dass skalierte Multivektoren weiterhin das gleiche Objekt beschreiben. Ein konformer Punkt P repräsentiert

somit den gleichen Punkt wie der Punkt $P_2 = 2 * P^1$. Ein Punkt mit einem e_0-Wert von 1 heißt *normalisiert*, gleiches gilt für Kugeln.

An dieser Stelle lässt sich festhalten, dass alle geometrischen Objekte, die für das Raytracing benötigt werden, direkt in KGA abgebildet werden können. Aus implementierungstechnischer Sicht kann außerdem erkannt werden, dass sich jedes Objekt als Tabelle darstellen lässt, welche die Zuordnung der einzelnen Blades mit einem Koeffizienten angibt. Blades, die im Multivektor nicht vorkommen, erhalten den Koeffizienten 0. Tabelle 2.1 zeigt ein Beispiel einer derartigen Repräsentation für eine Kugel mit Mittelpunkt $(1, 2, 3)$ und einem Radius von 2.

Blade	Value
e_1	1
e_2	2
e_3	3
e_0	1
e_∞	5
Rest	0

Tabelle 2.1.: Tabellarische Darstellung einer Kugel

Einzelne Einträge dieses Multivektors sind voneinander unabhängig. Diese Unabhängigkeit ist für die vorliegende Arbeit deshalb wichtig, weil sie es ermöglicht die einzelnen Einträge parallel zu berechnen, was sich auf entsprechenden Architekturen, wie zu zeigen ist, effizient realisieren lässt. Prinzipiell ist es somit möglich jedes Objekt als ein Array mit 32 Werten zu beschreiben. Nachteil dieses Ansatzes ist, dass bei den oben erwähnten Objekten der überwiegende Teil der Koeffizienten 0 ist, so dass bei einer solchen Repräsentation viel Speicherplatz durch die nicht gebrauchten Koeffizienten belegt wird. Vorteil dagegen ist, dass für alle denkbaren Objekte derselbe Datentyp verwendet werden kann und dass alle Operationen, wie die Produkte der Algebra, nur einmal implementiert werden müssen. Um allerdings eine möglichst hohe Berechnungsgeschwindigkeit zu erreichen, ist es sinnvoller, sowohl für jedes Objekt einen speziellen Datentyp mit optimiertem Speicherverbrauch zu erstellen, als auch einzelne Operationen auf diesen Datentypen gezielt zu optimieren. Daher wird diese Vorgehensweise für die Implementierungen gewählt. Auch in Gaigen2 [Fon06] werden verschiedene Datentypen statt einem generellen Objekt verwendet.

2.5 IPNS und OPNS

Eine wichtige Entscheidung für den Übergang von der Clifford Algebra zur anschaulichen Geometrischen Algebra ist die Frage, wie die Multivektoren visualisiert werden können. Hier gibt es im Prinzip zwei Möglichkeiten. Zum einen den *Inner Product Null Space (IPNS)* und zum anderen den *Outer Product Null Space (OPNS)*. Beide beantworten die Frage, wie ein Multivektor graphisch interpretiert wird, unterschiedlich. Zur Definition wird ein beliebigen Nullvektor x und der darzustellende Multivektor v herangezogen. Im Inner Product Null Space (IPNS) muss daraufhin die Gleichung:

[1] Der Beweis hierzu wird in [DFM07b, S. 362] erbracht.

$$v \cdot x = 0$$

gelöst werden. Die Lösungen sind alle Punkte x, deren Inneres Produkt mit dem Multivektor 0 ergibt, woraus der Name des IPNS resultiert. Geometrisch bedeutet dies folgendes: Es werden mit der Gleichung alle Punkte im Raum beschrieben, die nicht auf v liegen. Beschreibt v zum Beispiel eine Ebene $\pi = \mathbf{n} + d * e_\infty$ nach der oben gegebenen Definition, so repräsentiert die IPNS-Gleichung alle Punkte, die nicht auf der Ebene liegen, eine sogenannte *Duale Ebene* oder *dual plane*. Durch sie ist die eigentliche Ebene eindeutig festgelegt.

Der Outer Product Null Space (OPNS) ist das genaue Gegenstück dazu. Dort muss die Gleichung

$$v \wedge x = 0$$

gelöst werden, also alle Punkte x gefunden werden, deren Äußeres Produkt mit v 0 ergibt. Geometrisch sind so alle Punkte beschrieben, die auf der Ebene p liegen, also die Ebene selbst. Tiefergehende mathematische Grundlagen zu IPNS und OPNS können in [PH03, 17-22] gefunden werden. Sie sind für das Verständnis der vorliegenden Arbeit aber nicht notwendig. Die obigen Objektdefinitionen gelten für den IPNS.

Wie sich aus den Definitionen der beiden Räume erkennen lässt, besteht zwischen ihnen ein enger Zusammenhang, der am Beispiel der Ebene sichtbar wird. Sie sind *dual* zueinander und somit ist es möglich, von einem zum anderen überzugehen. Die Transformation von einer Darstellungsart in die andere erfolgt mittels des *Dualitätsoperators* $*$, der bei den Objektdefinitionen bereits verwendet wurde. Die Dualisierung eines Multivektors besteht in einer Division des Vektors durch den Pseudoskalar I bzw. in der Multiplikation mit dem Inversen von I. Technisch lässt sich eine Dualisierung sehr einfach durch Indexvertauschungen realisieren, so dass keine weiteren Operationen anfallen.

3 Theorie

Dieses Kapitel fasst die notwendige Theorie zusammen, die für das Verständnis der praktischen Implementierungen und der Untersuchung von Nöten ist. Der Vorgang des Raytracens wird beschrieben und die theoretische Leistung etablierter Verfahren mit den neuen Algorithmen der Geometrischen Algebra verglichen. Zuerst wird kurz die nötige Hard- und Software präsentiert.

3.1 Eingesetzte Software und ihre Benutzung

Dieser Abschnitt soll die wichtigen Softwaretools und ihre Verwendung bei der Betrachtung des Raytracing-Vorgangs erläutern.

3.1.1 CluCalc und Gaalop

In einem ersten Schritt zu dieser Arbeit werden theoretische Überlegungen und Untersuchungen mittels der beiden Tools *CluCalc* und *Gaalop*[1] vorgenommen, um bereits an dieser Stelle abschätzen zu können, in wie weit ein effizientes Raytracing von Dreiecken und Kugeln möglich ist.

Bei CluCalc handelt es sich um ein von Christian Perwass erstelltes Werkzeug zur Visualisierung und Berechnung von GA [Per11]. Zur Eingabe der Algorithmen steht die selbst entwickelte Skriptsprache *CluScript* zur Verfügung. Ein graphisches, sowie ein textbasiertes Fenster zeigen dabei jeweils die Ausgaben der durchgeführten Berechnungen. Eine kurze Einführung in das Programm und CluScript bietet [Per04]. Besonders vorteilhaft für die vorliegende Untersuchung von Raytracing-Verfahren ist die Kombination von graphischer und textueller Ausgabe, da je nach aktuell zu betrachtendem Algorithmus-Schritt mal die eine, mal die andere Ausgabeform nötig bzw. besser geeignet ist.

Gaalop (*Geometric Algebra ALgorithms OPtimizer*) [Hil11] ist eine an der TU Darmstadt entwickelte Software, welche es ermöglicht aus CluCalc-Skripten, unter Zuhilfenahme von Maple [Map11] und der Clifford Library [Abl11], optimierten C++-Code zu erzeugen. Im Verlauf dieser Arbeit wurde Gaalop außerdem noch um die Möglichkeiten erweitert direkt Verilog und OpenCL-Code zu erzeugen. Die erste dieser beiden Möglichkeiten wird kurz angetestet und der erzeugte Code mit den von Hand entwickelten Verilog-Implementierungen verglichen. Hierzu sei auf Kapitel 6.3.1 verwiesen.

[1] Die beiden Programme können auf `http://www.clucalc.info/` und `http://www.gaalop.de` heruntergeladen werden.

3.1.2 Untersuchung der Operationenzahl

Um bereits vor den ersten GA-C++-Implementierungen eine Vorstellung vom Unterschied im Rechenaufwand von LA und GA zu erhalten, werden die einzelnen Schritte der Raytracing-Algorithmen in CluScript formuliert und mittels Gaalop in C++ übersetzt. Anschließend werden sie optimiert und auf die von ihnen benötigten primitiven Operationen (Additionen, Multiplikationen, Divisionen, Quadratwurzeln) heruntergebrochen. Deren Anzahl wird bestimmt und gegen bereits vorhandene oder anhand von Pseudocode implementierte LA-Algorithmusstücke gestellt. Der letzte Teil dieses Kapitels gibt eine Zusammenfassung über die dabei gewonnenen Erkenntnisse.

3.2 Genutzte Hardware

Da für die Interpretation der Ergebnisse in den späteren Kapiteln das Verständnis der Architektur der einzelnen Komponenten nötig ist, wird diese für die wichtigen Hardwarebauteile an dieser Stelle kurz zusammengefasst. GPU-Rendering wird auf zwei verschiedenen Grafikkarten getestet; die C++-Raytracer werden auf drei unterschiedlichen Prozessoren ausgeführt.

3.2.1 Grafikkarten

Die Grafikkarten sind die wichtigste Komponente für die OpenCL-Raytacer. Auch wenn diese ebenfalls auf dem Prozessor ausgeführt werden können, sind die Grafikkarten auf Grund der deutlich höheren Ausführungsgeschwindigkeit die interessantere Plattform. Die Kenntnis ihrer Architektur ist entscheidend, um ihre Leistungsfähigkeit nachvollziehen zu können.

AMD Radeon HD5770

Die ältere und schwächere der beiden genutzten Grafikkarten, die für die OpenCL- Implementierungen und Messungen verwendet wird, ist AMDs HD5770. Diese basiert auf der *Juniper* GPU (*RV840*), welche eine beschnittene Variante der *Cypress*-Grafikprozessoren (R*V870*) ist. Der Grafikchip taktet mit 850 MHz. Seine Architektur ähnelt stark den Vorgänger-GPUs aus der *RV770*-Serie, welche in [Deu10, S. 14ff] beschrieben wird. Der Juniper besteht ebenfalls aus 10 *SIMD-Einheiten*, die je nach Sprache und Quelle auch *Engines* oder *Blöcke* genannt werden. Auch seine SIMD-Einheiten sind mit vier Textureinheiten zur Anbindung an den VRAM, lokalen Registern und *Local-Data-Share* (LDS) zum Austausch von Informationen zwischen Work-Items innerhalb einer Work-Group (s.h. dazu den folgenden Abschnitt) ausgestattet. An dieser Stelle sei angemerkt, dass der in [Deu10, S. 14] angesprochene Nachteil des langsamen LDS-Zugriffes, mit der Einführung der HD5XXX [2] Serie beseitigt ist, da diese Chips nun das Shadermodell 5 unterstützen.

Eine SIMD-Einheit beinhaltet 16 Untereinheiten, die von AMD *Stream Cores* oder *Shadercores* genannt werden. Diese wiederum bestehen aus fünf Recheneinheiten die als *Streamprocessors*

[2] Umfasst die GPUs Cypress, Juniper, Redwood und Cedar

oder *AMD Processing Units* bezeichnet werden. Wie erkennbar ist, ergibt sich aus den verschiedenen Benennungen der Elemente die Gefahr von Verwirrungen und Verwechslungen[3]. Daher verwendet diese Arbeit folgende Bezeichnungen:

> Auf der Juniper GPU befinden sich 10 *SIMD-Einheiten*, welche sich jeweils in 16 *Shadercores* untergliedern. Ein Shadercore besteht aus fünf *Streamprocessors*.

Der Aufbau eines Shadercores ist im Vergleich zum RV770 gleich geblieben. Sie werden aus vier gleichförmigen, kleineren Recheneinheiten und einer *Special-Function-Unit (SFU)* gebildet (5D). Die SFU dient zur Berechnung von komplexeren Funktionen wie Sinus oder Wurzelberechnungen. Jeder Shadercore verfügt außerdem über ein eigenes kleines Register und eine *Branching-Unit (B-Unit)* um Verzweigungen im Code behandeln zu können. Insgesamt befinden sich auf der HD5770 somit 800 Streamprocessors, die auf 160 Shadercores verteilt sind.

AMD Radeon HD6970

Die HD6970 basiert auf der *Cayman* GPU, welche mit 880 MHz getaktet ist. Da es sich bei der, in dieser Arbeit verwendeten Grafikkarte um eine freigeschaltete[4] HD6950 handelt, beträgt ihr Takt nur 800 MHz. Der Cayman besitzt mit 24 im Vergleich zu HD4850/HD5770 mehr als doppelt so viele SIMD-Einheiten. Diese werden, wie bei den Vorgängern auch, aus 16 Shadercores, sowie Registern, vier Textureinheiten und LDS gebildet. Die entscheidende Veränderung befindet sich allerdings innerhalb der Shadercores, da die 5D- auf eine 4D-Architektur abgeändert wurde. Die SFU entfällt vollständig und es verbleiben die vier gleichartigen Rechenwerke. Die Berechnungen, die bei den Vorläufern auf der SFU abgearbeitet werden, übernehmen im neuen Design drei der vier verbleibenden Rechenwerke. Das lokale Register sowie die B-Unit verbleiben im Shadercore. Der Cayman verfügt somit über 384 Shadercores und 1536 Streamprocessors.

3.2.2 Prozessoren

Als Prozessoren kommen folgende drei Modelle zum Einsatz:

1. **Intel Core 2 Quad Q6600**: Die *Kentsfield*-CPU verfügt über vier physikalische Kerne, die mit jeweils 2,4 GHz takten. Es werden SSE3-Befehlssätze unterstützt.
2. **AMD Athlon X2 250**: Die *Regor*-CPU ist ein Zweikerner der auf jeweils 3,2 GHz übertaktet ist. Er unterstützt SSE4a-Befehlssätze.
3. **Intel Core 2 Duo P7350**: Ein mobiler Zweikern-Prozessor mit SSE3-Unterstützung. Der Takt der Kerne beträgt jeweils 2 GHz.

Tabelle 3.1 zeigt alle drei Testsysteme im Überblick.

[3] Interessierte Leser können sich selbst ein Bild der diffusen Namensgebung machen, wenn sie die Review-Artikel zu HD4850, HD5870, HD5770 und HD6970 auf nachfolgenden Seiten vergleichen: `http://www.tomshardware.com`, `http://www.computerbase.de`, `http:www.3dcenter.org`, sowie `http:www.ht4u.org`

[4] Durch einen VGABios-Flash wurden deaktivierte Einheiten der HD6950 aktiviert. Der Takt wurde auf HD6950 Referenztakt belassen

	Testsystem 1 (TS1)	Testsystem 2 (TS2)	Testsystem 3 (TS3)
Prozessor	Intel Q6600	AMD Athlon 2 X2 250	Intel P7350
Grafikkarte	AMD HD6970	AMD HD5770	NVIDIA 9600M GT
Arbeitsspeicher	4x1 GB DDR2	2x1 GB DDR3	2x2 GB DDR2
Betriebssystem	Windows 7	Windows Vista	Windows Vista

Tabelle 3.1.: Die drei Testsysteme, die im Rahmen der Arbeit Verwendung finden

3.2.3 OpenMP, OpenCL und AMDs APP SDK

Bei dem für die C++-Raytracer-Implementierung verwendeten OpenMP, handelt es sich um eine Sammlung von Erweiterungen für bestehende Programmiersprachen. Verfügbar sind OpenMP-Implementierungen im Moment für C++ und Fortran. OpenMP basiert auf Compiler-Direktiven und lässt sich mit wenigen Codezeilen in ein vorhandenes C++-Projekt integrieren. Um über eine bereits existente *for*-Schleife zu parallelisieren, ist es in Microsofts Visual Studio notwendig den entsprechenden Header mit *#include <omp.h>* einzubinden. Außerdem muss dem Compiler durch ein *#pragma omp parallel for* unmittelbar über der *for*-Schleife mitgeteilt werden, dass die Anweisungen innerhalb des Schleifenkörpers parallel auszuführen sind. In den Compiler-Optionen muss außerdem OpenMP aktiviert werden. Um weitere Punkte, wie die Erzeugung von Threads und deren Verwaltung, kümmert sich der Compiler selbstständig.

OpenCL (Open Computing Language) ist eine Programmiersprache, die zur plattformunabhängigen Entwicklung von parallelen Programmen dient. Genau genommen ist OpenCL nicht nur eine Sprache, sondern ein Framework, welches Programmiersprache, API, Libraries und eine Laufzeitumgebung bietet [Khr11, S. 21]. Die Syntax der Sprache ist C-ähnlich [5] und der Code auf allen aktuellen Grafikkarten von AMD und NVidia sowie den Prozessoren von AMD und Intel ausführbar, sofern die passenden Treiber der genannten Hersteller auf dem entsprechenden System installiert sind. Die Art, wie Programme in der OpenCL-Umgebung ausgeführt werden, zeigt bereits, dass es das Ziel ist, parallele Architekturen möglichst gut zu unterstützen. Die nachfolgenden Beschreibungen versuchen wie in [Deu10, S. 16f] eine Beziehung zwischen den Bezeichnungen für AMDs Hardwarearchitektur und den OpenCL-Termini herzustellen. Grundlage der Ausführungen bildet AMDs APP Guide [Adv11, Chapter 1]. Auch sollen die Vorteile der Architektur für Raytracing-Algorithmen aufgezeigt werden.

Um Parallelität nutzbar zu machen, unterteilt OpenCL seine Ausführung in sogenannte *Work Items*. Jedes Work Item arbeitet denselben Code auf unterschiedlichen Daten ab. Die dabei ausgeführten Befehle müssen nicht identisch sein, falls der gemeinsame Code Verzweigungen enthält, die auf Grund der verschiedenen Daten auf unterschiedliche Weise durchlaufen werden. Jedes Work Item besitzt einen eindeutigen Index. Work Items werden zu *Work Groups* zusammengefasst, in welchen die Items in einem ein-, zwei- oder dreidimensionalen Array angeordnet werden. Items einer Group können untereinander Daten austauschen.

Jedes OpenCL Programm muss auf einem *OpenCL Device* ausgeführt werden, im vorliegenden Fall die Grafikkarte selbst. Das Device ist mit einem *Host* verbunden, der für ihn einen *Context* erstellt, in dessen Rahmen eine Kommunikation möglich ist. Auch ist der Context der Gültig-

[5] Daher heißt die Sprache eigentlich korrekterweise *OpenCL C*, aber diese Unterscheidung zwischen Sprache und Framework wird selbst in den OpenCL Spezifikationen [Khr11] nicht konsequent beibehalten

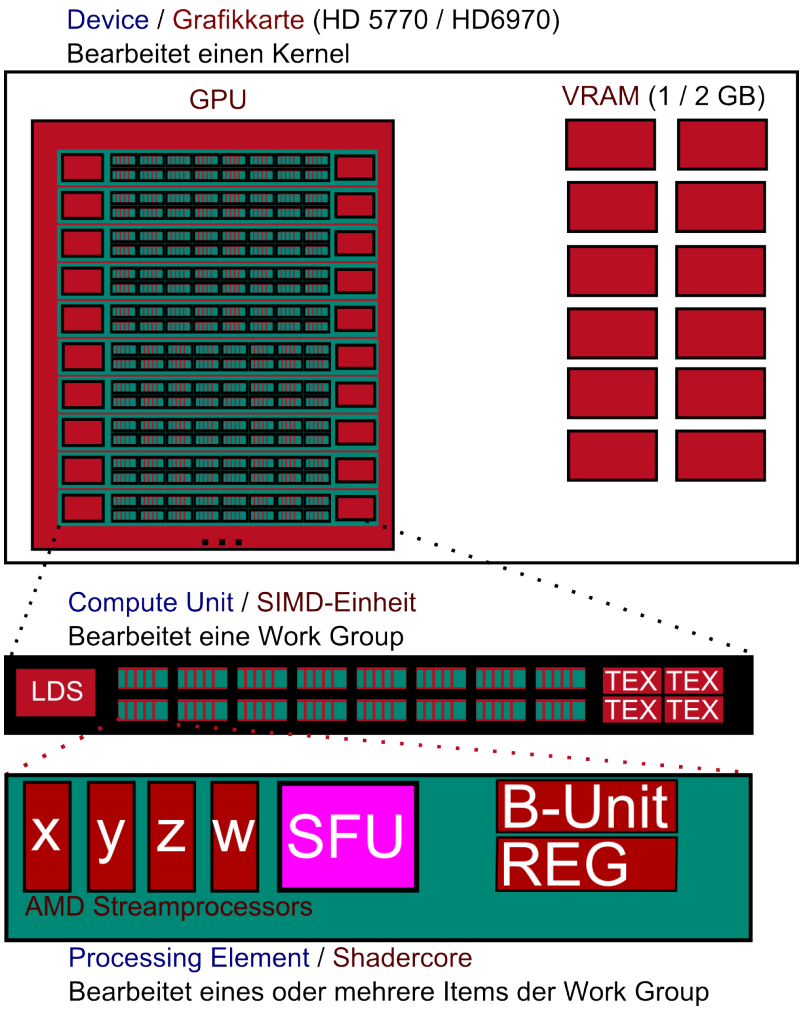

Abbildung 3.1.: OpenCL Ausführungshierarchie und Architektur der verwendeten AMD-Grafikkarten. In Blau sind die OpenCL Termini angegeben, in Rot die Entsprechungen auf AMDs Hardware. Schwarzer Text sind Erläuterungen und weitere Daten. Die pink unterlegte SFU entfällt bei der HD6970. Die Zahl der SIMD-Einheiten ist von der Karte anhängig. Grafik verändert, nach [Deu10] und [Adv11].

keitsbereich für die in ihm definierten Variablen. Durch *Commands* veranlasst der Host den Device zu Berechnungen und steuert deren Ausführung [Khr11, S. 23]. Die Unterprogramme, die an die Devices gesendet werden heißen *Kernel*.

Ein Device selbst ist eine Ansammlung von *Compute Units* (CUs), denen jeweils eine Work Group zugeordnet wird. Die Compute Units der AMD Grafikkarte sind die SIMD-Einheiten. Compute Units selbst bestehen aus einem oder mehreren *Processing Elements* (PEs), im vorliegenden Fall also die Streamcores der Radeons. Abbildung 3.1 ordnet all diese Begriffe und setzt sie mit der Radeon-Architektur in Verbindung.

Ein letzter einzuführender Begriff ist *Wavefront*. Eine Wavefront umfasst alle Items einer Group, die zusammen auf einer SIMD-Einheit ausgeführt werden [Adv11, S. 1-7ff]. Damit die langsamen Speicherzugriffe die Berechnung möglichst wenig verzögern, wird versucht diese durch eine gepipelinete Ausführung von Anweisungen zu überdecken. Je nach Grafikkarte besitzt die-

se Pipeline auf den Streamcores verschieden viele Stufen. Im Fall der HD5770 und HD6970 sind es jeweils vier. Das bedeutet, dass pro SIMD-Einheit 4*16 = 64 Items einer Group gleichzeitig aktiv sein können. Je nach Umfang des Codes, können unterschiedlich viele Wavefronts gleichzeitig instanziiert werden, da die vorhandenen Ressourcen limitieren.

Es werden von OpenCL bzw. AMDs Hardwaredesign für die Umsetzung und Ausnutzung der Parallelität des GA-Raytracings vor allem zwei Wege geboten:

1. OpenCL unterstützt sowohl das *Data Parallel Programming Model* als auch das *Task Programming Model* [Khr11, S. 28f]. Ersteres nutzt die Möglichkeit die gleichen Berechnungen parallel auf verschiedene Daten anzuwenden. Diese werden von Raytracing-Algorithmen in mehrfacher Hinsicht geboten. Die Pixel lassen sich unabhängig voneinander berechnen oder Bounding-Spheres und Dreiecke unabhängig auf Schnitt überprüfen. Das *Task Programming Model* spielt für diese Arbeit dagegen keine Rolle.
2. Der zweite Punkt ist AMDs Nutzung von *Very Long Instruction Words (VLIW)*, die durch den Compiler erzeugt werden. Dabei versucht dieser Parallelität innerhalb eines einzelnen Work Items auszunutzen (*Instruction Level Parallelism*), um alle Streamprocessors auszulasten. Theoretisch kommt dies den Operationen der GA mit ihren unabhängigen Multivektoreinträgen entgegen, da sich diese auf die unabhängigen Rechenwerke aufteilen lassen. Durch diese Vorgehensweise vereinfacht sich für AMD die Hardwarerealisierung, da die verfügbaren Ressourcen nicht dynamisch zugeteilt werden müssen, wie dies auf NVidia-Grafikkarten geschieht. Dafür liegt die Verantwortung für eine hohe Geschwindigkeit auf dem Compiler. Dies konnte im Rahmen der Arbeit mehrfach beobachtet werden, da sich mit jeder Treiberversion [6] die Messergebnisse teils deutlich veränderten [7]. Innerhalb der letzten drei bis vier Treiberversionen blieb die Geschwindigkeit relativ konstant, wenn auch nicht identisch. Alle Messungen der Arbeit erfolgen mit dem Catalyst-Treiber in der Version 11.8.

3.3 Der Ablauf des Raytracings

Raytracing ist eine schon lange bekannte, beliebte Methode der Bilderzeugung. Ihre Beliebtheit resultiert vor allem aus ihrem simplen Grundprinzip. Ihre Funktionsweise lässt sich ohne großen Aufwand bildlich darstellen und einfach nachvollziehen. Es wird das Pixelgitter des zu erzeugenden Bildes betrachtet und von einem Augpunkt, der vor dieser Bildebene liegt, ein Sichtstrahl durch die einzelnen Pixel auf die Szene geschossen, die sich hinter der Bildebene befindet. Nun muss geprüft werden, welche Objekte in der Szene von diesem Strahl getroffen werden und der nächstgelegene Schnittpunkt bestimmt werden. Für diesen wird anschließend getestet, von welchen Lichtquellen er beleuchtet wird, oder ob er im Schatten liegt. Bei reflektierenden Oberflächen wird außerdem geprüft, ob sich andere Objekte im Ausgangsobjekt spiegeln, auf welchem der gefundene Schnittpunkt liegt. Aus diesen Informationen wird schließlich der Farbwert des aktuellen Pixels bestimmt. Somit erhält man beim Raytracing automatisch Spiegelungen und Schattenwurfberechnungen. Auch lässt sich hieran erkennen, wieso Raytracing nicht auf eine bestimmte Art der Szenenbeschreibung festgelegt ist. Es ist jede Ob-

[6] Eingesetzt wurden stets die neueste Version des Stream bzw. jetzt APP SDK, sowie die aktuellste Treiberversion (14 Treiberversionen zwischen dem Catalyst 10.9 und dem 11.8)

[7] Bis zu zweistelliger Prozentbereich, Schwankungen in beide Richtungen

jektbeschreibung möglich, für die es möglich ist, einen Schnittpunkt mit dem Sichtstrahl zu berechnen. Abbildung 3.2 zeigt alle Elemente des Raytracing-Vorgangs mit den in dieser Arbeit verwendeten Bezeichnungen.

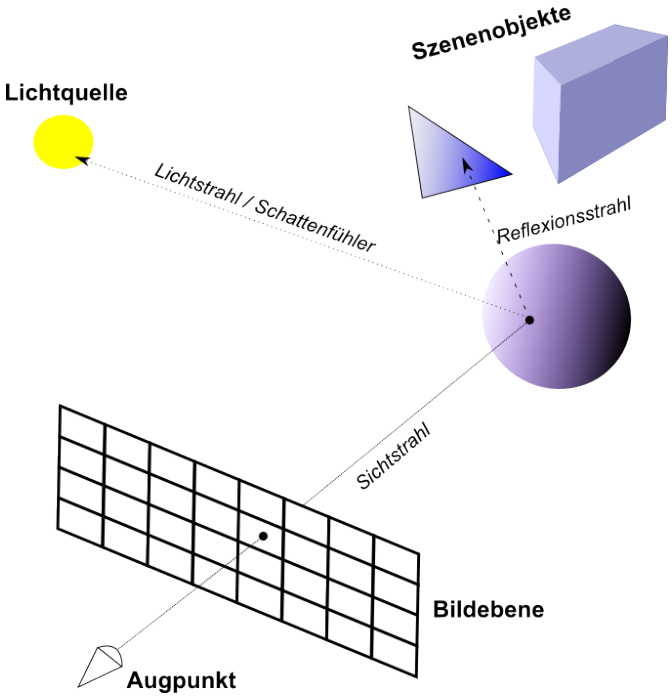

Abbildung 3.2.: Schematische Darstellung des Raytracing-Vorgangs. Wie im Text beschrieben, können verschiedene Szenenobjekte verwendet werden.

Um Missverständnisse zu vermeiden sei an dieser Stelle auch kurz der Begriff *Raytracing* definiert, da er in der Literatur nicht immer einheitlich verwendet wird. Unter *Raytracing* wird der vollständige Algorithmus verstanden, der sowohl Schattenfühler als auch Reflexionsstrahlen beinhaltet. Andere hierfür gebräuchliche Bezeichnungen sind *rekursives Raytracing* oder *echtes Raytracing*. Wie weit Strahlen verfolgt werden, legt die sogenannte Rekursionstiefe fest. Bei Rekursionstiefe 1 wird am Schnittpunkt mit dem Sichtstrahl der Reflexionsstrahl erzeugt. Für diesen wird geprüft, ob und wenn ja welches weitere Objekt von ihm getroffen wird. Bei Rekursionstiefe 2 würde am Schnittpunkt mit dem ersten Reflexionsstrahl ein weiterer Reflexionstrahl erzeugt und weiterverfolgt werden. Dies lässt sich beliebig tief schachteln. Die in dieser Arbeit verwendeten Raytracer arbeiten alle mit einer Rekursionstiefe von 1.

Im Gegensatz dazu steht das *Raycasting*, das auch *einfaches Raytracing* und *nicht-rekursives Raytracing* genannt wird. Hierbei wird lediglich der eigentliche Schnittpunkt berechnet und mittels Lichtquelleninformationen beleuchtet. Im Prinzip ist also ein Raycaster ein Raytracer mit Rekursionstiefe 0. Alle Raytracer werden auf ihre Geschwindigkeit und Bildqualität hin auch im Raycastingmodus untersucht. Die einzelnen Schritte, die für das Raytracen eines Bildes nötig sind, unterscheiden sich nicht zwischen Linearer und Geometrischer Algebra, so dass ein direkter Vergleich des Aufwandes möglich ist.

Beim Rendern eines Bildes mittels Raytracing werden die Pixel des Ergebnisbildes, wie beschrieben, getrennt voneinander berechnet. Somit ergibt sich die Gesamtzahl der nötigen primitiven Operationen zum Erzeugen des Bildes aus der Anzahl der Operationen, die zur Berechnung des

Farbwertes eines Pixels nötig sind, multipliziert mit der Menge der Pixel des Bildes. Der Farbwert eines Pixels lässt sich mittels folgendem Algorithmus berechnen:

1. Berechne für den aktuellen x- und y-Wert des Pixels den Sichtstrahl, der an dieser Stelle von der Kamera / dem Augpunkt durch die Bildebene geht.
2. Prüfe den erzeugten Strahl auf Schnitt mit den Objekten der Szene.
3. Finde den Schnittpunkt, welcher der Kamera / dem Auge am nächsten liegt.
4. Berechne für diesen Schnittpunkt den Farbwert mittels Beleuchtungs- und Punkteigenschaften. Beachte dabei mittels Schattenfühlern, ob ein gegebener Schnittpunkt nicht durch ein anderes Objekt im Schatten liegt.
5. Konstruiere den Reflexionsstrahl des Lichtes und führe mit ihm die Schritte 1-4 aus. Addiere die dabei gefundenen Farbwerte zu denen aus Schritt 4.

Was hierbei deutlich wird: Jede einzelne Pixelfarbwert-Berechnung ist unabhängig und es werden zu seiner Farbwertberechnung keinerlei Informationen aus den Berechnungen anderer Pixel benötigt. Somit ließen sich, entsprechende Hardware vorausgesetzt, sämtliche Pixel eines Bildes gleichzeitig berechnen. Ziel dieser Arbeit ist es, die dem Raytracing-Vorgang innewohnende Parallelität so weit wie möglich auszunutzen.

3.4 Vergleich der Schritte

Dieser Abschnitt vergleicht die Anzahl der nötigen Rechenoperationen innerhalb der einzelnen Teilschritte des Raytracing-Algorithmus zwischen LA und GA. Die Anzahl der Operationen im Fließtext wird zur besseren Übersicht immer als Ziffer angegeben, während weitere Zahlen bis zwölf ausgeschrieben sind.

3.4.1 Sichtstrahlenerzeugung

Für eine der frühen theoretischen Versionen des Raytracers war die Idee, die Sichtstrahlen auf der CPU zu erzeugen und sie von dort auf den Speicher des FPGA zu übertragen bzw. an den entsprechenden Algorithmus in OpenCL zu übergeben, der sie dann abarbeitet. Allerdings wirft dieses Vorgehen die Frage auf, wie die Reflexionsstrahlen und Schattenfühler erzeugt werden sollen. Wenn dies der FPGA bzw. OpenCL Kernel übernimmt, muss auf ihm dennoch die Logik zur Strahlenerzeugung untergebracht werden und er kann somit auch die primären Strahlen erzeugen. Wenn diese Berechnungen dagegen auf die CPU ausgelagert werden sollen, müssen die berechneten Schnittpunkte und Zwischenergebnisse zurück auf die CPU transferiert werden und die Menge der Sekundärstrahlen dann im weiteren Verlauf von dort aus zurück in den Speicher des FPGA bzw. der Grafikkarte geschrieben werden. Es erscheint daher nicht sinnvoll, einen Teil oder die gesamte Strahlenerzeugung auf die CPU auszulagern, da die Funktionalität zur Erzeugung dann entweder gleichzeitig auf CPU und FPGA/OpenCL untergebracht ist, oder viele Speicherzugriffe und Kommunikation von Nöten sind.

```
Ray::Ray(Camera* cam, int x, int y)
{
  // Zuerst Direction des Strahles erstellen
  Direction* dir = new Direction(cam, x, y);

  // Hier ist point6 eigentlich jeweils 1!
  // Daher die Multiplikation in für ray7,8 und 11 entfernen
  this->ray7 = -dir->direction4*point6;
  this->ray8 = dir->direction3*point6;
  this->ray9 = cam->pos->point4 * dir->direction3 -
    cam->pos->point3*dir->direction4;
  this->ray11 =-dir->direction2*point6;
  this->ray12 = -cam->pos->point4*dir->direction2 +
    cam->pos->point2 * dir->direction4;
  this->ray14 = cam->pos->point3*dir->direction2 -
    cam->pos->point2*dir->direction3;

  this->origin = cam->pos;
  delete dir;
}
```

Listing 3.1: Konstruktion eines konformen Strahles aus Punkt und Richtung

Strahlenerzeugung in GA

Die ursprüngliche Version der Sichtstrahlenerzeugung lief über konforme Punkte. Doch nach Betrachten der Gaalop-Ausgabe fällt auf, dass in den Berechnungen nur die ersten drei Komponenten des Multivektors benutzt werden, was in Listing 3.1 deutlich wird, da für den Sichtstrahl weder e_0 noch e_∞ benötigt werden. Die vierte Komponente e_∞ lässt sich bei Bedarf aus den ersten drei errechnen und die fünfte e_0 ist bei einem normalisierten Punkt immer 1. Somit müssen nur drei Werte für jeden Punkt gespeichert werden und auch die Strahlenerzeugung kann mittels eines 3D Punktes erfolgen.

Die zweite Komponente, neben seinem Ursprungspunkt, die für den Sichtstrahl benötigt wird, ist seine Richtung, die sich aus der aktuellen Pixelposition und den Kameraparametern bzw. der Bildebene ergibt. Die Berechnung der Richtung geschieht nach [DFM07a, S. 575]. Ihre Implementierung zeigt Listing 3.2. Berechnet werden müssen nur die x- und y-Richtung des Strahles, während die z-Richtung konstant 1 gesetzt werden kann.

Auch die Reflexionsstrahlen werden aus Punkt und Richtung konstruiert, da diese vorliegen. Die Information zur Berechnung der Lichtstrahlen/Schattenfühlern besteht dagegen aus zwei Punkten, die auf dem Strahl liegen. Um den Strahlenkonstruktor einheitlich zu halten, wird aus den beiden Punkten zuerst eine Richtung gebildet und mit dieser anschließend der Lichtstrahl erstellt.

Um den Sichtstrahl *eye* zu erzeugen sind 10 Multiplikationen und 5 Additionen notwendig.

```
Direction::Direction(Camera* cam, int x, int y)
{
  this->direction2 = x * cam->pixelWidth -
    0.5f * cam->cameraFOVwidth;
  this->direction3 = y * cam->pixelHeight -
    0.5f * cam->cameraFOVheight;
  this->direction4 = 1;
}
```

Listing 3.2: Konstruktion der Sichtstrahlenrichtung

Strahlenerzeugung in LA

Für die LA wird das gleiche Verfahren angewandt. Da die konformen Punkte der GA durch die Optimierungen zu 3D Punkten degenerieren, ist der Code zur Richtungserzeugung zwischen beiden Algebren identisch. Deshalb fallen auch hier vier Multiplikationen und zwei Additionen an. Der eigentliche Strahl in LA besteht nur aus Aufpunkt und Richtung, so dass an dieser Stelle nichts weiteres mehr gerechnet werden muss, was Tabelle 3.2 zusammenfasst.

	GA	LA	GA/LA
Erzeugung des Sichtstrahls			
Multiplikationen	10	4	2,5
Additionen/Subtraktionen	5	2	2,5

Tabelle 3.2.: Operationenvergleich bei Strahlenerzeugung

3.4.2 Schnitt mit Kugeln

Beim kugelbasierten Raytracer, der in Abschnitt 4 beschrieben wird, bilden Kugeln die Szenenobjekte. Für die Wichtigkeit des effizienten Kugelschnitts gilt in diesem Fall das, was über Dreiecke als Szenenobjekte in Abschnitt 3.4.3 gesagt wird. Aber auch in den, im Rahmen dieser Arbeit entwickelten, Raytracern, deren Szenenbeschreibungen auf Dreiecken basieren, spielt der Schnitttest mit Kugeln eine wichtige Rolle. Dort kommen Hüllkugeln zum Einsatz (vgl. Abschnitt 5.2), für die es wichtig ist, sie performant auf Schnitt mit dem Sichtstrahl prüfen zu können. Nicht relevant ist hierbei allerdings die genaue Lage des Schnittpunkts. Dieser Abschnitt beleuchtet sowohl Schnitttest als auch Schnittpunktbestimmung für die beiden eingesetzten Algebren.

Kugel-Schnitt in LA

Für die LA werden zwei verschiedene Verfahren implementiert und getestet, welche nun vorgestellt werden.

Verfahren von Stoll und Gumhold

Das erste Verfahren basiert auf einer Untersuchung von Stoll und Gumhold [SGS06], da es sich bei der Kugel um eine Quadrik handelt, deren effizientes LA-Raytracing die beiden Autoren für einen Raytracer untersuchen. Hierbei stellen die beiden fest, dass das Verfahren auf Grafikkarten sehr schnell ausgeführt werden kann, da die GPUs die enthaltenen Matrizen- und Vektorrechnungen sehr effizient bearbeiten können. Die Implementierung erfolgt mit der *OpenGL Shading Language (GLSL)*, welche hardwarenäher ist als OpenCL. Daher ist es von Interesse, das Verfahren in OpenCL umzusetzen, um zu prüfen, ob die in ihm enthaltene Parallelität auch mit dieser Sprache in Kombination mit modernen Grafikeinheiten ausgenutzt werden kann. Hier soll nun bestimmt werden, wie viele elementare Rechenoperationen nötig sind, um den angesprochenen Schnitt der beiden Objekte Strahl $\mathbf{r}$ und Kugel s zu finden. Eine Kugel lässt sich als Quadrik in folgender Form angeben:

$$\begin{aligned} f_I(x,y,z) = {} & k_{000} + k_{100} * x + k_{010} * y + k_{001} * z + \\ & k_{110} * x * y + k_{011} * y * z + k_{101} * x * z + \\ & k_{200} * x^2 + k_{020} * y^2 + k_{002} * z^2 \end{aligned} \tag{3.1}$$

Es ist aber auch eine Darstellung als 4x4 Matrix möglich, was den Recheneinheiten der Grafikkarte entgegen kommt:

$$\tilde{\mathbf{r}}^t \tilde{\mathbf{Q}} \tilde{\mathbf{x}} = 0 \qquad \text{wobei } \tilde{\mathbf{r}} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{3.2}$$

mit

$$\tilde{\mathbf{Q}} = \begin{pmatrix} k_{200} & \frac{1}{2}k_{110} & \frac{1}{2}k_{101} & \frac{1}{2}k_{100} \\ \frac{1}{2}k_{110} & k_{020} & \frac{1}{2}k_{011} & \frac{1}{2}k_{010} \\ \frac{1}{2}k_{101} & \frac{1}{2}k_{011} & k_{002} & \frac{1}{2}k_{001} \\ \frac{1}{2}k_{100} & \frac{1}{2}k_{010} & \frac{1}{2}k_{001} & k_{000} \end{pmatrix} \tag{3.3}$$

Für die Kugel werden einige der Einträge der Matrix 0 und sie hat folgende Form:

$$\tilde{\mathbf{Q}} = \begin{pmatrix} k_{200} & 0 & 0 & \frac{1}{2}k_{100} \\ 0 & k_{020} & 0 & \frac{1}{2}k_{010} \\ 0 & 0 & k_{002} & \frac{1}{2}k_{001} \\ \frac{1}{2}k_{100} & \frac{1}{2}k_{010} & \frac{1}{2}k_{001} & k_{000} \end{pmatrix} \tag{3.4}$$

Prinzipiell ließe sich sogar mittels einer Transformation der Quadrik auf Normalform nachfolgende Darstellung erreichen:

$$\widetilde{\mathbf{Q}} = \begin{pmatrix} k_{200} & 0 & 0 & 0 \\ 0 & k_{020} & 0 & 0 \\ 0 & 0 & k_{002} & 0 \\ 0 & 0 & 0 & k_{000} \end{pmatrix} \tag{3.5}$$

Doch das Verfahren hierzu (vgl. dazu [MV99, S. 345f]), benötigt deutlich mehr Operationen, als erforderlich sind, den Algorithmus mit der nicht transformierten Matrix ablaufen zu lassen, weshalb auf die Normalisierung verzichtet wird.

Der Sichtstrahl **r** besteht aus Aufpunkt **e** und Richtungsvektor **v**:

$$\mathbf{r}(\lambda) = \mathbf{e} + \lambda\mathbf{v} \tag{3.6}$$

Um den Schnittpunkt **p** zwischen dem Strahl und der Kugel (Quadrik) zu finden, setzt man den Sichtstrahl aus Gleichung 3.6 in die Matrixdarstellung aus Gleichung 3.2 ein und erhält dadurch Gleichung 3.7:

$$\begin{aligned} 0 &= (\mathbf{e} + \lambda\widetilde{\mathbf{v}}')^T \widetilde{\mathbf{Q}} (\mathbf{e} + \lambda\widetilde{\mathbf{v}}') \\ &= (\mathbf{e}^T + \lambda\widetilde{\mathbf{v}}'^T)(\widetilde{\mathbf{Q}}\mathbf{e} + \widetilde{\mathbf{Q}}\lambda\widetilde{\mathbf{v}}') \\ &= \mathbf{e}^T\widetilde{\mathbf{Q}}\mathbf{e} + \mathbf{e}^T\widetilde{\mathbf{Q}}\lambda\widetilde{\mathbf{v}}' + \lambda\widetilde{\mathbf{v}}'^T\widetilde{\mathbf{Q}}\mathbf{e} + \lambda^2\widetilde{\mathbf{v}}'^T\widetilde{\mathbf{Q}}\widetilde{\mathbf{v}}' \\ &= \lambda^2\widetilde{\mathbf{v}}'^T\widetilde{\mathbf{Q}}\widetilde{\mathbf{v}}' + 2\lambda * \mathbf{e}^T\widetilde{\mathbf{Q}}\widetilde{\mathbf{v}}' + \mathbf{e}^T\widetilde{\mathbf{Q}}\mathbf{e} \end{aligned} \tag{3.7}$$

Nun müssen die entsprechenden Werte für λ gefunden werden, was mit der bekannten Lösungsformel quadratischer Gleichungen (3.8) möglich ist:

$$\lambda_{\pm} = \frac{-\beta \pm \sqrt{\beta^2 - \alpha\gamma}}{\alpha} \tag{3.8}$$

Die Lösungsformel dient auch der Entscheidung, ob überhaupt ein Schnitt vorliegt. Wird die Diskriminante negativ, so trifft der Strahl die Kugel nicht. Bei einem Wert von 0 ist der Strahl eine Tangente der Kugel. Ist die Diskriminante positiv, schneidet der Sichtstrahl die Kugel in zwei Punkten.

Die Koeffizienten α, β und γ ergeben sich aus Gleichung 3.7 folgendermaßen:

$$\begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} \mathbf{v}^t\tilde{\mathbf{Q}}\mathbf{v} \\ \mathbf{e}^t\tilde{\mathbf{Q}}\mathbf{v} \\ \mathbf{e}^t\tilde{\mathbf{Q}}\mathbf{e} \end{bmatrix} \tag{3.9}$$

mit

$$\mathbf{e} = \begin{pmatrix} e1 \\ e2 \\ e3 \\ 1 \end{pmatrix} \qquad \mathbf{v} = \begin{pmatrix} v1 \\ v2 \\ v3 \\ 0 \end{pmatrix} \tag{3.10}$$

Zu beachten ist hier die 0 in der vierten Komponente des Vektors **v**.

Setzt man nun **v** und **e** in Gleichung 3.9 ein, so erhält man:

$\alpha = v_1^2 * k_{200} + v_2^2 * k_{020} + v_3^2 * k_{002}$
$\beta = v_1 * e_1 * k_{200} + v_2 * e_2 * k_{020} + v_3 * e_3 * k_{002}$
$\gamma = e_1^2 * k_{200} + e_2^2 * k_{020} + e_3^2 * k_{002} + k_{000}$

Zum Berechnen der Koeffizienten sind also $3 * 6 = 18$ Multiplikationen und $2 * 2 + 1 * 3 = 7$ Additionen notwendig. Für die Anwendung der Lösungsformel:

$$\lambda_{\pm} = \frac{-\beta \pm \sqrt{\beta^2 - \alpha\gamma}}{\alpha} \tag{3.11}$$

kommen 2 Multiplikationen, 1 Addition, 1 Subtraktion, 1 Division und 1 Quadratwurzel hinzu. Außerdem muss das kleinere der beiden λ ausgewählt werden, da dieses zu dem Schnittpunkt führt, der näher am Aufpunkt von r liegt.

Nun muss das gefundene λ noch in die Sichtstrahlgleichung 3.6 eingesetzt und der Schnittpunkt ausgerechnet werden. Da es sich hierbei um einen dreidimensionalen Punkt handelt sind dafür 3 Multiplikationen und 3 Additionen von Skalaren nötig, was aus folgender Gleichung zu erkennen ist:

$$\begin{pmatrix} e_1 \\ e_2 \\ e_3 \end{pmatrix} + \lambda * \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

Was bisher noch nicht berücksichtigt wurde, ist die Frage, wie man von der bekannten Mittelpunkt-Radius-Darstellung einer Kugel zur für dieses Verfahren benötigten Quadrikdarstellung kommt. Hierzu geht man einerseits von der Kugelgleichung

$$(x - x_o)^2 + (y - y_o)^2 + (z - z_0)^2 = r^2 \tag{3.12}$$

aus und betrachtet andererseits die Quadrikdarstellung

$$\begin{aligned} f_I(x, y, z) = k_{000} + k_{100} * x + k_{010} * y + k_{001} * z + \\ k_{200} * x^2 + k_{020} * y^2 + k_{002} * z^2 \end{aligned} \tag{3.13}$$

Nun wird Gleichung 3.13 umsortiert:

$$
\begin{aligned}
k_{000} = k_{200} * x^2 + k_{100} * x + \\
k_{020} * y^2 + k_{010} * y + \\
k_{002} * z^2 + k_{001} * z
\end{aligned}
\tag{3.14}
$$

Danach werden die drei einzelnen Zeilen jeweils quadratisch ergänzt:

$$
\begin{aligned}
k_{000} = k_{200} * x^2 + k_{100} * x + (\frac{k_{100}}{2})^2 - (\frac{k_{100}}{2})^2 + \\
k_{020} * y^2 + k_{010} * y + (\frac{k_{010}}{2})^2 - (\frac{k_{010}}{2})^2 + \\
k_{002} * z^2 + k_{001} * z + (\frac{k_{001}}{2})^2 - (\frac{k_{001}}{2})^2
\end{aligned}
\tag{3.15}
$$

An dieser Stelle wird von der Tatsache Gebrauch gemacht, dass k_{200}, k_{020} und k_{002} für eine Kugel immer 1 sind und anschließend werden die ergänzten Terme zusammengefasst, sowie überflüssige Summanden auf die rechte Seite gebracht:

$$
k_{000} + (\frac{k_{100}}{2})^2 + (\frac{k_{010}}{2})^2 + (\frac{k_{001}}{2})^2 = (x - \frac{k_{100}}{2})^2 + (y - \frac{k_{010}}{2})^2 + (z - \frac{k_{001}}{2})^2 \tag{3.16}
$$

Nach Koeffizientenvergleich und der Eigenschaft, dass die Werte k_{100}, k_{010} und k_{001} in der Matrix $\widetilde{\mathbf{Q}}$ mit dem Vorfaktor $\frac{1}{2}$ gespeichert werden, ergibt sich die Ergebnis-Matrix aus Gleichung 3.17.

$$
\widetilde{\mathbf{Q}} \quad = \quad \begin{pmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ -x_0 & -y_0 & -z_0 & -r^2 + x_0^2 + y_0^2 + z_0^2 \end{pmatrix} \tag{3.17}
$$

Zusammengefasst ergibt sich folglich für das Lösen des Schnittproblems mittels des Stoll/Gumhold-Verfahrens in LA folgender Rechenaufwand:

Additionen	16
Multiplikationen	27
Divisionen	2
Quadratwurzel	1

Tabelle 3.3.: Rechenoperationen

Verfahren nach Möller

Das zweite untersuchte Schnittverfahren wird in [Möl08, S. 739ff] erläutert. Ausgangspunkt ist hierbei eine Kugel **s** mit Mittelpunkt **c** und einem Radius r, welche durch die Funktion:

$$\mathbf{s}(\mathbf{p}) = ||\mathbf{p} - \mathbf{c}|| = r \tag{3.18}$$

beschrieben wird. Der Sichtstrahl **r** ist analog zum Verfahren von Gumhold definiert. Zum Lösen des Schnittproblems wird der Sichtstrahl in Gleichung 3.18 eingesetzt und die so entstehende Gleichung quadriert, ausmultipliziert und auf die linke Seite gebracht. Am Ende ergibt sich dadurch:

$$\lambda^2 + 2 * \lambda * (\mathbf{v}(\mathbf{e} - \mathbf{c})) + (\mathbf{e} - \mathbf{c})(\mathbf{e} - \mathbf{c}) - r^2 = 0 \tag{3.19}$$

Nun kann abkürzend definiert werden: $b = \mathbf{v}(\mathbf{e} - \mathbf{c})$ sowie $c = (\mathbf{e} - \mathbf{c})(\mathbf{e} - \mathbf{c}) - r^2$. Außerdem muss die Gleichung nach λ aufgelöst werden, womit für die Lösung des Problems folgt:

$$\lambda = -b \pm \sqrt{b^2 - c} \tag{3.20}$$

Auch in Gleichung 3.20 liefert der Wert unter Wurzel die Entscheidung, ob ein Schnitt zwischen **r** und **s** vorliegt. Ist er kleiner Null wird die Kugel s verfehlt, andernfalls wird s geschnitten bzw. berührt. Um den Schnittpunkt zu erhalten muss nun wie beim Verfahren von Gumhold vorgegangen werden und der Parameter t in die Schnittstrahlengleichung eingesetzt werden. Daher weisen beide Algorithmen ab diesem Zeitpunkt den selben Aufwand auf. Insgesamt summieren sich die Operationen zu den in Tabelle 3.4 dargestellten Ergebnissen.

Additionen	13
Multiplikationen	12
Divisionen	0
Quadratwurzel	1

Tabelle 3.4.: Rechenoperationen bei Möller

Schnitt in GA

Der eigentliche Schnitt läuft nach dem mit Gaalop optimierten Code ab, indem Kugel und Strahl durch OP geschnitten werden, so dass daraus ein Punktpaar entsteht. Dieses wird schon während der Berechnung dualisiert, da bei der Extraktionsfunktion für einen Punkt mehrfach das dualisierte und nicht das eigentliche Punktpaar benötigt wird. Listing 3.3 zeigt die Implementierung der Punktpaarkonstruktion im C++-Code.

Da die Kugeln durch ihre Konstruktion bereits normalisiert sind, gilt, dass *sphere6* gleich 1 ist und somit aus der Berechnung eliminiert werden kann, wodurch 6 Multiplikationen eingespart

```
PointPair::PointPair(Ray* r, Sphere* s)
{
  this->PP7 = -r->ray14;
  this->PP8 = r->ray12;
  this->PP9 = -s->sphere3*r->ray14 - r->ray11*s->sphere5 +
    s->sphere4*r->ray12;
  this->PP10 = r->ray11;
  this->PP11 = -r->ray9;
  this->PP12 = -s->sphere4*r->ray9 + r->ray8*s->sphere5 +
    s->sphere2*r->ray14;
  this->PP13 = -r->ray8;
  this->PP14 = -r->ray7*s->sphere5 - s->sphere2*r->ray12 +
    s->sphere3*r->ray9;
  this->PP15 = r->ray7;
  this->PP16 = -s->sphere3*r->ray8 + s->sphere4*r->ray7 +
    s->sphere2*r->ray11;

  this->inprod =  2*PP12*PP13 - PP11*PP11 - PP7*PP7 + 2*PP14*PP15 +
    2*PP9*PP10 + PP16*PP16 - PP8*PP8;
}
```

Listing 3.3: Schnitt Kugel und Strahl

werden können. Insgesamt fallen für diesen Algorithmus-Schritt noch 12 Multiplikationen und 8 Additionen an.

Nun muss noch der Punkt aus dem Punktpaare extrahiert werden, der näher am Augpunkt liegt. Hierzu dient die Formel:

$$P_{\pm} = \frac{\pm\sqrt{PP*.PP*} + PP*}{e_{inf}.PP} \tag{3.21}$$

die aus [Hil06, S. 74] entnommen ist. Vorteil dieser Gleichung ist, dass automatisch normalisierte, konforme Punkte erzeugt werden. Nachteil ist, zumindest für eine Realisierung des Algorithmus auf einem FPGA, die auftretende Division. Daher wird die Formel vereinfacht:

$$P = (\sqrt{PP.PP} - PP) * (e_{inf}.PP) \tag{3.22}$$

Die folgenden beiden Veränderungen werden im Vergleich zur ursprünglichen Form vorgenommen:

- Die Division wird gegen eine Multiplikation ersetzt, da eine Division in Hardware deutlich aufwändiger ist als eine Multiplikation. Dies ist auf Grund der Eigenschaft der Geometrischen Algebra möglich, dass skalierte Multivektoren dasselbe Objekt beschreiben (vgl. Abschnitt 2.4 zu den Multivektoren der GA).

```
Point * pminus = new Point(0,0,0);

pminus->point2 = -PP8*PP15 - PP7*PP13 + PP16*PP10 + w*PP10;
pminus->point3 = -PP11*PP15 + PP7*PP10 + w*PP13 + PP16*PP13;
pminus->point4 = PP8*PP10 + PP16*PP15 + PP11*PP13 + w*PP15;
pminus->point5 = w*PP16 + PP13*PP12 + PP15*PP14 + PP10*PP9 +
  PP16*PP16;
pminus->point6 = PP15*PP15 + PP10*PP10 + PP13*PP13;

pminus->normalize();
return pminus;
```

Listing 3.4: Extraktion des Schnittpunktes

- Schon durch Versuche in CluCalc kam die Vermutung auf, dass *pminus* stets der nächstgelegene Punkt ist. Bei der Implementierung des Raytracers in C++ bestätigt sich dieser Verdacht. Egal wie die Szene gedreht wird, ist *pminus* der korrekte Schnittpunkt. Somit wird immer *pminus* berechnet und verwendet; die Bestimmung von *pplus*, sowie ein Entfernungsvergleich der beiden Punkte zur Kamera kann vermieden werden.

Der C++-Code zur optimierten Schnittpunktextraktion findet sich in Listing 3.4.

Es existiert noch eine zusätzliche Vereinfachung. Werden die weiteren Berechnungen betrachtet, in denen der Schnittpunkt benutzt wird, so fällt auf, dass dessen Koeffizient *point5* nur in der Formel zur Berechnung des Abstandes zum Ursprung auftaucht. Da aber sowohl der Schnittpunkt als auch der Ursprung normalisierte, konforme Punkte sind, kann auch die Formel zur Bestimmung des euklidischen Abstandes genutzt werden. Somit ist es nicht nötig *point5* zu berechnen, wodurch 5 Multiplikationen und 4 Additionen gespart werden können. Somit kann in Listing 3.4 *pminus->point5* gleich 0 gesetzt werden.

Wird *pminus* nicht normalisiert, so muss *point6* mitgeführt und in weiteren Berechnungen des Raytracers berücksichtigt werden, wodurch zusätzliche Multiplikationen entstehen. Bei einer Normalisierung werden allerdings 3 Divisionen nötig (vorausgesetzt *point5* wird konstant 0 gesetzt), da jede der Punktkomponenten durch den Wert von *point6* geteilt werden muss. Eine Messung beider implementierter Varianten ergibt, dass der Weg über normalisierte Punkte in C++ und OpenCL minimale Vorteile bietet, wobei die Geschwindigkeitsunterschiede unter einem Prozent liegen. Da außerdem der Code bei normalisierten Punkten kürzer und somit übersichtlicher wird, ist in der fertigen Version diese Variante umgesetzt. Im Falle einer Verilog-Umsetzung ist dagegen die Implementierung des mitgeführten *point6*-Werts sinnvoller.

Tabelle 3.5 fasst die Ergebnisse dieses Abschnittes zusammen, wobei das Verhältnis GA/LA zwischen dem Verfahren von Möller und dem hier optimierten GA-Algorithmus gebildet wird.

Es lässt sich erkennen, dass das Verfahren von Möller sowohl beim reinen Schnitttest, als auch bei der Berechnung des eigentlichen Schnittpunktes wesentlich weniger Operationen benötigt als die beiden anderen. Bei der GA fallen jeweils mehr als doppelt so viele Multiplikationen und rund 50% mehr Additionen an. Das Verfahren von Stoll/Gumhold benötigt ebenfalls in beiden Bereichen mehr Operationen als Möller und liegt für den Test auf Schnitt sogar über der Anzahl der GA. Bemerkenswert ist beim Möller-Algorithmus, dass bereits für die Schnittentscheidung

	GA	Möller	Stoll	GA/LA
Test auf Schnitt				
Multiplikationen	19	9	24	2,11
Additionen	14	10	11	1,4
Wurzeln	0	1	0	0
Schnittpunkt berechnen				
Multiplikationen	31	12	27	2,58
Additionen	23	13	16	1,77
Divisionen	3	0	2	-
Wurzeln	1	1	1	1

Tabelle 3.5.: Operationenvergleich des Schnittproblems

eine Wurzel gezogen werden muss, was bei den anderen beiden nicht der Fall ist. Allerdings kommt Möllers Variante ohne Divisionen aus.

Legt man den Fokus ausschließlich auf die Zahlen, scheint das Möller-Verfahren daher überlegen zu sein. Doch fällt schon beim Betrachten des Sourcecodes bzw. der mathematischen Gleichungen auf, dass sich zum Beispiel die Komponenten des Punktpaar-Multivektors 9, 12, 14 und 16 parallel berechnen lassen und auch bei Stoll/Gumhold auf Grund ihrer Unabhängigkeit *alpha*, *beta* und *gamma* gleichzeitig bestimmt werden können. Möllers Algorithmus basiert dagegen auf sequentiellem Berechnen von untereinander abhängigen Zwischenergebnissen, was wiederum einen Vorteil für GA und Stoll bedeutet. Eine Entscheidung über die Leistungsfähigkeit ist somit nur anhand der Implementierungen möglich.

3.4.3 Schnitttest Dreieck/Strahl

Die Operation herauszufinden, ob ein Strahl ein Dreieck trifft, ist die mit Abstand häufigste beim Raytracing von Dreiecksnetzen. Geht man von einer Szene mit nur 5000 Dreiecken aus, die in der Auflösung 512x512 dargestellt werden soll und verzichtet auf den Einsatz von räumlichen Datenstrukturen und Hüllkörpern, so fallen 512*512*5000 Schnitttests mit den Dreiecken an, insgesamt rund $1,3 * 10^9$. Dem gegenüber stehen nur 262144 Erzeugungen von Sichtstrahlen (0,02 % der Anzahl von Dreiecksschnitttests) und maximal genauso viele Konstruktionen von Reflexionsstrahlen (mit denen wiederum die $1,3 * 10^9$ Schnitttests durchgeführt werden müssen). Tabelle 3.6 zeigt zwei weitere Beispiele, welche die Dominanz der Dreiecksschnitttests für die Gesamtgeschwindigkeit demonstrieren.

# Dreiecke	Pixel	# Tests	# Rays	% Anteil Rays
5000	512x512	$1,3 * 10^9$	262144	0,02
5000	800x800	$3,2 * 10^9$	640000	0,02
500000	512x512	$1,3 * 10^{11}$	262144	0,0002

Tabelle 3.6.: Dominanz der Schnitttests

Daher ist ersichtlich, dass die Performance des gesamten Algorithmus in großem Maße davon abhängt wie effizient der Test auf Schnitt zwischen Dreieck und Sichtstrahl durchgeführt werden

kann. Räumliche Datenstrukturen reduzieren sowohl für GA als auch LA in gleichem Maße die Anzahl der nötigen Tests und bieten daher theoretisch keiner der beiden Algebren einen Vorteil, weshalb sie in dieser Untersuchung nicht berücksichtigt werden.

Dreiecksschnitt in GA

Der für den Schnitttest in GA verwendete Algorithmus konstruiert zuerst drei DualRays, auf welcher die Dreiecksseiten liegen. Mittels des Inneren Produktes des Strahles wird mit jedem DualRay ein Indikator berechnet, der die relative Position des Strahles zur Dreiecksseite angibt:

$$\begin{aligned} edge1 &= p_0 \wedge p_1 \wedge e_\infty \\ edge2 &= p_1 \wedge p_2 \wedge e_\infty \\ edge3 &= p_2 \wedge p_0 \wedge e_\infty \\ \\ indi1 &= e_0.(ray.edge1) \\ indi2 &= e_0.(ray.edge2) \\ indi3 &= e_0.(ray.edge3) \end{aligned} \tag{3.23}$$

Hierbei muss natürlich vorausgesetzt werden, dass die Dreiecke einheitlich orientiert sind. Nur wenn dieser Indikator in allen drei Fällen positiv ist, wird das Dreieck vom Strahl getroffen. Hat einer der Indikatoren ein negatives Vorzeichen, steht an dieser Stelle fest, dass das Dreieck verfehlt wird und der Algorithmus kann beendet werden. Im positiven Fall wird nun der eigentliche Schnittpunktkandidat berechnet. Dazu wird mit Hilfe des OPs die Ebene berechnet, in der das Dreieck liegt:

$$plane = *(p_0 \wedge p_1 \wedge p_2 \wedge e_\infty) \tag{3.24}$$

Nun wird das OP zwischen der Dreiecksebene und dem Strahl gebildet, wodurch ein *FlatPoint* entsteht:

$$fp = *(plane \wedge ray) \tag{3.25}$$

der zu einem entsprechenden konformen Punkt umgewandelt wird.

Nun folgt noch eine Abstandsprüfung zwischen dem gefundenen Schnittpunkt und der Ebene im Strahlenursprung, um so *Self-Intersections* auszuschließen, also den Fall, dass ein Schnitt mit dem Dreieck gefunden wird, aus dem der Strahl entspringt, was bei Reflexionsstrahlen und Schattenfühlern passieren kann. Der Abstand wird durch das IP zwischen Punkt und der Ebene berechnet:

$$distance = rayplane.intersection\_point \tag{3.26}$$

Additionen	94
Multiplikationen	114
Divisionen	5
Quadratwurzel	0

Tabelle 3.7.: Rechenoperationen beim Dreiecksschnitt in GA

Ist der Schnitt zulässig wird mit Hilfe von baryzentrischen Koordinaten die Normale des Schnittpunkts aus den Normalen der drei Eckpunkte interpoliert. Für die Operationen der GA fallen vor allem viele Multiplikationen an, was Tabelle 3.7 veranschaulicht.

Die Menge der Operationen deutet an, dass der nötige C++-Code zu seiner Berechnung ebenfalls umfangreicher ist und daher an dieser Stelle auf die Wiedergabe des Codes verzichtet wird. Die Funktionsweise des Schnittalgorithmus in GA kann anhand des Raytracers in Anhang A nochmals nachvollzogen werden.

Dreiecksschnitt in LA

Im Fall der LA werden, wie bei den Kugeln, zwei bekannte Verfahren implementiert, um einen möglichst guten Vergleich zur GA zu bieten. Zum einen wird auf den Algorithmus von Möller und Trumbore [MT97] zurückgegriffen, zum anderen auf das von Badouel [Bad90] vorgestellte Verfahren. Beide werden von Segura und Feito [RSF01] auf einem SingleCore-Prozessor hinsichtlich ihrer Performance verglichen und es ergeben sich leichte Vorteile für Möllers Variante. Eigene Geschwindigkeitsmessungen sollen prüfen, wie sich beide Algorithmen auf parallelen GPU-Hardwarearchitekturen verhalten.

Möller / Trumbore

Dieser Algorithmus basiert auf der Tatsache, dass sich ein Punkt auf einem Dreieck $V_0V_1V_2$ mittels baryzentrischer Koordinaten u und v eindeutig beschreiben lässt:

$$T(u,v) = (1-u-v) * V_0 + u * V_1 + v * V_2 \tag{3.27}$$

Da der Schnittpunkt ebenfalls auf dem Sichtstrahl $\mathbf{r}$ liegen muss, der in parametrischer Darstellung vorliegt, lassen sich die beiden Gleichungen gleichsetzen:

$$\mathbf{e} + \lambda\mathbf{v} = (1-u-v) * V_0 + u * V_1 + v * V_2 \tag{3.28}$$

Dadurch entsteht ein 3x3 Gleichungssystem mit den Unbekannten u, v und λ, das sich wie in Gleichung 3.29 gezeigt auflösen lässt:

$$\begin{pmatrix} \lambda \\ u \\ v \end{pmatrix} = \frac{1}{(v \times edge2) \cdot edge1} * \begin{pmatrix} (T \times edge1) \cdot edge2 \\ (\mathbf{v} \times edge2) \cdot T \\ (T \times edge2) \cdot \mathbf{v} \end{pmatrix} \tag{3.29}$$

In Gleichung 3.29 werden folgende abkürzende Schreibweisen verwendet:

- *edge1*: $V_1 - V_0$
- *edge2*: $V_2 - V_0$
- T: $O - V_0$

Der Algorithmus berechnet nun zuerst den Bruch und hat an dieser Stelle einen Early Exit, falls der Nenner 0 wird. Anschließend werden zunächst u und v berechnet und die üblichen Bedingungen $u \geq 0$, $v \geq 0$ und $u + v \leq 1$ geprüft, wodurch jeweils ebenfalls ein früher Abbruch der Berechnung möglich ist. Ein letztes Entscheidungskriterium, ob ein zulässiger Schnitt gefunden wurde, bildet der Wert des Geradenparameters λ, da dieser nicht negativ werden darf. Sind alle Bedingungen erfüllt, kann mit λ der Schnittpunkt berechnet und mittels u und v seine Normale interpoliert werden. Tabelle 3.8 zeigt die für das Verfahren nötigen Rechenoperationen.

Additionen	37
Multiplikationen	45
Divisionen	2
Quadratwurzel	0

Tabelle 3.8.: Rechenoperationen beim Dreiecksschnitt mit Möller

Badouel

Badouels Verfahren [Bad90] kehrt den Weg von Möllers Algorithmus um, da zuerst ein Schnittpunktkandidat berechnet wird. Hierzu werden analog zu Möller die beiden Dreieckskanten $edge1$ und $edge2$ gebildet und aus deren Kreuzprodukt die Flächennormale **nf** des Dreiecks berechnet. Der Algorithmus benötigt an dieser Stelle die Ebene, welche durch die Dreieckseckpunkte V_0, V_1 und V_2 aufgespannt wird, in der *Hesseschen Normalform*, welche in Gleichung 3.30 gezeigt ist.

$$\mathbf{nf} \cdot \mathbf{p} + d = 0 \tag{3.30}$$

Um den Abstand d der Ebene zum Ursprung zu ermitteln, ist es notwendig das Skalarprodukt zwischen der Normalen **n** und einem beliebigen Punkt auf der Ebene, also in diesem Fall einen der Eckpunkte, zu bilden. Anschließend lässt sich durch das Einsetzen des Sichtstrahls **r** in die Ebenengleichung der Geradenparameter λ, wie in Gleichung 3.31 gezeigt, berechnen.

$$\lambda = -\frac{d + \mathbf{nf} * \mathbf{e}}{\mathbf{nf} \cdot \mathbf{v}} \tag{3.31}$$

Mittels λ und der Geradengleichung lässt sich der Schnittpunktkandidat **p** finden. Ein negativer λ-Wert gibt an, dass definitiv kein Schnitt mit der aufgespannten Ebene existiert. Nun muss mittels der baryzentrischen Koordinaten entschieden werden, ob **p** inner- oder außerhalb des Dreiecks liegt. Um den Berechnungsaufwand zu reduzieren, wird das Dreieck auf eine der Koordinatenebenen projiziert. Damit sichergestellt ist, dass nicht versucht wird auf eine Ebene zu

projizieren, die auf der Dreiecksebene senkrecht steht, wird nach einer Methode aus [SB87] ein Indexwert i_0 ermittelt. Das Auswahlverfahren zeigt Gleichung 3.32.

$$i_0 = \begin{cases} 0 \ falls & \left|\mathbf{nf}_x\right| = max(\left|\mathbf{n}_x\right|, \left|\mathbf{nf}_y\right|, \left|\mathbf{nf}_z\right|) \\ 1 \ falls & \left|\mathbf{nf}_y\right| = max(\left|\mathbf{nf}_x\right|, \left|\mathbf{nf}_y\right|, \left|\mathbf{nf}_z\right|) \\ 2 \ falls & \left|\mathbf{nf}_z\right| = max(\left|\mathbf{nf}_z\right|, \left|\mathbf{nf}_y\right|, \left|\mathbf{nf}_z\right|) \end{cases} \tag{3.32}$$

Es werden außerdem zwei weitere Indizes i_1 und i_2 definiert, welche mit den verbleibenden Werten aus $\{0, 1, 2\}$ belegt werden. So gilt beispielsweise im Fall $i_0 = 0$, dass $i_1 = 1$ und $i_2 = 2$ sind. Zur Steigerung der Übersichtlichkeit werden einige abkürzende Schreibweisen definiert, die in Gleichung 3.33 dargestellt sind.

$$\begin{aligned} s_0 &= \mathbf{p}_{i_1} - V_{0_{i_1}} & s_1 &= V_{1_{i_1}} - V_{0_{i_1}} & s_2 &= V_{2_{i_1}} - V_{0_{i_1}} \\ t_0 &= \mathbf{p}_{i_2} - V_{0_{i_2}} & t_1 &= V_{1_{i_2}} - V_{0_{i_2}} & t_2 &= V_{2_{i_2}} - V_{0_{i_1}} \end{aligned} \tag{3.33}$$

Gleichung 3.34 zeigt schließlich, wie u und v aus diesen Werten ermittelt werden.

$$u = \frac{det\begin{pmatrix} s_0 & s_2 \\ t_0 & t_2 \end{pmatrix}}{det\begin{pmatrix} s_1 & s_2 \\ t_1 & t_2 \end{pmatrix}} \qquad v = \frac{det\begin{pmatrix} s_1 & s_0 \\ t_1 & t_0 \end{pmatrix}}{det\begin{pmatrix} s_1 & s_2 \\ t_1 & t_2 \end{pmatrix}} \tag{3.34}$$

Die beim Möller-Verfahren genannten Bedingungen für baryzentrische Koordinaten ermöglichen eine Aussage darüber, ob **p** ein zulässiger Schnittpunkt ist. Zum Abschluss wird mit Hilfe von u und v die Normale **n** im Schnittpunkt interpoliert. Die Anzahl, der für den Algorithmus nötigen Operationen, zeigt Tabelle 3.9.

Additionen	35
Multiplikationen	36
Divisionen	3
Quadratwurzel	1

Tabelle 3.9.: Rechenoperationen beim Dreiecksschnitt mit Badouel

Tabelle fasst 3.10 die Ergebnisse für alle drei besprochenen Verfahren zusammen.

Die Operationenanzahl ist bei der GA vor allem bei den Multiplikationen bis zu dreimal höher als bei den etablierten Standardverfahren, unabhängig davon, ob nur die Schnittentscheidung oder die Schnittpunktberechnung betrachtet werden. Die Verfahren von Möller und Badouel liegen dagegen gleich auf. Den größten Unterschied macht die Quadratwurzel, die bei Badouel schon zur Schnittentscheidung benötigt wird, da es unumgänglich ist die Flächennormale zu normalisieren. Somit eignet sich der Badouel-Algorithmus weniger für eine direkte Realisierung in Hardware. Anhand des Themengebietes der Dreiecksschnitte lässt sich die kompakte Formulierbarkeit von Algorithmen in GA sehr gut verdeutlichen. Die Listings 3.5 und 3.6 stellen den OpenCL-Code von Möllers Algorithmus gegen das GA-Verfahren in CluScript. Die Formulierung in CluScript ist deutlich kürzer und einfacher lesbar.

Listing 3.5: Möller-Dreiecksschnitttest

```
bool rayIntersectsTriangle(float8 p0, float8
    p1, float8 p2, float4 r, float4 o, float*
    u_t, float* v_t)
{
  float4 edge1; float4 edge2; float4 pvec;
  edge1.x = p1.s0 - p0.s0;
  edge1.y = p1.s1 - p0.s1;
  edge1.z = p1.s2 - p0.s2;
  edge2.x = p2.s0 - p0.s0;
  edge2.y = p2.s1 - p0.s1;
  edge2.z = p2.s2 - p0.s2;

  pvec.x = r.y*edge2.z - r.z*edge2.y;
  pvec.y = r.z*edge2.x - r.x*edge2.z;
  pvec.z = r.x*edge2.y - r.y*edge2.x;

  float det = edge1.x*pvec.x + edge1.y*pvec.y
      + edge1.z*pvec.z;
  if(det > -2e-6f && det < 2e-6f)    return 0;

  float inv_det = 1.0f / det;

  float4 tvec;
  tvec.x = o.x - p0.s0;
  tvec.y = o.y - p0.s1;
  tvec.z = o.z - p0.s2;

  float u = (tvec.x*pvec.x + tvec.y*pvec.y +
     tvec.z*pvec.z) * inv_det;
  if(u < 2e-6f || u > 1.00f + 2e-6f) return
     0;

  float4 qvec;
  qvec.x = tvec.y*edge1.z - tvec.z*edge1.y;
  qvec.y = tvec.z*edge1.x - tvec.x*edge1.z;
  qvec.z = tvec.x*edge1.y - tvec.y*edge1.x;

  float v = (r.x*qvec.x + r.y*qvec.y + r.z*
     qvec.z) * inv_det;
  if(v < 2e-6f || u + v > 1.0f + 2e-6f)
     return 0;

  float t = (edge2.x*qvec.x + edge2.y*qvec.y
     + edge2.z*qvec.z) * inv_det;
  if (t < 2e-1f) return 0;

  *u_t = u; *v_t = v;
}
```

Listing 3.6: GA-Dreiecksschnitttest

```
edge1 = p0^p1^einf;
edge2 = p1^p2^einf;
edge3 = p2^p0^einf;

indi1 = e0.(ray.edge1);
indi2 = e0.(ray.edge2);
indi3 = e0.(ray.edge3);

side = -1;

if(indi1 > 0 && indi2 > 0 &&
    indi3 > 0)
{
  return 0;
}
```

	GA	Möller	Badouel	GA/LA
Test auf Schnitt				
Multiplikationen	84	27	27	3,11
Additionen	64	24	27	2,67
Divisionen	3	2	4	1,5
Wurzeln	0	0	1	0
Schnittpunkt berechnen				
Multiplikationen	114	45	36	3,17
Additionen	94	37	35	2,54
Divisionen	5	2	3	2,5
Wurzeln	0	0	1	0

Tabelle 3.10.: Operationenvergleich des Schnittproblems bei Dreiecken

3.4.4 Normalenberechnung

Der nächste Schritt beim Raytracing ist die Berechnung der Normalen im Schnittpunkt, welche für die Berechnung der Beleuchtung und des Reflexionsstrahles erforderlich wird. Dieser Abschnitt betrachtet nur die Normalenberechnung an Kugeln. Für das Raytracen von Dreiecksnetzen, wird wie im vorangehenden Abschnitt 3.4.3 dargelegt, auf Interpolation mittels baryzentrischer Koordinaten zurückgegriffen.

Normale in LA

In [SGS06] wird zur Berechnung der Normalen der Gradient der Funktion genutzt, so dass sich folgende Berechnung ergibt:

$$gradf_I(x,y,z) = \begin{pmatrix} 2*k_{200}*x + k_{110}*y + k_{101}*z + k_{100} \\ 2*k_{020}*y + k_{110}*x + k_{011}*z + k_{010} \\ 2*k_{002}*z + k_{011}*y + k_{101}*x + k_{001} \end{pmatrix} \tag{3.35}$$

Allerdings wird in [SGS06] dieser Weg beschritten, da dort mit allgemeinen Quadriken und nicht nur mit Kugeln gerechnet wird. Für Kugeln bietet sich ein einfacheres Verfahren an, um die Richtung der Normalen zu finden, der deutlich weniger Operationen benötigt, als die oben geschilderte: Die Subtraktion von Schnittpunkt auf der Kugeloberfläche und Mittelpunkt. Hierdurch resultieren zur Berechnung lediglich drei Subtraktionen.

Normale in GA

Der für die KGA offensichtlichste Weg die Normale zu erhalten, ist eine konforme Gerade durch Kugelmittelpunkt und Schnittpunkt zu konstruieren. Dies geschieht indem das OP der beiden konformen Punkte gebildet wird. Anschließend kann aus dieser Geraden, die Richtung der Normalen extrahiert werden, da nur diese für die folgenden Schritte benötigt wird.

```
// In CluCalc
DefVarsN3();
:IPNS;

?a=VecN3(x1, y1, z1);
?b=VecN3(x2, y2, z2);

n = *(a ^ b ^ einf);

?dir = n*(e1 ^ e2 ^ e3);

// Optimiert mit Gaalop:
dir(2)=b(2)*a(6)-a(2)*b(6);
dir(3)=b(3)*a(6)-a(3)*b(6);
dir(4)=b(4)*a(6)-a(4)*b(6);
```

Listing 3.7: Normalenberechnung

Allerdings stellt sich bei der Implementierung heraus, dass es auch in GA einen einfacheren Weg der Normalenberechnung gibt. Nach Betrachtung der Gaalopausgabe fällt auf, dass zur Berechnung der Richtung jeweils nur drei der fünf Koeffizienten der 5D Punkte benötigt werden, wenn voraussetzt wird, dass die Punkte normalisiert sind. Der Koeffizient von e_0 kann erneut aus der Berechnung eliminiert werden. Daher entfallen in Listing 3.7 die Faktoren *a(6)* und *b(6)*. Somit ist die Normalenberechnung deckungsgleich mit dem Fall in Linearer Algebra und benötigt 3 Differenzen.

Tabelle 3.11 vergleicht die gefundenen Ergebnisse dieses Abschnitts mit der Implementierung aus [SGS06]. Wie dargelegt kann aber auch in LA mit demselben Aufwand die Normale berechnet werden, wenn nur Kugeln betrachtet werden. Für die Geschwindigkeitsmessungen wird daher die zweite, kürzere Variante gewählt.

	GA	LA	GA/LA
Berechnung der Normalen			
Multiplikationen	0	12	0
Additionen/Subtraktionen	3	9	0,33

Tabelle 3.11.: Operationenvergleich bei Normalenberechnung

3.4.5 Reflexionsstrahlen

Reflexionsstrahlen und deren Richtung werden beim Raytracing, wie weiter oben dargelegt, zur Konstruktion von Sekundärstrahlen und zur Beleuchtungsrechnung benötigt. Pro Pixel können bei Rekursionstiefe 1 maximal zwei Berechnungen von Reflexionsstrahlen auftreten, nämlich dann, wenn der ursprüngliche Sichtstrahl eines der Szenenobjekte trifft und wenn der dort entspringende Reflexionsstrahl ein weiteres Objekt schneidet. Somit spielt der Gesamteinfluss der Reflexionsrechnungen eine eher untergeordnete Rolle für die Gesamtperformance des Programms. Dennoch werden die dazu benötigten GA-Berechnungen im folgenden Abschnitt etwas

ausführlicher betrachtet, da hierbei Optimierungsschritte am Ausgabecode von Gaalop auftreten, die sich auf beliebige andere Algorithmen übertragen lassen.

Reflexionen in LA

Die Formel zur Berechnung eines Reflexionsvektors $\mathbf{r}_r$ in Linearer Algebra ist:

$$\mathbf{r}_r = \mathbf{l} - 2 * \mathbf{n} \cdot \mathbf{l} * \mathbf{n} \tag{3.36}$$

Ausgeschrieben sieht die Formel folgendermaßen aus:

$$\mathbf{r}_r = \begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix} - 2 * \begin{pmatrix} n_1 \\ n_2 \\ n_3 \end{pmatrix} \cdot \begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix} * \begin{pmatrix} n_1 \\ n_2 \\ n_3 \end{pmatrix} \tag{3.37}$$

Zusammengefasst ergibt sich:

Additionen/Subtraktionen	5
Multiplikationen	7

Tabelle 3.12.: Rechenoperationen Reflexion in LA

Reflexion in 5D an Ebene

Schon erste, direkt an die GA angelehnte, Versuche einen performanten Weg zur Reflexion in 5D zu finden zeigen, dass es sich dabei um einen komplexeren Themenbereich handelt. Dadurch, dass in 5D ursprungunabhängig gerechnet werden kann und mehr geometrische Primitive vorhanden sind als in 3D, fallen weit mehr Berechnungen an als im Fall von LA.

Die intuitive Reflexion in 5D geschieht an einer Ebene, die im Schnittpunkt konstruiert wird. Mit Hilfe des sogenannten *Sandwichproduktes* kann anschließend aus der Ebene und dem einfallendem Lichtstrahl der Reflexionsstrahl erhalten werden. Listing 3.8 zeigt hierzu eine kommentierte CluCalc-Formulierung der benötigten Schritte. Listing 3.9 gibt die direkte Gaalopausgabe des optimierten CluCalc-Skriptes in C++ wider.

Wie erkennbar ist, sind dies deutlich mehr Berechnungen als im Fall von LA. Allerdings steckt darin noch einiges an Potential für eine manuelle Optimierung, um so den Bedarf an Operationen zu senken. Die Quadrate der Koeffizienten *p[2]*, *p[3]* und *p[4]* kommen jeweils sechsmal vor und können daher in einer ersten Stufe berechnet werden, um die Multiplikation nur einmal ausführen zu müssen. Ebenso kommen die Produkte $2 * p[2] * p[3]$, $2 * p[2] * p[4]$ und $2 * p[3] * p[4]$ viermal vor und daher besteht die Möglichkeit sie vorzuberechnen. Ähnliches gilt für die Produkte $2 * p[2] * p[5]$, $2 * p[3] * p[5]$ und $2 * p[4] * p[5]$, welche im Verlauf der Berechnung jeweils zweimal auftreten. Des Weiteren ist ersichtlich, dass bei jedem Koeffizienten die Quadrate von p[2], p[3] und p[4] mit demselben Eintrag von l multipliziert werden. Der Faktor l kann somit jeweils ausgeklammert werden. Dies führt zu kürzerem und übersichtlicherem C++-Code, der in Listing 3.10 gefunden werden kann.

Dies ist in Tabelle 3.13 zusammengefasst.

```
// Ebene konstruieren
?nor = xn*e1 + yn*e2 + zn*e3;
?p = nor + d*einf;

// Zu reflektierenden Strahl definieren
?p0 = VecN3(x0, y0, z0);
?p1 = VecN3(x1, y1, z1);
?g = *(p0^p1^einf);

// Relfexionsstrahl durch Sandwichprodukt
?rr = p*g*p;
```

Listing 3.8: Reflexion in 5D an Ebene in CluCalc

Additionen/Subtraktionen	32
Multiplikationen	39

Tabelle 3.13.: Rechenoperationen 5D Reflexion an Ebene

5D Reflexion nach Vince

In diesem Abschnitt soll untersucht werden, ob die ob die 3D-LA-Formel aus [Vin09, S. 109] auch im konformen Raum anwendbar ist und zu welchem Ergebnis ihre Verwendung führt. Hierzu seien der einfallende, konforme Lichtstrahl L, sowie die Reflexionsebene Pl gegeben. Die eingesetzte Formel zur Berechnung des Reflexionsstrahles R_r zeigt Gleichung 3.38.

$$R_r = -(Pl.L) * Pl - (Pl \wedge L) * Pl \tag{3.38}$$

Sie ist allerdings länger und vor allem im Vergleich zum Sandwichprodukt weniger intuitiv. Aus ihr resultiert der CluCalc-Code aus Listing 3.11. Gaalop optimiert diese Eingabe wie in Listing 3.12 gezeigt nach C++.

Tabelle 3.14 fasst die nötigen Rechenoperationen zusammen.

Additionen/Subtraktionen	12
Multiplikationen	36

Tabelle 3.14.: Rechenoperationen 5D Reflexion nach Vince

Die Quadrate der Pl-Koeffizienten kommen jeweils sechsmal vor und können daher vorberechnet werden was Listing 3.13 veranschaulicht.

Daraus resultiert die Anzahl der Berechnungen, die in Tabelle 3.15 aufgelistet sind.

Hier kann also eine deutliche Verbesserung erzielt werden, allerdings auf Kosten der Verständlichkeit und Intuitivität der CluScript-Eingabe.

```
rr[7] = -p[3]*p[3]*l[7] + p[4]*p[4]*l[7] +
  2*p[2]*p[4]*l[11] - p[2]*p[2]*l[7] -
  2*p[3]*p[4]*l[8];

rr[8] = -2*p[3]*p[4]*l[7] - p[2]*p[2]*l[8] -
  2*p[2]*p[3]*l[11] - p[4]*p[4]*l[8] +
  p[3]*p[3]*l[8];

rr[9] = -2*p[3]*p[5]*l[7] + p[3]*p[3]*l[9] -
  2*p[2]*p[4]*l[14] + p[4]*p[4]*l[9] -
  p[2]*p[2]*l[9] - 2*p[4]*p[5]*l[8] -
  2*p[3]*p[2]*l[12];

rr[11] = -p[4]*p[4]*l[11] + 2*p[2]*p[4]*l[7] -
  2*p[3]*p[2]*l[8] - p[3]*p[3]*l[11] +
  p[2]*p[2]*l[11];

rr[12] = 2*p[2]*p[5]*l[7] + p[2]*p[2]*l[12] -
  2*p[2]*p[3]*l[9] - 2*p[5]*p[4]*l[11] -
  2*p[3]*p[4]*l[14] + p[4]*p[4]*l[12] -
  p[3]*p[3]*l[12];

rr[14] = -2*p[4]*p[3]*l[12] - p[4]*p[4]*l[14] -
  2*p[2]*p[4]*l[9] + p[3]*p[3]*l[14] +
  p[2]*p[2]*l[14] + 2*p[2]*p[5]*l[8] +
  2*p[5]*p[3]*l[11];
```

Listing 3.9: Reflexion in 5D an Ebene in C++

Additionen/Subtraktionen	12
Multiplikationen	21

Tabelle 3.15.: Rechenoperationen 5D Reflexion nach Vince nach Handoptimierungen

Reflexion in 3D nach dem bisherigen Schema

Da festgestellt werden kann, dass Reflexionen in 5D Konformer Geometrischer Algebra bezüglich der Anzahl der Operationen nicht mit der LA Schritt halten kann, soll nun untersucht werden, wie sich die 3D-GA im Vergleich zur 3D-LA verhält. Da in diesem Fall beide ursprungsbezogen sind, ist der Vergleich in gewisser Hinsicht gerechter.

Bisher wird ein Sandwichprodukt aus Ebene und Sichtstrahl gebildet: $-plane*ray*plane$. Für 3D lässt sich daher vermuten, dass das gleiche Schema dort mittels dem Sandwichprodukt aus Ebenennormale und der 3D Richtung des eingehenden Strahles wiederholbar ist: $-n*raydir*n$. Dies ist auch das Ergebnis, das John Vince in seinem Buch [Vin09, S. 108] für die Spiegelung erhält. Auf dieser Basis steht der CluCalc-Code aus Listing 3.14.

Durch eine Optimierung mit Gaalop ergibt sich der Code Listing 3.15.

```
// 1. Schritt
double p2Q = p[2]*p[2];
double p3Q = p[3]*p[3];
double p4Q = p[4]*p[4];

double two23 = 2*p[2]*p[3];
double two24 = 2*p[2]*p[4];
double two34 = 2*p[3]*p[4];

double two25 = 2*p[2]*p[5];
double two35 = 2*p[3]*p[5];
double two45 = 2*p[4]*p[5];

// 2. Schritt
rr[7] = two24*l[11] - two34*l[8] + l[7]*(p4Q - p2Q - p3Q);

rr[8] = -two34*l[7] - two23*l[11] + l[8]*(p3Q - p2Q - p4Q);

rr[9] = -two35*l[7] - two24*l[14] - two45*l[8] - two23*l[12] +
  l[9]*(p3Q + p4Q - p2Q);

rr[11] = two24*l[7] - two23*l[8] + l[11]*(p2Q - p3Q - p4Q);

rr[12] = two25*l[7] - two23*l[9] -  two45*l[11] - two34*l[14] +
  l[12]*(p2Q - p3Q + p4Q);

rr[14] = two35*l[11] - two34*l[12] - two24*l[9] + two25*l[8] +
  l[14]*(p2Q + p3Q - p4Q)
```

Listing 3.10: Optimierte Reflexion in 5D an Ebene in C++

An dieser Stelle lässt sich konstatieren, dass hier weitaus mehr Berechnungen durchgeführt werden müssen als in 3D-LA. Sicherlich lässt sich die Anzahl der anfallen Operationen noch reduzieren (z.B. diverse Quadrate und Produkte kommen mehrfach vor und können in einem ersten Schritt vorberechnet werden), doch eine Reduktion auf nur 7 Multiplikationen ist nicht möglich. Auch durch die Ausnutzung der Eigenschaft der normalisierten Normale $x^2+y^2+z^2=1$ lassen sich einige Berechnungen vereinfachen. Allerdings muss dann vorausgesetzt werden, dass eine Normalisierung durchgeführt wurde, was zu einer aufwändigen Berechnung mit Quadratwurzel und Division führt. Gerade Wurzeln und Divisionen sollen aber vermieden werden, weshalb an dieser Stelle darauf verzichtet wird.

Reflexion in 3D nach Vince

Des Weiteren soll versucht werden, ob die Formel $R_r = -(Pl.L) * Pl - (Pl \wedge L) * Pl$ auch im Fall von 3D eine Verbesserung im Vergleich zum Sandwichprodukt bringt. Jedoch resultiert hier aus der Optimierung durch Gaalop dasselbe Ergebnis wie im zuvor beschriebenen Fall. Durch

```
N = n1*e1 - n2*e2 + n3*e3;
P1 = VecN3(P1a,P1b,P1c);
P2 = VecN3(P2a,P2b,P2c);

Pl = N + d*einf;
L = *(P1^P2^einf);

Rr = -(Pl.L)*Pl - (Pl^L)*Pl;
```

Listing 3.11: Reflexion in 5D nach Vince

```
Rr[7] = Pl[4]*Pl[4]*L[7] + Pl[2]*Pl[2]*L[7] + Pl[3]*Pl[3]*L[7];
Rr[8] = Pl[4]*Pl[4]*L[8] + Pl[3]*Pl[3]*L[8] + Pl[2]*Pl[2]*L[8];
Rr[9] = Pl[4]*Pl[4]*L[9] + Pl[3]*Pl[3]*L[9] + Pl[2]*Pl[2]*L[9];
Rr[11] = Pl[3]*Pl[3]*L[11] + Pl[4]*Pl[4]*L[11] + Pl[2]*Pl[2]*L[11];
Rr[12] = Pl[3]*Pl[3]*L[12] + Pl[2]*Pl[2]*L[12] + Pl[4]*Pl[4]*L[12];
Rr[14] = Pl[4]*Pl[4]*L[14] + Pl[2]*Pl[2]*L[14] + Pl[3]*Pl[3]*L[14];
```

Listing 3.12: Gaalopausgabe für Reflexion in 5D nach Vince

```
// 1. Schritt
Pl2Q = Pl[2]*Pl[2];
Pl3Q = Pl[3]*Pl[3];
Pl4Q = Pl[4]*Pl[4];

// 2. Schritt
R2[7] = Pl4Q*L[7] + Pl2Q*L[7] + Pl3Q*L[7];
R2[8] = Pl4Q*L[8] + Pl3Q*L[8] + Pl2Q*L[8];
R2[9] = Pl4Q*L[9] + Pl3Q*L[9] + Pl2Q*L[9];
R2[11] = Pl3Q*L[11] + Pl4Q*L[11] + Pl2Q*L[11];
R2[12] = Pl3Q*L[12] + Pl2Q*L[12] + Pl4Q*L[12];
R2[14] = Pl4Q*L[14] + Pl2Q*L[14] + Pl3Q*L[14];
```

Listing 3.13: Manuell weiter optimierte Gaalopausgabe für Reflexion in 5D nach Vince

```
pnorm = x1*e1^e2 + y1*e2^e3 + z1*e1^e3;
ray = x2*e1^e2 + y2*e2^e3 + z2*e1^e3;

?reflectedray = pnorm*ray*pnorm;
```

Listing 3.14: CluCalc für 3D Reflexion nach Vince

```
reflectedray_opt[7]= -2*x1*y1*y2 - 2*x1*z1*z2
  + y1*y1*x2 - x1*x1*x2 + z1*z1*x2;
reflectedray_opt[8]= -2*y1*z1*y2 - 2*x1*z1*x2
  + x1*x1*z2 - z1*z1*z2 + y1*y1*z2;
reflectedray_opt[11]= x1*x1*y2 - 2*y1*z1*z2
  - y1*y1*y2 - 2*x1*y1*x2 + z1*z1*y2;
```

Listing 3.15: Gaalopausgabe für 3D Reflexion nach Vince

Umstellungen der Formeln, Ausklammern und Vorberechnen von Teilergebnissen wird der in Listing 3.16 gezeigte C-Code erzeugt.

```
Direction* reflect3D(Direction* l, Direction* n)
{
  Direction* dir_ref = new Direction(0.1,0,0);

  // 1. Stufe
  double n2Q = n->direction2*n->direction2;
  double n3Q = n->direction3*n->direction3;
  double n4Q = n->direction4*n->direction4;

  // 2. Stufe
  dir_ref->direction2 = 2*n->direction2*n->direction3*l->direction3 +
  2*n->direction2*n->direction4*l->direction4 +
  l->direction2*(n2Q - n4Q - n3Q);

  dir_ref->direction3 = 2*n->direction3*n->direction2*l->direction2 +
  2*n->direction3*n->direction4*l->direction4 +
  l->direction3 * (n3Q - n2Q - n4Q);

  dir_ref->direction4 = 2*n->direction4*n->direction2*l->direction2 +
  2*n->direction4*n->direction3*l->direction3 +
  l->direction4*(n4Q - n2Q - n3Q);

  return dir_ref;
}
```

Listing 3.16: Gaalopausgabe für 3D Reflexion nach Vince

Er beinhaltet 24 Multiplikationen und 12 Additionen und damit rund dreimal so viele wie die Entsprechung in LA.

3.4.6 Beleuchtungsrechnung

Die Beleuchtungsrechnung funktioniert nach dem konventionellen Phong-Modell, welches von Phong bereits 1975 in [Pho75] vorgestellt wurde. Die Beleuchtung eines Punktes ergibt sich als Summe eines ambienten, diffusen und spekularen Anteils. Der ambiente Anteil ist für alle Punkte eines Objekts identisch und unabhängig von der Position der Lichtquelle. Für den diffusen

Wert spielt dagegen die Position der Lichtquelle eine Rolle, da der Winkel zwischen Flächennormale **n** und dem Vektor vom Schnittpunkt zur Lichtquelle **l** in die Berechnung einfließt. Für den spekularen Anteil sind die Richtung des Reflexionsvektors **r** und der Strahl vom Schnittpunkt zum Augpunkt der Szene $\mathbf{r}_{inv}$ entscheidend. Eine Modifikation des Phong-Modelles wurde 1977 von Blinn [Bli77] vorgeschlagen. Hierbei wird der Reflexionsstrahl durch eine Annäherung mit einfacherer Berechnung, dem sogenannten *half-vector*, ersetzt. Da beim rekursiven Raytracing der exakte Reflexionsvektor für die Sekundärstrahlen bestimmt werden muss, wird das klassische Phong-Modell implementiert. Ein weiterer Wert, der *Shininess-Faktor s* gibt außerdem an, wie stark die Spiegelung des Lichtes ausfallen soll. Außerdem ist es für das Phong-Modell nötig die Farbeigenschaften $\begin{pmatrix} r & b & g \end{pmatrix}^T$ des Lichtes für alle drei beschriebenen Anteile zu spezifizieren. Ebenso verfügt das zu beleuchtende Objekt über Materialeigenschaften M, die ebenfalls für jeden Anteil und Farbkanal getrennt angegeben werden. Die vollständige Beleuchtung ist in Gleichung 3.39 zu finden.

$$Color = M_{amb} * L_{amb} + M_{dif} * L_{dif} * \cos(l, n) + M_{spec} * L_{spec} * (\cos(r, e_{inv})^s \tag{3.39}$$

Dabei werden die Winkel zwischen Vektoren jeweils durch ihr Skalarprodukt bestimmt. Möglich wäre für die GA auch eine Winkelberechnung mittels des IPs von 5D Strahlen. Wenn dort ebenfalls normalisierte Punkte vorausgesetzt werden, ähnelt diese Berechnung wieder stark dem LA-Fall und lässt sich auf diesen herunterbrechen. Somit ist die Operationenzahl für beide Algebren bei der Beleuchtungsrechnung identisch.

3.5 Zusammenfassung der Optimierungen

Im Verlauf dieses Kapitels wurden einige manuelle Veränderungen an der Gaalopausgabe vorgenommen, mit denen es gelingt, die Operationenanzahl zu reduzieren. Diese sind zusammengefasst:

- **Konstanten finden:** In praktisch allen betrachteten Berechnungen ist e_0 gleich 1 und da es nur in Multiplikationen vorkommt, kann es ganz vernachlässigt werden. Damit Gaalop diese Optimierung durchführen kann, müsste es bestimmte Eigenschaften von Objekten, wie in diesem Fall von normalisierten Punkten und Kugeln, kennen. Außerdem muss bekannt sein, aus welchen Berechnungen nur normalisierte Objekte entstehen.
- **Nicht benötigte Komponenten erkennen:** Ein Fall für eine nicht benötigte Komponente ist das e_∞ der Punkte. Im Vergleich zu e_0 ist sein Wert zwar nicht konstant, wird aber in keiner weiteren Berechnung benötigt. Letzteres ist aber nur da Fall, da das Verfahren zur Abstandsmessung verändert wurde. Dieser Schritt ist nur manuell und nach theoretischen Überlegungen ersichtlich.
- **Ausklammern:** Besonders viele Multiplikationen können durch das Ausklammern von mehrfach vorkommenden Faktoren vermieden werden. Wenn eine möglichst geringe Operationenanzahl das Ziel der Optimierungen ist, ist eine Erweiterung von Gaalop um die Fähigkeit derartige Faktoren zu finden sinnvoll.
- **Berechnung in mehreren Stufen:** In engem Zusammenhang mit dem Ausklammern steht die, bei den Reflexionen verdeutlichte, Möglichkeit einstufige Berechnungen auf mehrere

Stufen zu verteilen und dadurch eine Verringerung der anfallenden Multiplikationen zu erreichen. Um dieses Vorgehen zu automatisieren wäre es nötig, dass Gaalop alle Multivektoreinträge der aktuellen Berechnung nach gemeinsamen Faktoren absucht, diese in jedem Eintrag ausklammert und gegen eine neue Variable ersetzt, die in einer früheren Stufe berechnet wird.

- **Besondere Optimierungen:** Die Verbesserung der Schnittpunktextraktion ist dagegen ein Beispiel das sich kaum automatisieren lässt. Sowohl die Umstellung der Formel, als auch die Tatsache, dass nur die Bestimmung von $p\_$ erforderlich ist, werden erst nach theoretischen Überlegungen erkennbar.

Wie sich diese Reduzierungen auf die Laufzeit der Raytracer auf parallelen Systemen auswirkt, wird in den nun folgenden Kapiteln geprüft.

4 Raytracen von Kugeln

Die Betrachtung des Themengebietes des Kugel-Raytracings erfolgt vor allem, um zu prüfen, wie sich die Geschwindigkeit von mit Gaalop optimierten GA-Algorithmusstücken im Vergleich zu bestehenden Lösungen verhält. Für die Arbeit ist die Betrachtung deshalb interessant, weil auch für das Dreiecks-Raytracing Kugelschnitte durchgeführt werden. Dort kommt das Konzept der Bounding-Spheres zum Einsatz, bei welchem die Modelle mit Hüllkugeln umschlossen werden. Somit bringt es Vorteile, wenn in der jeweiligen Algebra Schnittentscheidungen mit Kugel und Strahl effizient umgesetzt werden können. Eine weitere praktische Verwendung des Raytracens von Kugeln stellt das, von Don Herbisons-Evans dargestellte, Modellieren von Cartoon-Szenen mittels Ellipsoiden dar [HE74].

Zu Beginn der Untersuchung des Kugel-Raytracings werden zwei Raytracer in C++ entwickelt, welche als Szenenobjekte mit Kugeln umgehen können. Diese werden anschließend mittels OpenMP [CJP07] so erweitert, dass sie zur Ausführung mehrere Kerne von aktuellen CPUs in Anspruch nehmen. Schließlich werden die Algorithmen auch in OpenCL [Khr11] umgesetzt, um zu prüfen, inwiefern diese in der Lage sind, die parallele Architektur der heutigen Grafikkarten auszunutzen bzw. der Compiler in der Lage ist die Parallelität der Berechnungen auf die Hardware umzusetzen. Für GA und LA ist der Grundaufbau des Programmes identisch, und verläuft nach dem in Theorieabschnitt 3.3 dargelegten Prinzip.

Dieses Kapitel soll implementierungsbedingte Unterschiede und verschiedene Messungen bzgl. Leistungsfähigkeit und Bildqualität liefern.

4.1 Repräsentation im Speicher

Während in LA zur Repräsentation einer Kugel ein Kugelmittelpunkt als 3D Vektor, sowie der Radius gespeichert wird, müssen für GA die Werte *s2*, *s3*, *s4* und *s5* im Speicher gehalten werden. Des Weiteren ist jeder Kugel ein Material zugeordnet, um beim Shading die nötigen Farbinformationen zu liefern. Im Material werden jeweils drei Fließkommawerte für ambiente, diffuse sowie spekulare Farbeigenschaften und der Shininess-Wert des Materials gespeichert, woraus zehn Werte resultieren. Beide Algebren benötigen somit pro Kugel gleich viel Speicherplatz, da jeweils vier Fließkommawerte für die geometrischen Kugeldaten, sowie die identischen Materialwerte vorhanden sein müssen.

Die in Abschnitt 3.4.2 besprochenen Verfahren zum Kugelschnitt werden zuerst in C++ umgesetzt, um die Ergebnisse der theoretischen Überlegungen zu prüfen. Für das Verfahren von Stoll/Gumhold [SGS06] ist zunächst eine Umrechnung der vorliegenden Speicherrepräsentation in die Quadrikdarstellung nötig. Bei den Zeitmessungen wird diese Umrechnungsdauer dem Gesamtalgorithmus hinzuaddiert, da die bekannten Verfahren zum Erzeugen von Hüllkugeln oder Kugelannäherungen bei Punktwolken als Ergebnis keine Quadrik, sondern die normale Kugeldarstellung aufweisen [Deu10]. Das LA Verfahren aus [Möl08, S. 739ff], sowie das für

diese Arbeit entwickelte GA-Schnittverfahren können direkt auf den vorhandenen Repräsentationen arbeiten.

4.2 Rechenungenauigkeiten und Numerische Stabilität

Die Grundversion des LA-Kugel-Raytracers basiert bei der Repräsentation der Szenendaten auf Fließkommawerten mit doppelter Genauigkeit, wohingegen die eigentlichen Schnittberechnungen mit einfacher Genauigkeit ausgeführt werden. Die von ihm erzeugten Bilder sind auch zusammen mit dem Einsatz von Sekundärstrahlen artefaktfrei, was Abbildung 4.1 veranschaulicht. Die Kugeln spiegeln sich gegenseitig ineinander und in den Spiegelbildern sind keine fehlenden Pixel zu erkennen. Es werden also alle wirklichen Schnitte von Reflexionsstrahlen mit den Szenenobjekten gefunden. Falsche Schnitte werden dagegen keine berechnet, was an falschfarbigen Pixeln erkennbar wäre. Dass auch die Berechnung des Schattenwurfs bildfehlerfrei möglich ist zeigt Abbildung 4.2.

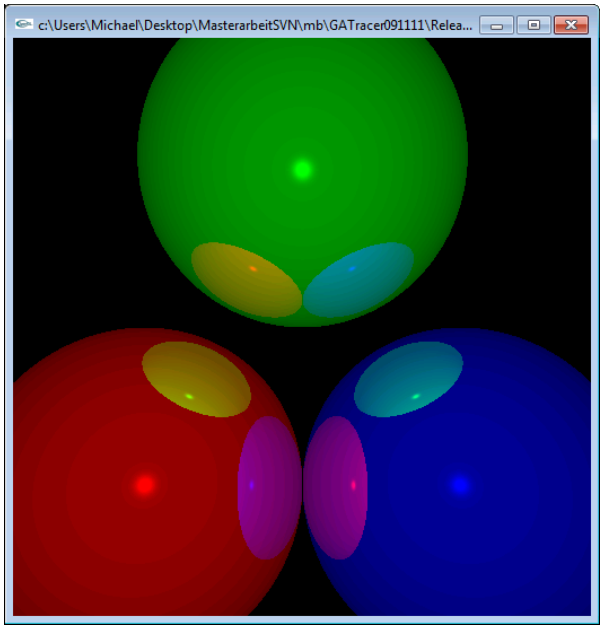

Abbildung 4.1.: Artefaktfreies Bild mit double Genauigkeit

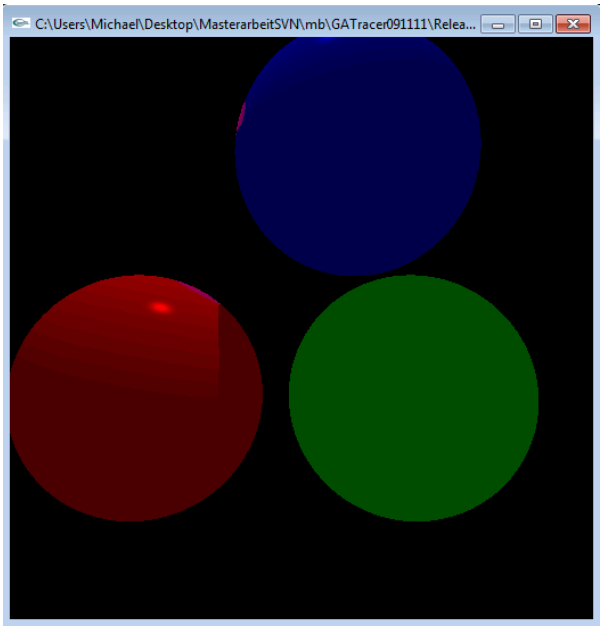

Abbildung 4.2.: Schattenwurf beim Kugel-Raytracer

OpenCL dagegen arbeitet mit Single-Precision, so dass die Darstellung der übergebenen Kugeln in den dortigen Raytracern mit 32bit Fließkommazahlen auskommen muss. Bilder, die lediglich durch Raycasting erzeugt werden, sind auch bei den beiden benutzten LA-Verfahren ohne Bildfehler. Nach dem Zuschalten von Schattenfühlern treten weiterhin keine sichtbaren Artefakte auf. Jedoch führt die Aktivierung der Reflexionsstrahlen, unabhängig vom Vorhandensein von Schattenfühlern, in manchen Szenen bei dem Verfahren von Möller zu wenigen falschen Schnitten, so dass dort einige zu helle Pixel auftreten. Dies passiert aber sehr selten und lässt sich in entsprechenden Szenen durch Anpassung der Schranke für den Parameter t meist vollständig verhindern. Beide Verfahren liefern optisch somit auch ein äquivalentes Bild zu Abb. 4.1. Im Hinblick auf eine Realisierung auf einem FPGA wird auch untersucht wie sich das GA- Verfahren beim Einsatz von Festkommazahlen verhält. Hierzu wird der von den Algorithmen verwendete Datentyp von *double* auf den Festkommatyp der frei verfügbaren *Fixed Point Math Class* von Erik

H. Gawtry [Gaw01] geändert. Anschließend werden die Bilder erneut sowohl visuell als auch mittels Debugger auf Artefakte überprüft. Die Genauigkeit der verwendeten Festkommazahlen beträgt hierbei acht Dezimalstellen nach dem Komma.

Die Ergebnisse zeigen, dass beim Raycasting in den Bildern keine Fehler auftreten, so dass sich die Verfahren auch bei Festkommaberechnung eignen. Die reine Schnittentscheidung wird somit immer korrekt gefällt. Selbst beim Raytracing tauchen nur ganz vereinzelt Artefakte auf, so dass alle drei Verfahren numerisch stabil genug sind um entstehende Rundungsfehler kompensieren zu können.

4.3 Leistungsmessungen am Kugel-Raytracer

Entscheidend für einen Raytracer ist neben der optischen Qualität natürlich auch seine Geschwindigkeit. Daher sollen in diesem Abschnitt die implementierten Versionen auf ihre Performance hin untersucht werden. Hierzu werden zwei Testszenen erstellt, um die Leistungsfähigkeit der CPU- und GPU-basierten Programme messen und vergleichen zu können. Abbildung 4.3 zeigt die Szene, die für die GPU-Messungen verwendet wird. Für die CPU wird dabei die Größe des Gitters auf 216 Kugeln reduziert. Zur Ermittlung der Ausführungsdauer werden im Fall der C++-Programme die Zeitmessfunktionen des Amsterdamer Raytracers benutzt, bei OpenCL wird auf die vorhandene Funktion *clGetEventProfilingInfo* [Khr11, S. 117] zurückgegriffen.

Beim Messen der Zeiten der OpenCL-Kernel stellt sich ein Problem heraus: Alle aktuellen Grafikkarten von AMD und NVidia takten im Windowsbetrieb herunter um Strom zu sparen und schalten erst bei Beanspruchung auf normale Taktraten um. Daher kann das Ergebnis der ersten und evtl. auch der darauf folgenden Zeitmessung negativ beeinflusst werden, je nachdem wie lange deren Berechnung dauert. Daher wird dieselbe Szene in einer Schleife mehrfach erzeugt und erst ab dem fünften Einzelbild mit in die Geschwindigkeitsermittlung einbezogen. Die Zeiten der fünften bis zur achten Messung werden daraufhin gemittelt und auf drei Nachkommastellen genau gerundet. Um das Ergebnis möglichst frei von Messfehlern und einzelnen Ausreißern zu halten, die auch trotz dieses Verfahrens, eventuell auf Grund von Treibereigenschaften, ab und zu auftreten, wird dieses Messverfahren für jede mögliche Konfiguration dreimal wiederholt. Auch aktuelle Desktop-Prozessoren takten im Idle-Betrieb niedriger als unter Last. Obwohl bei allen Stichproben keine Beeinflussung der ersten Messzeit festzustellen ist, wird das für OpenCL beschriebene Verfahren auch für die Messung der C++-Programme übernommen. In den folgenden Tabellen repräsentiert die Spalte *LA 1* das Verfahren von Möller, die Spalte *LA 2* dasjenige von Gumhold und die dritte schließlich den Algorithmus in GA. Die einzelnen Zeilen stehen für:

- **RC**: Raycasting
- **RR**: Reflexionsstrahlen aktiviert
- **SR**: Schattenfühler aktiviert
- **RRT**: Rekursives Raytracing
- **RRTMP**: Rekursives Raytracing mit OpenMP

4.3.1 Auswertung der CPU Performance

Die für die Auswertung gemessenen Zeiten beziehen sich lediglich auf den eigentlichen Renderingvorgang. Das Vorbereiten der Szenendaten wird dabei nicht berücksichtigt. Ebenso wird die Initialisierung der OpenGL-Umgebung und das eigentliche Anzeigen des Bildes nicht mit einbezogen. Tabelle 4.1 gibt die Messwerte für die C++-Raytracer wider. Hierbei sind alle Compilereinstellungen für den Viusal-C++-Compiler auf Performance/Speed gestellt.

	LA 1	LA 2	GA	Diff
RC	0,662	0,757	0,875	-24,34 %
RR	0,852	0,952	1,156	-26,82 %
SR	0,866	0,941	1,152	-24,82 %
RRT	1,115	1,213	1,464	-23,84 %
RRTMP	0,621	0,704	0,811	-23.43 %

Tabelle 4.1.: Kugel-Raytracing auf der CPU. Zeitangaben in Sekunden. Die letzte Spalte gibt den Geschwindigkeitsvorteil der jeweils schnellsten LA-Version gegenüber GA wider. Optimierte Compiler-Einstellungen. Ausgeführt auf TS1.

Das Betrachten der Werte ergibt, dass die Unterschiede in den Berechnungsgeschwindigkeiten der drei Raytracer ziemlich genau mit den in Abschnitt 3.4.2 ermittelten Unterschieden in der Operationenanzahl skalieren. Ein rekursiver Raytracer auf Basis von Möllers Algorithmus erzeugt das Bild fast ein Drittel schneller als die GA Version. Diese Unterschiede gelten ebenfalls für das Raycasting und die Zwischenglieder (nur Reflexionsstrahlen und nur Schattenfühler). Das Stoll/Gumhold-Verfahren platziert sich in Sachen Performance zwischen den beiden anderen, was seiner Operationenzahl entspricht. Auf einem Kern berechnet, ohne Autovektorisierungs-Unterstützung im Compiler, kann die GA also nicht mit bestehenden Lösungen konkurrieren. Bei der Hinzunahme von OpenMP bleibt die prozentuale Differenz ebenfalls gleich. Allerdings dreht sich das Bild bei der Aktivierung der Streaming SIMD Extensions (SSE2) bei den Compileroptionen, was in Tabelle 4.2 zu sehen ist.

	LA 1	LA 2	GA	Diff
RC	0,504	1,397	0,377	+33,69 %
RR	0,759	1,882	0,506	+50,00 %
SR	0,683	1,878	0,497	+37,42 %
RRT	0,889	2,383	0,647	+37,40 %
RRTMP	0,466	1,269	0,368	+26,63 %

Tabelle 4.2.: Kugel-Raytracing auf der CPU. Zeitangaben in Sekunden. Die letzte Spalte gibt den Geschwindigkeitsrückstand der jeweils schnellsten LA-Version gegenüber GA wider. SSE2 aktiviert. Ausgeführt auf TS2.

Am Auffälligsten ist die zweite Spalte, da ihre Werte deutlich über denen ohne aktivierte SSE2 liegen. Um auszuschließen, dass dieses Problem nur bei der vorhandenen Testszene vorliegt, wurden verschiedene andere Situationen gemessen. Aber sowohl in Fällen mit nur drei Kugeln (0,151sec gegen 0,220sec), als auch bei 1600 Kugel (6,385sec gegen 24,889sec) bleibt die Tatsache bestehen, dass Stoll/Gumhold mit SSE langsamer arbeitet als ohne. Die nächstliegende

Erklärung hierfür ist, dass die SIMD-Register nicht ausreichen und somit kein effizienter Code erzeugt werden kann. Die entsprechende Tabellenspalte wird daher bei der Auswertung nicht beachtet. Interessant ist vor allem die dritte Spalte. Die GA Version profitiert enorm von SSE2, da sich die Geschwindigkeit in jeder Testeinstellung mehr als verdoppelt. Die Parallelisierung, welche die Einführung von SSE zum Ziel hatte, kann hier optimal umgesetzt werden. Möllers Kugelschnitt profitiert zwar auch von der Nutzung von SSE2, doch der Vorteil fällt mit 30% Geschwindkeitszuwachs im Vergleich zur GA moderat aus.

4.3.2 Auswertung der GPU Performance

Die Library-Funktion *clGetEventProfilingInfo* misst lediglich die reine Laufzeit des Kernels, weshalb die Zeiten von der Übertragung der Daten auf die Grafikkarte und das Auslesen der Ergebnisse nicht mit eingerechnet werden. Vor allem bei kleinen Szenen, die dementsprechend schnell gerendert werden können, macht die Übertragungszeit den überwiegenden Anteil der Laufzeit aus, weshalb es auch sinnvoll erscheint, sie nicht mit zu berücksichtigen. Eine der verwendeten Testszenen zeigt Abbildung 4.3

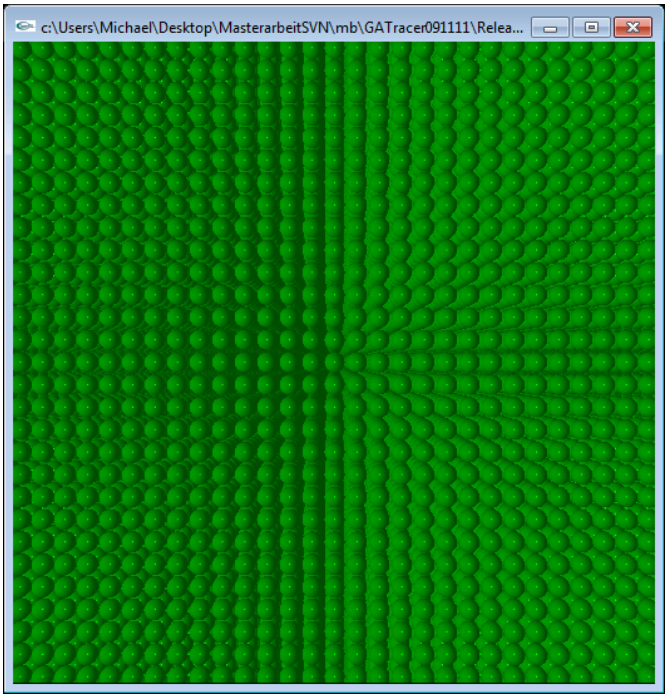

Abbildung 4.3.: Testszene 01 für den Kugelraytracer. Sie besteht aus rund 25000, in einem Gitter angeordneten, gleich großen Kugeln

Tabelle 4.3 gibt die gewonnenen Ergebnisse wider, wobei die Laufzeit jeweils in Sekunden angegeben ist.

In der ersten Zeile fällt auf, dass das GA-Verfahren schneller rendert als die beiden LA-Gegenstücke. Eine Untersuchung mittels des AMD APP Profiler Tools [Adv10] ergibt, dass die GA es trotz der theoretischen Mehrzahl an Operationen schafft, die ALUs der Grafikkarte besser auszulasten und somit eine kürzere Berechnungszeit erzielt. Eine ausführliche Untersuchung eines GPU-Kernels mittels des Profilers findet sich in Abschnitt 5.5.2, weshalb an dieser Stelle auf Details verzichtet wird. Bezüglich der Auslastung der ALUs schneidet das Verfahren

	LA 1	LA 2	GA	Diff
RC	0,169	0,160	0,149	+7,38 %
RR	0,397	0,382	0,378	+1,01 %
SR	0,335	0,306	0,305	+0,33 %
RRT	0,901	0,824	0,984	-8,43 %

Tabelle 4.3.: Kugel-Raytracing auf der GPU. Zeitangaben in Sekunden. Die letzte Spalte gibt den Geschwindigkeitsunterschied der jeweils schnellsten LA-Version gegenüber GA wider.

von Möller am schlechtesten ab und ist das langsamste, obwohl es die wenigsten Operationen aufweist. Somit arbeitet der GA Algorithmus rund 7% schneller, als der schnellste LA-Vertreter. Bei einer Deaktivierung der Schnittpunktbestimmung, also beim reinen Test auf Schnitt, bleibt der Unterschied von 7% erhalten. Wie aber Abschnitt 3.4.2 entnommen werden kann, benötigt das Stoll/Gumhold-Verfahren für den reinen Schnitttest mehr Multiplikationen als die GA und die Profilerausgabe bestätigt, dass der Stoll/Gumhold-Algorithmus die ALUs stärker auslastet als GA, in der Summe aber trotzdem etwas langsamer bleibt. Bei den Zeilen zwei und drei von Tabelle 4.3 kann festgestellt werden, dass die drei Verfahren hier deutlich näher zusammenrücken und der Vorsprung der GA verschwindet, da sie im Mittel jetzt nur noch einen Prozentpunkt schneller als das Verfahren von Stoll/Gumhold ist. Die ALU-Auslastung ist bei GA zwar immer noch am höchsten, kann die höhere Anzahl der Berechnung aber nur noch kompensieren und nicht mehr für einen wirklichen Geschwindigkeitsvorteil sorgen. Beim rekursivem Raytracing dreht sich schließlich das Bild und die LA überholt bezüglich Geschwindigkeit die GA. Aber auch hier bedeutet die Tatsache, dass das Möller-Verfahren die wenigsten Operationen hat, nicht, dass es deshalb das performanteste ist, denn die Stoll/Gumhold Version arbeitet 10% schneller. Auch hier gilt, dass die ALU-Auslastung bei letzterem deutlich höher ist, was zur kürzeren Berechnungsdauer führt. Die schlechtere Leistung von Möller kann darauf zurückgeführt werden, dass dieser Algorithmus die meisten Branching-Anweisungen enthält.

4.4 Fazit zum Kugel-Raytracen

Das Raytracing von Kugeln verhält sich bei den normalen C++-Versionen so, wie es anhand der theoretischen Untersuchungen zu erwarten war und die GA ist ein ganzes Stück langsamer als etablierte Verfahren. Unter SSE kann die GA dagegen deutlich die LA überholen. Mit OpenCL ist das Bild dagegen zweigeteilt. Werden die reinen Schnitttests und das Raycasting betrachtet, so arbeitet die GA performanter, beim Raytracing fällt sie dann hinter die Standardverfahren zurück.

Somit kann für dieses Kapitel festgehalten werden, dass die GA beim Rendern von Kugeln stets dort deutlich profitiert, wo Parallelität ihrer Berechnungen umgesetzt werden kann, sei dies mittels der SSE-Technologie oder AMDs OpenCL-Compiler.

5 Raytracen von Dreiecken

Neben den Raytracern für Kugeln werden auch solche für Dreiecksnetze auf Basis von LA und GA in C++ entwickelt, deren Aufbau und Funktionsweise in diesem Kapitel beschrieben wird. Es werden Implementierungsdetails erörtert und zum Abschluss des Kapitels die Performance der verschiedenen Programme analysiert und verglichen, sowie die Ergebnisse interpretiert.

5.1 Details der Implementierung

5.1.1 Objekte der Algebren

Für den LA-Raytracer werden die folgenden bekannten Konstrukte und geometrischen Objekte der Algebra implementiert. Dazu zählen die Datentypen:

- *Point3D*: Dreidimensionaler Punkt, der seine drei Koordinaten jeweils als *float* speichert und eine Referenz auf seine Normale hält, die als *Direction* realisiert ist.
- *Direction*: Dreidimensionale Richtung, die ihre Einträge ebenfalls in drei *floats* speichert.
- *Ray3D*: Strahl, der einen Aufpunkt und eine Richtung beinhaltet.
- *Kugeln*: Werden wie in Abschnitt 4 beschrieben gespeichert, allerdings ohne Materialeigenschaften.
- *Dreiecke*: Ihre Speicherung wird im Rahmen von Abschnitt 5.1.3 behandelt

Außerdem werden die Operationen der LA, zu denen vor allem Skalarprodukt und Kreuzprodukt gehören, implementiert. Weiterhin sind Funktionen zur Normalisierung, Addition / Subtraktion von Vektoren sowie entsprechende benötigte Konstruktoren vorhanden. Somit ist es möglich, dass sich ein *Ray3D* zum einen aus zwei 3D Punkten und zum anderen aus einem Punkt und einer Richtung erstellen lässt. Für die GA ist die Anzahl der Objekte etwas höher:

- *Point5D*: Für die im Theorieteil dargestellten konformen Punkte, müssten fünf Werte gespeichert werden. Da allerdings bei sämtlichen Berechnungen des Algorithmus nur normalisierte Punkte entstehen können, wird *s6* stets den Wert 1 haben und muss somit nicht gespeichert werden. *s5* lässt sich bei Bedarf aus *s2*, *s3* und *s4* berechnen, wird allerdings in der momentanen Programmversion zu keinem Zeitpunkt benötigt. Somit werden für einen GA-Punkt nur die Koeffizienten *s2*, *s3* und *s4* als drei float-Einträge gespeichert. Außerdem speichert jeder Punkt seine Normale als *Direction*.
- *Plane5D*: Für das Schnittverfahren wird, wie im Theorieteil dargelegt, eine Ebene zur Gültigkeitsprüfung benötigt. Diese braucht zur Speicherung vier float-Werte.
- *Ray5D* und *DualRay*: Sowohl ein konformer Strahl als auch seine duale Entsprechung existieren als Objekte. Beide werden durch sechs float-Werte beschrieben und unterscheiden

sich vor allem durch ihre Konstruktoren. Ein *Ray5D* speichert zusätzlich einen Punkt, der als Strahlenursprung für Sicht- / Reflexionsstrahlen, sowie die Schattenfühler interpretiert wird.

- *Dreiecke*: Es wird das gleiche Objekt wie für die LA verwendet, welches im folgenden Abschnitt 5.1.3 beschrieben wird.

Wie in Abschnitt 2.4 erläutert, werden nicht nur spezielle Objekte einem allgemeinen GA-Objekt mit 32 Einträgen vorgezogen, sondern auch alle vorkommenden Operationen speziell implementiert und optimiert. Somit unterscheiden sich die Realisierungen des IPs anhand ihrer Operanden. Listing 5.1 zeigt hierzu ein Beispiel.

```
// RAY5D.DUALRAY
float indi1 = r->ray7*edge1.ray18 + r->ray8*edge1.ray20 +
  r->ray9*edge1.ray22 + r->ray11*edge1.ray23 +
  r->ray12*edge1.ray25 + r->ray14*edge1.ray26;


// Point5D.Plane5D
float dist = (pl2*p->point2 + pl3*p->point3 + pl4*p->point4)
  - pl5*p->point6;
```

Listing 5.1: Vergleich zweier Innerer Produkte

Im oberen Teil von Listing 5.1 wird das IP zwischen einem *Ray5D* und einem *DualRay* berechnet, im unteren Teil dagegen das IP zwischen einem *Point5D* und einer *Plane5D*. Beide Produkte werden bei der Schnittpunktberechnung benötigt. Es lässt sich anhand des Codes erkennen, dass jeweils verschiedene Blades an den Produkten beteiligt sind und unterschiedlich viele Multiplikationen anfallen.

5.1.2 Die Szene

Der zentrale Bestandteil für alle Raytracer ist das *scene*-Objekt, welches für alle implementierten Dreiecks-Raytracer identisch ist. Dadurch wird sichergestellt, dass keine Version auf Grund eines leicht variierten Szenenobjektes eine andere Geschwindigkeit erzielt. Pro Aufruf des Programms gibt es nur eine einzige Instanz des Szenenobjekts. Sie ist nach dem Entwurfsmuster des *Singleton* [GHJV94, S. 157-167] realisiert. Die Szene dient vor allem der Speicherung der Dreiecksdaten, welche für die CPU- und GPU-Raytracer in verschiedenen Formen vorgehalten werden. Durch diese Designentscheidung erhöht sich der Arbeitsspeicherbedarf der Implementierung deutlich, dafür sind Änderungen, die alle Raytracer betreffen, schneller durchzuführen. Außerdem werden die Kamera, die Beleuchtung, die Materialeigenschaften und die Hüllkugeln von der Szene zur Verfügung gestellt.

5.1.3 Dreiecksnetze

Für die Dreiecksnetze müssen zwei Arten der Repräsentation unterschieden werden. Zum einen das Dateiformat, in dem die Dreiecksnetze auf der Festplatte vorliegen und zum anderen die Datenstruktur, in der die Netze während der Programmausführung im Arbeitsspeicher des

Rechners für die Bearbeitung durch die Raytracing-Algorithmen gehalten werden. Diese beiden Punkte werden im Folgenden behandelt.

Quelldateien

Die Raytracer können Dreiecksnetze in zwei verschiedenen Formaten verarbeiten. Das auch vom Amsterdamer Raytracer (A-Tracer) verwendete, von Dorst et al. selbst entwickelte, rtm-Format und das in der Graphischen Datenverarbeitung allgemein genutzte obj-Format.

Das rtm-Format besteht aus zwei Blöcken in der Datei. Zuerst wird definiert, aus wie vielen Dreiecken ein Netz besteht und welche Punkte zu welchem Dreieck gehören. Außerdem wird ein eindeutiger Index pro Dreieck vergeben. Die einzelnen Punkte werden dabei mit einem Index referenziert, was Listing 5.2 veranschaulicht.

```
1 number_of_faces 4096 // gibt Anzahl der Dreiecke an
  face 0 vtx 0 1 2 // das erste Dreieck des Netzes
```

Listing 5.2: Die ersten beiden Zeilen einer rtm-Datei

Im Anschluss daran werden die einzelnen Punkte, sowie deren Normalen und Texturkoordinaten definiert, was in Listing 5.3 zu sehen ist.

```
  number_of_vertices 2713 // Anzahl der Knoten im Netz
2 vertex  0
    pos 0.384615 0.000000 0.659341 // Definition des ersten Punktes
4   normal -0.935454 0.030608 -0.352121
    uv 0.035754 0.075000
```

Listing 5.3: Beginn des zweiten Blocks einer rtm-Datei

Als Basis für den Einlesecode für rtm-Netze wird das entsprechende Modul des Amsterdamer Raytracers verwendet und der Code für die benutzte Repräsentation der Netze im Speicher angepasst. Der Quellcode für den A-Tracer und weitere ihn betreffende Informationen können in [DFM07a] gefunden werden. Für das rtm-Format liegen nur zwei Testdateien (*Geosphere* und *Teapot*) vor, welche dem Archiv des Amsterdamer Raytracers beiliegen. Die Möglichkeit sie zu verarbeiten ist deshalb integriert, um einen direkten Vergleich in Sachen Geschwindigkeit und Bildqualität zwischen dem A-Tracer und den selbst entwickelten Versionen zu ermöglichen. Um die Anzahl an Testobjekten im rtm-Format zu erhöhen, sind die eigenen Raytracer um die Fähigkeit erweitert eingelesene Dreiecksnetze in rtm zu konvertieren.

Das von der Firma *Wavefront Technologies* entwickelte obj-Format ist ähnlich aufgebaut. Wie Listing 5.4 zeigt, existieren Zeilen für Punkt (v), Normalen (vn) und Dreiecksdefinitionen (f). Bei der Festlegung der Dreiecke trennen // die einzelnen Informationen für die Eckpunkte. Der Wert links des // gibt den Index des Punktes an, während rechts davon der Index der Normalen im entsprechenden Eckpunkt folgt. Des Weiteren ist es mit dem obj-Format ebenfalls möglich Texturkoordinaten anzugeben, sowie Referenzen auf Materialien, doch diese beiden Möglichkeiten sind nicht in den Einlesealgorithmus integriert, da sie von den Raytracern selbst nicht unterstützt werden.

```
vn −0.554625 0.831346 −0.035431 // Normale mit Index 1
v 0.433520 0.387984 −0.000003 // Punkt mit Index 1

f 107//107 118//118 1//1 // eine Fläche, die Punkt 1 enthält
```

Listing 5.4: Das obj-Dateiformat

Das Einlesemodul für obj-Dateien basiert auf der frei erhältlichen SmallVR Bibliothek [Pin06]. Auch diese ist auf die verwendete Dreiecksdatenstruktur hin adaptiert, welche im weiteren Verlauf des Abschnitts beschrieben wird.

Erweitert ist das Modul zum Einlesen von obj-Dateien vor allem um die Funktion Normalen für die Punkte zu interpolieren, da einige der Testobjekte, in ihren Quelldateien keine Normaleninformationen enthalten. Dies betrifft vor allem die Objekte, die von der Dreiecksanzahl und den sonstigen Anforderungen her gut für die Untersuchung geeignet sind. Um die Normalen zu erhalten wird, wie in Linearer Algebra üblich, die Flächennormale mittels Kreuzprodukt der Seitenvektoren für jedes vorhandene Dreieck berechnet. Anschließend werden für jeden Punkt die Flächennormalen der Dreiecke, in denen er vorkommt, gemittelt und als Eckpunktnormale zugewiesen. Dies ist auch der Weg, der beim traditionellen Phong-Shading benutzt wird. Mittlerweile existieren zwar Erweiterungen dieser Methode, wie sie beispielsweise von van Overveld et al. [OW97] vorgeschlagen werden, oder auch neue Konzepte wie von Reshetov et al. [RSM10] dargelegt. Diese bieten vor allem eine bessere Glattheit der Normalen an Übergängen. Die Benchmarks von Reshetov et al. legen nahe, dass ihre Lösung in Sachen Geschwindigkeit sowohl auf CPUs (i5-450M) als auch auf GPUs (CUDA-Realisierung auf einer NVidia GTX480 [NVI10]), den Phong-Normalen ebenbürtig sind. Da die Normaleninterpolation in den Raytracern als Vorarbeit auf der CPU für den eigentlichen Bilderzeugungsvorgang nicht mit in die Performancemessung einbezogen wird und exaktere Normalen keinen Einfluss auf die Berechnungsgeschwindigkeit haben, werden die Phong-Normalen eingesetzt.

Repräsentation im Speicher

Für die Dreiecks-Objekte liegen zwei verschiedene Implementierungen vor, je nachdem, ob sie in einem C++- oder einem OpenCL-Raytracer zum Einsatz kommen. Die Dreiecke in C++ speichern ihre Eckpunkte explizit, so dass die Daten für Punkte, die in mehreren Dreiecken vorkommen, mehrfach im Speicher gehalten werden. Ein Manko ist ein höherer Speicherverbrauch; der Vorteil die direkte Verfügbarkeit der Koordinaten während des Raytracing-Algorithmus. Leistungsmessungen ergeben, dass die momentan eingesetzte Variante bei den untersuchten Testszenen schneller ist. Die weiter unten getätigten Leistungsmessungen für die OpenCL-Raytracer basieren dagegen auf Kerneln, welche in den Dreiecksdaten nur Referenzen auf ihre Eckpunkte halten. Dadurch wird die Datenmenge der Netze verringert. Dies bietet bei der Datenübergabe vom Programm an den OpenCL-Kernel den Vorteil, dass weniger Kommunikation nötig ist, welche, gemessen an der Laufzeit, eine der größten Posten ist. Es wird aber auch eine Version betrachtet, die zur C++-Implementierung analoge Datenstrukturen aufweist.

5.2 Hüllkörper

Um die Anzahl der durchzuführenden Schnitttests zwischen Strahlen und Dreiecken zu reduzieren, werden zunächst die einzelnen Dreiecksnetze während des Einlesevorganges mit einer Hüllkugel umgeben. Beim Verfolgen eines Strahles wird zuerst geprüft, ob die Hüllkugel geschnitten wird und nur im Fall eines Schnittes werden die von der Kugel umgebenen Dreiecke auf einen Schnitt hin geprüft. Die Konstruktion der Bounding-Sphere (BSphere) übernimmt auch im Falle der OpenCL-Raytracer die CPU und übergibt diese dann zur weiteren Berechnung an die Grafikkarte. Zu Beginn der Implementierung, wurde der gleiche Algorithmus wie im A-Tracer verwendet. Dieser wird in [DFM07b, S. 563 - 564] beschrieben und die C++-Realisierung gezeigt. Er basiert darauf, dass über alle Punkte des Dreiecksnetzes iteriert wird und der minimale und maximale Wert des x-, y- sowie des z-Abstandes zum Ursprung gesucht wird. Die Koordinaten des Mittelpunkts der BSphere errechnen sich durch Mittelwertbildung der Minimal- und Maximalwerte der zuvor berechneten Abstände. Der Radius ist der maximale Abstand des Mittelpunktes zu allen Netzpunkten. Der Algorithmus ermöglicht zwar eine schnelle Berechnung einer Hüllkugel, doch es lässt sich demonstrieren, dass die Annäherung je nach Objekt ungenügend ist und somit viele unnötige Dreiecksschnittberechnungen durchgeführt werden. Daher wird das Verfahren zur Berechnung gegen Bernd Gärtners *Miniball* Algorithmus ersetzt [GÖ9]. Die Grundlage für den C++ Code bildet die im Internet verfügbare Implementierung durch Gärtner [GÖ6]. Abbildung 5.1 zeigt die berechnete Hüllkugel für das Objekt *cow1.obj*.

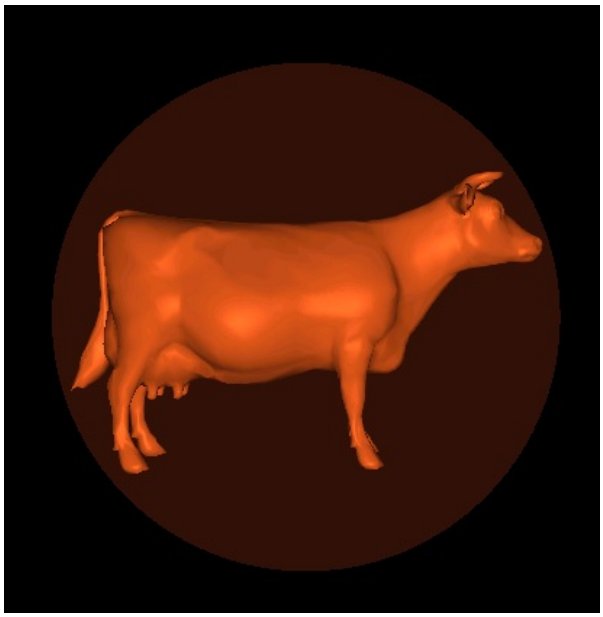

Abbildung 5.1.: BSphere für ganzes Netz (GA)

Abbildung 5.2.: BSpheres für unterteiltes Netz (GA)

Bei größeren Modellen schwindet der Nutzen einer einzigen BSphere allerdings, da die umschließende Kugel mit dem Modell wächst und mehr Pixel abdeckt. Somit wächst im Mittel die Anzahl der Dreiecksschnitte, mit negativem Ergebnis pro zu berechnendem Pixel. Daher wird eine weitere Verbesserung für den Raytracingvorgang eingeführt. Bevor die Dreiecksnetze an die Grafikkarte gesendet bzw. an den C++-Raytracing-Algorithmus übergeben werden, erfolgt

eine Zerlegung in kleinere Unternetze, die jeweils mit einer BSphere umgeben werden. Ein vom Benutzer festzulegender Faktor, der *Splitfactor*, gibt an, wie viele Dreiecke zu einem derartigen Unternetz zusammengefasst werden. Bei mehreren Objekten wird für jedes neue Objekt auch ein neues Netz begonnen; verbleibende Dreiecke des vorigen Objektes werden nicht übernommen. In einer Szene mit Splitfactor 500 und zwei Objekten, bei der das erste aus 800 und das zweite aus 900 Dreiecken besteht, werden vom Algorithmus daher vier Unternetze erstellt. Für Objekt Nummer 1 eines bestehend aus 500 und eines aus 300 Dreiecken; Objekt 2 wird in 500 und 400 Dreiecke zerlegt. Ein anderer Ansatz ist es in jede BSphere pro Objekt über gleich vielen Dreiecken zu konstruieren. Wie sich der Splitfactor genau auf die Leistungsfähigkeit der Algorithmen auswirkt, wird später betrachtet.

Eine weitere Idee zur Erhöhung der Geschwindigkeit ist der Versuch beide Verfahren zu kombinieren. Es wird für jedes Objekt sowohl die eigentliche Hüllkugel als auch die Hüllkugeln der Unternetze übergeben. Beim Rendern wird zuerst geprüft, ob die Objektkugel vom Strahl getroffen wird und erst im Trefferfall die Unterkugeln bzw. anschließend die darin liegenden Dreiecke auf Schnitt geprüft. Somit liegt eine zweistufige Hüllkugel-Hierarchie vor. Eine signifikante Leistungssteigerung für dieses Verfahren bleibt allerdings sowohl bei OpenMP als auch bei OpenCL aus. Die Laufzeit der OpenMP Version der Raytracer wird für kleinere Modelle um rund 10 % geringer, bei größeren Modellen schmilzt der Vorteil allerdings mit zunehmender Objektzahl, bis er sich schließlich auflöst. Die reduzierte Anzahl der Kugelschnitte fällt bei genügend großer Anzahl von Dreiecksschnitten nicht mehr ins Gewicht und allein die Menge der letztgenannten bestimmt die Leistung. Bei OpenCL verhält es sich ähnlich. Hier verliert die Version mit dem neuen Hüllkugel-Verfahren für größere Modelle allerdings bereits den Performancevergleich zur Standardversion. Dies liegt mit hoher Wahrscheinlichkeit an der erhöhten Anzahl von Sprungbefehlen, welche das „optimierte“ Verfahren erzeugt. Diese sind von der Grafikhardware schwer zu handhaben, da sie auf Grund der Architektur viele Taktschritte zur Ausführung benötigen. Der Vorteil der reduzierten Kugelschnittzahl wird daher nicht nur aufgehoben, sondern durch die erhöhte Laufzeit der Sprungbefehle verdeckt. Eine genaue Messung des Kernelverhaltens mittels AMDs Stream Profiler ist für das neue Verfahren nicht möglich, da beim Analysevorgang der Rechner komplett abstürzt. Weder eine Aktualisierung des Stream SDK, noch des ATI Catalysttreibers bringen in dieser Hinsicht eine Verbesserung. Eine Antwort des AMD/ATI-Supports zu diesem Problem steht aus. Somit ist in der aktuellen Version das Verfahren eingeschaltet, das die Modelle in Unternetze unterteilt und diese mit Bounding-Spheres umgibt.

5.3 Das Benutzerinterface

Zur Erstellung der Szenen und der Auswahl des Renderers ist eine spezielle Benutzeroberfläche vorhanden, welche in Abbildung 5.3 zu sehen ist.

Zur Darstellung der Oberfläche kommt das frei verfügbare QT-Framework [Nok11] von Nokia zum Einsatz. Auf der linken Seite existiert ein Vorschaufenster der Szene, welches über die normale OpenGL Rendering Pipeline erstellt wird [OSW+05, S. 10-13]. Die Szene lässt sich in diesem Fenster mit Hilfe der Maus drehen. Das rechte Fenster zeigt die Ausgabe des gewählten Raytracers, der sich im DropDown-Menü darunter auswählen lässt. Die Bildfehler in Form kleiner violetter Striche, die man an der Teekanne erkennt, stammen aus der Kombination des QT-Frameworks und der HD6970. Sie sind nicht durch den Algorithmus bedingt. Sie sind unabhängig von der verwendeten Version des AMD Treibers und treten bei einer HD5770 nicht

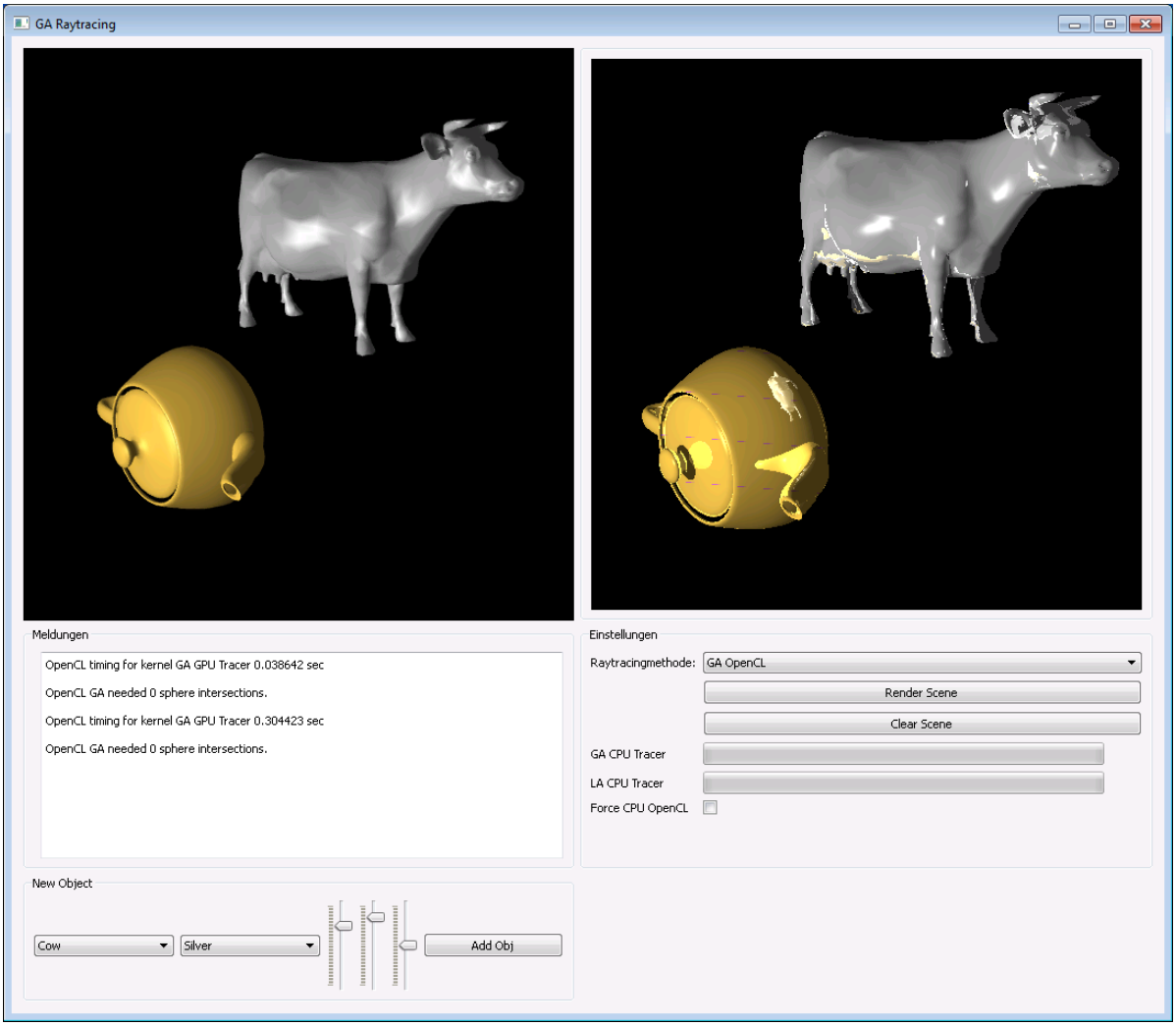

Abbildung 5.3.: GUI des Renderers

auf. Eine Beseitigung gelang auch mit den Ratschlägen seitens des Nokiasupports nicht. Der Button *Render Scene* veranlasst das Programm die generierte Szene zu raytracen, während der *Clear Scene* Button alle Objekte entfernt. Um neue Objekte hinzuzufügen, wird links unten das gewünschte Objekt gewählt, seine Position im Raum mittels der drei Slider festgelegt und die Auswahl anschließend mit *Add Obj* bestätigt. Im Meldungsfenster wird angezeigt, wie lange der Renderingvorgang gedauert hat.

5.4 Bildqualität

Bevor die Geschwindigkeit der einzelnen Tracer-Versionen ausgewertet wird, soll sich dieser Abschnitt mit der Bildqualität beschäftigen. Hierzu wird eine kleine Testszene erstellt, bei der sowohl Schattenwurf als auch Spiegelungen vorhanden sind. Abbildung 5.4 verdeutlicht dies.

Die Kuh wirft einen Schatten auf die Kugel und spiegelt sich in dieser. Außerdem wird die Kugel an der Kuh reflektiert. Bei dieser Ansicht ist zwischen den Raytracern kein Unterschied zu erkennen, weshalb die Abbildungen 5.5, 5.6, 5.7 und 5.8 einen Ausschnitt zeigen, bei dem die

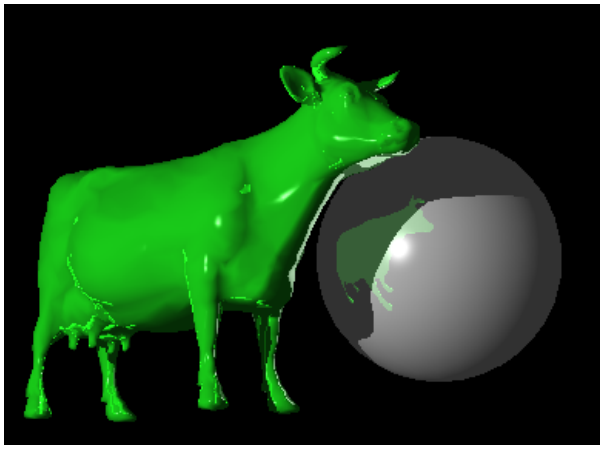

Abbildung 5.4.: Testszene für die Bildqualität (LA-GPU)

Kamera deutlich näher an der Szene positioniert ist. Erkennbar ist, dass alle vier Programme die Modelle vollständig zeichnen und nicht durch falsche Schnittentscheidungen Dreiecke ausgelassen werden. Die Reflexion der Kuh in der Kugel ist in allen Fällen einheitlich und korrekt, sowohl der Teil im Licht, als auch derjenige im Schatten. Der Schatten selbst verläuft ebenfalls einheitlich. Die Spiegelungen der Kugel auf der Kuh sehen jeweils gleich aus. Unterschiede gibt es in den Selbstreflexionen an der Kuh, vor allem unterhalb des Ohres. Die GA-Raytracer erzeugen jeweils das gleiche Bild. Auch nach einer Erhöhung der Rechengenauigkeit auf *double* gibt es bei den Algebren keine Unterschiede im Bild. Da selbst nach einer Umstellung der Berechnung der Reflexionsstrahlen auf das LA-Verfahren keine Änderungen am Bild auftreten, resultieren die Differenzen aus den eigentlichen Schnittalgorithmen. Bei beiden Algebren ist für ein fehlerfreies Bild die Einstellung einiger Parameter wichtig, da ein direktes Übernehmen der Bedingung $x > 0$ zu Artefakten führt. Der passende Wert für GA ist $2e - 3$, da für diesen keine Bildfehler erzeugt werden konnten. Auch für die LA ist er passend.

Im nachfolgenden Kapitel 5.5 über die Geschwindigkeit der Versionen sind weitere Screenshots zu finden, welche die Qualität der erzeugten Bilder belegen. Dabei wird stets angegeben, mit Hilfe welcher Algebra die Szenen gerendert werden.

5.5 Leistungsmessungen am Dreiecks-Raytracer

5.5.1 Auswertung der CPU Performance

Für die CPU Performance werden vier Fassungen der beiden Raytracer untersucht. Die ersten beiden unterscheiden sich lediglich in den Compilereinstellungen des Visual Studio 2008. Die erste Version wird ohne aktivierte Optimierungen kompiliert, wohingegen bei der zweiten Version folgende Optionen gesetzt werden:

- **Optimization**: *Full Optimization*
- **Favor Size or Speed**: *Favor Fast Code*
- **Whole Program Optimization**: *Enable link-time-code generation*

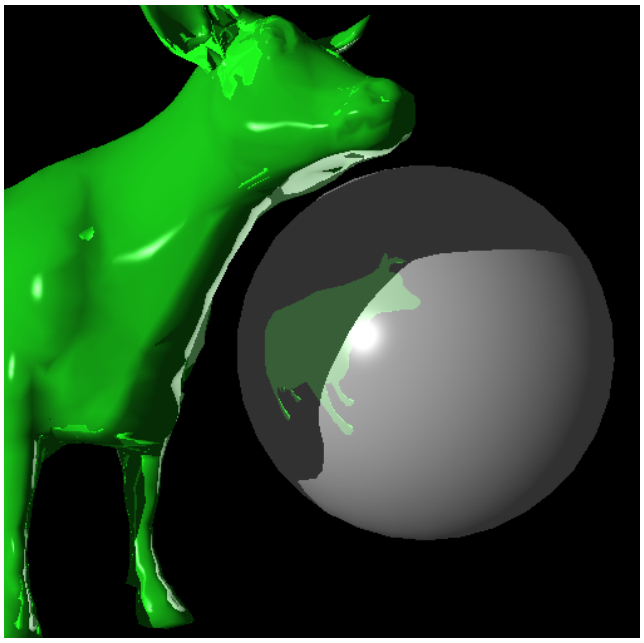

Abbildung 5.5.: Detail der Szene mit GA-GPU

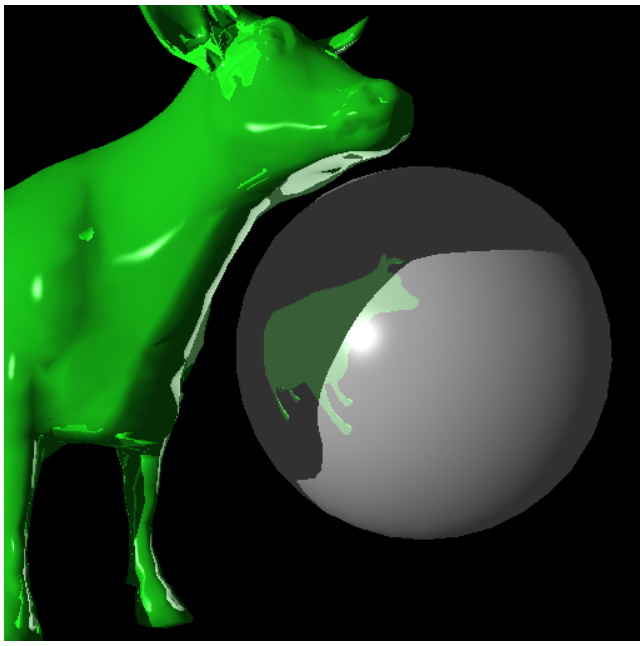

Abbildung 5.6.: Detail der Szene mit GA-CPU

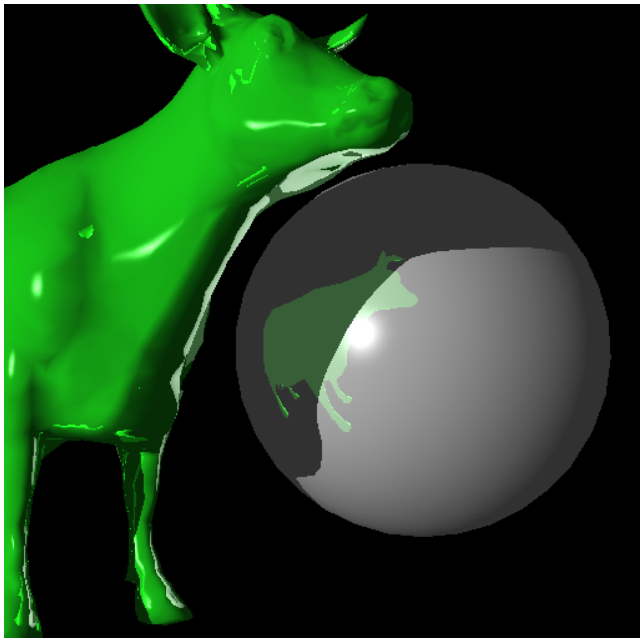

Abbildung 5.7.: Detail der Szene mit LA-GPU

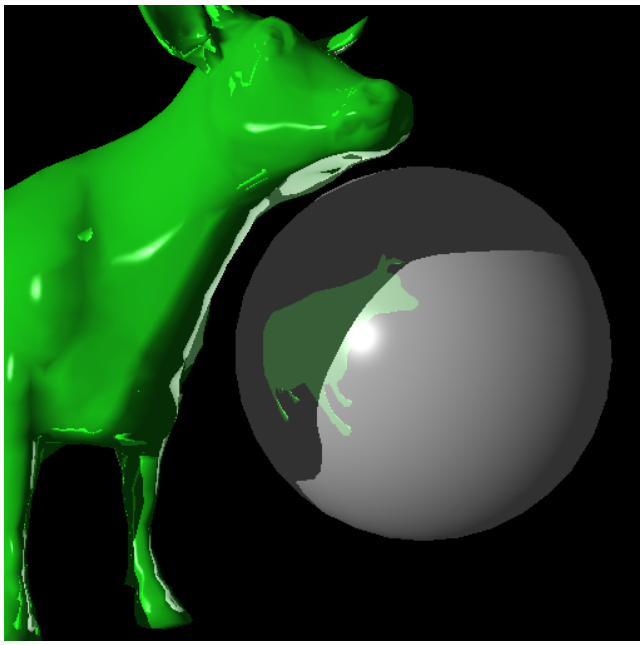

Abbildung 5.8.: Detail der Szene mit LA-CPU

OpenMP ist in beiden Fällen aktiviert. Version 3 wird, abweichend von den anderen drei, mit Intels ICC 11 und dessen spezifischen Compileroptimierungen erstellt. Zusätzlich wird der CPU-Fallback des APP SDK in Version 4 genutzt, um die OpenCL-Kernels auf dem Prozessor ausführen zu können.

Die Abbildungen 5.9, 5.10, 5.11 zeigen die drei verwendeten Testszenen.

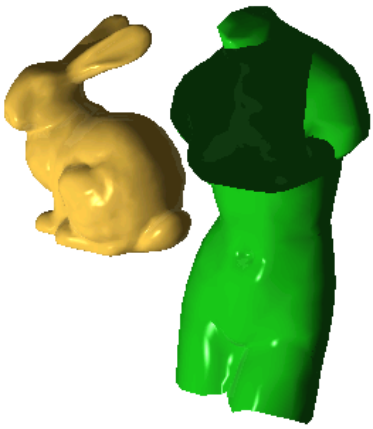

Abbildung 5.9.: *Venus* (GA)

Abbildung 5.10.: *Fishes* (LA)

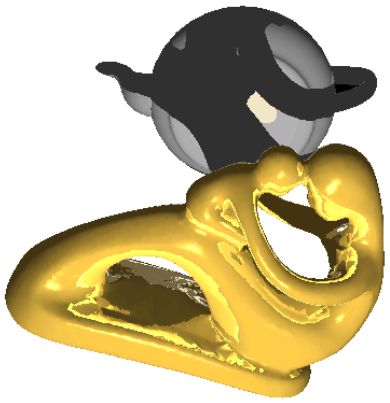

Abbildung 5.11.: *Fertility* (LA)

Die Dreiecksanzahl der drei Szenen beträgt 7200, 20000 und 55000. Alle Messungen werden fünfmal wiederholt und anschließend ein Mittelwert gebildet, der auf zwei Nachkommastellen gerundet wird. Auch hier werden die ersten zwei Messwerte verworfen, um Fehlmessungen durch den Taktwechsel der CPU auszuschließen. Tabelle 5.1 zeigt die Ergebnisse für die eingesetzten Prozessoren und die erste Version ohne Optimierungen.

	Venus	Fishes	Fertility
Dreiecke	7200	20000	55000
GA (Q6600)	67,83	130,61	526,22
LA (Q6600)	40,06	70,76	339,38
GA / LA (Q6600)	69,32 %	84,58 %	55,05 %
GA (X2 250)	55,76	129,63	587,88
LA (X2 250)	37,19	89,01	454,35
GA / LA (X2 250)	49,93 %	45,6 %	29,38 %
GA (Core2)	115,90	232,59	1139,25
LA (Core2)	62,49	130,10	675,71
GA / LA (Core2)	85,47 %	78,78 %	68,60 %

Tabelle 5.1.: Renderingergebnisse auf den drei eingesetzten Prozessoren mit deaktivierten Compilieroptimierungen. Zeitangaben in Sekunden.

Im Schnitt ist die GA hier um den Faktor 1,6 langsamer als die LA. Die Mehrzahl an Berechnungen der GA schlägt sich praktisch 1:1 in der Renderingzeit nieder. Der Vierkerner kann den AMD X2 erst beim großen Fertility-Modell überholen, ansonsten sind die beiden trotz der Nut-

zung von OpenMP nahezu gleich schnell. Der alte Core2Duo arbeitet nur halb so schnell wie die beiden anderen und der Rückstand der GA ist auf ihm noch etwas mehr ausgeprägt.

In Tabelle 5.2 sind die erzielten Renderingzeiten mit der zweiten Version zu finden, bei der die Compileroptimierungen dazugeschaltet werden.

	Venus	Fishes	Fertility
Dreiecke	7200	20000	55000
GA (Q6600)	27,02	52,36	211,35
LA (Q6600)	20,42	33,20	163.36
GA / LA (Q6600)	32,32 %	57,71 %	29,38 %
GA (X2 250)	17,61	41,68	198,08
LA (X2 250)	13,43	31,53	172,49
GA / LA (X2 250)	31,12 %	32,19 %	14,8 %
GA (Core2)	45,69	93,48	454,39
LA (Core2)	31,64	62,56	349,68
GA / LA (Core2)	44,4 %	49,42 %	29,94 % %

Tabelle 5.2.: Renderingergebnisse auf den drei eingesetzten Prozessoren mit aktivierten Compilieroptimierungen. Zeitangaben in Sekunden.

Mit den Optimierungen rücken die beiden Raytracer näher zusammen und GA ist im Schnitt nur noch um den Faktor 1,3 langsamer. Der AMD X2 kann sich etwas vom Q6600 absetzen und der Rückstand der GA ist auf ihm am kleinsten. Mittels der einfachen OpenMP-Einbindung, die über die geschachtelte for-Schleife parallelisiert, welche die Pixel des Ergebnisbildes hochzählt, werden die vier Kerne des Q6600 nicht voll ausgenutzt. Eine Betrachtung des Task-Managers ergibt, dass zwar alle Kerne arbeiten, aber die Auslastung nicht bei 100 % liegt. Praktisch das gleiche Bild ergibt sich bei Version 3 deren Ergebnisse auf TS1 von Tabelle 5.3 gezeigt werden.

	Venus	Fishes	Fertility
Dreiecke	7200	20000	55000
GA (Q6600)	19,76	42,81	186,66
LA (Q6600)	14,17	29,79	153,71
GA / LA (Q6600)	39,44 %	43,71 %	21,44 %

Tabelle 5.3.: Renderingergebnisse auf TS1 mit aktivierten Compilieroptimierungen des ICC. Zeitangaben in Sekunden.

Dass der Intel-Prozessor dem AMD dennoch überlegen sein kann zeigen die Ergebnisse des OpenCL-Fallbacks auf die CPU aus Tabelle 5.4.

Hier kann der Vierkerner den Zweikerner überholen und die Rechenzeiten sind generell um ein Vielfaches geringer als mit der OpenMP-C++-Implementierung. Für GA beträgt der Beschleunigungsfaktor 5. OpenCL gelingt das Ausnutzen der vier Kerne besser als OpenMP. Auch ist erkennbar, dass die beiden Algebren in dieser Laufzeitumgebung fast gleich schnell arbeiten, was anzeigt, dass die Parallelität der GA umgesetzt werden kann.

	Venus	Fishes	Fertility
Dreiecke	7200	20000	55000
GA (Q6600)	6,21	10,01	39,62
LA (Q6600)	6,19	9,87	38,81
GA / LA (Q6600)	0,32 %	1,42 %	2,09 %
GA (X2 250)	6,76	15,62	60,29
LA (X2 250)	6,65	15,07	59,05
GA / LA (X2 250)	1,65 %	3,65 %	2,10 %

Tabelle 5.4.: Renderingergebnisse auf dem Intel Q6600 und dem AMD Athlon X2 250 mittels des OpenCL Fallbacks auf CPU. Zeitangaben in Sekunden.

5.5.2 Auswertung der GPU Performance

Die Leistung auf der GPU soll detailliert betrachtet werden. Dank AMDs Profiler lässt sich sehr gut verfolgen, wie genau die Algorithmen auf die Hardware umgesetzt werden. Von dieser Möglichkeit soll bei den Leistungsmessungen Gebrauch gemacht werden.

Detaillierte Untersuchung einer Testszene

Um sich ein Bild des Ressourcenverbrauchs und der Stärken bzw. Schwächen der Algorithmen machen zu können, soll die Testszene aus Abbildung 5.12 in diesem Abschnitt mittels des AMD Profilers untersucht und die erzielten Ergebnisse bewertet werden. Die Szene enthält sowohl Reflexionen als auch Schattenwürfe und ist somit für eine repräsentative Untersuchung gut geeignet. Tabelle 5.5 gibt die wichtigsten Ausgaben des Profilers wider.

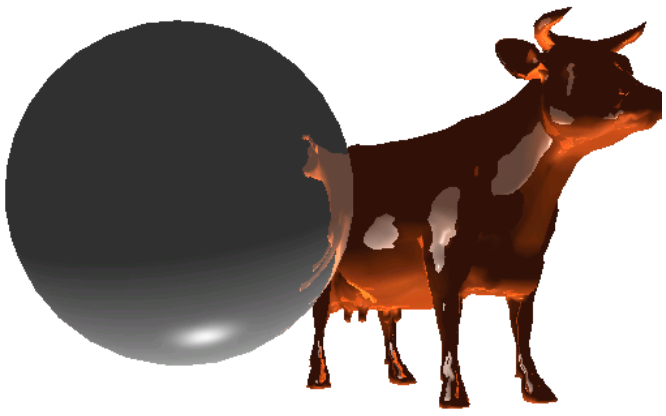

Abbildung 5.12.: Testszene *CowSphere*, gerendert vom LA-GPU-Raytracer

Die letzte Zeile gibt den Unterschied zwischen GA und LA für den entsprechenden Wert an. Genauere Informationen zu den einzelnen Profilerwerten können der AMD Dokumentation [Adv11, S. 61ff] entnommen werden.

	Time	GPRs	ScratchRegs	FCStacks	Wavefronts	ALUInsts
GA	160,86	25	0	5	4096	46015,71
LA	173,38	17	0	6	4096	44976,79
Abweich.	-7,22%	47,06%	0%	-16,67%	0%	2,31%

	FetchInsts	ALUBusy	ALUFetchRatio	ALUPacking	FetchSize	FetchUnitBusy
GA	5536,53	25,72	8,31	58,66	598271,31	9,60
LA	5477,14	23,40	8,21	44,31	589104,13	8,84
Abweich.	1,08%	9,91%	1,22%	32,33%	1,56%	8,60%

Tabelle 5.5.: Vergleich der Leistung der beiden Algebren auf der HD6970 in der Testszene CowSphere. Zeitangabe in Millisekunden.

- **Time**: Die reine Laufzeit des Kernels ohne Initialisierungs- und Kommunikationszeiten. Der GA-Kernel arbeitet in dieser Szene rund 7% schneller und kann somit im Vergleich zu den Kugel-Raytracern die LA-Variante überholen. Zwar ist der Unterschied nicht wirklich groß, aber dennoch stets reproduzierbar.
- **GPRs**: Diese Spalte gibt an wie viele der General Purpose Register durch den Kernel benutzt werden. Je höher der Ressourcenverbrauch, desto geringer ist im Normalfall die Anzahl der gleichzeitig aktiven Wavefronts, was sich negativ auf die Laufzeit auswirkt. Durch die Mehrzahl an Berechnungen und Zwischenwerten verbraucht die GA-Variante mehr Register, weshalb sich die Zahl der gleichzeitig aktiven Wavefronts zwischen beiden Kerneln unterscheidet.
- **Wavefronts**: Beide Kernel erzeugen natürlich die gleiche Anzahl an Wavefronts, was die angegebene Spalte verdeutlicht. Anhand einer Tabelle aus [Adv11, 4-47 / 4-48] lässt sich die Anzahl der gleichzeitig aktiven Wavefronts bestimmen. Für GA sind dies neun, für LA 14.
- **ScratchRegs**: Scratch-Register werden von einem Kernel benötigt, wenn sein Bedarf so hoch ist, dass die vorhandenen Register nicht ausreichen. Es wird dann in den VRAM ausgelagert, was die Geschwindigkeit des Algorithmus deutlich bremst, so dass es das Ziel sein muss diesen Wert auf 0 zu halten. Dies gelingt bei beiden Varianten. Abhilfe beim Auftreten von Scratch-Registern schafft am einfachsten eine Reduzierung der WorkGroup-Größe, was aber ebenfalls nachteilig für die Geschwindigkeit ist. Eine weitere Alternative ist es dem Compiler über Schlüsselwörter mitzuteilen, dass pro SIMD-Einheit weniger als vier aktive Wavefronts verwendet werden. Damit haben die verbleibenden mehr Register zur Verfügung und die Grenze, ab der Scratch-Register verwendet werden müssen, wird verschoben.
- **ALUInsts**: Gibt die durchschnittliche Anzahl an ALU-Instruktionen an, die pro Work-Item ausgeführt werden. Diese ist bei GA nur um 2% höher als bei Linearer Algebra, obwohl sie mehr Operationen benötigt. Die beiden nachfolgenden Werte können dieses Ergebnis verständlich machen.
- **ALUBusy**: Gibt an zu wieviel Prozent der Kernellaufzeit die ALUs aktiv waren. Der niedrige Wert, den beide Raytracer hier erreichen zeigt, dass die ALUs nur zu einem Viertel

der Laufzeit wirklich rechnen. In der restlichen Zeit muss auf Speicherzugriffe oder Verzweigungen im Code gewartet werden. Dennoch erreicht die GA einen fast 10% höheren Anteil.

- **ALUPacking**: Gibt die Rate an, mit der es gelingt, alle fünf (HD5770) bzw. vier (HD6970) Streamprocessors der Shadercores auszulasten. Je höher der Wert, desto besser kann die Parallelität von Operationen ausgenutzt werden. Hier zeigt sich, dass die im Theorieteil dargelegte Parallelisierbarkeit der GA-Berechnungen von AMDs Compiler gut umgesetzt werden kann und der erreichte Wert ein Drittel höher liegt als bei der LA Konkurrenz. Da die ALUs durch den Kernel sowohl zu einem höheren Anteil der Kernellaufzeit, als auch mit höherer Auslastung benutzt werden, ist das GA-Programm in der Lage trotz der höheren Operationenzahl schneller zu arbeiten als sein Gegenstück.
- Die **Fetch-Werte** geben über die Anzahl und Größe der Datentransfers zum VRAM der Grafikkarte Auskunft. Die beiden Algebren verhalten sich hier sehr ähnlich, wobei die GA generell dazu neigt etwas intensiver mit dem Speicher zu kommunizieren.

Als Zwischenfazit lässt sich also festhalten, dass der GA-Kernel trotz höherem Rechenaufwand auf Grund von besserer Ressourcenausnutzung schneller arbeiten kann.

Dieselbe Szene kann auch auf der HD5770 mittels des Profilers detailliert untersucht werden. In Tabelle 5.6 sind die Profilerausgaben dazu präsentiert.

	Time	GPRs	ScratchRegs	FCStacks	Wavefronts	ALUInsts
GA	294,61	22	0	5	4096	43380,89
LA	320,55	20	0	6	4096	44870,50
Abweich.	-8,09%	10,00%	0%	-16,67%	0%	-3,32%

	FetchInsts	ALUBusy	ALUFetchRatio	ALUPacking	FetchSize	FetchUnitBusy
GA	5536,53	30,39	7,84	49,75	598715,88	21,50
LA	5477,14	28,47	8,19	34,40	589521,50	19,25
Abweich.	1,08%	6.73%	-4,27%	48,95%	1,56%	11,69%

Tabelle 5.6.: Vergleich der Leistung der beiden Algebren auf der HD5770 in der Testszene CowSphere.

Das Bild ist dem auf der HD6970 ähnlich, besitzt allerdings einige Besonderheiten. Der Geschwindigkeitsunterschied bleibt relativ gleich, wobei GA auf der HD5770 im Schnitt bei dieser Szene 8% statt 7% schneller arbeitet. Die Zeiten im Vergleich zur HD6970 liegen im Erwartungsfenster. Da die HD6970 über etwas weniger als doppelt so viele Streamprocessors verfügt und mit dem gleichen Takt arbeitet, kann man annehmen, dass ein Kernel dementsprechend auch in der Hälfte der Zeit ausgeführt werden kann. Mit einem Faktor von $1,83$ wird diese Erwartung erfüllt.

Auffällig ist, dass die GA hier weniger, die LA dagegen mehr GPRs belegt und sich beide Algebren nur noch um zwei Register unterscheiden. Dadurch verändert sich die Anzahl der gleichzeitig aktiven Wavefronts. Bei der GA sind es elf, bei der LA zwölf. Die ALUBusy-Werte fallen beide jeweils etwas höher aus und die beiden Algebren rücken näher zusammen. Ein deutlicher

Unterschied ergibt sich allerdings bei der Packungsdichte der ALUs. Die Werte beider Algebren verringern sich im Vergleich zur neueren Grafikkartenarchitektur, doch derjenige des LA Kernels bricht viel deutlicher ein. Die 5D-Architektur der Juniper-Shadercores kann für LA vom Compiler nicht genutzt werden und im Schnitt sind pro Taktschritt, in dem der Core rechnet, über drei der Streamprocessors inaktiv, bzw. im Mittel sind gerade einmal zwei der fünf Streamprocessors am Arbeiten. Die generell etwas höhere Laufzeit der ALUs kann dieses Problem zwar weitestgehend kompensieren, aber dennoch führt dieser Sachverhalt zu dem weiteren Prozentpunkt, den GA auf die LA gutmachen kann. In Bezug auf Kommunikation mit dem VRAM verhalten sich beide Kernel erneut nahezu gleich.

Bei der Leistungsmessung tritt noch ein weiterer Punkt auf, der angesprochen werden soll, und zwar das Verhalten der beiden Kernel mit deaktivierten Schattenfühlern. Bei derartigen Testszenen verdoppelt sich der Vorsprung der GA nahezu. Tabelle 5.7 zeigt die dazu wichtigsten Daten.

	Time	GPRs	ALUInsts	ALUBusy	ALUPacking
GA	103,36	24	35414,61	33,41	58,12
LA	122,57	18	36019,69	27,34	44,33
Abweich.	-15,67%	33,33%	1,68%	22,20%	31,11%

Tabelle 5.7.: Vergleich der Leistung der beiden Algebren auf der HD6970 in der Testszene *CowSphere* mit deaktivieren Schattenfühlern.

Es ist sofort erkennbar, dass der Geschwindigkeitsvorteil von der längeren ALU-Laufzeit und höheren ALU-Auslastung kommt. In diesem Szenario kann der Compiler die Parallelität der GA am Besten umsetzen. Durch das Ausschalten der Schattenfühler sinkt die Anzahl an Verzweigungen im Kernel-Code, was dessen Ausführung beschleunigt. Dennoch liegt auch hier die Auslastung noch sehr niedrig.

Optimierungen des GA-Kernels mit Hilfe des Profilers

Mittels des Profilers ist es möglich einige Stellen im OpenCL-Code zu finden, bei denen eine kleine Veränderung bereits messbare Unterschiede in der Laufzeit des Kernels verursacht. Ein Beispiel hierfür ist die Normalisierung der interpolierten Normale für einen Schnittpunkt. Wann diese Normalisierung zwischen der Schnittberechnung und dem Shading stattfindet, ist für den Algorithmus unerheblich. Durch Variation der Position im Code und anschließenden Messungen stellt sich heraus, dass der Kernel am schnellsten ausgeführt wird, wenn die Normierung direkt vor der Bestimmung des diffusen Beleuchtungsanteils erfolgt. Im Vergleich zur ursprünglichen Position direkt nach der Schnittpunktbestimmung, resultiert daraus ein Geschwindigkeitsvorteil von durchschnittlich 3%, obwohl sich die Anzahl der Berechnungen durch die Umstellung des Codes nicht verändert. Interessant ist außerdem, dass sich beim LA-Kernel bei einer analogen Verschiebung keine messbaren Performanceunterschiede ergeben.

Auf Grund der Tatsache, dass Verzweigungen im Code für die Grafikkartenarchitektur schwierig zu handhaben sind, kann mit den Early Exits im Code der Schnitttests experimentiert und diese an verschiedene Stellen des Codes geschoben werden. Durch Ausprobieren kann herausgefunden werden, dass die höchste Geschwindigkeit erreicht wird, wenn die Prüfung mittels

der *DualRays* (vgl. Anhang A) zwischen die Berechnung der Dreiecksebene und die eigentliche Schnittpunktberechnung geschoben wird. Obwohl die Anzahl der notwendigen Berechnungen pro Pixel hierdurch erhöht wird, verringert sich die Rechenzeit des Kernels um durchschnittlich 4%. Ein weiterer Fall für den das gleiche zutrifft sind die Schattenfühler. Hier arbeitet der Kernel schneller, wenn er für die Schattenfühler den Schnittpunkt mit dem getroffenen Objekt berechnet, obwohl dieser für den Algorithmus nicht benötigt wird.

Ein dritter Punkt sind die verschiedenen Wege der Reflexionsberechnungen. Die 5D-Reflexionen sind langsamer als die 3D-Varianten. Bemerkenswert ist dagegen, dass die 3D-GA-Reflexionen trotz ihrer Mehrzahl an Multiplikationen zu einem schnelleren Kernel führen als die 3D-LA-Methode. Außerdem kann noch mit der Entfernungsberechnung experimentiert werden. Die wenigsten Operationen benötigt der euklidische Abstand, während in der GA auch die Möglichkeit des IPs zweier Punkte gegeben ist. Für dieses sind die Koordinaten e_0 und e_∞ nötig, so dass e_∞ aus e_1, e_2 und e_3 berechnet werden muss. Dennoch zeigen die Messungen, dass die Realisierung über IP fast 2% schneller ist als Euklid. Die Variante e_∞ immer mitzuführen ist dagegen um 2% langsamer als der Weg über Euklid.

Alle hier genannten Wege zur Erhöhung der Kernel-Laufzeitgeschwindigkeit basieren auf dem *Trial and Error*-Prinzip. Es scheint nicht vorhersagbar zu sein, wie die Kernel-Performance durch gewisse Änderungen beeinflusst wird. Daher lässt sich dieses Vorgehen auch nicht automatisieren. Jede Änderung bewirkt für sich genommen zwar nur minimale Erhöhungen der Geschwindigkeit; in der Summe wird aber der zweistellige Prozentbereich erreicht.

Außerdem existieren zwei weitere Kernel-Varianten des GA-Raytracers, die mittels Profiler untersucht werden können und interessante Ergebnisse liefert:

- *Datentyp-optimierte Fassung*: Bei diesem Kernel wird darauf geachtet möglichst wenig Speicher für Daten zu belegen. In der Standardversion werden die Daten für einen Punkt und seine Normale in einem OpenCL float8-Array abgelegt. Das bedeutet, dass nur sechs der acht Feldeinträge belegt sind, während die zwei restlichen auf Null gesetzt werden. Bei der Datentyp-optimierten Fassung werden Punkt und Normale auf eine float4- und eine float2-Variable aufgeteilt, so dass alle Einträge belegt sind. Dies wird für alle Variablen im Kernel entsprechend angepasst. Die Messung mit dem Profiler ergibt, dass dieser Kernel im Vergleich zur Standardvariante 20-30 % langsamer arbeitet. Dies liegt vor allem an seinem deutlich höheren Registerverbrauch, der bis zu 70 % höher liegen kann. Dadurch sinkt die Anzahl der gleichzeitig aktiven Wavefronts auf 5-6 im Vergleich zu 9-10. Eine Umsetzung dieser Kernel-Variante für LA wird daher nicht vorgenommen. Somit ist es für den Raytracer besser weniger Variablen mit größeren Feldern anzulegen, auch wenn diese nicht voll belegt werden können.
- *Speicher-optimierte Fassung*: Hierbei handelt es sich um einen Kernel, der auf die gleiche Datenstruktur setzt, wie die C++-Raytracer. Jedes Dreieck speichert seine Eckpunkte explizit. Eine Betrachtung mit dem Profiler ergibt, dass die zu übertragende Datenmenge deutlich größer ist als bei der Standardversion. Der Buffer für die Dreiecksdaten weist eine sechsfach höhere Größe auf. Die reine Laufzeit dieser Version ist dagegen in großen Szenen um ein Drittel niedriger. Für eine Szene mit 20000 Dreiecken benötigt die Standardvariante 1100ms, während die Speicher-optimierte Fassung das Bild in 700ms rendern kann. In der Profilerausgabe ist ersichtlich, wie dieser Unterschied zu Stande kommt. Die Menge der Speicherzugriffe ist deutlich geringer, was die Werte in Tabelle 5.8 veranschaulichen.

	FetchInstructions	FetchSize	CacheHit	FechUnitBusy
GA (Std)	86745,07	1114535,13	59,40	22,52
GA (RAM)	12428,87	574738,38	89,19	5,79
Abweich.	+597,93 %	+93,92%	-33,40%	+ 288,95 %

Tabelle 5.8.: Vergleich der Standardversion des GA-Kernels (oben) gegen die Speicher-optimierte Fassung (unten) in einer Testszene mit 20000 Dreiecken.

Der LA-Kernel mit dieser Speichervariante verhält sich genauso. Der prozentuale Unterschied zwischen GA und LA bleibt konstant, während die beiden Speicher-optimierten Kernel jeweils schneller sind als die Standardversionen. Dafür steigt wie angesprochen die Übertragungszeit vor der eigentlichen Ausführung.

Weitere Ergebnisse

Eine Szene, bei der sich der eindeutige Einfluss von Schattierungen auf die GA bemerkbar macht, wird in den Abbildungen 5.13 und 5.14 gezeigt.

Abbildung 5.13.: Testszene *3 Bunnys and Elephant* gerendert vom GA-GPU-Raytracer, frontal beleuchtet

Abbildung 5.14.: Testszene *3 Bunnys and Elephant* gerendert vom GA-GPU-Raytracer, von unten beleuchtet

Die Szene und Kamerapositionierung bleibt in beiden Fällen unverändert. Sie besteht aus fast 16150 Dreiecken, wobei der Hase mit 4968 das deutlich detailliertere Objekt ist. Das einzige was geändert wird ist die Lage der Lichtquelle. In Abb. 5.13 wird von vorne beleuchtet und die Lichtquelle fällt mit dem Augpunkt zusammen. Da alle Objekte die gleiche Distanz zum Lichtursprung aufweisen, kommt es nicht zum Schattenwurf. In Abb. 5.14 wird das Licht unterhalb der Szene positioniert, weshalb der untere Hase und der Elephant einen Schatten auf die Objekte werfen, die über ihnen liegen. Diese Umpositionierung der Lichtquelle wirkt sich, wie in Ta-

belle 5.9 ersichtlich wird, sehr deutlich auf den Geschwindigkeitsunterschied zwischen beiden Raytracern aus.

	Time	ALUBusy	ALUPacking
GA	533,04	20,28	59,09
LA	597,10	17,25	44,43
Abweich.	-12,02%	17,57%	33,00%
	Time	ALUBusy	ALUPacking
GA	657,55	17,35	59,30
LA	694,55	16,31	44,44
Abweich.	-5,33%	6,38%	33,44%

Tabelle 5.9.: Ergebnisse der Szene *Three Bunnies and Cow*. Die obere Hälfte gibt dabei die Ergebnisse bei Frontalbeleuchtung, die untere diejenigen bei Beleuchtung von unten wieder. Zeitangaben in Millisekunden.

So bewirkt die Beleuchtungsänderung, dass der GA-Kernel nur noch 5% schneller ausgeführt wird, als die LA Variante, während der Vorsprung im anderen Fall sogar über 12% beträgt. Die Erklärung hierfür liefert erneut ein Blick auf die Ausgaben des AMD Profilers. Die Laufzeit der ALUs ist im ersten Fall für GA um fast 20% höher, im zweiten nur noch etwas über 6%.

Eine letzte Szene, die für Raytracing auf der CPU betrachtet wird, und bei der beide Kernels nahezu die gleiche Leistung aufweisen, ist in Abbildung 5.15 zu sehen.

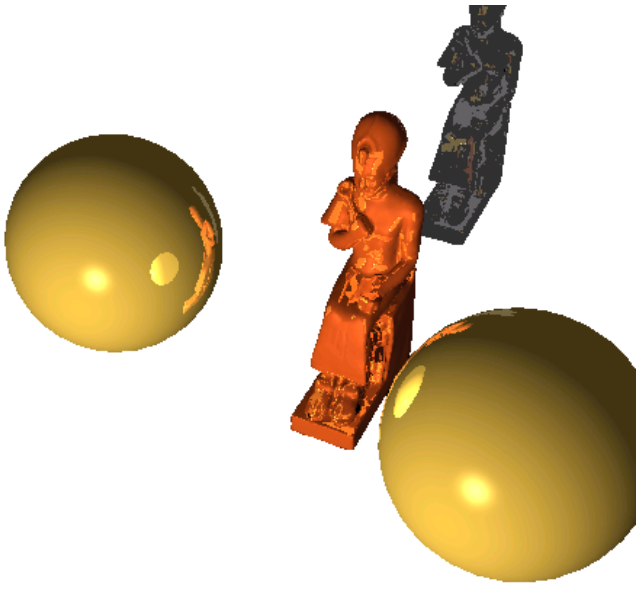

Abbildung 5.15.: Szene *Sonnenpharao* gerendert mit dem GA-GPU-Raytracer

Durch die mittige Lage des vorderen Ramses-Modells, zwischen den beiden Kugeln, werden viele Reflexionen verursacht und der hintere Pharao liegt vollständig im Schatten, weshalb viele Schattenfühler einen positiven Schnitt finden. Das komplette Bild besteht aus 42000 Dreiecken. Mit 1871,64 ms Berechnungszeit ist GA nur 1,37% schneller als LA, welche zum Rendern des Bildes 1897,66 ms benötigt.

	Raptor 200 000	Raptor 500 000
GA	2,292373 sec	3.437648 sec
LA	2,515765 sec	3.653528 sec
Abweich.	-8,8 %	-5,9 %

Tabelle 5.10.: Ergebnisse des *Raptor*-Modells mit Splitfactor 50000 auf HD6970.

Zum Schluss dieses Abschnittes, soll noch die Raycasting-Leistung der beiden Kernel miteinander verglichen werden. Hierzu dient zum einen das Modell *Raptor* [1], welches im Original aus 2000000 Dreiecken besteht. Es wird mittels MeshLab [2] auf eine Version mit 200000 und eine Version mit 500000 Dreiecken heruntergerechnet und in das obj-Format umgewandelt. Abbildungen 5.16 und 5.17 zeigen den kleinen Dinosaurier in zwei verschiedenen Versionen.

Abbildung 5.16.: Modell *Raptor* aus 50k Dreiecken, in 0,31sec mit Splitfactor 500 gerendert von LA-GPU.

Abbildung 5.17.: Modell *Raptor* aus 1Mio Dreiecken, in 1,15sec mit Splitfactor 500 gerendert von LA-GPU.

Detailunterschiede sind vor allem an seinem Kopf und seiner Flanke zu erkennen. Abbildungen 5.18 und 5.19 zeigen einen vergrößerten Ausschnitt der Kopfpartie. Vor allem die schuppige Haut des Reptils ist bei der 1 Mio Dreiecke umfassenden Version deutlich besser ausgebildet. Diese Rauheit seiner Oberfläche wird mit abnehmender Dreieckszahl immer mehr geglättet.

In Tabelle 5.10 sind die Geschwindigkeitsunterschiede für Versionen mit 200000 und 500000 Dreiecken verdeutlicht.

Beide Male kann der GA-Kernel die schnellere Zeit erreichen, was erneut zeigt, dass die AMD Hardware auch bei diesem großen Modell besser genutzt werden kann, als von der LA. Der Vorsprung ist beim größeren Modell etwas geringer. Eine Analyse mittels Profiler ist bei diesem Modell nicht mehr möglich, da dies zu einem Timeout und einem Neustart des Grafikkartentreibers führt. Die vorangehenden Untersuchungen der Kernel-Algorithmen mit kleineren Meshes sind auf die großen Netze übertragbar.

[1] Es kann hier heruntergeladen werden: `http://shapes.aim-at-shape.net/view.php?id=178`
[2] `http://meshlab.sourceforge.net/`

Abbildung 5.18.: Kopf des *Raptors* bei 50k Dreiecken (LA)

Abbildung 5.19.: Kopf des *Raptors* bei 1 Mio. Dreiecken (GA)

	Splitfac 50000	Splitfac 5000	Splitfac 500
GA	3.437648 sec	2.592454 sec	0.557545 sec
LA	3.653528 sec	2.868918 sec	0.628686 sec
Abweich	-5,90 %	- 9,64 %	- 11,32 %

Tabelle 5.11.: Ergebnisse des *Raptor*-Modells mit Splitfactors 50000/5000/500 auf HD6970.

Ein weiterer Punkt, der Betrachtung verdient, ist die Auswirkung des *Splitfactors* auf die Renderinggeschwindigkeit. Hierzu wird die 500000 Dreiecke-Version des *Raptors* mit einem Unterteilungsfaktor von 500, 5000 und 50000 gebenchmarkt, was in Tabelle 5.11 zusammengefasst ist.

Je nach Faktor lässt sich die Berechnungszeit für eine Szene deutlich verbessern. Bei falscher Einstellung für diesen Faktor sinkt die Performance dagegen teils deutlich ab. Wie an den Ergebnissen in Tabelle 5.11 zu sehen ist, sind selbst für große Dreiecksnetze kleine Splitfactors nötig, um kurze Rechenzeiten zu erhalten. So steigt die Berechnungsgeschwindigkeit bei einer Verzehnfachung des Splitfactors von 500 auf 5000 um den Faktor 5, wohingegen sich eine weitere Verzehnfachung des Splitfactors auf 50000 lediglich in einer Verlangsamung um einen Faktor kleiner als $1,5$ niederschlägt. Dies lässt sich damit erklären, dass die Unterteilung der Dreiecksuntermengen lediglich in der Reihenfolge des Einlesens aus der Quelldatei erfolgt. In der Liste aufeinanderfolgende Dreiecke sind zwar oft Nachbarn im Netz, aber es ist meist nicht der Fall, dass sie auch einen möglichst kleinen Volumenanteil des Modells einnehmen. In einem denkbaren Extremfall liegen alle Dreiecke für ein Unternetz in einer langen Kette aneinander, die vom Beginn des Modells zu seinem Ende reicht. Ihre umschließende Hüllkugel ist somit mit derjenigen des vollständigen Objektes identisch und es werden viele unnötige Schnittprüfungen hervorgerufen. Generell lässt sich feststellen, dass mit steigender Splitfactor-Größe deutlich größere BSpheres generiert werden, so dass die Dreiecksschnittzahl im Endeffekt weniger deut-

lich, bis fast garnicht mehr reduziert wird. Im Worst-Case wird somit jedes Dreieck wie bei der Brute-Force-Implementierung pro Pixel einmal auf Schnitt überprüft und zusätzlich muss für jede der BSpheres einmal berechnet werden, ob sie vom aktuellen Sichtstrahl getroffen wird. Da bei den getesteten Dreiecksnetzen die Bedingung, dass untereinander stehende Dreiecke nebeneinander liegen, in vielen Fällen erfüllt ist, ergeben sich für kleine Untermengen von 100-500 Dreiecken stets kleine Hüllkugeln, durch welche der Renderingvorgang beschleunigt werden kann. Je nach Größe des Netzes ist die Vorbereitungszeit durch die CPU allerdings enorm. Um das 1 Mio. Dreiecke-Modell des Raptors mit Splitfactor 500 zu rendern, benötigen beide Algebren knapp über eine Sekunde. Vom Programmstart bis zum Aufruf des Kernels vergeht dagegen eine Viertelstunde. Eine Alternative zur aktuellen Vorgehensweise stellt die Zerlegung der Modelle in Untermengen von Dreiecken anhand der räumlichen Nachbarschaft von Dreiecken und nicht der Einlesereihenfolge aus der Quelldatei dar. Allerdings ist diese Berechnung für sich selbst genommen wiederum sehr aufwändig. Da die Qualität der Hüllkugeln bei kleinem Splitfactor in der vorliegenden Fassung bereits gut ist und somit die Berechnungszeit der Kernel an sich kaum gesteigert werden kann, wird die aktuell umgesetzte Lösung beibehalten.

Ein weiteres Modell mit dessen Hilfe die Raycasting-Geschwindigkeit gemessen wird, ist das *Kitten*[3]. Es ist aus 274196 Dreiecken aufgebaut und in den Abbildungen 5.20 und 5.21 unter Verwendung zweier verschiedener Materialeigenschaften zu sehen.

Abbildung 5.20.: Modell *Kitten* aus 274196 Dreiecken gerendert mit GA-GPU-Raycasting

Abbildung 5.21.: Modell *Kitten* aus 274196 Dreiecken gerendert mit LA-GPU-Raycasting

[3] Es kann unter `http://shapes.aim-at-shape.net/viewgroup.php?id=366` gefunden werden.

5.6 Vergleich mit dem Raytracer der Uni Amsterdam

Da Daniel Fontijne im Rahmen seiner Dissertation bereits diverse Raytracer in LA und GA entwickelte [Fon07], werden seine 3D-LA- und die 5D-GA-Version für die vorliegende Masterarbeit etwas genauer betrachtet, auch mit dem Ziel einen Vergleich zwischen der hier eingesetzten Gaalop-Optimierung und der Gaigen2-Bibliothek von Fontijne zu gewinnen, sofern dies möglich ist und sinnvoll erscheint. In Amsterdam wurden Raytracer basierend auf der Gaigen1- und Gaigen2-Bibliothek erstellt. Erstere sind deutlich langsamer als ihre Nachfolger[4] und werden daher im folgenden nicht berücksichtigt. Für Gaigen2 existieren zwei KGA-Raytracer. Die erste Version nennen die Autoren *nice* und beschreiben sie wie folgt:

> This is a variant of the regular conformal (5-D GA) version in which we used all the semantically nice representations, instead of focusing on efficiency ([Fon07, S. S. 196]).

Die zweite dagegen soll besonders leistungsfähig sein und setzt daher auf komplexere Konstrukte der KGA, weshalb ihre Funktionsweise schwerer zu verstehen ist. Da sie aber deutlich schneller als die *nice*-Variante ist, wird sie zum Vergleich herangezogen. Laut [Fon07, S. S. 196] ist diese Raytracer-Variante in KGA um den Faktor 1,26 langsamer als die Version in 3D Linearer Algebra. Außerdem ist es das Ziel ein durch Gaigen2 optimiertes Programm auf aktueller Hardware zu testen, da die von Fontijne eingesetzte Hardware[5] bereits 2007 nicht mehr den aktuellen Stand der Technik darstellte. Im Gegensatz zum vorliegenden Ansatz zielen die Raytracer aus Amsterdam nicht auf parallele Architekturen, sondern auf Single-Core-CPUs, so dass ein Vergleich nur dann versucht werden kann, wenn die im Rahmen dieser Arbeit entwickelten Raytracer ohne OpenMP- bzw. OpenCL-Unterstützung kompiliert und benutzt werden. Getestet wird auf System TS1.

Da die Gaigen-Raytracer als Datenstruktur einen BSP-Tree [Ebe01, S. 354-364] einsetzen, wird dieser für die Vergleichsmessungen deaktiviert, indem der Code dementsprechend verändert wird. Die Hüllkugeln bleiben aktiviert. Wird für ein Pixel die Hüllkugel getroffen, wird jedes Dreieck darin auf Schnitt geprüft. Analoge Änderungen werden in den eigenen Tracern vorgenommen. Das Hüllkugel-Erzeugungsverfahren wird auf dasjenige bei Fontijne zurückgesetzt, so dass für ein Dreiecksnetz in allen Varianten dieselbe BSphere vorhanden ist. Für die Messungen werden vier Objekte herangezogen: Die *Sphere* und der *Teapot*, welche bereits vorhanden sind, sowie die ins rtm-Format konvertierten Modelle *WFish* und *Cow*. Außerdem ist anzumerken, dass beim LA-Raytracer aus Amsterdam für die Schnittberechnung das im GA-Tracer benutzte Verfahren verwendet wird und nicht ein optimiertes wie dasjenige von Möller oder Badouel. Testweise wird daher der Schnittalgorithmus ersetzt. Für die erste Messung wird der Visual C++-Compiler eingesetzt und alle Optimierungen ausgeschaltet. Die Ergebnisse der Vergleichsmessungen zeigt Tabelle 5.12.

Bei den Programmen aus Amsterdam zeigt sich ein klarer Unterschied zwischen LA und GA, woraus sich erkennen lässt, dass auch Gaigen2 viel mehr Operationen erzeugt, als für LA anfallen. Dies entspricht den Erwartungen. Deutlich fällt auch der Unterschied zwischen dem Amsterda-

[4] [Fon07, S. 195] gibt an, dass Gaigen1 um den Faktor 4,5 langsamer arbeitet als die 3D-LA-Variante, die mit Gaigen2 optimiert ist.

[5] Notebook mit Pentium IV 3 GHz, 1024 MB RAM

	Cow	WFish	Teapot	Geosphere
Dreiecke	5804	10000	4096	720
GA (A)	331,88	140,39	290,08	18,64
LA (A)	209,46	87,93	183,64	11,91
LA (A mit M)	160,88	71,85	149,96	8,19
GA (selbst)	325,91	126,31	256,86	17,72
GA (A) / LA (A)	-36,89 (-51,52) %	-37,37 (-48,82) %	-36,69 (-48,32) %	-37,37 (-48,82) %
GA (A) / GA (s)	-1,80 %	-4,94 %	-11,45 %	-10,03 %

Tabelle 5.12.: Vergleich des Amsterdamer Raytracers mit den eigenen Implementierungen. Kompiliert mit dem Visual-C++-Compiler und ausgeschalteten SSE-Optimierungen. Die *Zeile GA (A) / LA (A)* zeigt den prozentualen Geschwindigkeitsunterschied zwischen dem Amsterdamer LA- und dem GA-Raytracer. In Klammern der Unterschied zur optimierten Amsterdamer LA-Version mit Möllers Algorithmus. Die letzte Zeile gibt den Unterschied zwischen der eigenen GA Implementierung und derjenigen von Fontijne an. Zeitangaben in Sekunden.

	Cow	WFish	Teapot	Geosphere
Dreiecke	5804	10000	4096	720
GA (A)	43,95 sec	19,18 sec	34,29 sec	2,18 sec
LA (A)	45,14 sec	21,54 sec	35,98 sec	2,29 sec
GA (selbst)	81,75 sec	34,22 sec	60,38 sec	3,75 sec
LA (selbst)	74,48 sec	35,33 sec	58,01 sec	3,59 sec
Abw. GA (A) / LA (A)	+2,71 %	+12,30 %	+4,93 %	+5,05 %
Abw. GA (A) / GA (selbst)	+86,01 %	+78,42 %	+76,09 %	+72,02 %

Tabelle 5.13.: Vergleich des Amsterdamer Raytracers mit den eigenen Implementierungen. Kompiliert mit Intels Parallel Composer und eingeschalteten SSE-Optimierungen

mer Verfahren und Möllers Algorithmus aus, da letzterer im Schnitt rund 20% schneller arbeitet als die ursprüngliche Version. Ein Vergleich zwischen Möller und Gaigen2 erscheint fairer, wenn man die neuen GA-Algorithmen mit den etablierten Verfahren vergleichen will.
Für eine zweite Messreihe wird der Intel Parallel Composer eingesetzt und das *Streaming SIMD Extensions 2* Instruction Set genutzt. Ebenso werden Intelprozessor-spezifische SSE Optimierungen eingeschaltet. Die erreichten Zeiten sind in Tabelle 5.13 veranschaulicht.

Es ist auffällig, dass Gaigen2 vom Einsatz von optimiertem SSE deutlich profitieren kann. Die über 35% Geschwindigkeitsvorteil der LA werden vollständig aufgeholt und beide Raytracer arbeiten praktisch gleich schnell. Gaigen2 zieht deutlich größere Vorteile aus der Autovektorisierung als es LA kann. Die eigenen Raytracer erreichen durch SSE zwar ebenfalls schnellere Ausführungszeiten, arbeiten aber in dieser Konfiguration im Vergleich langsamer als die Amsterdamer Versionen. Dafür ist es bei diesen nicht möglich eine einfache Optimierung durch OpenMP zu erreichen, da bei dessen Einsatz keine korrekten Bilder mehr erzeugt werden. Es scheinen durch die Optimierungen in der Gaigen2-Library Datenabhängigkeiten in den Berechnungen der einzelnen Pixel zu entstehen, so dass diese einfache Parallelisierung nicht mehr funktioniert. Die eigenen Versionen mit OpenMP und ohne SSE brauchen zum Rendern der Testszenen in etwas die gleiche Zeit, wie die A-Tracer ohne OpenMP und mit SSE.

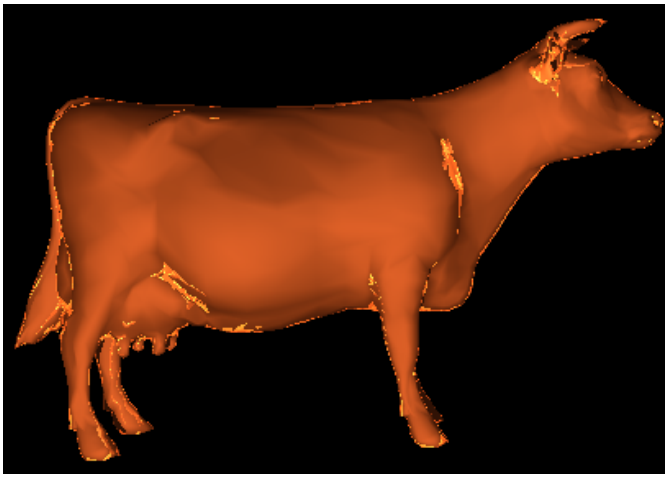

Abbildung 5.22.: *Cow* gerendert mit dem c3ga-Raytracer aus Amsterdam

Beim Test der Bildqualität lässt sich allerdings feststellen, dass bei der GA-Variante Artefakte auftreten, die an den Umrandungen der Objekte sichtbar werden. Abbildung 5.22 zeigt das Auftreten dieser Bildfehler. Der Bildhintergrund wird schwarz belassen, da die leuchtenden Pixel auf dem Rand der *Cow* so deutlicher werden. Auch bei *Teapot*, *WFish* und der *Geosphere* treten solche Pixel auf. In Bezug auf die Bildqualität sind die eigenen Implementierungen überlegen.

Gleiches gilt für die Verständlichkeit des Codes. Durch die vorhandenen Optimierungen und benutzten Konstrukte im A-Tracer leidet die Übersicht deutlich. Um den Verlauf der GA-Berechnungen nachvollziehen zu können, muss eine deutlich höhere Kenntnis der Algebra (und auch von C++) vorausgesetzt werden, als bei den eigenen Implementierungen. Die Intuitivität als Vorteil der GA ist im vorliegenden Fall nicht mehr vorhanden.

5.7 Fazit zum Raytracen von Dreiecken

Das Bild, das sich beim Kugel-Raytracen zeigt wiederholt sich. Ohne parallele Ausführung sind die Programme in Geometrischer Algebra langsamer als etablierte Verfahren. Bei Verwendung von OpenCL dagegen kann GA zur LA aufschließen oder sie sogar überholen. Mit Hilfe von AMDs Profiler lassen sich außerdem weitere Optimierungen finden, die im Voraus nicht immer voraussehbar sind. Die Leistungsschwankungen, die durch verschiedene Treiberversionen verursacht werden, fallen bei den Dreiecken stets deutlicher aus, als bei den Kugeln.

6 Raytracen auf FPGAs

Als weiterer Test, inwiefern sich die Parallelität der GA auf verschiedenen Architekturen umsetzen lässt, werden Implementierungen in Verilog für die FPGAs Virtex2p, Spartan3E und Virtex5 angefertigt. Hierbei wird ein Raycaster umgesetzt, der auf verschiedenen Plattformen realisiert, simuliert und getestet wird. Allerdings wird nicht der vollständige Funktionsumfang eines Raycasters auf dem FPGA platziert. Der FPGA ist nur dafür verantwortlich pro Pixel den nächsten Schnittpunkt und die dazugehörige Normale zu finden. Das Shading wird nicht implementiert, da es sich zwischen GA und LA nicht unterscheidet und somit keinen Nutzen aus GA ziehen kann. Außerdem wird somit vermieden die Funktionalität für Quadratwurzel und Exponentialfunktion auf den FPGA zu bringen, wodurch viele Ressourcen gespart werden können. Ebenfalls nicht enthalten ist das Konzept der Hüllkugeln, da hierbei schon im Voraus absehbar ist, dass der Platz auf dem verwendeten Virtex fehlt. Die Kameraposition ist fest bei $\begin{pmatrix} 0 & 0 & -4 \end{pmatrix}^T$. Die Position der Lichtquelle, sowie das Material des Objektes werden nachträglich im Software-Renderer hinzugefügt. Trotz dieser Ressourceneinsparungen ist der Raycaster zu groß für den Virtex2p und wird daher am Ende dieses Kapitels auf einem Virtex5 umgesetzt.

Zum Entwickeln und Synthetisieren wird das Xilinx ISE in den Versionen 11.2 (Spartan3E und Virtex5) bzw. 10.1 (Virtex2p) verwendet. Beim Simulieren kommen VCS und ISIM zum Einsatz.

6.1 Datentypen und Rechenungenauigkeiten

Als Datentyp für die Speicherung der Dreiecksdaten, Schnittpunkte und Zwischenwerte wird ein selbst umgesetztes Festkomma-Format genutzt, welches 32bit breit ist. Die sechs höchstwertigen Bits stellen dabei die Zahl vor dem Komma und das Vorzeichen dar; die 26 verbleibenden Stellen dienen als Repräsentation für die Nachkommastellen. Bei Additionen und Subtraktionen wird der Festkommatyp wie eine normale Ganzzahl behandelt. Durch die sechs Vorkomma-Bits wird sichergestellt, dass kein Overflow auftreten kann, weil Beobachtungen an der C++-Version des Raytracers ergeben, dass bei normalisierten Dreiecksnetzen und den entscheidenden Berechnungen nur Zahlen im Intervall $(-2, 2)$ auftreten. Bei Multiplikationen werden die Festkommawerte ebenfalls zunächst als 32bit Ganzzahlwerte aufgefasst und das Ergebnis in ein 64bit breites Register geschrieben. Dieses Zwischenergebnis wird anschließend um die Anzahl der Nachkommastellen nach rechts geshifted und in ein 32bit Ergebnisregister geschrieben. Bei der einzigen vorkommenden Division im Verlauf des Algorithmus muss lediglich ein Kehrwert gebildet werden. Dies geschieht durch Teilen der Konstante $32'h80000000$ durch den Divisor, von dem der Kehrwert benötigt wird. Auch bei dieser Division werden die Eingabedaten als Ganzzahlen angesehen. Das ganzzahlige Ergebnis wird in ein 32bit Register geschrieben und der Rest, der bei der Division entsteht, verworfen. Durch die Shifts entstehen Rundungsfehler, die sich im Verlauf der Berechnung summieren. Vergleiche mit dem C++-Programm mit *double*-Genauigkeit ergeben für die Schnittpunkte eine Abweichung zwischen $0,05\%$ und $0,20\%$. Die Zwischenwerte, die für die Schnittentscheidung selbst gebraucht werden, weichen weniger als $0,05\%$ vom *double*-Wert ab, so dass es beim reinen Raycasten nicht zu Artefakten kommt.

6.2 Struktur der Module

Dieser Abschnitt erläutert zwei verschiedene Designs des Raycasters. Dabei soll deutlich werden, welche Aufgaben für den Raycasting-Vorgang anfallen und in welcher Reihenfolge sie im jeweiligen Design ausgeführt werden. Wie genau die Berechnungen schließlich implementiert und in Hardware realisiert werden, ist Thema von Abschnitt 6.3.

6.2.1 Design der Version 1

Die ursprüngliche, in Verilog geschriebene Version des Raycasters basiert auf dem Flussdiagramm, das in Abbildung 6.1 gezeigt ist. An die berechnenden Prozesse sind die jeweiligen Operationen der GA geschrieben. Außerdem werden die Prozessbezeichnungen bei der Erläuterung der Module im übernächsten Absatz wieder aufgegriffen und im Detail beschrieben.

Bei diesem Entwurf wird vor allem ausgenutzt, dass der Sichtstrahl für jedes Dreieck pro Pixel gleich bleibt. Somit kann er beim ersten Dreieck parallel zum Auslesen der Dreiecksdaten aus dem Speicher berechnet werden und schließlich bis zum Ende dieses Pixelschleifendurchlaufs gehalten werden. Dies lässt sich an dem parallelen Block in der Mitte von Abb. 6.1 erkennen. Daher ist es nicht notwendig, dass die Strahlenerzeugung so schnell wie möglich abläuft, da ihre Berechnungszeit durch das Auslesen der Dreiecke verdeckt wird. Deshalb können an dieser Stelle Ressourcen eingespart werden. Besonders wichtig ist in diesem Modell die effektive Schnittpunktberechnung, da der weitere Verlauf so lange blockiert ist, bis das Ergebnis zur Verfügung steht. Dies ist in Abb. 6.1 an der langen sequentiellen Kette zwischen den Prozessen *C_DualRays* und *Save IP* zu erkennen, denn dort können keine Prozesse parallel ausgeführt werden.

Das erste Modul dieses Designs ist das Ray-Generator-Modul *RayGen*, das in Abb. 6.2 gezeigt ist. Als Eingang erwartet es den X- und Y-Wert des aktuellen Pixels, sowie die Kameraparameter inklusive der zu rendernden Auflösung. Auf eine positive Flanke von *START_RAYGEN* hin beginnt es die Berechnung des entsprechenden Sichtstrahls. Liegen die Ergebnisse vor, wird dies durch ein Setzen von *RAY_READY* auf 1 signalisiert. Die Ergebnisse werden so lange an den Ausgängen gehalten, bis *START_RAYGEN* zurück genommen wird. Daraufhin folgt ein Setzen der Datenausgänge und von *RAY_READY* auf 0.

Die berechnenden Prozesse, die mit diesem Modul assoziiert sind, sind:

- *C_Direction*: Es wird die Richtung des Sichtstrahls konstruiert.
- *C_Ray*: Aus der zuvor berechneten Richtung und dem Ursprungspunkt wird der Sichtstrahl gebildet.

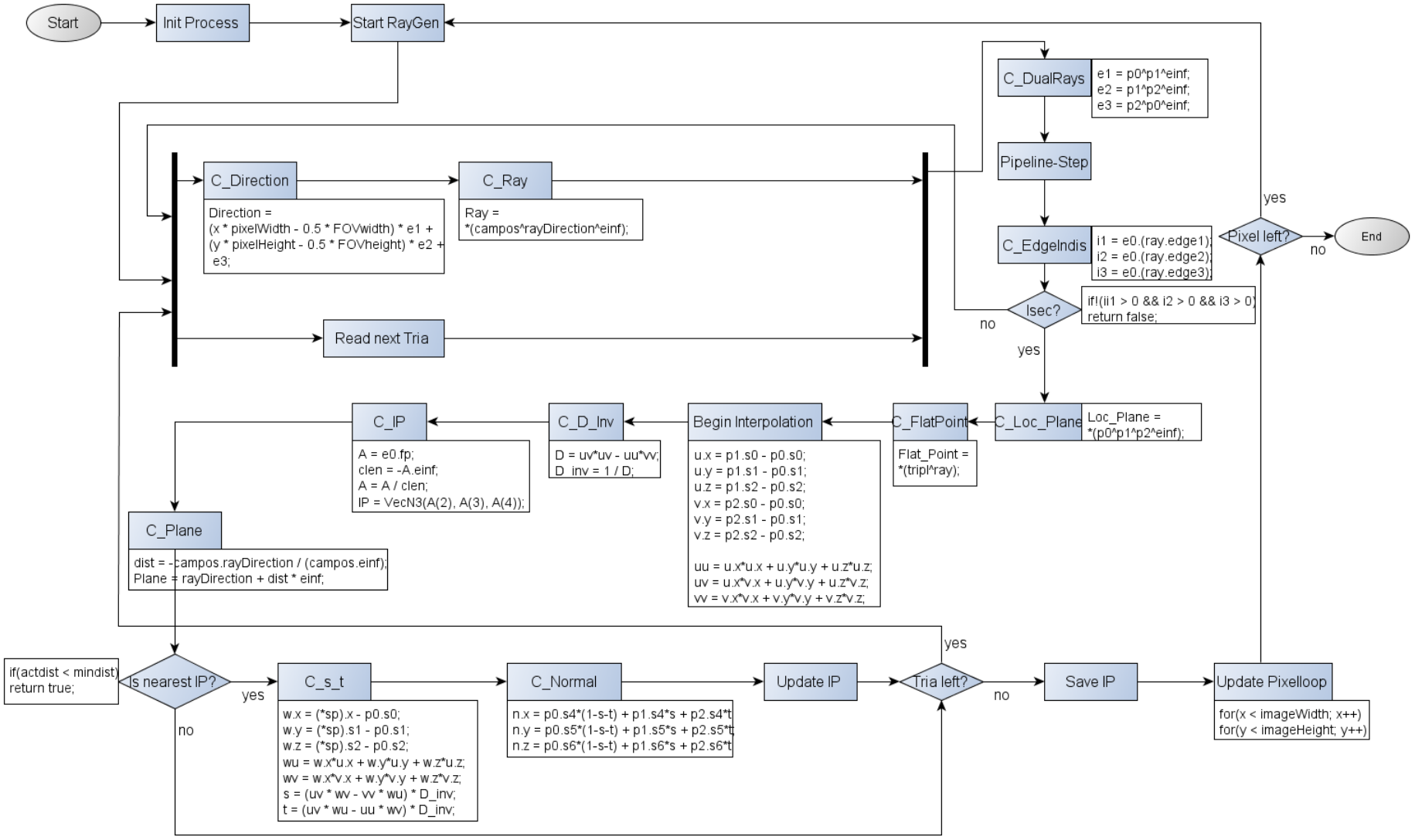

Abbildung 6.1.: Ablauf der Version 1. Folgende Abkürzungen werden verwendet: C_... = Calculate ..., IP = Intersection Point, Isec = Intersection, Tria = Triangle.

Das zweite Modul dieses Designs ist das Point-Intersector-Modul *Intersector*, welches Abb. 6.3 zeigt. Es erwartet den Strahl vom *RAYGEN* und braucht als weiteren Eingangswert das auf Schnitt zu untersuchende Dreieck. Damit es mit der Funktionsausführung beginnt, muss das Startsignal *START_ISEC* gesetzt und auf 1 gehalten werden. Als Ausgabe liefert es den fertigen Schnittpunkt *SP* und seine Distanz *DIST* zum Strahlenursprung. Es meldet das Ende der Berechnung durch *ISEC_READY* = 1. Der Indikator *HIT* gibt an, ob ein Schnittpunkt gefunden wurde, oder ob der Strahl das Dreieck verfehlt. *HIT* wird gleichzeitig mit *ISEC_READY* gesetzt und bei Rücknahme von *START_ISEC* auf 0 zurück gesetzt. Mit ihm sind alle verbleibenden Berechnungsprozesse verbunden:

- *C_DualRays*: Die drei Kanten des Dreiecks aus den Eckpunkten als *DualRays* darstellen.
- *C_EdgeIndis*: Die drei Indikatoren berechnen, die angeben auf welcher Seite der Dreieckskante der Sichtstrahl verläuft.
- *C_Loc_Plane*: Die Ebene konstruieren, auf der das Dreieck liegt.
- *C_FlatPoint*: Den Schnittpunkt als FlatPoint berechnen.
- *C_Begin Interpolation*: Die für die Interpolation nötigen Zwischenwerte uu, uv und vv berechnen.
- *C_D_Inv*: Den für die Interpolation gebrauchten Wert D und dessen Kehrwert D_Inv berechnen.
- *C_IP*: Den FlatPoint in einen konformen Punkt umwandeln.
- *C_Plane*: Ebene im Strahlenursprung berechnen, die senkrecht zu dessen Richtung liegt.
- *C_s_t*: Interpolationsparameter s und t bestimmen.
- *C_Normal*: Normale im Schnittpunkt interpolieren.

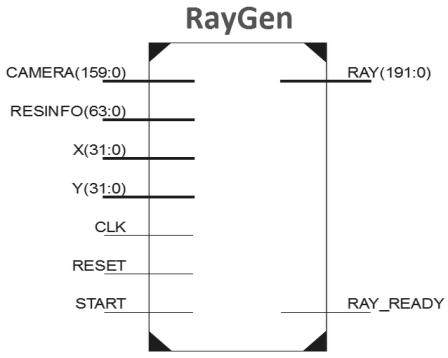

Abbildung 6.2.: RayGen V1

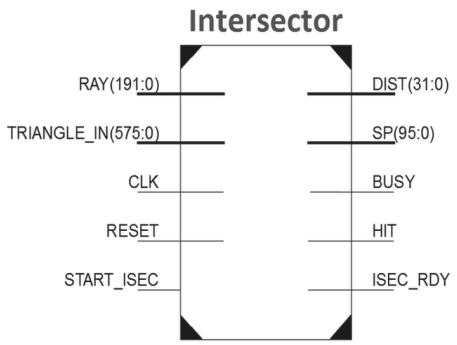

Abbildung 6.3.: Intersector V1

Drittes Modul ist der *Controller*. Seine Aufgabe ist die Steuerung des Raytracing-Ablaufs und die Kommunikation mit der Peripherie. Es prüft alle in Abb. 6.1 gezeigten Bedingungen und behandelt die Verzweigungen im Kontrollfluss. Es stellt die X-/Y-Werte, die Kameraparameter, sowie das auf Schnitt zu prüfende Dreieck bereit und verwaltet die Steuer- bzw. Antwortsignale der anderen beiden Module. Zu seinen Aufgaben gehört außerdem die Verbindung zum DDR-Speicher mit den Dreiecksdaten auf dem Virtex2p.

Tabelle 6.1 erläutert die Bedeutung der einzelnen Busse, welche die Module miteinander verbinden.

Bus-Name [Leitung]	Bedeutung	Datentyp
CAMERA [31:0]	x-Wert des Augpunkts	FP
CAMERA [63:32]	y-Wert des Augpunkts	FP
CAMERA [95:64]	z-Wert des Augpunkts	FP
CAMERA [127:96]	Pixel-Width	FP
CAMERA [159:128]	Pixel-Height	FP
RESINFO [31:0]	max. x-Auflösung	INT
RESINFO [63:32]	max. y-Auflösung	INT
X [31:0]	aktueller x-Wert	INT
Y [31:0]	aktueller y-Wert	INT
RAY [191:0]	sechs Multivektoreinträge à 32bit	FP
TRIANGLE_IN[31:0],[63:32],[95:64]	x,y,z-Wert von P0	FP
TRIANGLE_IN[191:160],[159:128],[127:96]	x,y,z-Richtung der Normale in P0	FP
TRIANGLE_IN[287:192]	x,y,z-Wert von P1 analog P0	FP
TRIANGLE_IN[383:288]	x,y,z-Richtung der Normale N0 in P0 analog N0	FP
TRIANGLE_IN[575:384]	drittes Dreieck analog zu den ersten beiden	FP
SP [95:0]	x,y,z-Wert des Schnittpunktes	FP
DIST [31:0]	Abstand des Schnittpunktes zum Strahlenursprung	FP

Tabelle 6.1.: BUS-Signale für die Version 1 des Designs. FP = 32bit-Festkommaformat, INT = 32bit-Ganzzahl.

Vorteil dieses Designentwurfs ist zum einen seine Einfachheit, da vor allem die Kommunikation sehr gering gehalten werden kann und über einfache Handshake-Protokolle realisiert wird. Den Verlauf der Kommunikationssignale für die Berechnung eines Pixels bei einem Dreiecksnetz, bestehend aus einem einzelnen Dreieck, zeigt Abb. 6.4.

Im Falle eines größeren Dreiecksnetzes wiederholen sich die Schritte 3-6 für jedes Dreieck. Weiterer Vorteil dieser Variante ist die Tatsache, dass alle Multiplizierer, die im *Intersector* gekapselt sind, sowohl für den Schnitttest als auch für die Schnittberechnung zur Verfügung stehen, so dass insgesamt weniger Multiplikationseinheiten auf dem FPGA instanziiert werden müssen. Nachteil ist zum einen die Größe des *Intersector*-Moduls. Durch seinen immensen Aufgabenbereich wird es zu einem, viele Zustände umfassenden, Automaten. Dies macht es schwer diesen Automaten zu modifizieren oder Fehler in ihm zu finden. Zum anderen ist die generelle Laufzeit des *Intersectors* dieses Entwurfs nachteilig, da der Algorithmus, wie oben angesprochen, so lange stehen bleibt, bis im Schnittfall der Punkt fertig berechnet ist. Bei einem Modell mit vielen Schnitten, wirkt sich derartiges Verhalten sehr negativ auf die Geschwindigkeit aus. Daher ist ersichtlich, dass dieser Stelle in einer verbesserten Version besonders große Aufmerksamkeit geschenkt werden muss.

6.2.2 Design der Version 2

Das Ziel der zweiten Version ist es, die Nachteile zu beseitigen, die bei der ersten Variante auftreten; vorwiegend die Dauer der Schnittberechnung. Durch die anfallende Division bei der Schnittpunktberechnung, die viele Takte benötigt, ist das Optimierungspotential an dieser Stelle ziemlich beschränkt. Allerdings ergibt sich beim Betrachten des Raycasting-Algorithmus, dass die Schnittprüfung auch als einzelner Schritt gesehen werden kann und nicht mit der eigentlichen Schnittpunktberechnung zusammenfallen muss. Daher ist es möglich die beiden Aufgaben voneinander zu entkoppeln und in einzelne Module aufzuteilen. Daraus ergibt sich für den gesamten Algorithmus ein neues Flussdiagramm, wie es Abb. 6.5 zeigt. An seinem Umfang lässt sich bereits erkennen, dass der Ablauf komplexer ist. Die auftauchenden Prozesse sind die

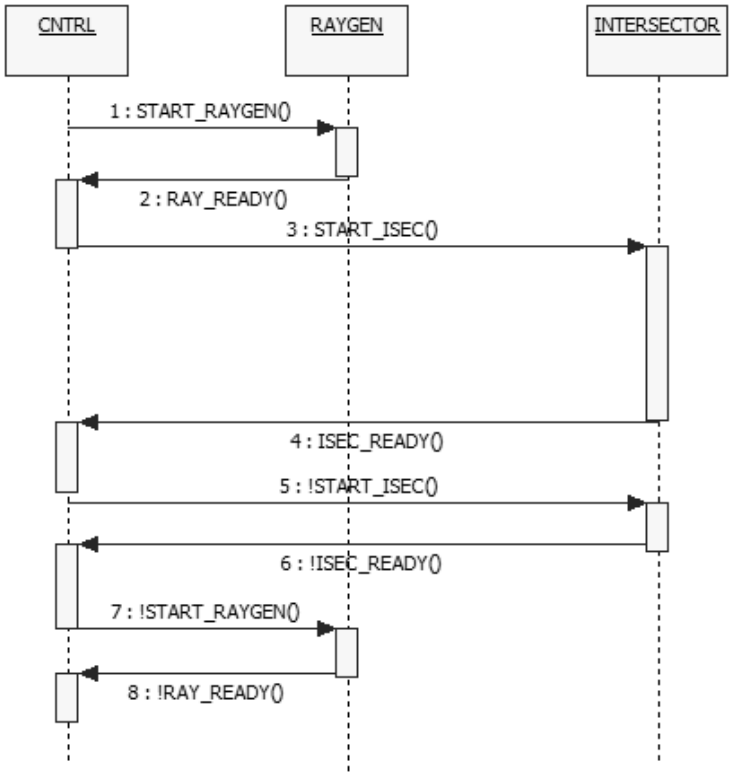

Abbildung 6.4.: Signalverlauf der Version1

gleichen wie im ersten Design, allerdings verändert sich die Ausführungsreihenfolge. Daraus resultieren deutliche Geschwindigkeitsvorteile, da jetzt mehrere Aufgaben parallel mit Synchronisation ausgeführt werden. Die Synchronisation ist durch die entsprechenden *WAIT* ...-States angegeben, da ein Prozess erst dann weiterläuft, wenn der parallel dazu arbeitende Prozess einen gewissen Fortschritt erreicht hat. Daher sind die Prozesse nicht vollkommen unabhängig voneinander. In Abb. 6.5 sind der Übersicht halber Zustände paarweise farblich markiert, die miteinander in Verbindung stehen.

Wie bereits bei Version 1 wird der Sichtstrahl parallel zur Vorbereitung des ersten zu testenden Dreiecks berechnet und steht dann bis zum Ende des betrachteten Pixels zur Verfügung. Erst wenn alle Berechnungen für das aktuelle Pixel abgeschlossen sind, setzt sich dieser Prozesszweig bis zum Ende fort.

Die Dreiecksvorbereitungen verlaufen nun parallel zu den Berechnungen, die mit den Dreiecken ausgeführt werden. Schon während sich die Schnittprozesse initialisieren, wird damit begonnen das erste Dreieck auszulesen. Auf Anforderung wird dieses Dreieck zur Verfügung gestellt und nach seiner Übergabe sofort damit begonnen das nächste auszulesen. Daraufhin wird so lange gewartet, bis das nun vorbereitete Dreieck angefordert wird. Somit wird die Kommunikation mit dem Speicher von den Schnittberechnungen verdeckt, sofern ihre Laufzeit kürzer als die des Schnitttests ist.

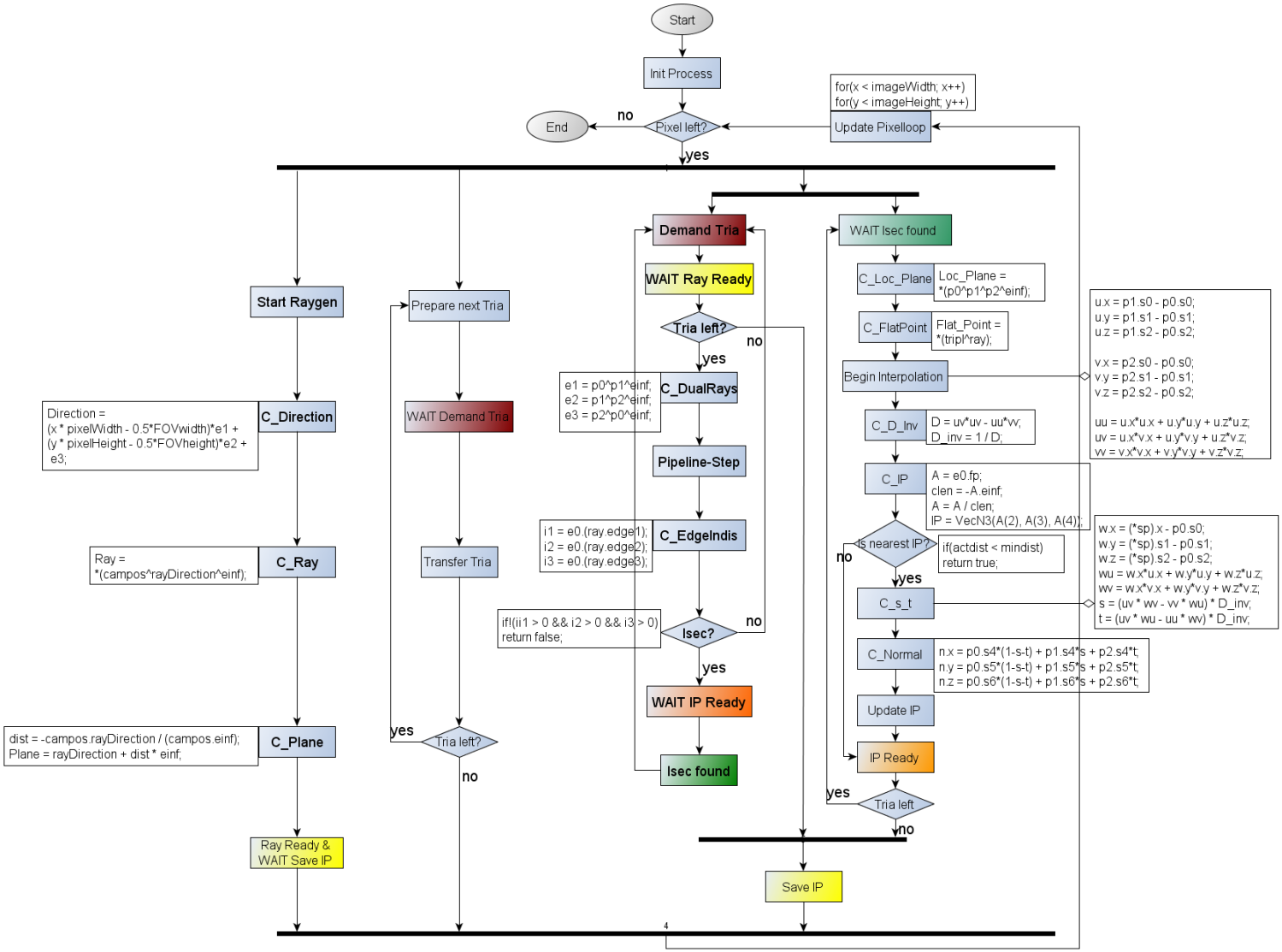

Abbildung 6.5.: Ablauf der Version 2. Folgende Abkürzungen werden verwendet: C_... = Calculate..., IP = Intersection Point, Isec = Intersection, Tria = Triangle. Voneinander abhängige Prozesse sind nach Farbe gruppiert.

Die größte Änderung zeigt sich in dem, aus zwei Prozessen bestehenden, parallelen Block auf der rechten Seite des Diagramms da dort die Trennung von Schnitttest und Schnittpunktsuche realisiert ist. Die linke Seite des Blocks ist für die Tests und für das Anfordern der Dreiecksdaten verantwortlich. Außerdem muss vor dem ersten Test auf den fertigen Sichtstrahl gewartet werden. Sind diese beiden Voraussetzungen erfüllt, kann der Test durchgeführt werden. Im Fall, dass kein Schnitt gefunden wird, ändert sich bezüglich der Laufzeit nichts im Vergleich zur Version 1, da auch der *Intersector* seine Berechnung frühzeitig abbrechen und ein neues Dreieck behandeln kann. Im Schnittfall werden nun allerdings die Dreiecksdaten und der Sichtstrahl an die eigentliche Schnittpunktberechnungseinheit weitergegeben, sofern diese nicht gerade noch durch eine Vorgängerberechnung blockiert ist. Tritt diese Situation ein muss auch hier die komplette Ausführung abgewartet werden, bis der Schnittpunkt vorhanden ist. Wenn die Einheit dagegen frei ist und die Daten übernommen hat, kann der Schnitttester das nächste Dreieck anfordern und auf Schnitt untersuchen. Es können daher beliebig viele Dreiecke betrachtet werden, für die kein Schnitt vorliegt, während die Berechnungseinheit den Schnittpunkt ermittelt. Erst wenn ein weiterer Schnitt gefunden wird, muss auf das Ende der Punktberechnung gewartet werden. Für die Performance dieses Designs ist es daher von entscheidender Bedeutung, wie oft es eintritt, dass beim Raycasten eines Dreiecksnetzes aufeinander folgende Schnitte ermittelt werden. Hierzu kann festgestellt werden, dass diese Fälle selten sind. Da die Reihenfolge der Dreiecke in einem standardmäßigen Netz so festgelegt ist, dass Dreiecke, die in der Liste aufeinander folgen im Netz Nachbarn sind, folgt für denselben Strahl auf einen Hit automatisch, dass er das nächste Dreieck verfehlen wird. Ausnahmen hiervon gibt es lediglich in folgenden Fällen:

- Der Schnittpunkt liegt genau auf der berührenden Kante von zwei Dreiecken, weshalb für beide ein Hit gefunden wird. Somit würde für beide (weitere Rundungsfehler außen vorgelassen) der gleiche Schnittpunkt gefunden, der zweimal berechnet wird. Dieses Problem ist allerdings sehr selten und kann aus praktischer Sicht vernachlässigt werden.
- Die komplette Szene besteht nicht nur aus einem Objekt. Somit kann es passieren, dass zwischen zwei, in der Reihenfolge nacheinander kommenden Dreiecken ein Objektwechsel stattfindet. Wenn eines der Objekte in der Szene, vom Augpunkt aus gesehen, vor dem anderen liegt und sich die beiden betrachteten Dreiecke direkt hintereinander befinden, werden auch zwei Schnittpunkte gefunden. Von den beiden wird derjenige mit dem geringeren Abstand zum Augpunkt verwendet. Dieses Szenario ist ebenfalls sehr selten zu erwarten.
- Die Dreiecke in der Eingabedatei sind nicht nach Nachbarschaft geordnet, so dass aufeinander folgende Dreiecke im Modell nicht nebeneinander liegen. Hier können dann Dreiecke wie im obigen Fall hintereinander liegen und somit zwei Berechnungen verursachen. Alle in dieser Arbeit untersuchten Netze besitzen die Eigenschaft der Nachbarschaft, so dass diese Situation nie eintreten kann.

Ein weiterer Faktor für die Leistungsfähigkeit des zweiten Designentwurfs ist das Verhältnis der Ausführungsdauer von Schnitttest- und Punktberechnungseinheit. Bei allen Implementierungsversuchen liegt das Verhältnis zwischen $4 : 1$ und $6 : 1$. Es können also in der Zeit, in der ein Schnittpunkt berechnet wird, durchschnittlich fünf Dreiecke auf Schnitt überprüft werden. Auf Grund der oben dargelegten Eigenschaften der Dreiecksnetze ist es dabei praktisch nie zu erwarten, dass innerhalb dieser vier bis sechs Dreiecke zwei Schnitte gefunden werden. Daher tritt der Fall, dass der Algorithmus auf das Punktberechnungsmodul warten muss, so gut wie nie

auf. In den untersuchten Timingdiagrammen kommt er zu keinem Zeitpunkt vor. Die Version 2 arbeitet im Vergleich zu Version 1 deutlich effektiver, da kaum noch Wartezeiten vorkommen.

Die neue Architektur besteht aus vier Modulen. Nahezu identisch zu Version 1 ist der *RayGen_V2*, den Abb. 6.6 zeigt. Die Eingaben, die er erwartet, sind dieselben wie bei Version 1. Auch seine beiden Steuersignale *START* und *RAY_READY* funktionieren auf analoge Weise. Der Unterschied liegt bei der zweiten Ausgabe, der *PLANE*, welche die Ebene im Strahlenursprung darstellt und für die Entfernungsberechnung des Schnittpunktes benötigt wird. Diese Aufgabe übernimmt in Designvariante 1 noch der *Intersector*. Durch diese Aufgabenverlagerung wird das neue Modul zum Berechnen des Schnittpunktes weniger komplex. Die Laufzeit des *RayGen_V2* wird hierdurch zwar nahezu verdoppelt, doch da diese Berechnung nur einmal pro Pixel durchgeführt wird und wie beschrieben parallel zum Vorbereiten des ersten Dreiecks läuft, wirkt sich diese Änderung nicht negativ auf die Gesamtperformance aus. Der Wert von *PLANE* wird wie derjenige von *RAY* nach seiner Berechnung konstant am Ausgang gehalten, bis das Startsignal auf 0 gesetzt wird.

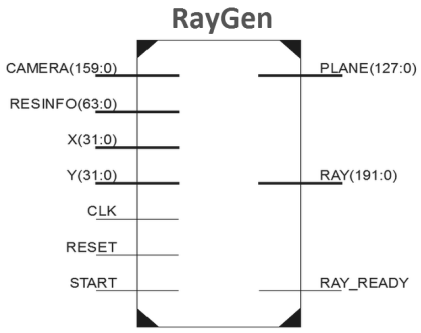

Abbildung 6.6.: RayGen V2

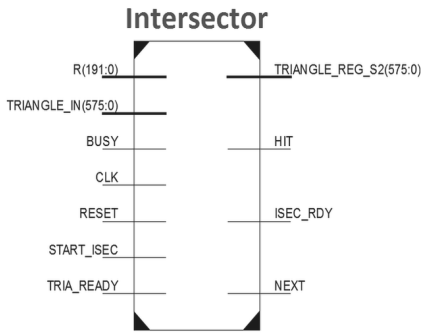

Abbildung 6.7.: Intersector V2

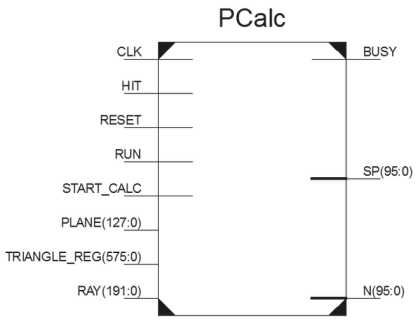

Abbildung 6.8.: PCalc V2

Das *Intersector*-Modul unterscheidet sich dagegen deutlich von der ursprünglichen Version. Es ist in Abb. 6.7 zu sehen. Gleich geblieben sind die Dateneingaben in Form vom Sichtstrahl *RAY* und dem aktuellen Dreieck *TRIANGLE_IN*. Auch das Signal *START_ISEC* zum Starten des Moduls wird beibehalten, ebenso wie der Indikator *HIT*. Neu ist der Datenausgang *TRIANGLE_REG_S2*, auf den im Falle eines Hits das Dreieck ausgegeben wird, für welches ein Schnitt gefunden wurde. Dies ist deshalb nötig, da in der Kontrolleinheit bereits das Nachfolgedreieck geladen ist, für die Schnittpunktberechnung dagegen das alte Dreieck gebraucht wird. Das *BUSY*-Signal zeigt dem Modul an, dass der Schnittpunktberechner noch arbeitet und somit momentan keine neue Berechnung durchführen kann. Die beiden Steuersignale *TRIA_READY* und *NEXT* dienen dazu neue Dreiecke anzufordern.

Das neu hinzugekommene Point-Calculation-Modul *PCalc* aus Abbildung 6.8 benötigt als Eingabedaten zum einen die Ursprungsebene *PLANE*, die der RAYGEN berechnet hat, sowie das Dreieck, für das ein Schnitt festgestellt wurde. Am Ausgabeport *SP* hält es den Schnittpunkt, der seit dem Start des Moduls, der nächste gefundene ist. Die zum Schnittpunkt gehörende Normale liegt am Ausgang *N* an. Das Verhalten des *PCalc* beeinflussen die Eingänge *HIT*, *START_CALC* und *RUN*. *RUN* deutet an, dass die Berechnung für ein neues Pixel gestartet wurde und der *PCalc* schaltet sich ein. Wird *RUN* auf 0 gesetzt schaltet sich *PCalc* ab und setzt seine Datenregister zurück. Sobald *HIT* und *START_CALC* auf high wechseln, werden das angelegte Dreieck und der Strahl übernommen und die Berechnung gestartet. Liegt der Schnittpunkt vor, wird dies durch *!BUSY* signalisiert. Der Punkt wird aber nur dann an den Ausgang gelegt, wenn

Bus-Name [Leitung]	Bedeutung	Datentyp
TRAINGLE_REG [575:0]	Dreieck für das Schnitt vorliegt. Aufbau analog zu TRIANGLE_IN	FP
PLANE [95:0]	x-,y- und z-Richtung der Normalen im Strahlenursprung	FP
PLANE [127:96]	Abstand der Ebene zum Koordinatenursprung	FP

Tabelle 6.2.: Neue BUS-Signale für die Version 2 des Designs. FP = 32bit-Festkommaformat, INT = 32bit-Ganzzahl.

er näher am Strahlenursprung liegt, als der bisher gespeicherte. *PCalc* ist daraufhin bereit eine weitere Berechnung durchzuführen. Die Aufgaben und Funktionsweise des *Controller*-Moduls sind gleich und werden nur an die neue Kommunikationsstruktur angepasst. Diese ist in Version 2 ebenfalls aufwändiger, was Abb. 6.9 veranschaulicht.

Das dortige Beispiel-Sequenzdiagramm geht davon aus, dass im Controller ein neuer Schleifendurchlauf gestartet wurde und das Dreiecksnetz aus zwei Dreiecken besteht, von denen für das erste ein Hit, für das zweite ein Miss erfolgt.

Die Module führen die gleichen Berechnungsprozesse aus, wie das erste Design. Nur deren Zuteilung zu den Modulen verschiebt sich:

- **RayGen**: *C_Direction*, *C_Ray*, *C_Plane*
- **Intersector**: *C_DualRays*, *C_EdgeIndis*
- **PCalc**: *C_Loc_Plane*, *C_FlatPoint*, *C_Begin Interpolation*, *C_D_Inv*, *C_IP*, *C_s_t*, *C_Normal*

Auch die Busse bleiben zum Großteil identisch und werden daher nicht wiederholt. Tabelle 6.2 gibt die neu hinzugekommenen Busse wider.

6.3 Implementierung der Module

Nachdem sich der vorhergehende Abschnitt mit der Architektur des Raycasters beschäftigt hat, soll es hier um die konkreten Implementierungen des zweiten Designs auf verschiedenen Plattformen gehen. Hauptaugenmerk liegt hierbei auf der Virtex2p-Plattform des Fachgebiets, da diese die ursprüngliche Zielplattform darstellt.

6.3.1 Realisierung auf einem Virtex2p

Die ML310 ACS (Adaptive Computing System) Boards sind eine Eigenentwicklung des Fachgebietes Eingebettete Systeme. Auf ihnen sitzt ein Virtex2p-Chip mit zwei PowerPC 405 CPUs. Von diesen beiden CPUs wird nur eine benutzt. Der Virtex2p wird mit 100 MHz betrieben. Außerdem sitzt auf dem Board ein 256 MB großes DDR-DRAM Modul, das von CPU und RCU genutzt werden kann. Die Boards werden über einen angeschlossenen Rechner betrieben [1].

Mittels des vorhandenen Softwarerahmens auf den Fachgebietsrechnern lassen sich zwei Grundgerüste für ein Projekt erstellen. Master- und Slave-Mode-Anwendungen. Bei Slave-Mode-Anwendungen läuft der eigentliche Algorithmus auf der CPU ab und einzelne Berechnungen werden von dort an die RCU übergeben und von dieser ausgeführt. Bei Master-Mode-

[1] Weitere Details zu den ML310 ACS finden sich in*Praktikum adaptive Computersysteme*, SS 2010, erstellt durch Thorsten Wink auf S. 31f.

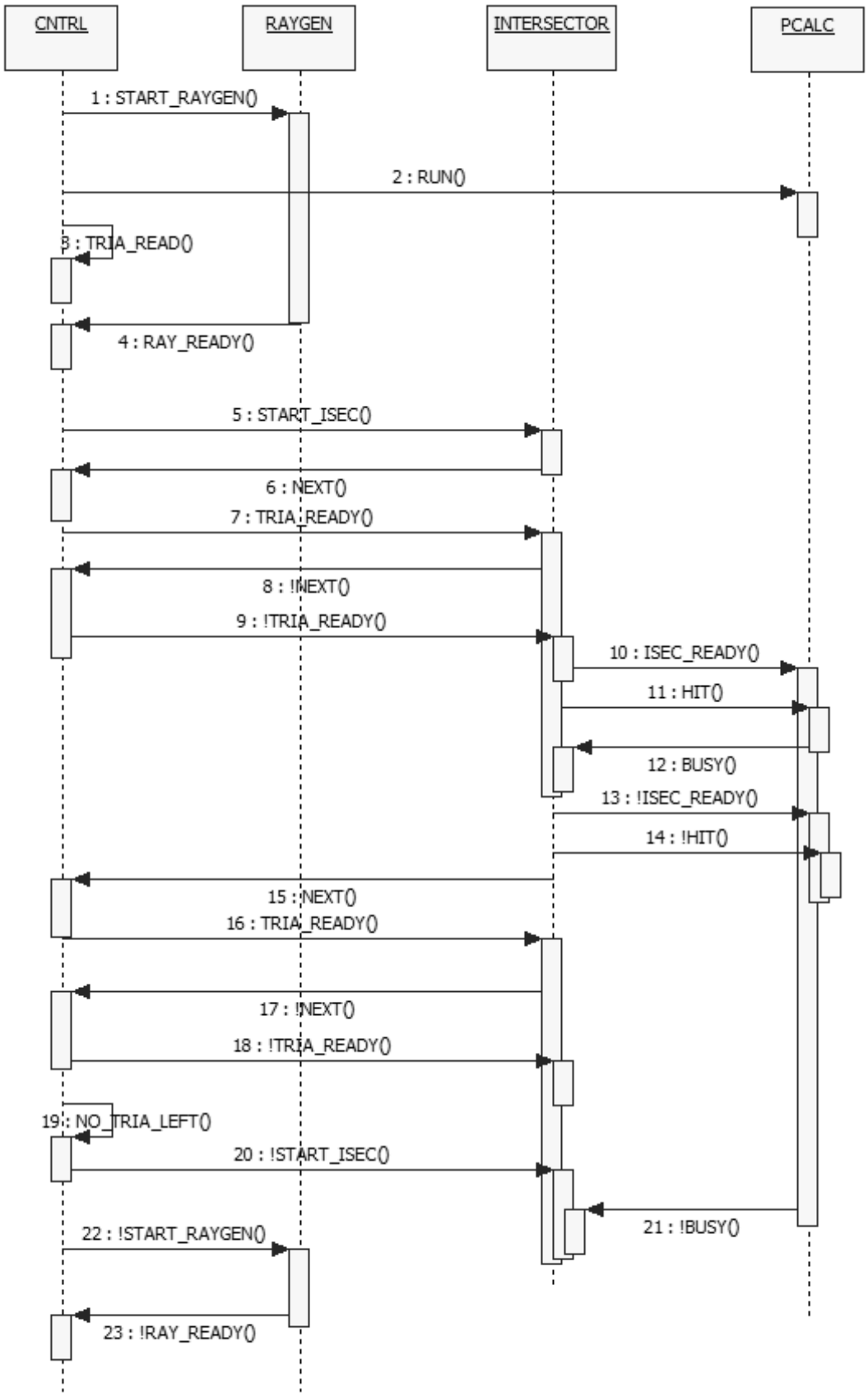

Abbildung 6.9.: Beispielsequenz der Version 2

Anwendungen dient der Softwareteil auf der CPU nur zur Initialisierung und dem Starten der Anwendung auf der RCU, auf der anschließend der eigentliche Algorithmus läuft.

Anbindung an den DDR-Speicher

Die eingesetzten ML310 ACS Boards verfügen zur Datenspeicherung über DDR-Speichermodule, die wie angesprochen von RCU und CPU benutzt werden können. Dieser Speicher dient daher, neben einigen bestimmten Registern der RCU, als Mittel zur Kommunikation zwischen der CPU und dem erstellten Raycasting-Modul. Der Raycaster ist als *Master*-Anwendung realisiert. Daraus folgt, dass die CPU nur dafür verantwortlich ist, vor dem Beginn der eigentlichen Berechnung die Eingangsdaten vorzubereiten, wichtige Parameter für das Raycasting an die RCU zu übergeben und nach Abschluss der Vorbereitungen diese zu starten.

Die Anbindung an den DDR-Speicher geschieht mittels der *Memory Architecture for Reconfigurable Computers (MARC)* des Fachgebiets. Diese besteht aus Daten- ‚Steuer-, und Flusskontrollsignalen. Der Datenfluss ist aus zwei voneinander getrennten Streams aufgebaut, die je nach Einstellung lesen oder schreiben können. Für die Flusskontrolle seitens der RCU ist das Array *STREAM_ENABLE[]* verantwortlich, das zwei 1bit-Einträge besitzt. *STREAM_ENABLE[0]* steuert den ersten Stream; *STREAM_ENABLE[1]* den zweiten. Eine 1 startet den Stream. MARC teilt der RCU durch das Signal *STREAM_STALL[]* mit, dass für den entsprechenden Stream der Datenstrom abgerissen ist und die RCU warten muss, bis die Daten wieder fließen können.

Um MARC in das Raycaster-Projekt zu integrieren, wird anhand einer Aufgabenstellung zu einem Praktikum des Fachgebiets [2] der darin existierende Rahmen für die MARC-Anbindung vervollständigt und auf die eigene Anwendung angepasst. Im Raycaster ist es sinnvoller die Speicheranbindung mehrfach herzustellen und sie danach wieder zu beenden, statt sie durchgehend aktiv zu halten. Die Dreiecksdaten müssen für jedes Pixel erneut einmal vollständig durchlaufen werden. Das Schreiben der Ergebnisse erfolgt erst im Anschluss an die Bearbeitung des letzten Dreiecks. Für die Bearbeitung eines Pixels sieht die Speicherverwaltung daher folgendermaßen aus:

Zuerst muss die Speicheranbindung konfiguriert werden. Dazu werden die wichtigen Parameter für den Ein- und Ausgabestream taktweise gesendet. Die ersten beiden Parameter sind Start- und Zieladresse der Streams. Die Startadresse für den Lesestream ist für jedes Pixel identisch, da sich die Dreiecke während des Raycaster-Durchlaufs an einer festen Position im Speicher befinden. Die Zieladresse muss dagegen für jeden Durchlauf hochgezählt werden, damit nach Abschluss der Berechnung für jedes Pixel der entsprechende Satz an Ergebnissen vorliegt. Das zweite Parameterpaar gibt die Länge bzw. Anzahl der Datensätze der Streams wider. Beide sind während eines Durchlaufs konstant. Die Anzahl der zu lesenden Datenwerte beträgt $18 * ANZAHL_{DREIECKE}$, da jeder der drei Eckpunkte aus drei Punkt- und drei Normalenkoordinaten besteht. Dem stehen acht Ausgabewerte gegenüber: Die x/y-Werte des Pixels, drei Schnittpunktkoordinaten und weitere drei für seine Normale. Die letzten drei Parameterpaare sind ebenfalls konstant: Pro Schritt wird ein Wert geschrieben / gelesen, die Datenwörter sind 32bit breit und einer der beiden Streams ist Lesestream, während der andere als Schreibstream fungiert.

[2] *Praktikum adaptive Computersysteme*, SS 2010, erstellt durch Thorsten Wink. Dort sind auch weitere Details zur Funktionsweise von MARC zu finden.

Nachdem die Initialisierung beendet ist, wird der Lesestream gestartet und die Daten für das erste Dreieck in ein entsprechend 576bit breites Register geschrieben. Anschließend wird bereits der nächste 576bit breite Datensatz gelesen und daraufhin der Stream durch Wegnehmen des *STREAM_ENABLE[0]* -Bits pausiert. Wenn das bereits im Voraus gelesene Dreieck durch das Steuerungsmodul angefordert wird, startet ein neuer Lesezyklus für einen weiteren Satz Dreiecksdaten durch das Setzen von *STREAM_ENABLE[0]*. Dies wird wiederholt, bis alle Daten für das aktuelle Pixel gelesen sind. Der Schreibstream muss nur am Ende einer Pixelberechnung genutzt werden, wenn der nächste Schnittpunkt bestimmt ist. Durch *STREAM_ENABLE[1]=1* wird der bisher pausierte Stream aktiviert und die acht Daten in aufeinanderfolgenden Takten übertragen. Zum Abschluss wird die Speicherverbindung getrennt und dem MARC-Modul angezeigt, dass alle Schreib- und Lesevorgänge beendet sind.

Besonders wichtig ist die Beachtung der Stalls, die während der Übertragung durch MARC ausgelöst werden können. MARC signalisiert damit, dass sie momentan nicht in der Lage ist, die angeforderten Daten momentan bereit zu stellen. Der Raycaster muss hier entsprechend reagieren und warten bis der Datenstrom wieder bereit ist. Abbildung 6.10 zeigt MARC.

Interner Aufbau der Module

Dieser Abschnitt soll den internen Aufbau und die Struktur der Module erläutern. Während der Raycasting-Berechnungen, die auf dem Virtex2p ausgeführt werden, treten sowohl Multiplikationen als auch Divisionen auf. Für diese beiden Operationen ist es daher nötig mittels *Xilinx Core Generator* entsprechende Cores zu erzeugen, welche diese Aufgaben übernehmen. Der eingesetzte Multiplizierer besitzt zwei 32bit Eingangssignale und einen 64bit breiten Ausgabewert. Er ist getaktet und besitzt ein Chip-enabled (CE) Signal. An Ressourcen benötigt er 124 LUT4s und vier der auf dem FPGA vorhandenen Mult18x18s Einheiten. Seine Latenz beträgt vier Takte und die Ausführung geschieht gepipelined. Die eingesetzten Cores zur Division haben zwei 32bit Eingänge und zwei 32bit Ausgänge für das ganzzahlige Ergebnis und den Rest. Auch er arbeitet erst nach *CE*.

Zur weiteren Erklärung der Struktur sind Blockschaltbilder der drei Module vorhanden. Diese sind in den Abbildungen 6.11, 6.12 und 6.13 zu finden. Die Schaltbilder zeigen, aus welchen Komponenten ein Modul aufgebaut ist und verdeutlichen den Datenfluss innerhalb des Moduls. Um die komplexen Module übersichtlich darstellen zu können, sind die Schaltbilder allerdings vereinfacht. Folgende Vereinfachungen sind dabei jeweils umgesetzt:

- Da es sich um Zustandsautomaten handelt, wird jedem Zustand eine Farbe zugewiesen. Signalleitungen, die nur in einem bestimmten Zustand aktiv sind, sind in der entsprechenden Farbe gekennzeichnet. An schwarzen Leitungen liegt unabhängig vom Zustand ein gültiges Signal an. Die Bezeichnung der einzelnen Zustände entspricht den Berechnungsprozessen in den Flussdiagrammen aus Abb. 6.5 und 6.1.
- Alle Register besitzen ein Enable-Signal, welches um die Zahl der Verbindungen zu reduzieren nicht eingezeichnet ist. Jedes Register übernimmt nur bei entsprechendem Zustand der Schaltung den Wert an seinem Eingang. Der Zustand der Schaltung ergibt sich aus den Registern STATE und COUNT, welche wiederum von den Steuersignalen am Eingang des Moduls abhängen. Die Logik zur Generierung der Enable-Signale ist ebenfalls nicht aufgeführt.

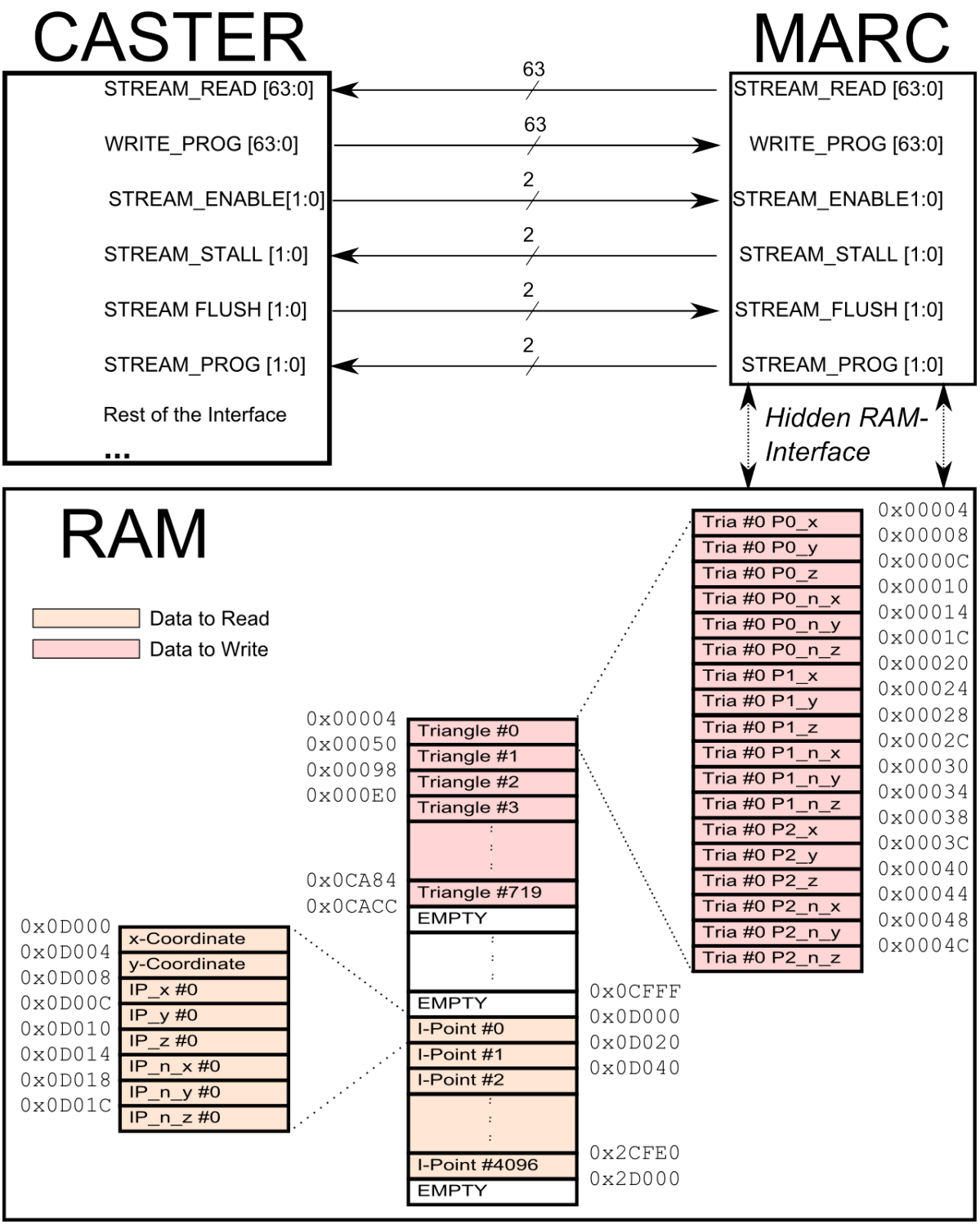

Abbildung 6.10.: MARC mit Memory-Map. Alle Daten liegen im selbst entworfenen 32bit Festkomma-Format vor.

- Die in den Schaltungen verwendeten De- / Multiplexer-Symbole, repräsentieren komplexere Einheiten. Neben der Multiplexerfunktionalität sind auch sie nur während bestimmter Zustände der Gesamtschaltung aktiv. Hinzu kommt, dass sie auch in der Lage sind, je nach Zustand, bestimmte Teile des Eingangssignales zu selektieren und an ihre Ausgänge zu legen. Ein derartiger Multiplexer, an dessen Eingang das 95bit breite Eingangssignal aus drei 32bit großen Eckpunkten anliegt, kann die einzelnen Punkte auswählen und weitergeben. Zudem führen diese Einheiten die Shifts durch, die für das Festkomma-Format benötigt werden.
- Der Anschluss des jeweiligen Moduls an die Peripherie geschieht über die schwarzen, rechteckigen Symbole.

Timingdiagramme zu den Modulen sind im Abschnitt 6.3.3 über den Virtex5 zu finden. Im Folgenden wird nur die Anzahl der Takte für die Berechnung durch ein Modul genannt.

RayGen

Der *RayGen* ist die einfachste der erstellten Komponenten. Aufgrund der untergeordneten Wichtigkeit seiner Laufzeit arbeitet in diesem Modul nur eine einzige Multipliziererereinheit. Intern ist sein Ablauf rein sequenziell. In einem ersten Schritt wird die Strahlenrichtung konstruiert, wie es im Theorieabschnitt beschrieben wird. Der entsprechende CluCalc-Code wird in Listing 6.1 wiederholt. Hierzu sind acht Takte nötig. Anschließend wird der eigentliche KGA-Strahl nach Listing 3.1, sowie die Ebene im Strahlenursprung berechnet. Hierzu fallen nochmals 14 Takte an. Insgesamt vergehen vom Setzen des Startsignals bis zum Anliegen des Ergebnisses am Ausgang 22 Taktschritte. Sein Blockschaltbild zeigt Abbildung 6.11. Die drei einzelnen Schritte der Rechnung lassen sich darin leicht erkennen. Die Direction wird für Strahlen- und Ebenenerzeugung benötigt, weshalb die Verbindung an ihrem Ausgang mit den Farben dieser beiden Berechnungen gezeichnet ist.

Der *RayGen* ist auch das Testmodul für einen Vergleich mit dem Verilog-Export von Gaalop. Die Beschreibung der Strahlenerzeugung in CluScript wird in Listing 6.1 gezeigt.

```
rayDirection = (x * pixelWidth - 0.5 * cameraFOVwidth) * e1 +
  (y * pixelHeight - 0.5 * cameraFOVheight) * e2 + e3;

ray = *(campos^rayDirection^einf);

dist = -campos.rayDirection / (campos.einf);

raypl = rayDirection + dist * einf;
```

Listing 6.1: Strahlenerzeugung in CluScript

Um zu demonstrieren, wie dieser Code manuell in Verilog übersetzt wird, ist der kommentierte Code für das RayGen-Modul in Anhang B widergegeben. Dort findet sich auch eine kurze Darstellung des Vergleichs mit dem durch Gaalop generierten Modul.

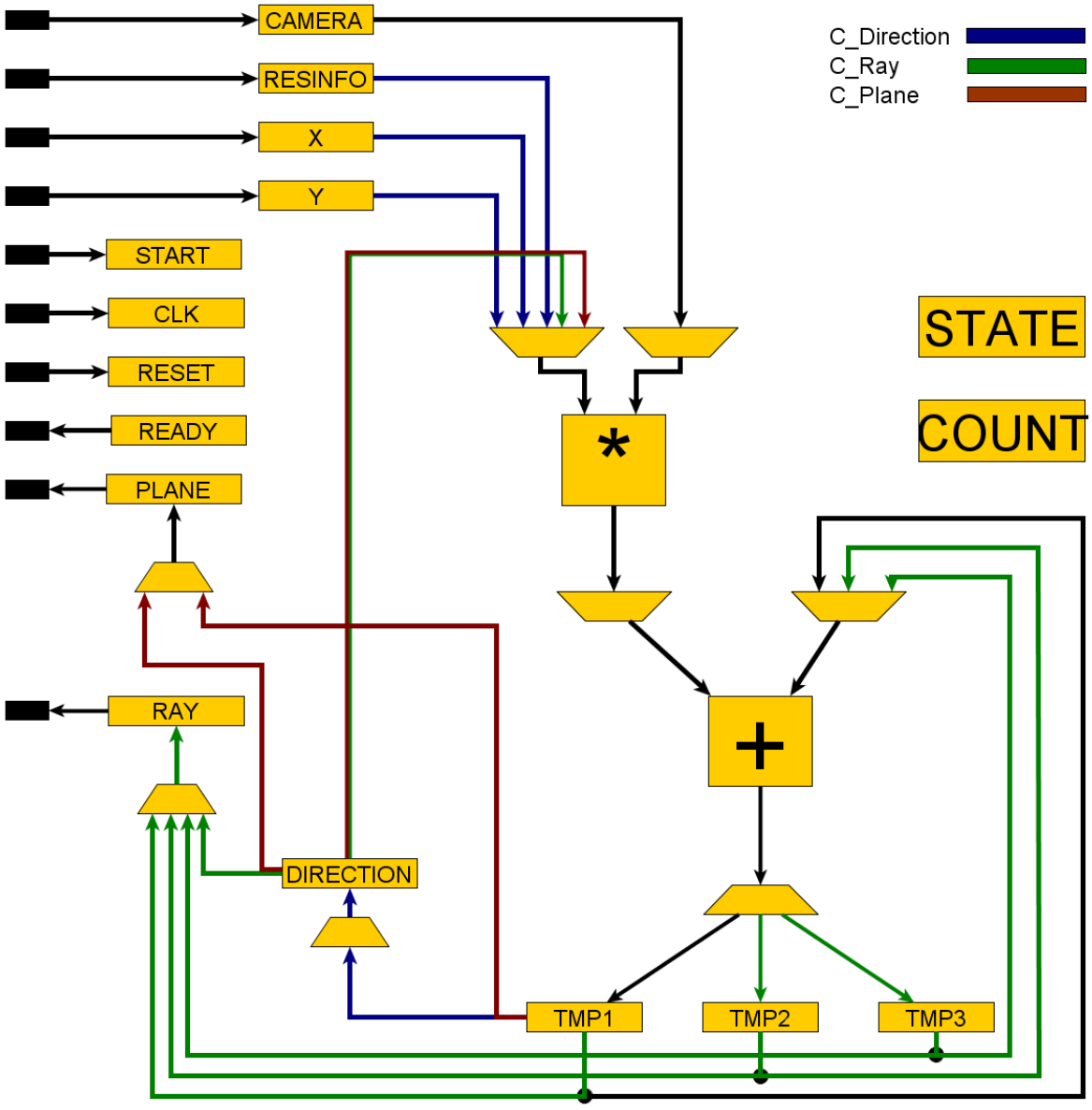

Abbildung 6.11.: Blockschaltbild des RayGen-Moduls. Details im Text.

Da die Leistungsfähigkeit des *Intersectors* zum überwiegenden Teil die Laufzeit des Algorithmus festlegt, ist die Realisierung dieses Moduls relativ aufwändig. Intern besitzt es fünf Multiplizierer. Auch arbeitet das Modul in einer zweistufigen Pipeline, um die Gesamtgeschwindigkeit so fast verdoppeln zu können. Listing 6.2 zeigt, basierend auf dem Raytracer im Anhang A, die nötigen KGA-Operationen des Schnitttests in CluScript.

```
// Beginn Pipelinestufe 1
edge1 = p0^p1^einf;
edge2 = p1^p2^einf;
edge3 = p2^p0^einf;

// Beginn Pipelinestufe 2
indi1 = e0.(ray.edge1);
indi2 = e0.(ray.edge2);
indi3 = e0.(ray.edge3);

side = -1;

if(indi1 > 0 && indi2 > 0 && indi3 > 0)
{
  side = 1;
}

if(side == -1)
{
  break;
}
```

Listing 6.2: Schnittentscheidung in KGA mittels CluScript

In der ersten Stufe der Pipeline werden die drei Kanten als duale Geraden konstruiert, aus denen im zweiten Schritt, zusammen mit dem zu prüfenden Sichtstrahl, die drei Kantenindikatoren berechnet werden. Diese legen fest, ob ein Hit vorliegt. Die Ausführung einer Pipelinestufe benötigt 15 Takte, so dass, zusammen mit der Kommunikations- und Weiterschaltzeit, für den Schnitttest eines Dreiecks 38 Takte anfallen. Die Weiterschaltzeit beträgt einen Takt. Dabei werden die Daten der Pipelineregister der ersten Stufe in diejenige der zweiten geschoben; in die der ersten werden die neuen Eingaben gelesen. Nachdem die Pipeline gestartet und mit Daten gefüllt ist, liegt alle 19 Takte ein Ergebnis am Ausgang des *Intersectors* an. Da das Auslesen eines Dreiecks aus dem Speicher 20 Takte benötigt, wird die Berechnungszeit des Schnittes durch diesen Vorgang überdeckt.

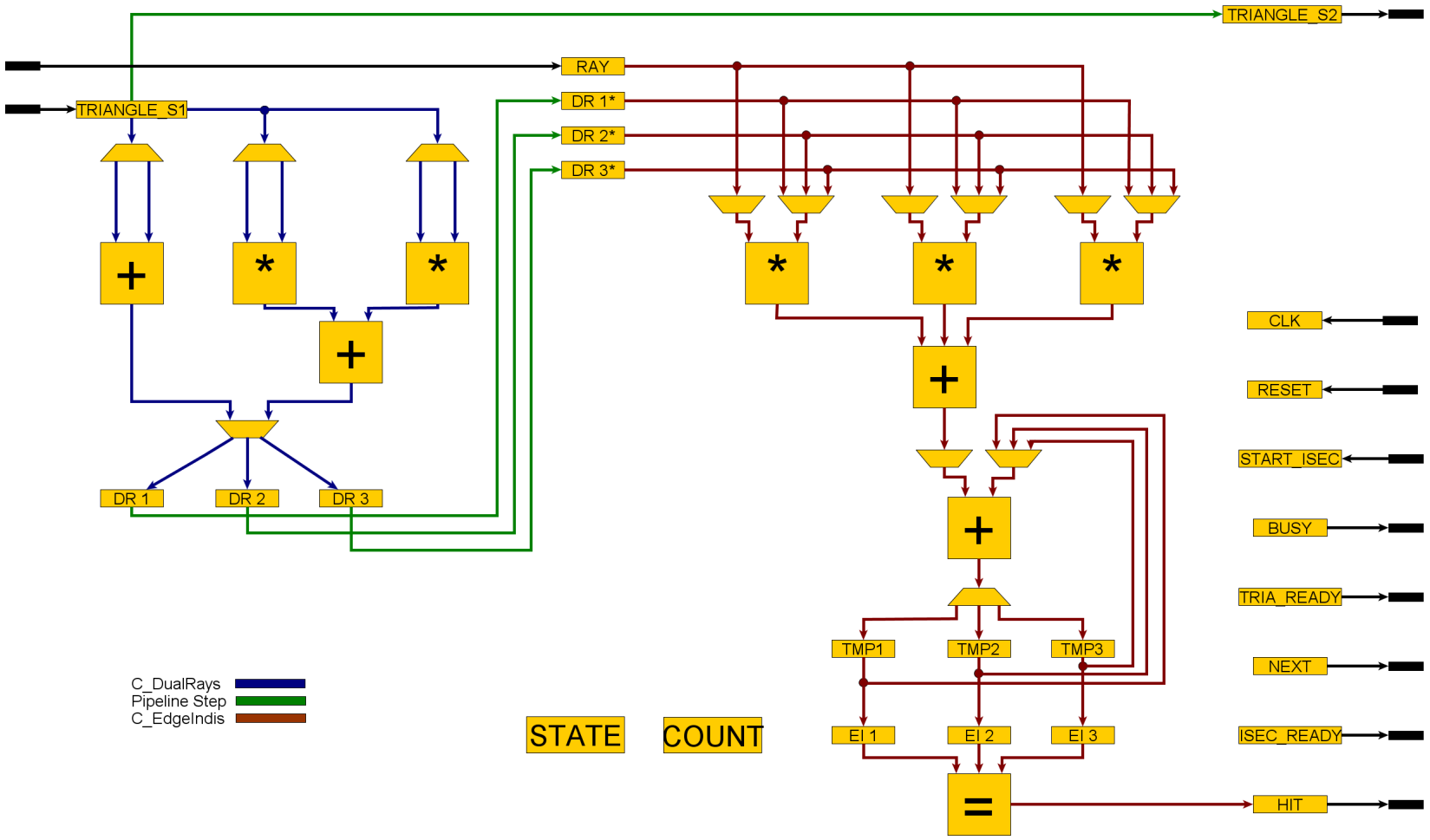

Abbildung 6.12.: Blockschaltbild des Intersector-Moduls. Details im Text.

Für den Point Calculator ist ein Kompromiss zwischen Ressourcenbedarf und Geschwindigkeit nötig, so dass er drei Multiplizierer und eine Divisionseinheit beinhaltet. In ihm wird der Schnittcode aus Listing 6.3 umgesetzt, welcher ebenfalls aus dem CluCalc-Raytracer aus Anhang A entnommen ist.

```
tripl = *(p0^p1^p2^einf);
fp = *(tripl^ray);

A = e0.fp;
clen = -A.einf;
A = A / clen;
sp = VecN3(A(2), A(3), A(4));

dist2 = raypl.sp;

if(dist2 < 0.0001)
{
  break;
}
```

Listing 6.3: Schnittberechnung in KGA

Die meisten Operationen, die hierbei anfallen, werden durch die Konstruktion der Ebene *tripl* verursacht. Da es sich hierbei um ein weiteres Beispiel handelt, wie der erzeugte CluCalc-Code noch auf besondere Anforderungen hin optimiert werden kann, soll dieser Schritt ausführlicher besprochen werden. Nachfolgend findet sich die durch Gaalop generierte Ausgabe (angepasst an die eingesetzten Objekte):

pl2 = -p2->point3*p0->point4 + p2->point4*p0->point3 + p0->point4*p1->point3 + p2->point3*p1->point4 - p2->point4*p1->point3 - p0->point3*p1->point4;

pl3 = - p2->point4*p0->point2 - p0->point4*p1->point2 + p0->point2*p1->point4 + p2->point2*p0->point4 + p2->point4*p1->point2 - p2->point2*p1->point4;

pl4 = p0->point3*p1->point2 - p0->point2*p1->point3 - p2->point3*p1->point2 + p2->point3*p0->point2 - p2->point2*p0->point3 + p2->point2*p1->point3;

pl5 = p2->point3* p0->point2*p1->point4 + p2->point4*p0->point3*p1->point2 + p2->point2* p0->point4*p1->point3 - p2->point4*p0->point2*p1->point3 - p2->point2*p0->point3*p1->point4 - p2->point3*p0->point4* p1->point2;

Es ist ersichtlich, dass einige Produkte mehrfach vorkommen und daher nur einmal berechnet werden müssen. Für die Gesamtlaufzeit gibt es aber eine noch entscheidendere Änderungsmöglichkeit: Um die Pipeline der Multiplizierer während der Berechnung stets gefüllt zu halten, müssen die jeweils farbig unterlegten Produkte für jeden Eintrag an die erste und zweite Stelle

gesetzt werden. pl.x sieht daraufhin folgendermaßen aus:

this->pl2 = p0->point4*p1->point3 + p2->point4*p0->point3 - p2->point3*p0->point4 + p2->point3*p1->point4 - p2->point4*p1->point3 - p0->point3*p1->point4;

Bei *pl3* und *pl4* wird analog verfahren. Wie die Produkte Nummer drei bis sechs pro Eintrag angeordnet werden ist dagegen nicht von Belang. Was hierdurch erreicht wird ist folgendes: Die drei Multiplizierer werden in den ersten beiden Takten mit den ersten beiden Spalten der ersten drei Ebeneneinträge gefüllt. Anschließend werden die Spalten drei bis fünf nachgeschoben. Im darauf folgenden Taktschritt stehen die Ergebnisse für Spalte eins von *pl.x*, *pl.y* und *pl.z* bereit und es lassen sich die Multiplikationen für die ersten drei Spalten von *pl.w* an die Eingänge der Multiplizierer anlegen. Einen Takt später sind nun die Werte für die Spalte zwei der ersten drei Einträge verfügbar und die drei letzten Spalten von *pl.w* können an die Multiplizierer übergeben werden. Daraufhin wird noch die sechste Spalte der ersten drei Einträge in die Pipeline geschoben. Parallel zu den letzten beiden beschriebenen und in den nachfolgenden Takten werden die pro Taktschritt eintreffenden Zwischenergebnisse in vier entsprechenden Registern aufsummiert und geshifted, bis die fertige Ebene zur Verfügung steht.

Diese Vorgehensweise bietet zwei Vorteile. Zum einen werden keine temporären Register benötigt, um die Produkte zwischenzuspeichern, die den vier Einträgen gemeinsam sind, da die Zwischenwerte sofort nach ihrer Berechnung weiterverarbeitet werden können. Zum anderen gelingt es so, alle für die Erzeugung der Ebene anfallenden 30 Multiplikationen innerhalb von nur 13 Takten zu erledigen und die Pipelines der Multiplizierer ohne Unterbrechung gefüllt zu halten. Für drei Multiplizierer stellt dies das theoretische Maximum dar. Um es zu ermöglichen, dass der Gaalop-Verilog-Compiler derartige, an die Zielarchitektur angepasste, Pipeline-Strukturen erzeugen kann, müssten zumindest die beiden Parameter *Anzahl der Multiplizierer* und *Pipelinestufen der Multiplizierer* zur Kompilierzeit bekannt sein.

Eine weitere wichtige Designentscheidung beruht auf der Tatsache, dass eine Division deutlich mehr Zeit benötigt als eine Multiplikation. Somit sollte, sofern realisierbar, die Division so bald wie möglich gestartet werden und während ihrer Ausführung möglichst viele andere Berechnungen getätigt werden, um die Divisionszeit zu nutzen. Dies ist im vorliegenden Beispiel möglich. Listing 6.4 zeigt hierzu den Code für den eigentlichen Schnitt von gerade konstruierter Ebene *pl* mit dem Strahl *r* in OpenCL.

```
float fp9  = -pl.s3*r.s3 + pl.s2*r.s4 - pl.s1*r.s5; // e1^einf
float fp12 =  pl.s3*r.s1 - pl.s2*r.s2 + pl.s0*r.s5; // e2^einf
float fp14 = -pl.s3*r.s0 + pl.s1*r.s2 - pl.s0*r.s4; // e3^einf
float fp16 =  pl.s2*r.s0 + pl.s0*r.s3 - pl.s1*r.s1; // einf^e0

sp.s0 = 1.0f/(-fp16) * fp9; // e1
sp.s1 = 1.0f/(-fp16) * fp12; // e2
sp.s2 = 1.0f/(-fp16) * fp14; // e3
sp.s3 = 1.0f;
```

Listing 6.4: Schnittberechnung in KGA mittels OpenCL

Die implementierte C++-Fassung unterscheidet sich in ihrer Funktionsweise nicht, aber die OpenCL Variante ist in diesem Fall einfacher zu lesen. Die anfallende Division teilt die ers-

ten drei Koordinaten des Schnittpunktes durch dessen vierte. Somit ist es das Ziel der Verilog-Realisierung diese nach Möglichkeit früh zu starten. Ersichtlich ist, dass *fp16* unabhängig von den anderen drei Koordinaten ist und nur die Ebene und den Strahl benötigt. Somit wird die Berechnung von *fp16* sofort gestartet, nachdem die ersten drei *pl*-Werte verfügbar sind und erst danach die verbleibenden Einträge betrachtet. Während die Division für den Kehrwert von *fp16* läuft, werden die Daten zur Interpolation der Normalen, so weit wie sie ohne den Schnittpunkt bestimmbar sind, berechnet.

Die Verzahnung der verschiedenen Arbeitsschritte führt zwar zu höher Geschwindigkeit, macht aber den Code und somit auch das Blockschaltbild (Abb. 6.13) sehr komplex. Dennoch ist auf dem Schaltbild erkennbar, zu welchem Zeitpunkt der Berechnung die Daten aus den internen Registern verwendet werden. Auch die temporären Register, die bei der Ebenenberechnung angesprochen werden, sind in Abb. 6.13 unter den Namen TMP1 - TMP4 zu finden. Außerdem befinden sich zentral die drei Multiplizierer und auf der linken Seite des Schaltbildes das Divisions-Modul. Da dieses für den Algorithmus nur verwendet wird, um Kehrwerte zu bilden, liegt an seinem einen Eingang eine konstante 1 an. Am linken Rand des Bildes ist die Entscheidungseinheit eingezeichnet, die bestimmt, ob ein errechneter Schnittpunkt der nächstgelegene ist und den bisherigen ersetzen muss. Der aktuell nächste Punkt wird auf die Ausgabepins gelegt.

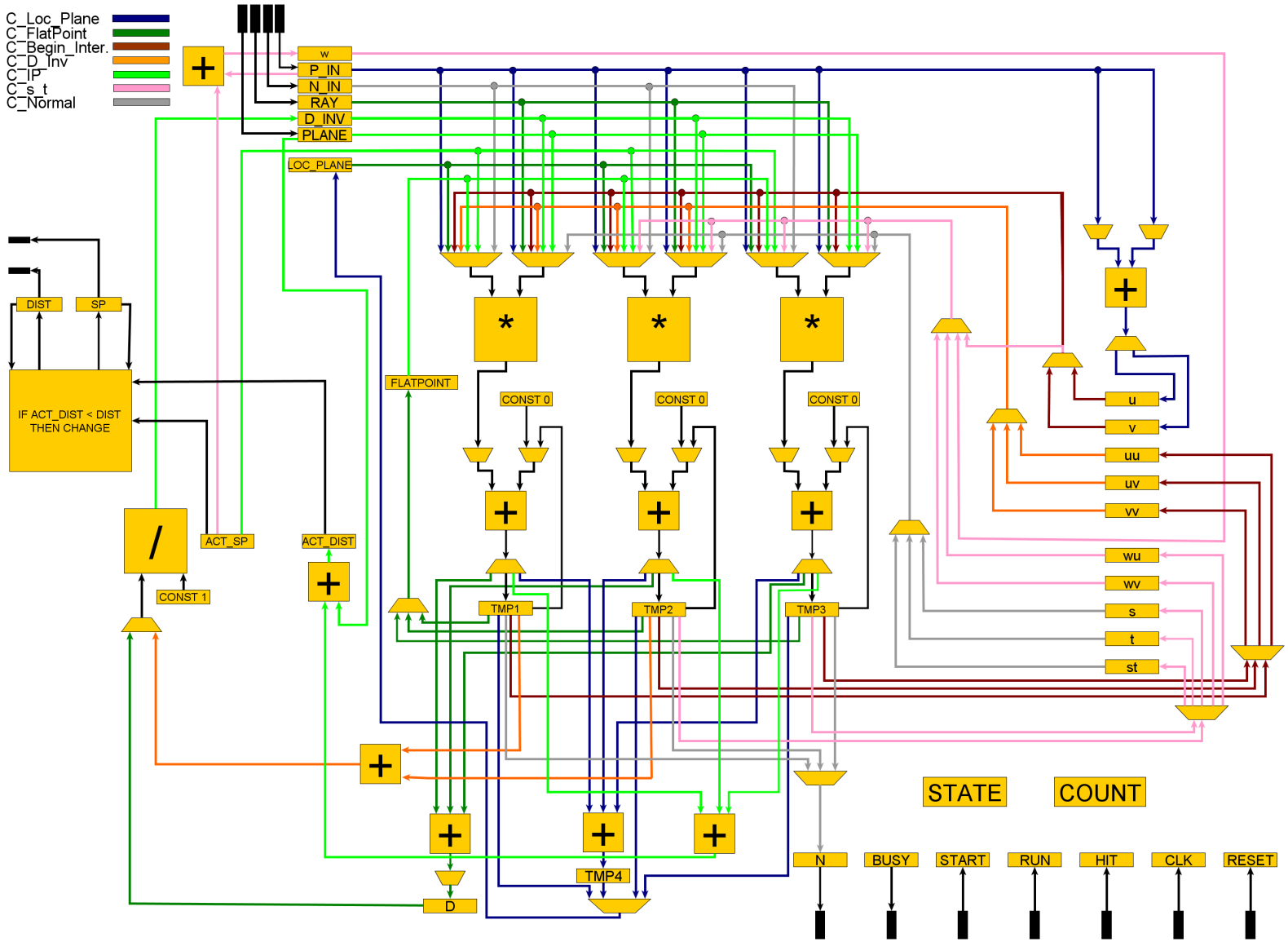

Abbildung 6.13.: Blockschaltbild des PCalc-Moduls. Details im Text.

Der Softwareteil

Da es sich bei dem Raycaster um eine Master-Anwendung handelt, ist der C-Code, der auf dem PowerPC-Prozessor ausgeführt werden muss, sehr kompakt. Den größten Anteil macht das Einlesen der Quelldaten und der Kommandozeilenparameter aus. Die Aufrufparameter sind in dieser Reihenfolge:

1. Dateiname der Quelldatei
2. Dateiname der Zieldatei
3. Auflösung des Bildes
4. Der FOV-Kameraparameter

zu übergeben. Das Format für die Eingangsdaten basiert auf den vorhandenen *mem*-Dateien. Erstellt werden die Dateien mittels des C++-Raytracers, der mit der Funktionalität des mem-Exports ausgestattet ist. Der erste Wert der Datei gibt die Adresse wieder, an welcher die Dreiecksdaten im Speicher beginnen sollen. Der zweite Wert, der sich noch in der selben Zeile befindet, gibt die Anzahl der Dreiecksdaten als Hex-Wert an. Anschließend folgt jede Zeile ein Dreieckswert in der Reihenfolge: Punkt 1 / Normale 1, Punkt 2 / Normale 2 und Punkt 3 / Normale 3. Diese Daten werden der Einlesereihenfolge entsprechend in den Speicher geschrieben und die Startadresse, sowie die Länge des Datenblocks, als Parameter an die RCU zu übergeben. Für die Ausgabewerte wird ebenso eine Startadresse und deren Länge weitergereicht, die von der CPU anhand der Auflösung berechnet wird. Außerdem werden noch die Aufrufparameter 3 und 4 in die Register der RCU geschrieben, bevor die RCU durch einen Zugriff auf deren *START*-Register aktiviert wird. Die CPU wartet nun darauf, dass die RCU durch einen Interrupt das Ende der Berechnung meldet. Daraufhin werden die Ausgabedaten aus dem zuvor reservierten Speicherbereich gelesen und sequentiell in die Ausgabedatei geschrieben.

Um das berechnete Bild sichtbar zu machen, muss die erzeugte *.mem-Datei einem Zusatzmodul des C++-Raytracers, dem *Picture-Renderer*, übergeben werden. Dieser liest die hexadezimalen Schnittpunkte, sowie deren Normalen, ein und konvertiert sie ins *float*-Format. Anschließend werden daraus konforme Punkte erzeugt und wie die Raytracing-Bilder unter Einsatz des Phong-Modells geshaded. Die Position des Lichtes, sowie das Material, aus welchem das Objekt bestehen soll, können vom Benutzer festgelegt und als Aufrufparameter übergeben werden.

Der C++-Raytracer bietet außerdem die Funktion, das von ihm berechnete Bild ins mem-Format zu exportieren, um so einen optischen Vergleich mit den FPGA-Bildern zu haben. Zusätzlich ist durch den *Picture-Renderer* die Möglichkeit geboten, die prozentualen Unterschiede in den Ergebnissen zwischen Hardware- und Software-Rendering auszuwerten.

Simulation und Synthese für den Virtex2p

Trotz der deutlichen Reduzierung der benötigten Multiplizierer-Cores durch die Verwendung von Pipelining, ist das Projekt zu umfangreich, um auf dem Virtex2p ausgeführt zu werden. Die Synthese des vollständigen Raycasters mit Speicheranbindung meldet, dass auf dem FPGA nicht genügend Ressourcen zur Verfügung stehen.

Daher wird folgendes Design untersucht: Die Speicheranbindung und die Schleifenkonstruktion wird in der Hauptdatei des Projekts auskommentiert. Die Kommunikation mit der CPU über Register bleibt erhalten, so dass diese das Startsignal sendet und den Interrupt abwartet. Testweise werden einige feste, beispielhafte Dreiecksdaten in Register geschrieben. Außerdem wird die Berechnung der Normalen im *PCalc* deaktiviert. Danach lässt sich ein bit-File generieren, doch dieses ist fehlerhaft und funktioniert nicht auf den FPGAs. Der obere Teil von Tabelle 6.3 zeigt den Ressourcenverbrauch dieses Designs.

Slice Logic Utilization	Used	Available	Utilization
Raycaster ohne Speicher und Normalenberechnung			
Number of Slice Flip Flops	19064	27392	69 %
Number of 4 input LUTs	19796	27392	72 %
Number of occupied slices	13694	13696	99 %
Number of MULT18X18s	35	136	25 %
ISE Projekt			
Number of Slice Flip Flops	12390	27392	45 %
Number of 4 input LUTs	15652	27392	57 %
Number of occupied slices	10698	13696	78 %
Number of MULT18X18s	35	136	25 %
Leere Master-Mode-Anwendung			
Number of Slice Flip Flops	6,757	27392	24 %
Number of 4 input LUTs	9,781	27392	35 %
Number of occupied slices	7,584	13,696	55 %

Tabelle 6.3.: Ressourcenverbrauch des Virtex2p-Raycasters. Aus dem Synthesereport des Xilinx ISE. Im oberen Teil finden sich die Daten für das Master-Mode-Design mit deaktivierter Speicheranbindung. Die Mitte zeigt die Belegung für das ISE Projekt. Der untere Tabellenteil gibt den Verbrauch einer leeren Master-Mode-Anwendung wider. Details siehe Text.

25 % der Slices enthalten dabei dem Synthesereport zu Folge *unrealated logic*, während 74 % rein mit related belegt sind. Diese beiden Arten sind wie folgt seitens Xilinx definiert:

> Related logic is defined as being logic that shares connectivity - e.g. two LUTs are related if they share common inputs. When assembling slices, Map gives priority to combine logic that is related. (...)
>
> Unrelated logic shares no connectivity. Map will only begin packing unrelated logic into a slice once 99% of the slices are occupied through related logic packing.
>
> (...) Unrelated logic packing will then begin, continuing until all usable LUTs and FFs are occupied. Depending on your timing budget, increased levels of unrelated logic packing may adversely affect the overall timing performance of your design. (aus der *_map.mrp Datei des Projekts)

Dass der Baustein schon sehr voll ist zeigt die Tatsache, dass bei Reaktivierung der Normalen, bzw. bereits schon der Berechnung von uu, uv und vv der Placing-Vorgang fehlschlägt.

Um den Ressourcenverbrauch weiter zu untersuchen, wird ein ISE Projekt für den Virtex2p erstellt und ein kleiner Rahmen mit festen Eingangsdaten geschrieben der die Ausgabe auf

LEDs umleitet. Die Ergebnisse des Mappings sind im unteren Teil von 6.3 gezeigt. Auch bei dieser Version sind bereits 80 % der Slices belegt. Welches Modul des Entwurfs dabei wieviel Platz auf dem FPGA belegt, lässt sich mittels des ISE für den Virtex2p nicht feststellen. Da die Module aber vom Aufbau mit denen des Virtex5 aus Abschnitt 6.3.3 identisch sind, ist es sehr wahrscheinlich, dass sie sich beim Platzverbrauch vergleichbar zu den Virtex5-Ergebnissen aus Tabelle 6.4 verhalten. Eine Betrachtung einer leeren Master-Mode-Anwendung ergibt, dass der Rahmen für diese schon einen relativ hohen Ressourcenbedarf hat, was ebenfalls aus Tabelle 6.3 ersichtlich wird. Für den Masterrahmen und den Raycaster gleichzeitig steht somit nicht genügend Platz zur Verfügung.

Da die Synthese des Raycasters für den Virtex2p nicht möglich ist, basieren die folgenden Aussagen und Tests in diesem Kapitel auf den Ausgaben des VCS-Simulators. Es lässt sich aufgrund der korrekten Funktion des Moduls in der Simulation annehmen, dass der Raycaster auf einem entsprechend großen Baustein funktioniert. Dies lässt sich auch aus dem Aufbau der Module als synchrone, endliche Zustandsautomaten und daraus, dass keine timing-kritischen Berechnungen enthalten sind, folgern. Auch bestätigen die Ergebnisse aus den Abschnitten 6.3.2 und 6.3.3 diese Annahme.

In Abbildung 6.14 sind eine Kugel in der Auflösung 128x128 und das Model *Cow* mit der Auflösung 256x256 zu sehen.

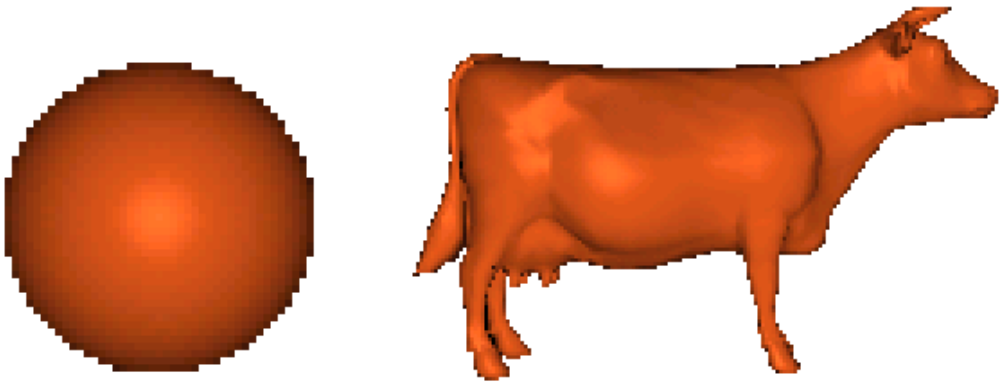

Abbildung 6.14.: Links *Geosphere* (128x128), rechts *Cow* (256x256). Gerendert mit Hilfe der Ausgaben des Simulators

Beide Modelle werden durch den *Picture-Renderer* auf die gleiche Größe skaliert. Die Laufzeit des Simulators zum Berechnen der Kugeldaten beträgt über acht Stunden. Für die *Cow* lief die Simulation er über zwei Tage. Um die Modelle überhaupt simulieren zu können, werden als Eingabe nur diejenigen Dreiecke übergeben, die im Ergebnisbild zu sehen sind und der Pixelbereich so eng wie möglich gewählt. Andernfalls stürzt der Simulator im Verlauf des dritten Tages mit einem Speicher-Allozierungs-Fehler ab. Die durchschnittlichen Abweichungen bei den drei Koordinaten der Schnittpunkte liegt in allen Fällen unter 0, 10%. Die Normalen weichen im Schnitt etwas über 1%, aber nie weiter als 5% ab.

6.3.2 Realisierung auf einem Spartan3E

Die drei eigentlichen Raycasting-Module werden zum Test ihrer Funktionsfähigkeit auf einem Spartan3E Starter Kit[3] ausgeführt. Dieser arbeitet mit einem Takt von 50 MHz. Der FPGA ist zu klein, um alle drei Module gleichzeitig fassen zu können. Daher werden zwei verschiedene Kombinationen implementiert und synthetisiert.

Für die erste Hälfte der Funktionsprüfung werden *RayGen* und *Intersector* miteinander verbunden und sämtliche Eingabedaten fest in den Verilog-Code geschrieben. Über eine der LEDs wird dabei ausgegeben, ob ein Schnitt gefunden wird. Über die *Slide Switches*[4] kann zwischen verschiedenen Eingabekombinationen gewählt werden. Alle betrachteten Testfälle liefern dabei zum C++-Raytracer identische Ergebnisse.

In der zweiten Testhälfte wird der *PCalc* untersucht. Auch für ihn sind die Eingaben fest im Quellcode vorhanden. Die *Slide Switches* werden in diesem Fall dazu benötigt, um zwischen den einzelnen Bytes der Ergebnisvektoren zu wählen. Diese werden auf den acht LEDs ausgegeben. Auch beim Spartan3E sind die Schnittpunktkoordinaten, bis auf die auftretenden Rechenungenauigkeiten, korrekt.

Tabelle 6.4 zeigt die verbrauchten Ressourcen der beiden Test-Teile, sowie diejenigen für die in ihnen integrierten Module. Wie erkennbar ist, belegt alleine Test 2 den FPGA schon ziemlich vollständig. Auffällig ist, dass der *RayGen* im Vergleich zum *Intersector* größer ist als auf dem Virtex2p. Dies liegt darin begründet, dass auf dem Spartan3E nicht genügend MULT18x18 verfügbar sind, um alle Multiplizierer-Cores über diese zu realisieren. Daher ist der Multiplizierer des *RayGen* nur mittels LUTs umgesetzt und benötigt deshalb mehr der anderen Ressourcen.

	Slices (4656)	Slice Reg (9312)	LUTs (9312)	MULT18x18 (20)
Test 1	2397 / 51 %	3099 / 33 %	2919 / 31 %	20 / 100 %
RayGen	593 / 25 %	891 / 28 %	838 / 29 %	0 / 0 %
Intersector	1468 / 61 %	2170 / 70 %	1739 / 60 %	20 / 100 %
Test 2	4461 / 90 %	4172 / 45 %	5005 / 52 %	12 / 60 %
PCalc	4207 / 94 %	4131 / 99 %	4861 / 97 %	12 / 10 %

Tabelle 6.4.: Ressourcenverbrauch der einzelnen Module auf dem Spartan3E. Aus dem Synthesereport des Xilinx ISE. In runden Klammern hinter dem Ressourcentyp findet sich die Gesamtzahl, die auf dem FGPA verfügbar sind. Pro Test ist für jeden Ressourcentyp die absolute Anzahl an benötigten Einheiten, sowie deren prozentualer Anteil an den Gesamtressourcen angegeben. Für jeden Test sind außerdem die darin enthaltenen Module gelistet. Für diese ist die Gesamtzahl eines Ressourcentyps, sowie der prozentuale Anteil am Gesamtverbrauch des jeweiligen Tests widergegeben.

Da sich die Latenzzeit der Multiplizierer beim Spartan3E auf nur drei Takte beläuft, müssen die Module verändert werden. Der *RayGen* benötigt zur Strahlenerzeugung 20 Takte; der *Intersector* fällt eine Schnittentscheidung innerhalb von 16 Takten. Die Laufzeit des *PCalc* verändert sich

[3] Die Produkthomepage bei Xilinx: `http://www.xilinx.com/products/boards-and-kits/HW-SPAR3E-SK-US-G.htm`

[4] Bezeichnungen zum Spartan3E Board sind [Xil11] entnommen.

nicht. Die Multiplikationen können zwar schneller ausgeführt werden, aber es existieren nun Takte, in denen sich Multiplizierer im Leerlauf befinden.

6.3.3 Realisierung auf einem Virtex5

Da der Virtex2p und der Spartan3E zu wenig Ressourcen bieten, um alle Module gleichzeitig auf ihnen zu platzieren, wird als Proof of Concept zum Abschluss der FPGA-Untersuchungen eine angepasste Version für eine Virtex5 (XC5VLX110T)-Platine (Xilinx ML505 Evaluation Platform) [5] entwickelt. Da auf der Plattform keine CPU zur Verfügung steht, werden die Testdreiecke fest in Register geschrieben. Mittels der auf dem Board vorhandenen *GPIO DIP Switches* 1-5 kann zwischen den verschiedenen Dreiecken gewechselt werden. Zur Ausgabe der berechneten Werte dienen die acht *GPIO LEDs*, auf denen jeweils 8bits einer der drei Punkt- oder Normalenkoordinaten ausgegeben werden. Welche Ergebnis-Bits durchgeschaltet werden bestimmt die Stellung der *GPIO DIP Switches* 6-8. In Tabelle 6.5 sind die Schalterstellungen und die entsprechenden Teile der Ausgabewerte gezeigt.

Schalterstellung	Ergebnisbit aus SP bzw. N	Entsprechung
00000 / 00001 / 00010 / 00011	[7:0] / [15:8] / [23:16] / [31:24]	x-Wert SP
00100 / 00101 / 00110 / 00111	[39:32] / [47:40] / [55:48] / [63:56]	y-Wert SP
01000 / 01001 / 01010 / 01011	[71:64] / [79:72] / [87:80] / [95:88]	z-Wert SP
10000 / 10001 / 10010 / 10011	[7:0] / [15:8] / [23:16] / [31:24]	x-Wert N
10100 / 10101 / 10110 / 10111	[39:32] / [47:40] / [55:48] / [63:56]	y-Wert N
11000 / 11001 / 11010 / 11011	[71:64] / [79:72] / [87:80] / [95:88]	z-Wert N

Tabelle 6.5.: Die Tabelle gibt den Zusammenhang der Schalter 1-4 und dem ausgegebenen Ergebnis auf den 8 LEDs an. Eine 0 steht hierbei für *Schalter unten* (low); eine 1 für *Schalter oben (high)*

Da sich die generierten Cores für den Virtex5 erneut von denen für Virtex2p und Spartan3E unterscheiden, müssen die Zustandsautomaten für die einzelnen Module wiederum auf den FPGA angepasst werden. Die Multiplizierer-Cores sind vollständig auf den *XtremeDSP slices* des FPGAs realisiert und haben bei gepipelineter Architektur eine Latenz von sechs Takten, womit sie zwei Takte langsamer arbeiten, als diejenigen auf dem Virtex2p. Für das *RayGen*-Modul hat dies zur Folge, dass es in dieser Konfiguration 27 Takte zur Berechnung eines Strahles benötigt. Auch die Ausführungszeit des *Intersectors* erhöht sich auf 20 Takte.

Eine Fassung des Raycasters mit fünf wählbaren Testdreiecken zeigt, dass ein FGPA von der Größe des Virtex5 für den Raycaster ausreicht, auch wenn man einen weiteres Modul zur Verbindung mit dem Speicher integrieren würde. Tabelle 6.6 zeigt die wichtigsten Daten des Synthesereports.

Lediglich die DSP48Es sind über 50% ausgelastet, wohingegen die anderen wichtigen Ressourcen nur bis zu einem Fünftel belegt sind. Eine Speicheranbindung würde keine weiteren Multiplizierer belegen und noch genügend freie Ressourcen vorfinden. Um sich einen Überblick über

[5] Der User Guide zu diesem FPGA kann auf `http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf` heruntergeladen werden. Der FPGA läuft mit 100 Mhz. Auf ihm beruhen die Bezeichnungen der einzelnen Komponenten des Boards.

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	9878	69120	14 %
Number of Slice LUTs	10736	69120	16 %
Number of occupied slices	4003	17280	30 %
Number of DSP48Es	35	64	54 %

Tabelle 6.6.: Ressourcenverbrauch des Virtex5-Raycasters. Aus dem Synthesereport des Xilinx ISE.

den Platzbedarf der drei selbst entwickelten Module verschaffen zu können, sind in Tabelle 6.7 die verbrauchten Ressourcen der drei Module gelistet.

	Slices (5197)	Slice Reg (9787)	LUTs (10736)	DSP48E (35)
RayGen	238 / 5 %	496 / 5 %	481 / 4 %	3 / 9 %
Intersector	1107 / 21 %	3049 / 31 %	3187 / 30 %	20 / 57 %
PCalc	2909 / 56 %	6000 / 61 %	6630 / 62 %	12 / 34 %

Tabelle 6.7.: Ressourcenverbrauch der einzelnen Module. Aus dem Synthesereport des Xilinx ISE. In R runden Klammern hinter dem Ressourcentyp findet sich die Gesamtzahl, die das Projekt auf dem FPGA belegt. Pro Modul ist für jeden Ressourcentyp die absolute Anzahl an benötigten Einheiten, sowie sein prozentualer Anteil am Gesamtverbrauch des Moduls gezeigt.

Der *RayGen* braucht erwartungsgemäß sehr wenig Platz und benötigt durch seinen einzigen Multiplizierer nur 3 % der DSP48E. Der *Intersector* belegt trotz seiner Pipelinearchitektur und den damit verbundenen temporären Registern und Zusatzlogik nur ein Drittel der gesamten Slice Regs und LUTs. Wegen seiner fünf internen Multiplizierer sind ihm allerdings über die Hälfte der DSP48E für das Gesamtmodul zugeteilt. Der *PCalc* stellt das größte Modul dar, was auch durch seinen Dividierer bedingt ist, der für sich betrachtet schon 3147 Slice Regs und 1246 LUTs in Anspruch nimmt. Der Dividierer bildet allein die Hälfte des Platzbedarfs für den *PCalc*.

Neben dem Platzbedarf ist auch noch das Timingverhalten der drei Module von Interesse, welches in den kommentierten Abbildungen 6.15, 6.16 und 6.17 veranschaulicht wird.

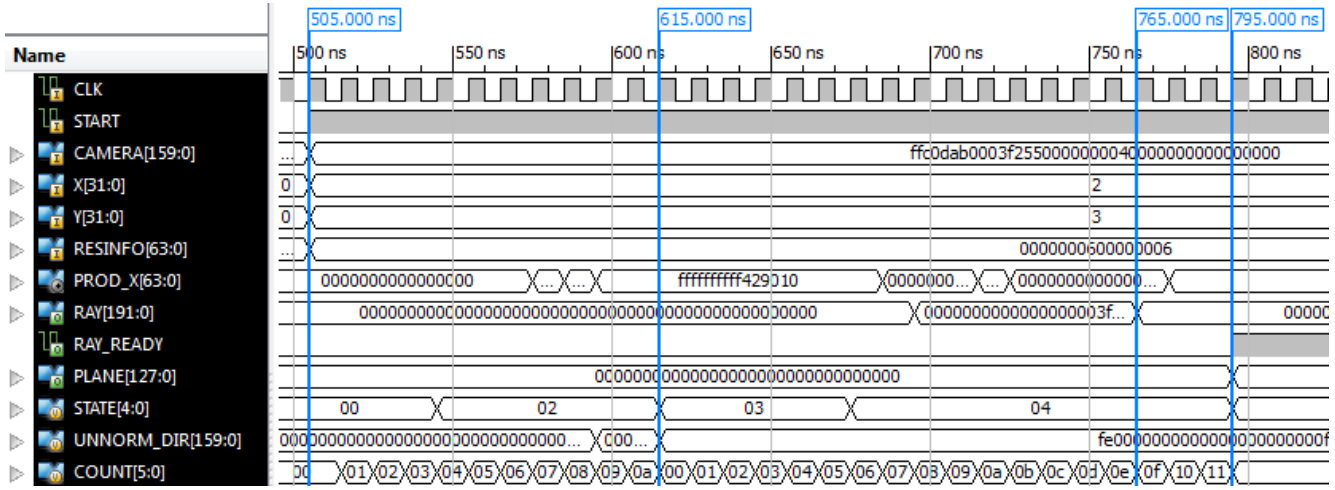

Abbildung 6.15.: Timingdiagramm des RayGen-Moduls für den Virtex5.

- **505ns**: Das Startsignal START wird von außen gesetzt, wenn alle nötigen Daten an den Eingängen CAMERA, X, Y und RESINFO anliegen. Das Modul beginnt mit der Berechnung, was am internen Registern COUNT erkannt werden kann. Nach sieben Takten liegt das erste Ergebnis am Ausgang PROD_X des internen Multiplizierers an.
- **615ns**: Als erstes Zwischenergebnis ist die Richtung des Sichtstrahls fertig berechnet und ihr Ergebnis im Register UNIFORM_DIR gespeichert.
- **765ns**: Der Sichtstrahl ist komplett und liegt am Ausgang RAY an. Die Arbeit des Moduls ist noch nicht beendet, da die Ebene im Strahlenursprung noch nicht vorhanden ist.
- **795ns**: Die Ebenen-Berechnung ist beendet und das Ergebnis liegt an PLANE an. Dies wird nach außen durch RAY_READY signalisiert.

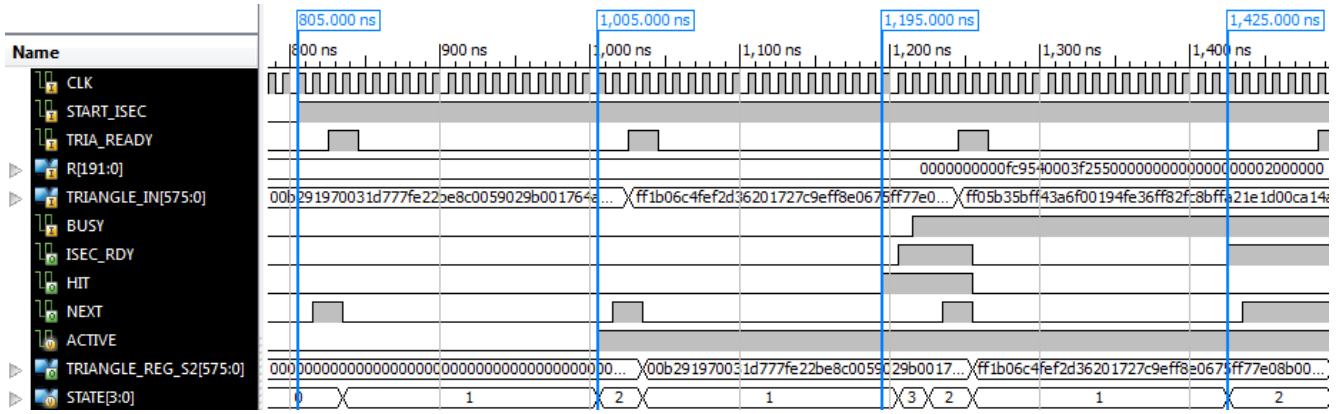

Abbildung 6.16.: Timingdiagramm des Intersector-Moduls für den Virtex5.

- **805ns**: Nachdem der korrekte Sichtstrahl am Eingang R angelegt ist, wird das Modul durch START_ISEC gestartet. Es fordert einen Takt später durch NEXT ein Dreieck an. Im darauffolgenden Takt wird dem Modul durch TRIA_READY signalisiert, dass entsprechende Daten am Eingang TRIANGLE_IN anliegen. Diese übernimmt das Modul in interne Register und wechselt in einen neuen STATE.
- **1005ns**: Durch ACTIVE zeigt das Modul an, dass seine Pipeline gefüllt ist. Es fordert im darauffolgenden Takt mittels NEXT ein weiteres Dreieck an.
- **1195ns**: Es wird festgestellt, dass der Strahl das Dreieck schneidet, weshalb HIT gesetzt wird. Einen Takt später wird durch ISEC_READY angezeigt, dass die Berechnung für das aktuelle Dreieck beendet ist, welches am Ausgang TRIANGLE_REG_S2 anliegt. Durch BUSY wird dem Modul mitgeteilt, dass der Schnitt registriert wurde und es mit der Arbeit fortfahren kann. Daraufhin werden HIT und ISEC_READY zurück genommen und durch NEXT ein neues Dreieck angefordert.
- **1425ns**: Für das nächste Dreieck wird kein Schnitt gefunden, was durch die Kombination ISEC_READY && !HIT signalisiert wird. Es kann mit dem nächsten Dreieck fortgefahren werden.

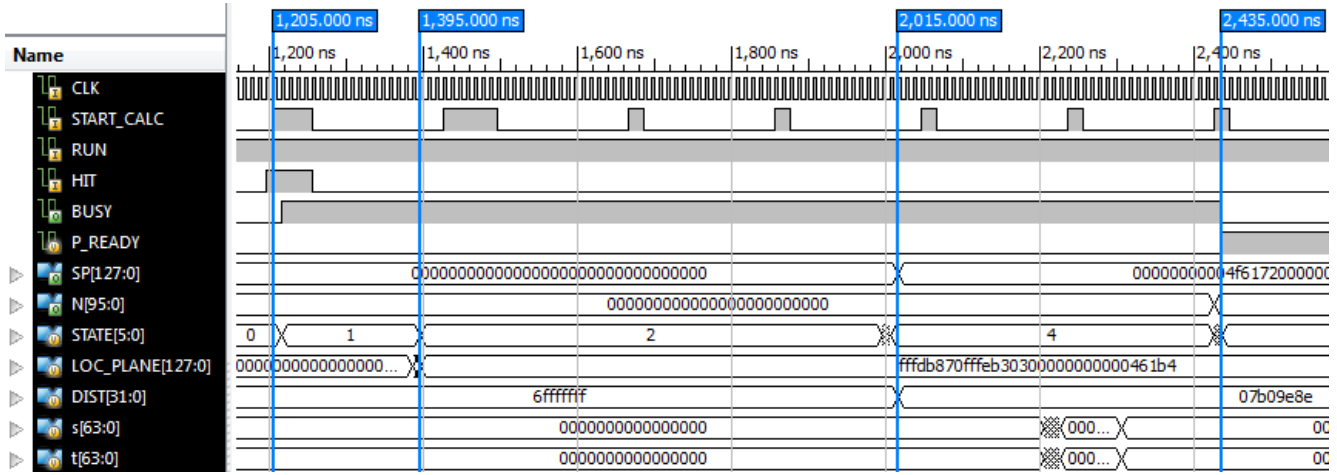

Abbildung 6.17.: Timingdiagramm des PCalc-Moduls für den Virtex5.

- **1205ns**: Durch die Kombination START_CALC && RUN && HIT wird der PCalc gestartet und übernimmt die Daten, die an den Eingängen R und TRIA_IN anliegen, in interne Register. Mit RUN signalisiert das Modul, dass das Übernehmen erfolgt ist.
- **1395ns**: Die Ebene, auf der das Dreieck liegt, ist berechnet und im Register LOC_PLANE gespeichert. Der STATE wechselt nun zur eigentlichen Schnittpunktberechnung.
- **2015ns**: Der Schnittpunkt und seine Distanz sind berechnet. Da er der erste Schnittpunkt für das Pixel ist, wird er als neuer nächstliegender Punkt in das Register SP übernommen. Sein Abstand zum Augpunkt wird in DIST gespeichert. Im nun folgenden STATE wird die Normale interpoliert.
- **2435ns**: Die Normale wurde mit Hilfe der Parameter s und t interpoliert und in Register N gespeichert. Der Abschluss der Berechnung wird durch P_READY && !BUSY angezeigt. Die positiven Flanken von START_CALC während der Ausführung signalisieren jeweils, dass der Intersector einen weiteren Schnitttest mit negativem Ergebnis beendet hat.

6.4 Abschätzungen zu Leistung

Zum Abschluss soll die Raycasting-Leistung der FPGA-Realisierungen mit den GPU- und CPU-Varianten verglichen werden. Bei den FPGAs werden der Virtex2p und der Virtex5 betrachtet. Der Spartan3E wird auf Grund seiner geringen Größte nicht mit einbezogen.

Die Simulationsergebnisse für *Geoshpere* und *Cow* aus Abschnitt 6.3.1 bestätigen, dass der Virtex2p in der Lage ist, alle 19 Takte das Ergebnis eines Schnitttests zu liefern. Auch zeigen sie, dass die Zeit zur Berechnung durch weitere Schnitttests verdeckt werden kann. Pro Pixel fällt die Zeit an, die Speicherverbindung aufzubauen. Die Berechnung des Sichtstrahls kann dagegen vernachlässigt werden, da sie schneller erfolgt als das Initialisieren der DDR-Anbindung mit anschließendem Auslesen des ersten Dreiecks. Nach 25 Takten steht das erste Dreieck nach dem Start des Moduls zu Verfügung. Die Anzahl der Takte pro Pixel ergibt sich daher zu: $Anzahl_{Dreiecke} * Taktzahl_{Schnitttest} + 25$. Insgesamt lässt sich die Laufzeit der Schaltung für große, nicht mehr simulierbare Modelle, wie folgt abschätzen: $Bildhoehe * Bildbreite * (Anzahl_{Dreiecke} * 19 + 25)$. Mittels der Taktrate (100 MHz bei Virtex2p und Virtex5) lässt sich anschließend die zu erwartende Rechenzeit bestimmen.

Für das Modell *Squirrel* werden die Rendering-Geschwindigkeit von GPU und CPU gemessen: Analog zum implementierten Algorithmus auf dem FGPA werden die Hüllkugeln ausgeschaltet. Es muss also für jedes Pixel jedes Dreieck geprüft werden. Außerdem wird das Shading deaktiviert. Für die CPU werden Compileroptimierungen und OpenMP ausgeschaltet. Somit arbeitet nur einer der CPU-Kerne an der Berechnung. Das Rendering-Ergebnis von CPU/GPU zeigt Abb. 6.18. Die HD6970 benötigt zur Erzeugung des Bildes 0,86 Sekunden, der Q6600 953,45 Sekunden. Nach Einsetzen der 18694 Dreiecke und der Auflösung von 512x512 ergibt sich, dass die Berechnung auf dem Virtex2p 931,31 Sekunden dauern wird. Auf dem Virtex5 braucht der Intersector einen Takt mehr. Nimmt man für ihn an, dass die Herstellung einer Speicheranbindung ebenfalls 25 Takte benötigt, ist seine Berechnungszeit 980,32 Sekunden. Der Virtex2p ist schon bei seinem geringerem Takt schneller als ein einzelner CPU-Kern, der Virtex5 etwas langsamer.

Rechnet man die Leistung der FPGAs auf die 2,4 GHz-Taktrate der Prozessoren hoch, so ergibt sich, dass der Virtex2p das Bild in 34,49 Sekunden berechnet, während der Virtex5 36,31 Sekunden benötigen würde. Die Rechengeschwindigkeit pro Takt ist bei den FPGA-Implementierungen deutlich höher, als auf dem CPU-Kern. Die beiden FGPAs würden das Bild bei gleichem Takt in einem Dreißigstel der Zeit berechnen. Die Kernel auf der GPU dagegen arbeiten nochmals um den Faktor 40 schneller.

Der Virtex5 bzw. generell größere Plattformen bieten außerdem noch weitere Möglichkeiten der Geschwindigkeitssteigerung. Eine schnelle Speicheranbindung vorausgesetzt, kann der Schnitttestvorgang zum einen dadurch beschleunigt werden, dass mehr Multiplizierer im *Intersector* genutzt werden, Somit können weitere Berechnungen parallel ausführt werden. Zum anderen existiert die Variante zwei *Intersector*-Module zu instanziieren und ihnen abwechselnd Dreiecke zu liefern. *RayGen* und *PCalc* können in diesem Fall von beiden Modulen geteilt werden. Durch die Erhöhung der Anzahl der Multiplizierer im *Intersector* ließe sich dessen Geschwindigkeit um ca. 50 % erhöhen. Mehr ist nicht möglich, da einige Berechnungen vom Ergebnis anderer abhängig sind und somit erst gestartet werden können, wenn die Zwischenergebnisse vorliegen. In diesem Fall beschränkt die Latenzzeit der Multiplizierer die erreichbare Geschwindigkeit. Durch

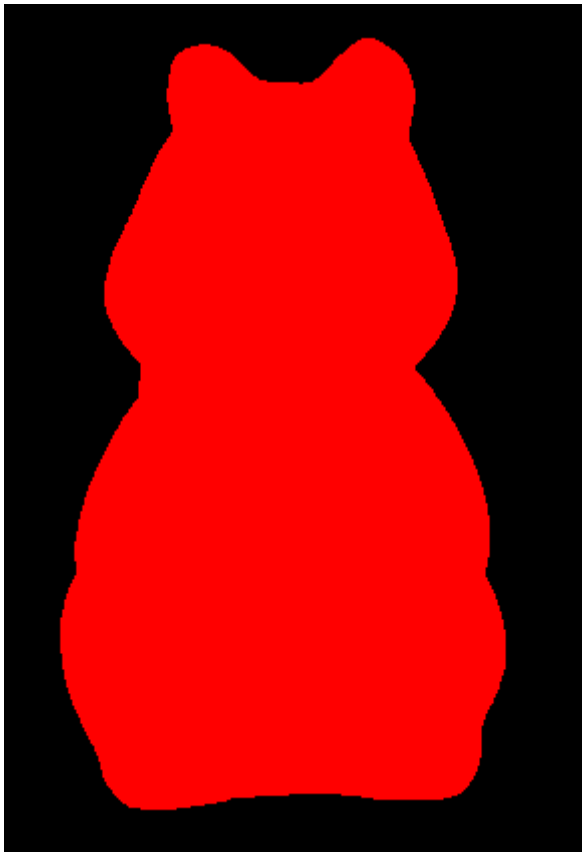

Abbildung 6.18.: Modell *Squirrel* ohne Shading

die Variante mit zwei *Intersectors* ließe sich die Ausführungszeit halbieren, wenn der Speicher schnell genug Dreiecke bereitstellen kann, um keine Wartezeiten zu verursachen.

7 Fazit und Ausblick

Die Untersuchungen dieser Arbeit zeigen, dass sich beim Raytracing die parallele Natur von Algorithmen in Konformer Geometrischer Algebra auf entsprechenden Plattformen ausnutzen lässt. Es ist bezüglich der Geschwindigkeit möglich, bestehende Lösungen in Linearer Algebra zu überholen. Durch den Einsatz OpenCL ist dies vor allem auf Grafikkarten zu erreichen. Mit CluCalc und Gaalop stehen zudem für die Entwicklung von Algorithmen in GA zwei sehr nützliche Tools zur Verfügung, welche eine Umsetzung erleichtern. Weitere Unterstützung bieten die Tools des AMD APP SDKs. Es sollte in Zukunft auch in weiteren Anwendungsfeldern geprüft werden, inwiefern die Kombination GA-Gaalop-OpenCL in der Lage ist mit Standardverfahren zu konkurrieren, da es durchaus realistisch erscheint, dass die Ergebnisse dieser Arbeit auf weitere Anwendungsfelder übertragen lassen.

Allerdings verdeutlichen die Ausführungen der vorliegenden Arbeit auch, dass eine vollautomatische Optimierung mittels Gaalop schwer erreichbar sein wird, zumindest wenn die Leistung so weit wie möglich maximiert werden soll. Für OpenCL-Kernel sind manche Verbesserungen erst durch theoretische Überlegungen zu finden. Beispiele hierfür wurden im Verlauf des Textes beschrieben. Andere Optimierungen wiederum lassen sich sogar nur Ausprobieren und Messen entdecken. Auch gilt es bei den Grafikkarten auf Hersteller-spezifische Eigenschaften zu achten, wohingegen die Prozessoren allgemeiner behandelt werden können. Besonders wichtig ist die Erkenntnis, dass die naheliegende Aussage: „Die Berechnung mit den wenigstens Operationen besitzt die kürzeste Rechenzeit.“ bzgl. der OpenCL-Kernel in vielen Fällen nicht zutrifft. Viel wichtiger ist das Vermeiden bzw. das Reduzieren von Verzweigungen im Code.

Für die FPGAs erscheint ein vollautomatisch generierter Code ebenfalls schwer erreichbar zu sein. Selbst wenn man die Verstehbarkeit und Erweiterbarkeit des erzeugten Codes ausklammert, bleiben noch weitere Probleme bestehen. Für große Projekte ist vor allem die Frage zu klären, wie der Compiler entscheiden soll, an welchen Stellen wieviele Ressourcen für parallele Berechnungen bereitgestellt werden und an welchen stellen Pipelining möglich ist, um den vorhanden Platz bestmöglich zu nutzen. Die volle Ausnutzung der Parallelität wird nur in wenigen Fällen zu erreichen sein, so dass stets ein Kompromiss zwischen Platzbedarf und Geschwindigkeit gefunden werden muss. Dennoch sind auch die FPGAs durchaus für die performante Realisierung von GA-Algorithmen geeignet, wie der entworfene Raycaster zeigt.

Zum Ende der Arbeit hin ergab ein Durchlauf der Benchmark-Suite der Dreiecks-Raytracer, dass die Leistungsunterschiede zwischen den beiden Algebren auf NVidia-Grafikkarten generell viel weiter schwanken, als auf den AMDs. Teilweise ist die LA schneller als die GA, auf anderen Modellen ist es umgekehrt. Ein extremer Unterschied besteht in einigen getesteten Szenen auf der GTX580. In einem Fall ist die GA um den Faktor 3 schneller als LA. Allerdings stürzen auch einige Testszenen ab. Da diese Ergebnisse dennoch sehr interessant sind, sollen sie im Rahmen der Masterarbeit von Patrick Charrier zusammen mit einer neuen Version des Gaalop-Compilers genauer untersucht werden. Aber auch die neue Generation von AMD Karten sollte betrachtet werden, da die HD7000er Serie beim GPU-Computing nicht mehr mit VLIW arbeiten wird, sondern wie NVidia bei seinen Karten auf eine dynamische Zuteilung der verfügbaren

Ressourcen setzt. Auch sind für neuere Revisionen von OpenCL wichtige Punkte wie double-Genauigkeit und Unterstützung von Rekursion in Planung, was die Programmierung der GPUs noch flexibler gestalten soll.

A Kompakter Raytracer in CluCalc

Nachfolgendes CluCalc-Beispiel verfolgt zwei Ziele: Zum einen soll es demonstrieren, dass es mit KGA möglich ist mit sehr wenigen Codezeilen die Kernfunktionalitäten eines Raytracers zu formulieren. So sind in den rund 35 Zeilen Code sowohl die Sichtstrahlenerzeugung, der Schnitttest mit der Bounding-Sphere, der Schnitttest und die Schnittpunktberechnung mit dem Dreieck, sowie die Abstandsberechnung zum Ursprung enthalten. Was der Code nicht zeigt sind die Normaleninterpolation und die Beleuchtungsrechnung, da diese analog zur LA verlaufen. Ihr Umfang ist auch jeweils fast so groß wie die restlichen Schritte zusammen. Zum anderen sollen einige Konzepte und Bedeutungen von Operationen anhand eines praktischen Beispiels gezeigt werden, ohne dass die theoretischen Grundlagen dafür erläutert werden müssen.

```
/*
Dieser Teil des Algorithmus würde in einer geschachtelte
for-Schleife über die einzelnen Pixel des Bildes iterieren
*/

// Die 3D Richtung des Sichtstrahles durch die ViewPlane
// wird berechnet. Dies geschieht in LA in gleicher Weise
rayDirection = (x * pixelWidth - 0.5 * cameraFOVwidth) * e1 +
  (y * pixelHeight - 0.5 * cameraFOVheight) * e2 + e3;

// Mittels äußerem Produkt aus Richtung und Kameraposition
// sowie Dualisierung wird der konforme Sichtstrahl erstellt
ray = *(campos^rayDirection^einf);

// Die Distanz zum Ursprung über inneres Produkt
// berechnen
dist = -campos.rayDirection / (campos.einf);

// Ebene durch den Kamerapunkt, mit Strahlenrichtung
// als Normale
raypl = rayDirection + dist * einf;

// Auf Schnitt mit BoundingSphere prüfen
// Punktepaar durch Äußeres Produkt
PP = *(ray^bound);

// Prüfung des Schnittindikators
// Zeigt an ob Boundingsphere
// verfehlt wurde und ermöglicht
// early exit
inprod = PP.PP;
if(inprod <0)
break;
```

```
35 // Auf Schnitt mit dem Dreicks-Netz testen
   /*
37 Test basiert darauf, dass bei gleicher Orierentierung
   der Kanten des Dreiecks (edgeX) der indiX angibt, auf
39 welcher Seite der Kante der Schnitt mit der
   Dreiecksebene liegt. Liegt er bei allen auf der
41 rechten Seite, liegt er im Dreieck.
   */
43
   // Seiten des Dreiecks als konforme Geraden
45 // mittels Eckpunkten und einf
   edge1 = p0^p1^einf;
47 edge2 = p1^p2^einf;
   edge3 = p2^p0^einf;
49
   indi1 = e0.(ray.edge1);
51 indi2 = e0.(ray.edge2);
   indi3 = e0.(ray.edge3);
53
   side = -1;
55
   if(indi1 > 0 && indi2 > 0 && indi3 > 0)
57 {
     side = 1;
59 }

61 // Weiterer early exit
   if(side == -1)
63 {
     break;
65 }

67 // Wenn Test positiv, dann Schnittpunkt berechnen
   // Schnittpunkt als so genannten FlatPoint durch
69 // Äußeres Produkt Dreick^Strahl berechnen
   // Ebene in der Dreieck liegt
71 tripl = *(p0^p1^p2^einf);
   fp = *(tripl^ray);
73
   // Umwandlung eines FlatPoints in einen
75 // normalen, konformen Punkt
   A = e0.fp;
77 clen = -A.einf;
   A = A / clen;
79 sp = VecN3(A(2), A(3), A(4));

81 // Prüfen ob Schnittpunkt zulässig und ggf verwerfen
   // Wenn der Punkt hinter der Kameraebene liegt ist
83 // er ungültig
```

```
dist2 = raypl.sp;
if(dist2 < 0.0001)
{
  break;
}

// Abstand des Punktes zum Augpunkt ebenfalls
// darauf prüfen ob, Punkt der näheste ist

// Normale in Schnittpunkt über baryzentische
// Koordinaten interpolieren
// Analog zu 3D Verfahren
```

Listing A.1: Raytracer Kernfunktionalität in CluCalc

B RayGen-Modul in Verilog

Die frühe, für diese Arbeit getestete Version des Gaalop-Verilog-Compilers unterstützt weder Schleifen noch Bedingungen im Code, so dass es nicht möglich ist, den vollständigen Algorithmus zu testen. Somit wird der Vergleich auf das *RayGen*-Modul beschränkt. Seine Realisierung in C++ kann in Abschnitt 3.4.1 gefunden werden, während das folgende Listing B.1 den selbst entwickelten Verilog-Code zeigt, der auf Basis des optimierten C++-Codes steht. Die Verilog-Umsetzung erfolgt nahe am C++-Code der Strahlenerzeugung. Sie führt jeweils zwei der Multiplikationen parallel aus. Zuerst wird die Richtung des Strahles berechnet und anschließend der Sichtstrahl konstruiert.

```
1  `timescale 1ns / 1ps

3  `define PRE 25
   `define SCALE 33554432
5
   module PipeRayGen(
7    CAMERA, // FOV-Winkel und Position der Kamera
     X, // X-Wert des aktuell betrachteten Pixels
9    Y, // Y-Wert des aktuell betrachteten Pixels
     RESINFO, // Maximale Auflösung des Bildes
11   START, // Enable-Signal für das Modul
     RESET,
13   CLK,
     RAY, // Die sechs Vektoreinträge für den Strahl
15   RAY_READY, // Berechnung beendet
     PLANE // Ebene senkrecht auf Strahlenrechnung
17     );

19 input [159:0] CAMERA;
   input [31:0] X;
21 input [31:0] Y;
   input [63:0] RESINFO;
23 input START;
   input RESET;
25 input CLK;
   output [191:0] RAY;
27 output RAY_READY;
   output [127:0] PLANE;
29
   // Inputs
31 wire [159:0] CAMERA;
   wire [31:0] X;
33 wire [31:0] Y;
   wire [63:0] RESINFO;
```

```
35
// Output
37 reg [191:0] RAY;
reg RAY_READY;
39
// Internal Registers
41 reg [4:0] STATE;
reg [159:0] UNNORM_DIR; // 3D-Richtung des Sichtstrahles
43 reg BEGIN_MULT;
reg [5:0] COUNT; // Takte für Rechenoperationen zählen
45 reg [127:0] PLANE;

47 // Eingabewerte für den Multiplizierer
reg [31:0] INPUT1_MULT1;
49 reg [31:0] INPUT2_MULT1;

51 // Internal wires
// Werte für die Multiplikationen
53 wire [63:0] PROD_X;

55 // Internal Components
// Mit Core-Gen erstellter Multiplizierer
57 v3_multiplier mult1(BEGIN_MULT, CLK, INPUT1_MULT1, INPUT2_MULT1, PROD_X);

59 // Zustände des Automaten definieren
`define WAIT_FOR_CAMVALS  0   // Warte auf Kamera und Auflösungsdaten
61 `define STAGE_2 2 // Berechne cam.s5/6 * (...)
`define PREPARE_MULT2 3 // Werte in Register der Multiplizierer schreiben
63 `define DO_MULT2 4 // Berechne Werte
`define READ_RAY 5 // Strahl aus bisherigen Zwischenwerten zusammenbauen
65
reg [63:0] TMP1; reg [63:0] TMP2; reg [63:0] TMP3;
67
always @(posedge CLK or posedge RESET) begin
69 // asyncroner Reset
if(RESET)
71   begin
       RAY <= 0;
73     RAY_READY <= 0;
       STATE <= `WAIT_FOR_CAMVALS;
75     UNNORM_DIR <= 0;
       INPUT1_MULT1 <= 2;     INPUT2_MULT1 <= 2;
77     TMP1 <= 0; TMP2 <= 0; TMP3 <= 0;
       BEGIN_MULT <= 0;
79     PLANE <= 0;
       COUNT <= 0;
81   end

83 // synchroner Zustandsautomat
else
```

```
85  begin
      case (STATE)
87      `WAIT_FOR_CAMVALS:
        begin
89        if(START) begin
            case (COUNT)
91          0: begin
              // x * pixelWidth
93            BEGIN_MULT <= 1;
              INPUT1_MULT1 <= X;
95            INPUT2_MULT1 <= CAMERA [127:96];
              COUNT <= COUNT + 1;
97          end

99          1: begin
              // 0.5 * imageWidth * pixelWidth
101           INPUT1_MULT1 <= CAMERA [127:96];
              INPUT2_MULT1 <= RESINFO [31:0] >> 1;
103           COUNT <= COUNT + 1;
            end
105
            2: begin
107           // y * pixelHeight
              INPUT1_MULT1 <= Y;
109           INPUT2_MULT1 <= CAMERA [159:128];
              COUNT <= COUNT + 1;
111         end

113         3: begin
              // 0.5 * imageHeight * pixelHeight
115           INPUT1_MULT1 <= CAMERA [159:128];
              INPUT2_MULT1 <= RESINFO [63:32] >> 1;
117           STATE <= `STAGE_2;
              COUNT <= COUNT + 1;
119         end

121         default: begin
              ;
123         end

125         endcase
          end
127     end

129
        `STAGE_2:
131     begin
          if(COUNT == 5) begin
133         TMP1 <= PROD_X[63:0];
            COUNT <= COUNT + 1;
```

```
135         end

137         else if(COUNT == 6) begin
              // e1 = x * pixelWidth - 0.5 * cameraFOVwidth
139           UNNORM_DIR[63:0] <= TMP1 - PROD_X[63:0];
              COUNT <= COUNT + 1;
141         end

143         else if(COUNT == 7) begin
              TMP1 <= PROD_X[63:0];
145           COUNT <= COUNT + 1;
            end
147
            else if(COUNT == 8) begin
149           // e2 = y * pixelHeight - 0.5 * cameraFOVheight
              UNNORM_DIR[127:64] <= TMP1 - PROD_X[63:0];
151           // e3 = -1
              UNNORM_DIR[159:128] <= -`SCALE;
153           COUNT <= 0;
              STATE <= `PREPARE_MULT2;
155         end

157         else
              COUNT <= COUNT + 1;
159         end

161       `PREPARE_MULT2: begin
            case(COUNT)
163         // p = campos, d = raydirection
            0: begin
165           // p.s2 * d.y;
              INPUT1_MULT1 <= CAMERA[95:64];
167           INPUT2_MULT1 <= UNNORM_DIR[95:64];
              COUNT <= COUNT + 1;
169         end

171         1: begin
              // p.s1 * d.z
173           INPUT1_MULT1 <= CAMERA[63:32];
              INPUT2_MULT1 <= UNNORM_DIR[159:128];
175           COUNT <= COUNT + 1;
            end
177
            2: begin
179           // p.s0 * d.z
              INPUT1_MULT1 <= CAMERA[31:0];
181           INPUT2_MULT1 <= UNNORM_DIR[159:128];
              COUNT <= COUNT + 1;
183         end
```

```
185       3: begin
            // p.s2 * d.x
187         INPUT1_MULT1 <= CAMERA[95:64];
            INPUT2_MULT1 <= UNNORM_DIR[31:0];
189         COUNT <= COUNT + 1;
          end
191
          4: begin
193         // p.s1 * d.x
            INPUT1_MULT1 <= CAMERA[63:32];
195         INPUT2_MULT1 <= UNNORM_DIR[31:0];
            COUNT <= COUNT + 1;
197       end

199       5: begin
            // p.s0 * d.y
201         INPUT1_MULT1 <= CAMERA[31:0];
            INPUT2_MULT1 <= UNNORM_DIR[95:64];
203
            // Aus den bisherigen Werten den Strahl zusammenbauen
205         RAY[31:0] <= - UNNORM_DIR[159:128]; // r.s0 = -d.z
            RAY[63:32] <= UNNORM_DIR[95:64]; // r.s1 = d.y
207         RAY[127:96] <= - UNNORM_DIR[31:0]; // r.s3 = -d.x
            TMP1 <= PROD_X;
209
            COUNT <= COUNT + 1;
211         STATE <= `DO_MULT2;
          end
213
        endcase
215   end


217
    `DO_MULT2: begin
219   case (COUNT)

221       6: begin
            // p.s2 * d.y - p.s1 * d.z
223         TMP1 <= TMP1 - PROD_X;

225         // Plane vorbereiten Teil 1
            INPUT1_MULT1 <= CAMERA[31:0];
227         INPUT2_MULT1 <= UNNORM_DIR[31:0];

229         COUNT <= COUNT + 1;
          end
231
          7: begin
233         TMP2 <= PROD_X;
```

```
235         // Plane vorbereiten Teil 2
            INPUT1_MULT1 <= CAMERA[63:32];
237         INPUT2_MULT1 <= UNNORM_DIR[127:64];

239         COUNT <= COUNT + 1;
          end
241
          8: begin
243         // r.s4 = p.s0 * d.z - p.s2 * d.x
            TMP2 <= TMP2 - PROD_X;
245
            // Plane vorbereiten Teil 3
247         INPUT1_MULT1 <= CAMERA[95:64];
            INPUT2_MULT1 <= UNNORM_DIR[159:128];
249
            COUNT <= COUNT + 1;
251       end

253       9: begin
            TMP3 <= PROD_X;
255         COUNT <= COUNT + 1;
          end
257
          10: begin
259         TMP3 <= TMP3 - PROD_X;
            COUNT <= COUNT + 1;
261       end

263       11: begin
            RAY[95:64] <= TMP1[31:0];
265         RAY[159:128] <= TMP2[31:0];
            RAY[191:160] <= TMP3[31:0];
267
            // Erster Planewert ist fertig
269         TMP1[63:0] <= PROD_X[63:0];
            COUNT <= COUNT + 1;
271       end

273       12: begin
            // Zweiter Planewert ist fertig
275         TMP1[63:0] <= TMP1[63:0] + PROD_X[63:0];
            COUNT <= COUNT + 1;
277       end

279       13: begin
            // Dritter Planewert ist fertig
281         TMP1[63:0] <= TMP1[63:0] + PROD_X[63:0];
            COUNT <= COUNT + 1;
283       end
```

```
285         14: begin
              // Die vier Werte für die Plane eintragen
287           PLANE[31:0] <= UNNORM_DIR[31:0];
              PLANE[63:32] <= UNNORM_DIR[95:64];
289           PLANE[95:64] <= UNNORM_DIR[159:128];
              PLANE[127:96] <= TMP1[31:0];
291
              // Jetzt kann im nächsten Schritt der Ray gelesen werden
293           RAY_READY <= 1;
              STATE <= `READ_RAY;
295           COUNT <= 0;
            end
297
            default:
299           COUNT <= COUNT + 1;
            endcase
301       end

303       // Für Handshakeprotoll
          `READ_RAY:
305       begin
            if(START == 0)
307         begin
              RAY_READY <= 0;
309           STATE <= `WAIT_FOR_CAMVALS;
            end
311
            else begin
313           RAY <= RAY;
              RAY_READY <= 1;
315         end
          end
317


319       default: ;
        endcase
321   end
end
323


325 endmodule
```

Listing B.1: Raytracer Kernfunktionalität in CluCalc

Der durch Gaalop generierte Code unterscheidet sich deutlich von Listing B.1. Größter Unterschied ist, dass kein eigentliches Multiplizierer-Modul existiert, sondern sämtliche Berechnungen direkt abgebildet werden. Hierdurch wird der Code deutlich umfangreicher und sehr schwer lesbar. Auch die automatisch erzeugten Namen für Register und Signale sind schwer verständlich. Eine Änderung am Verilog-Code bzw. Anpassungen sind somit kaum möglich. Ein Synthesedurchlauf ergibt, dass in etwa gleich viele Ressourcen belegt werden, wie beim selbst

geschriebenen Code. Allerdings erscheint es aufgrund der fehlenden Unterstützung bestimmter Kontrollanweisungen nicht sinnvoll, den kompletten Raytracer mittels generiertem Verilog-Code zu beschreiben. Die fehlenden Anweisungen nachträglich zu integrieren ist wegen der Unübersichtlichkeit des erzeugten Codes kaum mit angemessenem Aufwand realisierbar.

C Abkürzungsverzeichnis

A-Tracer Amsterdamer Raytracer
BSphere Bounding-Sphere
KGA Konforme Geometrische Algebra
GA Geometrische Algebra
LA Lineare Algebra
GDV Graphische Datenverarbeitung
OP Äußere Produkt
IP Innere Produkt
OPNS Outer Product Null Space
IPNS Inner Product Null Space
SSE2 Streaming SIMD Extensions

Literaturverzeichnis

[Abl11] Ablamowicz, Rafal: *CLIFFORD - A Maple Package for Clifford Algebra Computations*. http://math.tntech.edu/rafal/. Version: Juni 2011

[Adv10] Advanced Micro Devices, Inc.: *ATI Stream Computing, Rev. 1.03*, June 2010. http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf

[Adv11] Advanced Micro Devices, Inc.: *AMD Accelerated Parallel Processing OpenCL Programming Guide*, August 2011 2011. http://developer.amd.com/sdks/amdappsdk/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf

[Bad90] Badouel, Didier: Graphics gems. Version: 1990. http://dl.acm.org/citation.cfm?id=90767.90867. San Diego, CA, USA : Academic Press Professional, Inc., 1990. – ISBN 0–12–286169–5, Kapitel An efficient ray-polygon intersection, 390–393

[BC] Buck, David K. ; Collins, Aaron A.: *POV-Ray - The Persistence of Vision Raytracer*. http://www.povray.org/

[BKHS09] Brendel, Elmar ; Kalbe, Thomas ; Hildenbr, Dietmar ; Schäfer, Michael: *Simulation of Elastic Rods using conformal Geometric Algebra*. \url{http://www.gris.informatik.tu-darmstadt.de/~dhilden/FCS2008_E.Brendel.pdf}. Version: 2009

[Bli77] Blinn, James F.: Models of light reflection for computer synthesized pictures. In: *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM Press, 1977, S. 192–198

[CJP07] Chapman, Barbara ; Jost, Gabriele ; Pas, Ruud van d.: *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. – ISBN 0262533022, 9780262533027

[Deu10] Deul, Crispin: *Das Raytracen von Punktwolken mit Hilfe der Geometrischen Algebra*. Deutschland, Technische Universität Darmstadt, Diplomarbeit, 2010

[DFM07a] Dorst, Leo ; Fontijne, Daniel ; Mann, Stephen: *Conformal Model Raytracer*. http://www.geometricalgebra.net/raytracer.html, 2007. – zuletzt abgerufen am 18.10.2010

[DFM07b] Dorst, Leo ; Fontijne, Daniel ; Mann, Stephen: *Geometric Algebra for Computer Science: An Object-Oriented Approach to Geometry (The Morgan Kaufmann Series in Computer Graphics)*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2007. – ISBN 0123694655

[Ebe01] Eberly, David H.: *3D game engine design - a practical approach to real-time computer graphics*. Morgan Kaufmann, 2001. – I–XXVIII, 1–560 S. – ISBN 978–1–55860–

593–0

[FBD01] FONTIJNE, Daniel ; BOUMA, Tim ; DORST, Leo: Gaigen: a Geometric Algebra Implementation Generator. In: *FD03] FONTIJNE D., DORST L.: MODELING 3D EUCLIDEAN GEOMETRY. IEEE COMPUTER GRAPHICS AND APPLICATIONS 23, 2 (MARCHAPRIL 2003*, 2001, S. 68–78

[FD03] FONTIJNE, Daniel ; DORST, Leo: Modeling 3D Euclidean Geometry. In: *IEEE Comput. Graph. Appl.* 23 (2003), March, 68–78. http://dx.doi.org/http://dx.doi.org/10.1109/MCG.2003.1185582. – DOI http://dx.doi.org/10.1109/MCG.2003.1185582. – ISSN 0272–1716

[Fen03] FENDER, Joshua: A high-speed ray tracing engine built on a field-programmable system. In: *Proc. Int. conf. on Field-Programmable Technology, IEEE*, 2003, S. 188–195

[FGS+09] FRANCHINI, Silvia ; GENTILE, Antonio ; SORBELLO, Filippo ; VASSALLO, Giorgio ; VITABILE, Salvatore: An embedded, FPGA-based computer graphics coprocessor with native geometric algebra support. In: *Integr. VLSI J.* 42 (2009), June, 346–355. http://dx.doi.org/10.1016/j.vlsi.2008.09.010. – DOI 10.1016/j.vlsi.2008.09.010. – ISSN 0167–9260

[Fon06] FONTIJNE, Daniel: Gaigen 2:: a geometric algebra implementation generator. In: *Proceedings of the 5th international conference on Generative programming and component engineering*. New York, NY, USA : ACM, 2006 (GPCE '06). – ISBN 1–59593–237–2, 141–150

[Fon07] FONTIJNE, Daniel: *Efficient Implementation of Geometric Algebra*, University of Amsterdam, Diss., 2007

[GÖ99] GÄRTNER, Bernd: Fast and Robust Smallest Enclosing Balls. In: *ESA '99: Proceedings of the 7th Annual European Symposium on Algorithms*. London, UK : Springer-Verlag, 1999. – ISBN 3–540–66251–0, S. 325–338

[GÖ06] GÄRTNER, Bernd: *Smallest Enclosing Balls of Points - Fast and Robust in C++*. http://www.inf.ethz.ch/personal/gaertner/miniball.html, 2006. – zuletzt abgerufen am 18.10.2010

[Gaw01] GAWTRY, Erik H.: *Fixed Point Math Class*. http://www.koders.com/cpp/fidC41DBA66A5CACED1861E4DFDB7909A8AADBA8E46.aspx?s=mdef%3Agame, 2001. – zuletzt abgerufen am 24.08.2011

[Geb03] GEBKEN, C.: *Implementierung eines Koprozessors für geometrische Algebra auf einem FPGA*. Kiel, 2003 http://books.google.com/books?id=XIO4PgAACAAJ

[GHJV94] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John M.: *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. Addison-Wesley Professional, 1994 \url{http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0201633612}. – ISBN 0201633612

[GSS+05] GENTILE, Antonio ; SEGRETO, Salvatore ; SORBELLO, Filippo ; VASSALLO, Giorgio ; VITABILE, Salvatore ; VULLO, Vincenzo: CliffoSor: A Parallel Embedded Architecture for Geometric Algebra and Computer Graphics. In: *CAMP*, IEEE Computer Society,

2005. – ISBN 0–7695–2255–6, 90-95

[HBCZE05] Hildenbrand, Dietmar ; Bayro-Corrochano, Eduardo ; Zamora-Esquivel, Julio: Advanced Geometric Approach for Graphics and Visual Guided Robot Object Manipulation. In: *ICRA*, 2005, S. 4727–4732

[HE74] Herbison-Evans, Don: Animated Cartoons by Computer Using Ellipsoids. Version: 1974. http://www-staff.it.uts.edu.au/~don/pubs/cartoon.html. 1974 (94). – Forschungsbericht

[Hil06] Hildenbrand, Dietmar: *Geometric Computing in Computer Graphics and Robotics using Conformal Geometric Algebra*, Darmstadt University of Technology, Diss., 2006

[Hil11] Hildenbrand, Dietmar: *Gaalop - Geometic Algebra Algorithms Optimizer*. http://www.gaalop.de/. Version: Juni 2011

[HLSK08] Hildenbrand, Dietmar ; Lange, Holger ; Stock, Florian ; Koch, Andreas: Efficient Inverse Kinematics Algorithm Based on Conformal Geometric Algebra - Using Reconfigurable Hardware. In: *GRAPP*, 2008, S. 300–307

[HS87] Hestenes, David ; Sobcyk, Garret: *Clifford Algebra to Geometric Calculus: A Unified Language for Mathematics and Physics (Fundamental Theories of Physics)*. Kluwer Academic Publishers, 1987

[Khr11] Khronos OpenCL Working Group: *The OpenCL Specification, version 1.1*, 1 January 2011. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[LB06] Loop, Charles ; Blinn, Jim: Real-time GPU rendering of piecewise algebraic surfaces. In: *ACM Trans. Graph.* 25 (2006), Nr. 3, S. 664–670. – ISSN 0730–0301

[LWC+02] Luebke, David ; Watson, Benjamin ; Cohen, Jonathan D. ; Reddy, Martin ; Varshney, Amitabh: *Level of Detail for 3D Graphics*. New York, NY, USA : Elsevier Science Inc., 2002. – ISBN 1558608389

[Map11] Maplesoft: *Maple 15*. http://www.maplesoft.com/products/maple/. Version: Juni 2011

[Möl08] Möller, Tomas: *Real-Time Rendering, Third Edition*. 3. AK Peters, 2008 http://www.worldcat.org/isbn/1568814240. – ISBN 1568814240

[MSM04] Mattson, Timothy ; Sanders, Beverly ; Massingill, Berna: *Patterns for parallel programming*. First. Addison-Wesley Professional, 2004. – ISBN 0321228111

[MT97] Möller, Tomas ; Trumbore, Ben: Fast, Minimum Storage Ray-Triangle Intersection. In: *journal of graphics, gpu, and game tools* 2 (1997), Nr. 1, S. 21–28

[MV99] Meyberg, Kurt ; Vachenauer, Peter: *Höhere Mathematik 1*. Springer, 1999

[MW06] Mishra, Biswajit ; Wilson, Peter: Color Edge Detection Hardware based on Geometric Algebra. In: *The 2nd Institution of Engineering and Technology Multimedia Conference 2006*, 2006

[Nok11] Nokia: *Qt library 4.7*. http://developer.qt.nokia.com/doc/qt-4.7/, 2011. – zuletzt abgerufen am 28.09.2011

[NVI10] NVIDIA Corporation: *NVIDIA GeForce GTX 480/470/465 GPU Datasheet*, 2010. http://www.nvidia.de/docs/IO/90192/GTX-480-470-Web-Datasheet-Final4.

pdf

[OSW+05] OPENGL ; SHREINER, D. ; WOO, M. ; NEIDER, J. ; DAVIS, T.: *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, 2005. – ISBN 0321335732

[OW97] OVERVELD, C. W. A. M. ; WYVILL, B.: Phong normal interpolation revisited. In: *ACM Trans. Graph.* 16 (1997), October, 397–419. http://dx.doi.org/http://doi.acm.org/10.1145/263834.263849. – DOI http://doi.acm.org/10.1145/263834.263849. – ISSN 0730–0301

[Per04] PERWASS, Christian: *Teaching Geometric Algebra with CLUCalc University of Kiel, Cognitive Systems Group*. 2004

[Per11] PERWASS, Christian: *CLUCalc - Interaktive Virtualization*. http://www.clucalc.info/. Version: Juni 2011

[PGS03] PERWASS, C. ; GEBKEN, C. ; SOMMER, G.: Implementation of a Clifford algebra coprocessor design on a field programmable gate array. In: ABLAMOWICZ, R. (Hrsg.) ; 6th Int. Conf. on Clifford Algebras and Applications, Cookeville, TN (Veranst.): *CLIFFORD ALGEBRAS: Application to Mathematics, Physics, and Engineering* 6th Int. Conf. on Clifford Algebras and Applications, Cookeville, TN, Birkhäuser, Boston, 2003 (Progress in Mathematical Physics), S. 561–575

[PH03] PERWASS, C. ; HILDENBRAND, D.: Aspects of Geometric Algebra in Euclidean, Projective and Conformal Space / Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik. 2003 (Number 0310). – Technical Report

[Pho75] PHONG, Bui T.: Illumination for Computer Generated Pictures. In: *Commun. ACM* 18 (1975), Nr. 6, S. 311–317

[Pin06] PINHO, Márcio S.: *OBJ File Reader Library*. http://www.inf.pucrs.br/~pinho/CG/Aulas/LeitorDeObj/, 2006. – zuletzt abgerufen am 18.10.2010

[Rei07] REIS, G.: *GPU-based High-Quality Rendering of Quartic Splines*. Internet, March 2007

[RSF01] RAFAEL, Comparative S. ; SEGURA, Rafael J. ; FEITO, Francisco R.: Algorithms To Test Ray-Triangle Intersection. In: *Journal of WSCG*, 2001, S. 200–1

[RSM10] RESHETOV, Alexander ; SOUPIKOV, Alexei ; MARK, William R.: Consistent Normal Interpolation. New York, NY, USA : ACM, 2010

[SB87] SNYDER, John M. ; BARR, Alan H.: Ray tracing complex models containing surface tessellations. In: *SIGGRAPH Comput. Graph.* 21 (1987), August, 119–128. http://dx.doi.org/http://doi.acm.org/10.1145/37402.37417. – DOI http://doi.acm.org/10.1145/37402.37417. – ISSN 0097–8930

[SGS06] STOLL, Carsten ; GUMHOLD, Stefan ; SEIDEL, Hans-Peter: Incremental Raycasting of Piecewise Quadratic Surfaces on the GPU. In: WALD, Ingo (Hrsg.) ; PARKE, Steven G. (Hrsg.) ; IEEE (Veranst.): *IEEE Symposium on Interactive Raytracing 2006 Proceedings*. Salt Lake City, USA : IEEE, September 2006 (IEEE Symposium on Interactive Raytracing Proceedings). – ISBN 1–4244–0693–5, S. 141–150

[SWS02] SCHMITTLER, Jörg ; WALD, Ingo ; SLUSALLEK, Philipp: SaarCOR: a hardware architecture for ray tracing. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2002 (HWWS '02). – ISBN 1–58113–580–7, 27–36

[SWW+04] SCHMITTLER, Jörg ; WOOP, Sven ; WAGNER, Daniel ; PAUL, Wolfgang J. ; SLUSALLEK, Philipp: Realtime ray tracing of dynamic scenes on an FPGA chip. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. New York, NY, USA : ACM, 2004 (HWWS '04). – ISBN 3–905673–15–0, 95–106

[Uni04] UNIVERSITY OF TORONTO: *TM3 Documentation*, 2004. http://www.eecg.utoronto.ca/~tm3/

[Vin09] VINCE, John: *Geometric algebra: An algebraic system for computer games and animation*. London: Springer. xviii, 195 p, 2009. http://dx.doi.org/10.1007/978-1-84882-379-2. http://dx.doi.org/10.1007/978-1-84882-379-2

[WSS05] WOOP, Sven ; SCHMITTLER, Jörg ; SLUSALLEK, Philipp: RPU: a programmable ray processing unit for realtime ray tracing. In: *ACM SIGGRAPH 2005 Papers*. New York, NY, USA : ACM, 2005 (SIGGRAPH '05), 434–444

[Xil02] XILINX, INC.: *Virtex-E 1.8 V Field Programmable Gate Arrays*, 2002. http://www.xilinx.com/support/documentation/data_sheets/ds022-1.pdf

[Xil11] XILINX: *Spartan-3E FPGA Starter Kit Board User Guide (UG230)*, January 2011. http://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf