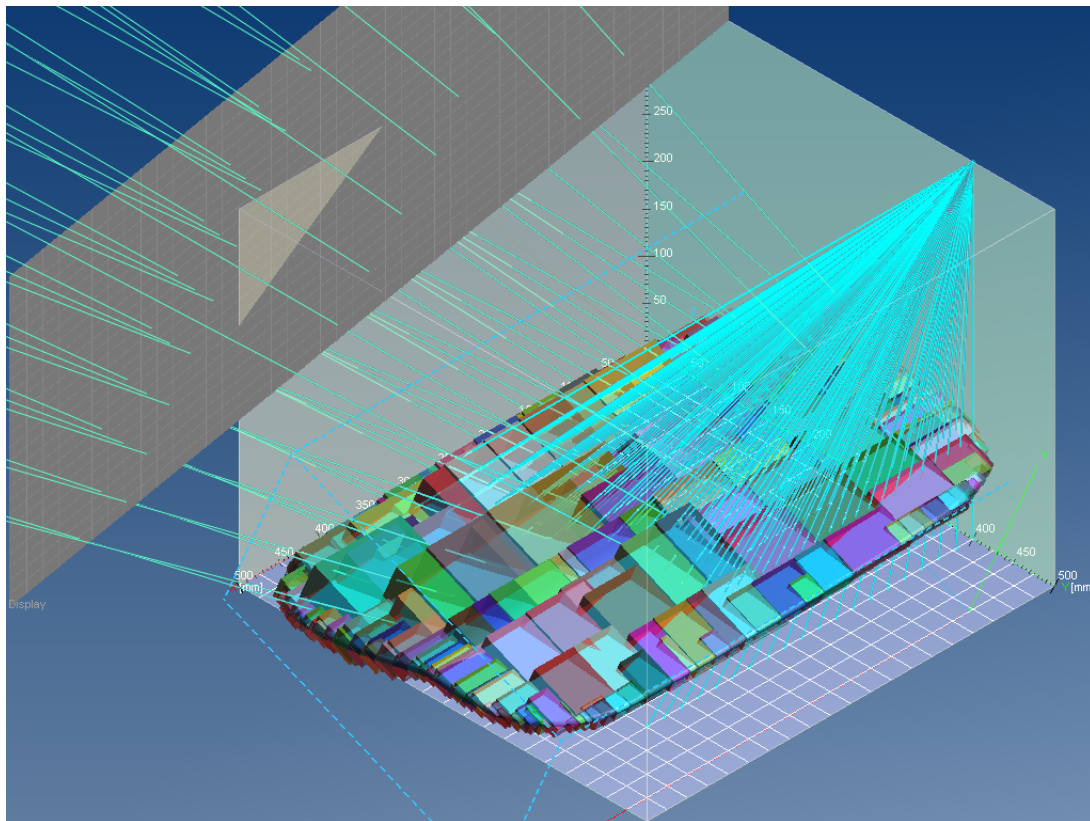


Deflektometrische Positionsbestimmung gegebener Geometrien mit Grafikkartenbeschleunigung

Diplomarbeit



Verfasser	Christoph K. Riesinger
Aufgabensteller	Prof. Dr. Klaus Donner Lehrstuhl für Numerische Mathematik und Analysis
Zweiter Prüfer	Prof. Dr. Paul Lukowicz Lehrstuhl für Eingebettete Systeme
Betreuer	Dr. Alexander Zimmermann
Datum	Passau, den 7. April 2010

Inhaltsverzeichnis

Problemaufriss	7
1 Einführung	9
1.1 Motivation	10
1.2 Aufgaben	12
1.2.1 Positionsbestimmung	12
1.2.2 Grafikkartenbeschleunigung	12
1.2.3 Modell zur Beschreibung der Geometrie	13
1.3 Aufbau	15
1.3.1 Positionsbestimmung einer gegebenen Geometrie	15
1.3.2 Grafikkartenprogrammierung mit CUDA	15
1.3.3 Dreiecksmengen und STL	16
1.3.4 Ergebnisse und Leistungsmessungen	16
1.3.5 Abschließendes Fazit und Ausblick	17
2 Positionsbestimmung einer gegebenen Geometrie	19
2.1 Deflektometrie	20
2.1.1 Aufbau eines Deflektometriesystems	20
2.1.2 Messverfahren in der Deflektometrie	20
2.1.3 Pixelpositionsbilder	21
2.2 Algorithmus zur Positionsbestimmung	23
2.2.1 Simulation	23
2.2.2 Vektordifferenzen in Pixpos-Bildern	24
2.2.3 Nichtlineare Optimierung	26
2.3 Modifikation des Algorithmus zur Verbesserung der Rekonstruktionsgüte	28
2.3.1 Kritik am bisherigen Rekonstruktionsalgorithmus	28
2.3.2 Ursache der mangelhaften Rekonstruktionsgüte	29
2.3.3 Elimination der Ursache	31
2.4 Beschreibung des Simulationaufbaus	35
2.4.1 Konventionen für Transformationen	35
2.4.2 Modellierung der Kamera	36
2.4.3 Sonstige Komponenten des Deflektometriesystems	36
2.4.4 Inverse Transformationen	36
3 Grafikkartenprogrammierung mit CUDA	39
3.1 Einführung und Kontext	40
3.1.1 Historie der Grafikprozessoren	40

3.1.2	Eigenschaften von Grafikkarten und Abgrenzung zur CPU	42
3.1.3	Komponenten von CUDA	43
3.2	Das CUDA-Modell	44
3.2.1	Compute Capabilities	44
3.2.2	Kernel, Threads, Blocks und Grids	44
3.2.3	Speicherorganisation	47
3.2.4	Warps, Half-Warps und Warp-Divergence	52
3.2.5	Occupancy und Instruction Throughput	53
3.3	Klassische Probleme und Trade-Offs bei CUDA-Programmierung	55
3.3.1	Limitierung schneller Speicher vs. Occupancy	55
3.3.2	Coalescing vs. komplexe Datenstrukturen	56
3.3.3	Kleine Kernel vs. Instruction Throughput	56
3.3.4	Speicherzugriffe vs. Neuberechnung	57
3.3.5	Bedingte Anweisungen führen zu Warp-Divergence	58
3.3.6	Speichertransfers zwischen Host und Device	60
3.3.7	Größe von Blocks und dem Grid	61
3.4	Wann ist der Einsatz von CUDA sinnvoll?	62
4	Dreiecksmengen und STL	67
4.1	Einführung und Herausforderungen	68
4.1.1	Eigenschaften von Dreiecksmengen und des STL-Modells	68
4.1.2	Vor- und Nachteile von Dreiecksmengen und des STL-Modells	70
4.1.3	Raycasting	72
4.2	Elimination von Vertexduplikaten	73
4.3	Hierarchisierung der Facettenmenge zur Laufzeitverbesserung	76
4.3.1	Notwendigkeit der Hierarchisierung	76
4.3.2	Octree	76
4.3.3	kd-Tree	78
4.3.4	BVH	79
4.4	Aufbau einer BVH	81
4.4.1	Aufbau durch die CPU	81
4.4.2	Aufbau durch die GPU	83
4.4.3	Heuristiken für performante Hierarchien	83
4.4.4	Konvertierung einer Hierarchie für die Grafikkarte	85
4.5	Traversierung einer BVH	87
4.5.1	Notwendige Berechnungen	87
4.5.2	Traversierung auf der CPU	89
4.5.3	Traversierung auf der GPU	90
4.6	Glättung der Facetten	94
4.6.1	Notwendigkeit von Vertexnormalen	94
4.6.2	Heuristiken unter Berücksichtigung der gesamten Facette	95
4.6.3	Heuristiken unter Berücksichtigung des Vertex	96
4.6.4	Reapproximation mit der Taylorreihe	98
4.6.5	Probleme bei der Bestimmung von Vertexnormalen	98
5	Ergebnisse und Leistungsmessungen	101
5.1	Spezifikation des Testsystems	102

5.2	Auswertung des Rekonstruktionsalgorithmus	104
5.2.1	Einführende Erläuterungen	104
5.2.2	Einfluss der Anzahl der Iterationen	105
5.2.3	Relevanz weggenommener Randschichten	106
5.2.4	Auswirkungen der Pixpos-Bild-Auflösung	107
5.2.5	Größe des Startsimplex	108
5.3	Laufzeitvergleich zwischen CPU- und CUDA-Implementierung	110
5.3.1	Einführende Erläuterungen	110
5.3.2	Sichtstrahlgenerierung	110
5.3.3	Simulation der Pixpos-Bilder	111
5.3.4	Wegnahme von Randschichten einer Maske	113
5.3.5	Berechnung der normierten Summe der Pixelpositionsabweichungen	114
5.3.6	Beurteilung der Gesamtberechnung	115
5.4	Vergleich unterschiedlicher CUDA-Handhabungen	118
5.4.1	Einführende Erläuterungen	118
5.4.2	Einfluss von gecachten Texturen	118
5.4.3	Relevanz des Shared Memory	119
5.4.4	Verwendung schneller mathematischer Operationen	120
6	Abschließendes Fazit und Ausblick	121
6.1	Bewertung des Rekonstruktionsalgorithmus	122
6.2	Nutzen der Grafikkartenprogrammierung	124
6.3	Ausblick auf die zukünftige Entwicklung des CUDA-Frameworks	125
	Literaturverzeichnis	127
	Abbildungsverzeichnis	133
	Tabellenverzeichnis	135
	Listingverzeichnis	137
	Abkürzungen	139
	Danksagung	141
	Eidesstattliche Erklärung	143

Problemaufriss

*„Inmitten der Schwierigkeit liegt die Möglichkeit.“
Albert Einstein, deutscher Physiker (*14.03.1879, †18.04.1955)*

Deflektometrie hat sich zu einem etablierten Verfahren in der Qualitätsprüfung entwickelt. Sie bietet die Möglichkeit, spiegelnde Oberflächen, die beispielsweise bei metallischen Bauteilen oder Komponenten vorkommen, präzise zu inspizieren. Mit ihrer Hilfe lassen sich Mängel wie Lackfehler, Haarrisse oder kleinste Dellen detektieren.

In dieser Diplomarbeit wird ein Verfahren vorgestellt, das es mit Hilfe der Deflektometrie ermöglicht, die Position und Lage eines zu prüfenden Bauteils zu rekonstruieren. Ein solcher Schritt ist notwendig, wenn man Qualitätskriterien prüfen will, die über die bloßen Eigenschaften der Oberfläche hinausgehen. So ist es oftmals von Interesse, ob die Form des produzierten Bauteils innerhalb gegebener Toleranzen liegt oder ob die industriell gefertigte Komponente zu weit von einem meist als CAD-Entwurf vorliegenden Ideal abweicht. Solche Informationen sind umso wichtiger, je exakter ein Teil gefertigt sein muss, wie es im Maschinenbau oder in der Medizin der Fall ist. Um derartige Aussagen zu treffen, ist es zunächst jedoch erforderlich, genau zu wissen, in welcher Position der zu vermessende Gegenstand unter dem Deflektometriemessaufbau liegt.

Zwar gibt es bereits zahlreiche Verfahren in der maschinellen Bildverarbeitung, die eben jene Positionsrekonstruktion durchführen, allerdings ist es wünschenswert, mit nur einem einzelnen Messvorgang möglichst viele Aussagen machen zu können. Anstatt also Deflektometrie für die Oberflächeninspektion und gesondert modellbasierte Stereorekonstruktion für die Bestimmung von Lage und Position zu verwenden, ist es erstrebenswert, beide Informationen aus nur einem deflektometrischen Messdatensatz zu erhalten.

Neben der bloßen Positionsrekonstruktion liegt ein zweiter Schwerpunkt dieser Diplomarbeit in der Beschleunigung der Algorithmik durch programmierbare Grafikprozessoren. Diese können unter gewissen Umständen extrem performant sein und Berechnungen signifikant schneller durchführen als der Hauptprozessor. Die in dieser Arbeit vorgestellte Algorithmik erfüllt an vielen Stellen die Eigenschaften, um effizient von Grafikprozessoren abgearbeitet zu werden. Ziel ist es zu zeigen, dass die noch sehr junge Technologie programmierbarer Grafikprozessoren in einem industriellen Anwendungsszenario klassischen Berechnungsarten durch den Hauptprozessor deutlich überlegen sein kann.

Kapitel 1

Einführung

1.1 Motivation

Unter maschineller Bildverarbeitung fasst man Verfahren zusammen, die voll- oder halb-automatisch Informationen aus Bildern extrahieren. Solche Bilder können aus unterschiedlichsten Quellen stammen. Im Bereich der Medizin können mit Röntgenstrahlen, radioaktiven Stoffen, magnetischen Feldern, Schallwellen und den entsprechenden Sensoren Röntgen-, Computertomographie-, Magnetresonanztomographie und Ultraschallbilder generiert werden. Sensoren in Zeilen- und Matrixkameras erfassen sichtbares, infrarotes oder ultraviolettes Licht und liefern so Abbilder der visuell wahrnehmbaren Umwelt. Wie das Beispiel der Medizin zeigt, muss das Ausgangsmaterial für maschinelle Bildverarbeitung keineswegs immer ein rechteckiges digitales Bild sein. 3D-Punktwolken und Volumenaufnahmen kommen ebenso in Frage wie sich zeitlich ändernde oder durch den physikalischen Messprozess mit zusätzlichen Daten angereicherte Aufnahmen.

Somit ist maschinelle Bildverarbeitung ein Teilbereich der Signalverarbeitung und hat Berührungspunkte mit den unterschiedlichsten Disziplinen. Physik, Sensorik und Optik spielen eine wichtige Rolle. Zahlreiche Teilgebiete der Mathematik, wie Numerik, finden Verwendung. Genauso zahlreich wie die Art der Informationen, die man mit Hilfe maschineller Bildverarbeitung gewinnen kann, sind auch die Bereiche der Informatik, die zum Einsatz kommen.

In vielen Bereichen findet sich maschinelle Bildverarbeitung. In der Medizin dient sie der Diagnostik und Therapieplanung. Ein weiteres Feld ist die Robotik. Dort dient sie der optischen Erfassung der Umwelt und stellt somit oftmals, genau so wie beim Menschen, den wichtigsten Sinn dar. Große Bedeutung hat maschinelle Bildverarbeitung darüber hinaus vor allem in der Industrie, wo sie einen zentralen Stellenwert in der Qualitätsprüfung hat. Überall dort, wo Waren produziert werden, hat man ein Interesse an hoher Qualität der erzeugten Produkte. Dieser Wunsch wird umso größer, je teurer die Herstellungsprozesse oder je kritischer die Aufgabenbereiche sind, in denen die Erzeugnisse Verwendung finden. Der Maschinenbau ist hierfür ein gutes Beispiel. Hergestellte Komponenten werden nicht selten in niedriger Stückzahl oder als Einzelteile produziert, was sie teuer macht. Selbst wenn ein Teil zur Massenkomponekte wird, muss es immer noch höchsten Präzisions- und Qualitätskriterien entsprechen. Das Modell eines Kolbens in einem Motor veranschaulicht dies sehr gut. Er muss mit einer extrem kleinen Toleranz in den Zylinder passen. Darüber hinaus wird er im Laufe seines Lebens Milliarden Male im Zylinder hin und herbewegt und das unter hohem Druck. Die Qualität, Stabilität und Präzision eines Kolbens zu gewährleisten, ist somit ein nachvollziehbares Ziel.

Genau in diese Kerbe schlägt maschinelle Bildverarbeitung, mit der es möglich ist, zahlreiche Qualitätsparameter von vorliegenden Erzeugnissen schnell und detailreich zu erfassen. Aus diesem Grund spielt sie in der industriellen Qualitätssicherung schon seit langem eine wichtige und etablierte Rolle. Eine Form der maschinellen Bildverarbeitung ist die Deflektometrie.

Deflektometrie ist in erster Linie ein Verfahren zur Inspektion von spiegelnden und daher meist metallischen oder lackierten Oberflächen. Mit ihrer Hilfe lassen sich Schäden und Mängel wie Lackkratzer, Dellen und Haarrisse detektieren. Lackkratzer können nicht nur ein ästhetischer Makel sein, sondern sind nicht selten die Keimpunkte für Rost. Dellen sind in Hochpräzisionsspiegeln, wie sie in großen Teleskopen vorkommen, absolut inakzeptabel. Haarrisse stellen eine strukturelle Schwäche im Material dar, an denen es versagen kann,

und sind somit unerwünscht. Jedoch kann Deflektometrie nur Gegenstände prüfen, die eine spiegelnde Oberfläche besitzen. Außerdem beschränkt sich Deflektometrie zunächst nur auf die Oberflächeninspektion. Informationen über Form und Lage des zu prüfenden Bauteils stehen zunächst nicht zur Verfügung. Dafür sind zusätzliche Informationen wie der Abstand des Objekts zur Kamera erforderlich [Kic06]. Aber auch andere Disziplinen der maschinellen Bildverarbeitung wie modellbasierte Stereorekonstruktion oder Laservermessung können zur Ermittlung von Form und Lage herangezogen werden.

Man ist allerdings daran interessiert, möglichst viele Informationen aus möglichst wenigen Messungen mit möglichst wenigen Messgeräten zu erhalten. Je weniger Messungen gemacht werden, desto schneller kann der Gesamtmessvorgang ausgeführt werden. Je weniger Messgeräte benötigt werden, desto kostengünstiger ist der Messaufbau. Beides sind ökonomische Vorteile. Mit diesem Hintergrund ist der Versuch also nur verständlich, aus deflektometrischen Messdaten, neben den bloßen Informationen über die Oberfläche, auch weiterführende Aussagen treffen zu können. In welcher Lage und Position liegt die zu prüfende Form unter dem Messaufbau? Wurde sie innerhalb geometrischer Toleranzen gefertigt? Entspricht ihr Aussehen einem meist als Computer Aided Design-Entwurf (kurz CAD) vorliegenden Ideal?

Um all diese Fragen zu beantworten, ist eine Information notwendig, die Deflektometrie nicht implizit liefert: In welcher räumlichen Ausrichtung wird das zu analysierende Teil vom deflektometrischen Messaufbau erfasst? Oder anders ausgedrückt: Wie muss man das vorgegebene Ideal rotieren beziehungsweise drehen und translieren beziehungsweise verschieben, damit es mit dem real vermessenen Fertigungsprodukt möglichst exakt übereinstimmt? In dieser Arbeit wird ein Algorithmus vorgestellt, der genau diese Frage positiv beantwortet und dabei nur auf deflektometrische Messdaten und den vorgegebenen CAD-Entwurf des zu überprüfenden Objekts zurückgreift. Die Berechnung der erwähnten Rotation und Translation wird auch als Positionsrekonstruktion oder Positionsbestimmung bezeichnet.

Getreu dem Motto „Zeit ist Geld“ ist es wünschenswert, dass Aufgaben möglichst schnell abgearbeitet werden. Bei der hier vorliegenden Positionsrekonstruktion ist dies nicht anders. Eine noch relativ junge Technologie könnte hierbei zu einer wesentlichen Leistungssteigerung bezüglich der Performanz (nicht unbedingt bezüglich der Präzision) beitragen: Programmierbare Grafikprozessoren. Sie ermöglichen es, die seit jeher in Grafikprozessoren (Graphics Processing Unit, kurz GPU) vorhandenen massiv parallelen Rechenwerke für andere Zwecke als das Rendering von virtuellen 3D-Szenen zu nutzen. Erfüllen Probleme gewisse Eigenschaften, können sie durch Grafikprozessoren wesentlich effizienter und somit schneller und stromsparender gelöst werden als durch den Hauptprozessor (Central Processing Unit, kurz CPU). Der hier vorgestellte Algorithmus zur Positionsrekonstruktion erfüllt an vielen Stellen die nötigen Eigenschaften, weshalb das zweite große Ziel dieser Arbeit darin besteht zu überprüfen, ob die Algorithmik zur Positionsrekonstruktion von programmierbaren GPUs profitieren kann.

1.2 Aufgaben

Nachdem im vorigen Abschnitt geklärt wurde, warum ein Interesse an der Positionsbestimmung eines zu prüfenden Bauteils herrscht, soll dieser Abschnitt zusammenfassen, welche Aufgaben Teil dieser Diplomarbeit sind.

1.2.1 Positionsbestimmung

Positionsbestimmung beziehungsweise Positionsrekonstruktion bedeutet die Ermittlung der Lage des real unter dem deflektometrischen Messaufbau liegenden Teiles. Zwar kann diese Lage im Voraus grob auf wenige Grad Rotation und wenige Millimeter (kurz: mm) Translation geschätzt werden, um eine vernünftige Qualitätssicherung durchführen zu können, sollten jedoch Rotationen auf Bogenminuten, besser noch auf Bogensekunden genau angegeben werden können, Translationen auf wenige Mikrometer (kurz: μm) genau.

Technischer ausgedrückt heißt das, dass man einen Entwurf der gefertigten Komponente vorliegen hat, der die ideale Form beschreibt. Ein solcher Entwurf stammt zumeist von einer technischen Zeichnung oder, heute eher üblich, aus einem CAD-Entwurf. Dort wird die ideale Vorlage virtuell bezüglich eines eigenen Koordinatensystems angegeben, dem CAD-Koordinatensystem (kurz: CAD-KoSy). Die Realität, in der die Qualitätsprüfung durchgeführt wird und in der sich der deflektometrische Messaufbau befindet, hat allerdings ihr eigenes Koordinatensystem, das Welt-Koordinatensystem (kurz W-KoSy) genannt wird. Die Positionsrekonstruktion entspricht also exakt der Aufgabe, diejenige Transformation zu finden, die das CAD-KoSy in das W-KoSy überführt. In beiden Koordinatensystemen gibt es ein gemeinsames Objekt, nämlich das fabrizierte Produkt, das inspiziert werden soll. Zum einen liegt es virtuell und ideal im CAD-KoSy vor, zum anderen real und mit möglichst wenig Produktionsfehler im W-KoSy.

Erste und wichtigste Aufgabe ist demzufolge das Finden eines Algorithmus, der diejenige Transformation errechnet, die das CAD-KoSy korrekt in das W-KoSy überführt. Dabei soll er nur auf deflektometrische Messergebnisse und den Entwurf des Teils zurückgreifen. Eine Transformation ist hier die Hintereinanderausführung einer Rotation und einer Translation. Sie wird durch insgesamt sechs Parameter beschrieben. Drei dieser Parameter $r = (\phi_1, \phi_2, \phi_3)$ sind Winkel, die jeweils eine Drehung um eine der drei Achsen des kartesischen Koordinatensystems im $\mathbb{R}^3$ ausdrücken und bilden somit die Rotation. Solche Winkel werden auch als Eulerwinkel [Wik10b] bezeichnet. Die anderen drei Parameter $t = (t_1, t_2, t_3)$ sind die Komponenten einer Verschiebung im $\mathbb{R}^3$, die die Translation darstellt. Mit diesen sechs Parametern lassen sich alle form- und größenhaltenden Transformationen im $\mathbb{R}^3$ durchführen. Abbildung 1.1 zeigt eine einfache Rotation und eine einfache Translation.

1.2.2 Grafikkartenbeschleunigung

Die zweite zentrale Aufgabe dieser Arbeit ist nach dem Finden eines Algorithmus für die Positionsrekonstruktion die Beschleunigung des gefundenen Verfahrens mit programmierbaren Grafikprozessoren. Diese können unter gewissen Umständen Berechnungen wesentlich zügiger bewerkstelligen als die CPU. Da es sich jedoch beim Rekonstruktionsalgorithmus zur Positionsbestimmung um ein nichttriviales, mehrstufiges Verfahren handelt,

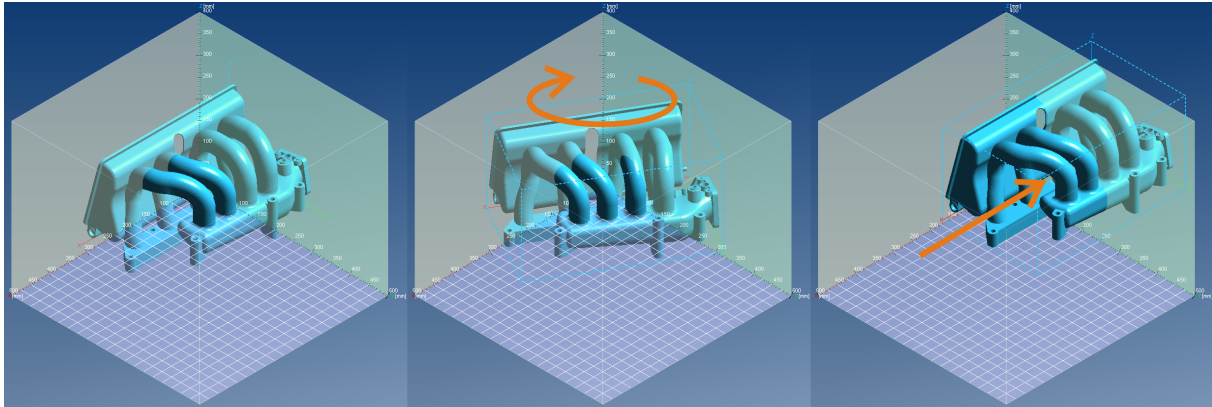


Abbildung 1.1: Das linke Bild zeigt das Bauteil in der Ausgangsposition. Im mittleren Bild wurde es um 30° um die z-Achse gedreht. Rechts wird eine Translation in der yz-Ebene illustriert.

lassen sich die geforderten Umstände, in denen programmierbare Grafikprozessoren dem Hauptprozessor überlegen sind, nicht ohne Weiteres überprüfen. Diese Tatsache führt zu zwei wichtigen Aufgaben:

- Einerseits soll konkret für den vorliegenden Algorithmus geklärt werden, ob er von Grafikkartenbeschleunigung profitieren kann. Da er mehrstufig ist, muss eine Evaluation für jede einzelne Verarbeitungsstufe durchgeführt werden.
- Durch die Mehrstufigkeit des Rekonstruktionsalgorithmus werden eine große Menge verschiedener Szenarien getestet, die für Berechnungen auf der Grafikkarte in unterschiedlichem Maße geeignet sind. So wird es möglich, Erfahrungen zu sammeln, die über die in der Einstiegsliteratur [NVi09a] vorhandenen Beispiele weit hinausgehen. Diese Erfahrungen sind überaus nützlich für die Beurteilung zukünftiger Projekte bezüglich ihrer Eignung für Grafikkartenbeschleunigung.

1.2.3 Modell zur Beschreibung der Geometrie

An mehreren Stellen wurde bereits erwähnt, dass ein Ideal der hergestellten Komponente als Vorlage in Form eines CAD-Entwurf existiert. Dabei stellt sich im nächsten Schritt die Frage, auf welche Art und Weise das Bauteil im CAD-Entwurf beschrieben wird. CAD-Programme stellen dabei dem Benutzer eine Vielzahl von Modellierungsmöglichkeiten zur Verfügung, um die Oberfläche eines Gegenstandes zu beschreiben. In diesem Zusammenhang wird das modellierte Objekt oft auch als Geometrie bezeichnet, da es sich hierbei um einen allgemeineren Begriff handelt. Schließlich können in einem CAD-Programm auch Objekte modelliert werden, die kein Äquivalent in der Wirklichkeit besitzen.

Zu den Oberflächenmodellen gehören neben Bézierpatches auch unterschiedliche Arten von Splines (kubische Splines, B-Splines) und die darauf aufbauenden Non Uniform Rational B-Splines (kurz: NURBS). Auch eine Modellierung der Oberfläche mit Polygonen ist möglich. Vor allem Dreiecksmengen dienen häufig der Darstellung von dreidimensionalen Objekten am Computer. Ein weit verbreiteter Vertreter solcher Dreiecksmengen ist das Surface Tessellation Language-Modell (kurz: STL). Eine detaillierte Einführung von

Dreiecksmengen im Allgemeinen und des STL-Modells im Speziellen findet separat in Kapitel 4 statt, da sie die in dieser Arbeit verwendeten Methoden sind, um die virtuelle Form von Objekten wiederzugeben.

Somit liegt der Kern der Aufgabe nicht im Finden eines neuen Modells für die Oberflächenmodellierung, sondern in der Beherrschung von Dreiecksmengen bei komplexen Geometrien.

1.3 Aufbau

Da nun demonstriert wurde, wofür Positionsrekonstruktion benötigt wird und welche Aufgaben in dieser Arbeit erledigt werden, soll dieser Abschnitt den strukturellen Aufbau dieser Arbeit wiedergeben.

1.3.1 Positionsbestimmung einer gegebenen Geometrie

Inhalt des ersten Kapitels nach der Einführung ist der Algorithmus, der die Positionsbestimmung durchführt. Dazu wird anfangs im einführenden Abschnitt 2.1 das Verfahren der Deflektometrie genauer vorgestellt, von dem bisher nur bekannt ist, dass es sich um ein Verfahren der maschinellen Bildverarbeitung zur Inspektion spiegelnder Oberflächen handelt. Bevor der Rekonstruktionsalgorithmus vorgestellt werden kann, ist allerdings ein tieferes Verständnis von Deflektometrie und deren Arbeitsweise von Nöten.

Ist dieser Grundstein gelegt, kann im folgenden Abschnitt 2.2 begonnen werden, den Rekonstruktionsalgorithmus ausführlich zu erläutern. Zu diesem Zeitpunkt wird klar, dass die Positionsbestimmung mehrstufig abläuft und der Algorithmus auf diverse Techniken der maschinellen Bildverarbeitung zurückgreift.

Es stellt sich jedoch heraus, dass eine bestimmte Modifikation des Algorithmus unerlässlich ist, damit dieser zuverlässig die Positionsbestimmung durchführt. Warum diese Anpassung notwendig ist und wie diese aussieht, erklärt Abschnitt 2.3. In dieser Modifikation steckt das primäre Forschungsergebnis dieser Arbeit.

Im letzten Abschnitt 2.4 werden eben jene Technologien der maschinellen Bildverarbeitung erläutert, auf denen der Rekonstruktionsalgorithmus aufbaut. Dabei handelt es sich hauptsächlich um diverse Modelle, die im Zuge des Lösungsalgorithmus Verwendung finden.

1.3.2 Grafikkartenprogrammierung mit CUDA

Nachdem der Rekonstruktionsalgorithmus präsentiert wurde, kann damit begonnen werden, diesen unter Verwendung programmierbarer Grafikprozessoren zu beschleunigen. In Kapitel 3 wird darauf eingegangen. Um Grafikkartenprogrammierung im richtigen Kontext zu sehen, beleuchtet Abschnitt 3.1 historische Meilensteine, die zu den heute verfügbaren Grafikprozessoren geführt haben.

Im Anschluss daran wird in Abschnitt 3.2 das theoretische Fundament für die Programmierung von Grafikkarten des Herstellers NVidia mit dessen Compute Unified Device Architecture-Framework (kurz: CUDA) gelegt. Einerseits legt man sich somit auf die Technologie eines bestimmten Herstellers fest, andererseits ist es nur so möglich, Aspekte der Grafikkartenprogrammierung tiefgehend zu ergründen. Aufgrund der Tatsache, dass es sich bei dieser Technologie noch um eine sehr junge Entwicklung handelt, liegt auf diesem Abschnitt ein besonderer Fokus.

Unter Verwendung dieses theoretischen Wissens können in Abschnitt 3.3 klassische Probleme und Trade-Offs der Grafikkartenprogrammierung unter die Lupe genommen werden. Hierbei zeigt sich auch, unter welchen Umständen programmierbare Grafikprozessoren klassischen Hauptprozessoren bezüglich ihrer Performanz überlegen sind.

Abgeschlossen wird Kapitel 3 mit Abschnitt 3.4, der als eine Art Resumée drei Entschei-

dungsbäume zur Verfügung stellt, die auch komplexere Probleme auf ihre Eignung für Grafikkarten überprüfen. Sie bilden somit eine Zusammenfassung der Erfahrungen im Bereich Grafikkartenprogrammierung.

1.3.3 Dreiecksmengen und STL

Um das digitale Ideal des Bauteils zu beschreiben, wird ein Oberflächenmodell benötigt. In dieser Arbeit wird dafür die Zerlegung der Oberfläche in Dreiecke gewählt, mit der sich Kapitel 4 ausführlich befasst. Zunächst dient Abschnitt 4.1 der Vorstellung solcher Dreieckszerlegungen, ihrer Eigenschaften, den sich daraus ergebenden Herausforderungen und ihren Vor- und Nachteilen. Dabei dient das STL-Modell als Anschauungsobjekt einer in der Industrie eingesetzten Dreiecksmenge.

Einer dieser Nachteile ist die Art und Weise der Speicherung der Menge von Dreiecken, denn zahlreiche Informationen werden redundant abgelegt und verbrauchen so deutlich mehr Speicher als nötig. Sowohl die dauerhafte Speicherung auf persistenten Medien als auch das Halten im Arbeitsspeicher zur Laufzeit ist davon betroffen. Welche Möglichkeiten der Speicherverbrauchreduktion existieren, zeigt Abschnitt 4.2.

Dreiecksmengen approximieren die Oberfläche eines Objekts mit Dreiecken. Will man effizient mit dem STL-Modell arbeiten, muss die Menge von Dreiecken hierarchisiert werden. Nur so lässt sich logarithmische anstatt linearer Laufzeit bezüglich der Anzahl der Dreiecke für typische Operationen mit der Dreiecksmenge erreichen. Mehrere Möglichkeiten einer solchen Hierarchisierung führt Abschnitt 4.3 vor.

Anhand einer Bounding Volume Hierarchy (kurz: BVH) konkretisiert Abschnitt 4.4, auf welche Weise eine derartige Hierarchisierung erstellt werden kann. BVHs sind die in dieser Arbeit gewählte Hierarchisierungsmethode. Andere Ansätze der Hierarchisierung werden nur der Vollständigkeit halber erwähnt, da es nicht „die“ beste Hierarchisierungsmethode gibt und alternative Hierarchisierungsverfahren durchaus ihre Existenzberechtigung haben.

Hat man schließlich eine Hierarchisierung der Dreiecksmenge aufgebaut, muss diese auch entsprechend traversiert werden. Vor allem unter Berücksichtigung des Versuchs der Grafikkartenbeschleunigung ist dies ein interessantes Thema, da Grafikkartenprogrammierung und komplexe Datenstrukturen im Konflikt zueinander stehen. Folglich beschäftigt sich Abschnitt 4.5 mit der Findung eines Kompromisses, um diesen Konflikt aufzulösen.

Zuletzt wird in Abschnitt 4.6 ein letzter Nachteil der Approximation von Oberflächen durch Dreiecke behandelt. Da die Oberfläche der Geometrie nur mit planaren Objekten modelliert wird, können gekrümmte Oberflächen nicht fehlerfrei dargestellt werden. Diverse Tricks erlauben aber immerhin die ungefähre Nachbildung von gekrümmten Oberflächen. Je nachdem, wie viel Aufwand man in die Glättung der Dreiecksmenge stecken will, reichen diese Verfahren von einfachen Heuristiken bis hin zu komplexeren Rekonstruktionen mit der Taylor-Formel.

1.3.4 Ergebnisse und Leistungsmessungen

Die Auswertung der in dieser Arbeit behandelten Technologien findet in Kapitel 5 statt. Viele der Ergebnisse stammen aus Leistungsmessungen, sogenannten Benchmarks. Ihre Grundlage ist die Spezifikation des Testsystems in Abschnitt 5.1, mit dem die Messungen

erhoben werden.

Ganz besonders wichtig für die Güte des Rekonstruktionsalgorithmus ist dessen Parametrisierung. Güte bedeutet hierbei die Rate, mit der das Verfahren die Position korrekt bestimmt. Zahlreiche Größen haben Einfluss auf diese Güte, genauso wie auf die Geschwindigkeit der Terminierung des Verfahrens. Deren optimale Wahl klärt Abschnitt 5.2.

Ziel dieser Arbeit ist es unter anderem zu vergleichen, in welchen Fällen die GPU-Implementierung der CPU-Implementierung überlegen ist und wie deutlich der Unterschied ausfällt. Am leichtesten erreicht man dies durch den Laufzeitvergleich beider Implementierungen. Die Tabellen in Abschnitt 5.3 stellen die Zahlen dieses Duells für jeden einzelnen Schritt des Rekonstruktionsverfahrens gegenüber.

Grafikkartenprogrammierung ermöglicht es an vielen Stellen mehrere Wege zu gehen. Zusätzlich lassen sich Kalkulationen auf Kosten der Berechnungsgenauigkeit beschleunigen. Ob solche abweichenden Ansätze beziehungsweise alternative Berechnungsmethoden schlussendlich die Performanz signifikant erhöhen, analysiert Abschnitt 5.4.

1.3.5 Abschließendes Fazit und Ausblick

Den Abschluss dieser Arbeit bildet Kapitel 6, das die Inhalte dieser Arbeit letztmalig zusammenfasst und einen Ausblick in die zukünftigen Entwicklungen der Grafikkartenprogrammierung wagt.

Dazu gibt Abschnitt 6.1 die Leistungsfähigkeit des Rekonstruktionsalgorithmus wieder und beantwortet letztendlich die Frage, ob das präsentierte Verfahren die an ihn herangetragenen Anforderungen zufriedenstellend erfüllt.

Ob die erarbeitete Algorithmik in ihrer Gesamtheit durch programmierbare Grafikkarten profitiert und falls ja, wie stark, resümiert Abschnitt 6.2.

Ein Ausblick in die nähere Zukunft der Entwicklung der Grafikkartenprogrammierung soll abschließend in Abschnitt 6.3 im Mittelpunkt stehen. Dabei spielen sowohl Aspekte der Hardware, vor allem aber der Software eine Rolle.

Kapitel 2

Positionsbestimmung einer gegebenen Geometrie

2.1 Deflektometrie

Bevor mit der Präsentation des Rekonstruktionsalgorithmus begonnen werden kann, wird zuvor das Verfahren der Deflektometrie erläutert.

2.1.1 Aufbau eines Deflektometriesystems

Deflektometrie ist ein optisches Messverfahren zur Inspektion spiegelnder Oberflächen. Ein Deflektometriesystem ist grundsätzlich aus drei Objekten aufgebaut:

- Eine Kamera, die die Szene aufnimmt.
- Das gefertigte Teil, das geprüft werden soll.
- Ein Bildschirm, der ein spezielles Muster anzeigt.

Dabei werden diese drei Objekte so angeordnet, dass der Monitor das zu kontrollierende Teil beleuchtet. Die Kamera betrachtet ebenfalls die zu prüfende Komponente und ist dabei auf sie fokussiert. Aufgrund der geforderten spiegelnden Oberfläche des Prüfobjekts, erfasst die Kamera die Reflexion dessen, was auf dem Bildschirm angezeigt wird. Abbildung 2.1 illustriert den schematischen Aufbau eines Deflektometriesystems, während 2.2 einen realen Aufbau zeigt.

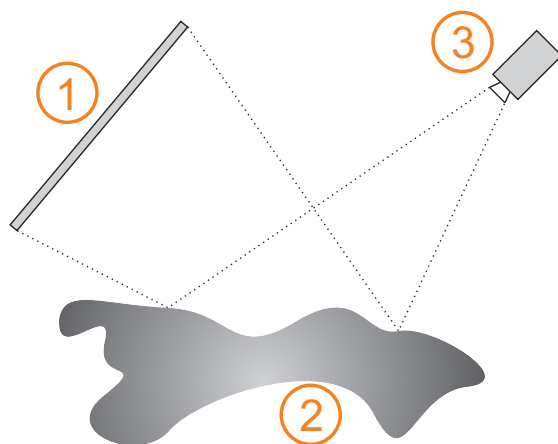


Abbildung 2.1: Der Monitor (1) links ist so ausgerichtet, dass er das zu inspizierende Objekt (2) bestrahlt. Die Kamera (3) betrachtet ebenfalls die Komponente und sieht so die Reflexion des Bildschirms.

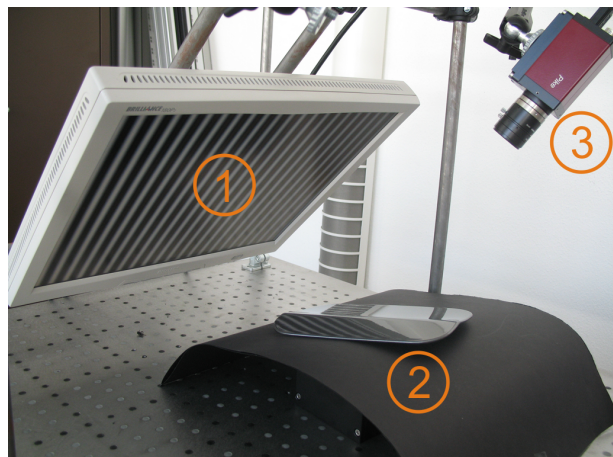


Abbildung 2.2: Auf dem Monitor (1) wird ein vertikales Streifenmuster angezeigt, das sich auf einem Tankdeckel (2) spiegelt. Diese Spiegelung wird von der Kamera (3) erfasst und aufgenommen.

2.1.2 Messverfahren in der Deflektometrie

Das Wesentliche der Deflektometrie liegt in dem Muster, das der Bildschirm anzeigt. Dieses wird von der spiegelnden Oberfläche der zu prüfenden Komponente reflektiert und von

der Kamera erfasst. Bei diesem Muster handelt es sich grundsätzlich um schwarz-weiße, sinusförmige Streifen. Der Übergang von einem weißen zu einem schwarzen Streifen und umgekehrt ist also nicht abrupt, sondern hat einen Grauwertübergang, der der Sinusfunktion entspricht.

Während eines Messvorgangs wird dieses Streifenmuster phasenverschoben. Es wandert also über den Bildschirm. Dabei ändert sich zunächst nur die Phase, nicht jedoch die Wellenlänge der Streifen. Alle anderen Objekte des Deflektometriesystems bleiben unverändert. Vergleicht man nun die von der Kamera erfassten, phasenverschobenen Streifenmuster, die an der Oberfläche der Testkomponente gespiegelt werden, so entstehen daraus sogenannte Phasenbilder. Mit diesen Phasenbildern lassen sich im nächsten Schritt diskrete Gradienten in Richtung des zugehörigen Streifenmusterwinkels berechnen. Mit ihnen ist es möglich, größere Krümmungsänderungen an der Oberfläche zu erkennen. Derartige Krümmungsänderungen treten beispielsweise bei Kratzern auf. Jedoch ist diese Detektion nur qualitativ. Möchte man eine quantitative Aussage über die Tiefe des Kratzers treffen, muss diese als Standardverfahren bezeichnete Vorgehensweise erweitert werden.

Ändert man das Standardverfahren und variiert nun auch die Wellenlänge beziehungsweise Breite der Streifen, so können zusätzliche Phasenbilder berechnet werden. Sie erlauben es, den zu einer Reflexion gehörigen Bildpunkt auf dem Bildschirm, der das Streifenmuster anzeigt, zu bestimmen. Dazu wird ein Phase-Unwrapping [GP98] genanntes Verfahren benötigt.

Tiefergehendes Wissen, wie die Wahl der Phase oder Wellenlänge, soll an dieser Stelle nicht vermittelt werden. Auch die genaue Berechnung der Phasenbilder, der diskreten Gradienten und das Phase-Unwrapping werden nicht genauer beleuchtet, da in dieser Arbeit nur die Berechnungsergebnisse benötigt werden, nicht jedoch die Berechnungswege. Für eine detailliertere Einführung sei stattdessen auf [Bro09] verwiesen, wo sich auch Bildmaterial zu den Streifenmustern findet. Eine umfassende Wissenssammlung zu Deflektometrie findet sich in [Sch04b].

2.1.3 Pixelpositionsbilder

Es ist also möglich, diejenige Position in Bildschirmkoordinaten zu ermitteln, die der Schnittpunkt eines an der Objektoberfläche reflektierten Kamerasichtstrahls mit dem Monitor hat. Umgekehrt heißt das, dass jeder Bildpunkt der Kamera einen Sichtstrahl erzeugt. Verwendet man das Lochkameramodell, so geht dieser einerseits durch das optische Zentrum der Kamera und andererseits vereinfacht durch den Mittelpunkt des Sensors, der den Grau- beziehungsweise Farbwert des Pixels liefert. Somit ist der Sichtstrahl definiert. Die meisten Sichtstrahlen treffen die Oberfläche des zu validierenden Bauteils und werden von ihr reflektiert. Einige Sichtstrahlen jedoch, vor allem die, die zu Pixeln im Randbereich des Kamerabildes gehören, treffen das Bauteil nicht und werden somit auch nicht reflektiert. Von nun an werden nur noch die an der Oberfläche reflektierten Sichtstrahlen betrachtet. Sie lassen sich wiederum in zwei Kategorien einteilen. Es ist möglich, dass ein an der Oberfläche reflektierter Sichtstrahl die Bildschirmoberfläche mit dem Muster trifft. Der Schnittpunkt von Bildschirmoberfläche und reflektiertem Sichtstrahl lässt sich dann im Screen-Koordinatensystem (kurz: S-KoSy), einem zweidimensionalen Koordinatensystem, das in der Bildschirmebene liegt, angeben. Solch ein Schnittpunkt wird als Pixelposition (kurz: Pixpos) bezeichnet. In die zweite Kategorie von reflektierten

Sichtstrahlen fallen diejenigen, die den Monitor nicht treffen und sozusagen an ihm „vorbeischießen“. Für sie kann folglich keine Pixelposition berechnet werden.

Eine Pixelposition ist also eine zweidimensionale Koordinate, die einem Pixel des Kamerabildes zugeordnet wird. Damit lassen sich Pixpos-Bilder einführen. Anstatt einem einzelnen Kamerapixel seine Pixelposition zuzuordnen, speichert ein Pixpos-Bild für jeden Punkt eines Kamerabildes seine korrespondierende Pixelposition. Pixpos-Bilder sind also eine Zuordnung

$$f^{img} : (\mathbb{Z}_{[0, c_{hori}-1]} \times \mathbb{Z}_{[0, c_{vert}-1]}) \longrightarrow \left\{ \perp, (\mathbb{R}_{[0, s_{hori}-1[} \times \mathbb{R}_{[0, s_{vert}-1[}) \right\} \quad (2.1)$$

bei der c_{hori} die horizontale und c_{vert} die vertikale Auflösung der Kamera ist, s_{hori} die horizontale und s_{vert} die vertikale Auflösung des Bildschirms und $\perp$ denjenigen Sichtstrahlen der Kamera zugeordnet wird, die entweder nicht die zu prüfende Oberfläche treffen oder deren Reflexionen am Bildschirm vorbeischießen.

Die Abbildungen 2.3 und 2.4 stellen jeweils die Welt aus Sicht des Bildschirms dar. Trifft

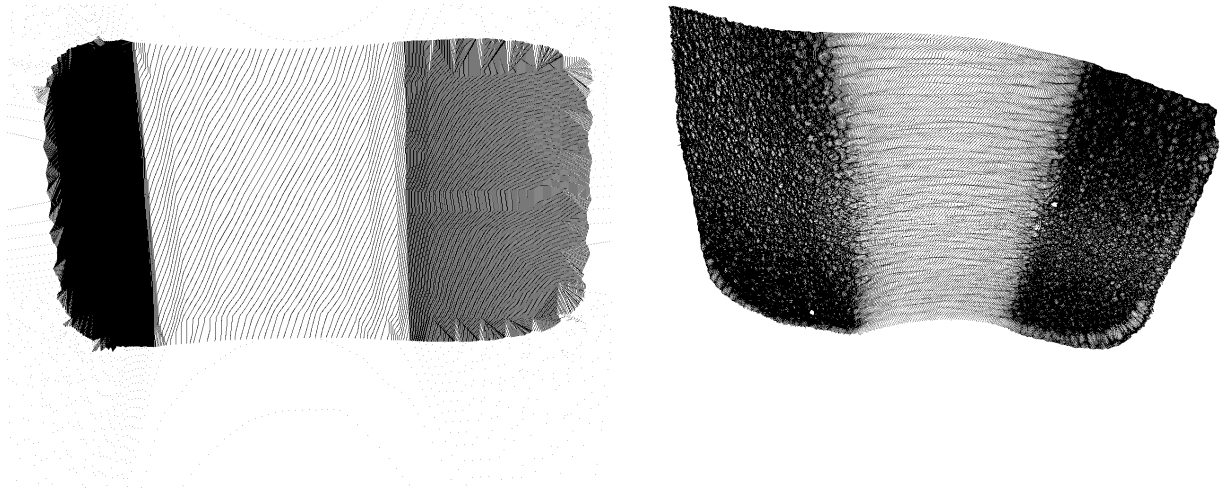


Abbildung 2.3: Dieses Bild stellt den Bildschirm dar. Jeder Sichtstrahl der Kamera, der von der Prüfteiloberfläche reflektiert wird und den Monitor trifft, ist an der entsprechenden Stelle mit einem schwarzen Pixel markiert.

Abbildung 2.4: Im Gegensatz zu Abbildung 2.3 zeigt dieses Bild nicht das Ergebnis einer Simulation, sondern stammt aus einem realen Messprozess. Jeder schwarze Pixel wurde von einem Sichtstrahl der Kamera erzeugt.

ein Sichtstrahl den Monitor, so wird an der entsprechenden Pixelposition ein schwarzes Pixel gesetzt. Bei der zu überprüfenden Komponente handelt es sich in beiden Fällen um einen Tankdeckel.

Es leuchtet sofort ein, dass die Koordinaten der Pixelpositionen von der Position des Bildschirms, der Kamera und des Prüfteiles abhängen. Sind die ersteren beiden Objekte starr und ihre Lage und Ausrichtung bekannt, so hat nur noch das zu inspizierende Objekt Einfluss auf die Pixelpositionen. Diesen Umstand macht man sich für den Rekonstruktionsalgorithmus zu Nutze.

2.2 Algorithmus zur Positionsbestimmung

Nachdem erklärt wurde, was Pixpos-Bilder sind, kann der Rekonstruktionsalgorithmus zur Positionsbestimmung präsentiert werden.

2.2.1 Simulation

Im Kern löst der Algorithmus das Rekonstruktionsproblem durch Simulation. Das bedeutet, dass am Computer die Realität möglichst exakt nachmodelliert wird. Genauer gesagt soll die reale deflektometrische Messung simuliert werden, deren Ergebnisse in vollem Umfang zur Verfügung stehen. Das hat zur Folge, dass die drei essentiellen Objekte eines Deflektometriesystems in die Virtualität übertragen werden müssen: Der Monitor, der das sinusförmige Streifenmuster anzeigt, das zu prüfende Bauteil und die Kamera, die die Szene aufnimmt.

Wesentlicher Vorteil einer Simulation ist es, dass sich ohne viel Aufwand sämtliche Parameter der Szene verändern lassen. Von zentralem Interesse sind die sechs Parameter, die die Transformation beschreiben, die das CAD-KoSy in das W-KoSy überführen. Diese sollen vom Rekonstruktionsalgorithmus ermittelt werden.

Nur diese sechs gesuchten Parameter werden in der Simulation variabel gehalten. Alle anderen Parameter werden von der Realität in die Simulation übertragen. Für die einzelnen Komponenten des Deflektometriesystems bedeutet dies:

- *Bildschirm*: Er wird einerseits durch seine Position im Raum beschrieben, andererseits durch die Eigenschaften des Panels, also des Bereichs, der den Monitorinhalt anzeigt. Zwei senkrecht aufeinander stehende Vektoren, die die Bildelebene vorgeben und die Position des Ursprungs des S-KoSy, meist die linke obere Ecke des Monitors, charakterisieren die Position. Horizontale und vertikale Auflösung sowie Breite und Höhe eines einzelnen Bildschirmpixels definieren das Panel.
- *Geometrie*: Die ideale Spezifikation der Form des Bauteils muss vorliegen. Da auf einen CAD-Entwurf zurückgegriffen werden kann, ist dies der Fall. Wie allerdings diese Vorlage transformiert werden muss, um mit dem real gefertigten Bauteil übereinzustimmen, muss noch beantwortet werden.
- *Kamera*: Sie wird durch die intrinsischen und extrinsischen Kameraparameter modelliert. Bei den intrinsischen Kameraparametern handelt es sich um einen Satz von Werten, die die Projektion eines Punktes im Raum in die Retinaebene der Kamera ausdrücken. Lage der Kamera und ihre Blickrichtung werden dagegen durch die extrinsischen Kameraparameter dargestellt.

Bildschirm, Form der Geometrie und Kamera sind also in der Simulation fest. Ihre Beschreibungen stammen entweder aus Kalibrierungsprozessen oder aus dem CAD-Entwurf. Nur die Position und Lage der Geometrie ist variabel. Ziel der Simulation ist es nun, durch Ausprobieren diejenigen sechs Transformationsparameter zu finden, die die Geometrie so rotieren und translieren, dass die gesamte virtuelle Modellierung möglichst exakt mit der realen Messung übereinstimmt.

Dazu wird eine passende Metrik benötigt, die den Unterschied zwischen Realität und

Simulation misst. Auch das Ausprobieren, mit dem man die sechs gesuchten Transformationsparameter finden will, soll möglichst intelligent gestaltet sein. Sowohl beim Finden einer passenden Metrik, als auch für das intelligente Ausprobieren können Pixpos-Bilder hilfreich sein.

2.2.2 Vektordifferenzen in Pixpos-Bildern

Gesucht ist eine Metrik, die bewertet, wie weit die Transformation in der Simulation von der Transformation in der Wirklichkeit entfernt ist. Ein direkter Vergleich der sechs virtuellen mit den sechs realen Transformationsparametern ist nicht möglich, da die reale Rotation und Translation gesucht, also unbekannt ist.

Stattdessen werden Pixpos-Bilder miteinander verglichen. Leicht erkennt man, dass zwei unterschiedliche Transformationen der Geometrie zu zwei unterschiedlichen Pixpos-Bildern führen. Zwar sind Fälle denkbar, in denen sich die Geometrie invariant gegenüber bestimmten Transformationen verhält, aufgrund der Komplexität der in der Praxis vorkommenden Geometrien und der Tatsache, dass bei solchen Konstellationen das vom Rekonstruktionsalgorithmus gefundene Ergebnis immer noch gültig ist, können solche Fälle aber ignoriert werden. Nichtsdestotrotz seien der Vollständigkeit halber zwei simple invariante Formen erwähnt:

- Kugeln können beliebig um ihr Zentrum rotiert werden, es ergibt sich immer eine gleiche Kugel, die den selben Strahl, der auf sie trifft, in die selbe Richtung reflektiert.
- Transliert man die xy-Ebene in x- oder y-Richtung, so hat dies ebenfalls keinerlei Auswirkung auf deren Reflexionsverhalten.

Zwei unterschiedliche Transformationen führen also in der Regel zu zwei unterschiedlichen Pixpos-Bildern. Man kann vermuten, je ähnlicher die beiden Pixpos-Bilder sind, desto ähnlicher sind auch die beiden Transformationen, die zu ihnen geführt haben. Auf jeden Fall ist es so, dass zwei gleiche Transformationen auch zu zwei gleichen Pixpos-Bildern führen. Natürlich immer unter der Annahme, dass nur die Transformationen der Geometrie variabel sind und das Deflektometriesystem in beiden Fällen gleich ist. Schafft man es, durch Simulation ein Pixpos-Bild zu finden, das dem real aufgenommenen Pixpos-Bild entspricht, so stimmen die virtuellen Transformationsparametern, die zum simulierten Pixpos-Bild führen, mit den gesuchten realen Transformationsparametern überein.

Die Frage, ob zwei Transformationen gleich sind, ist also in der Praxis äquivalent zu der Frage, ob zwei Pixpos-Bilder gleich sind. Quantitativ lässt sich eine derartige Ähnlichkeit durch die normierte Summe der quadratisch gewichteten Abstände der Pixelpositionen beider Pixpos-Bilder berechnen (siehe Formel 2.2). Im Einzelnen bestimmt man den quadratisch gewichteten, euklidischen Abstand zweier Pixelpositionen, die durch ein und den selben Sichtstrahl der Kamera hervorgerufen werden, jedoch das eine Mal aus dem realen und das andere Mal aus dem virtuellen Pixpos-Bild stammen. Vektordifferenz ist in diesem Kontext ein gleichwertiger Begriff für euklidischen Abstand zwischen zwei Pixelpositionen. Für alle Pixelpositionen in den beiden Pixpos-Bildern wird dieser Abstand berechnet. Natürlich ist dies nur möglich, wenn der selbe reflektierte Sichtstrahl sowohl in der Realität als auch in der Simulation den Bildschirm trifft und dadurch beide Male

eine Pixelposition erzeugt. Aufgrund der unterschiedlichen Transformationen in der Realität und in der Simulation muss dies keineswegs der Fall sein. Solche Pixelpositionen, für die das dennoch zutrifft, werden im Folgenden auch als gemeinsame Pixelpositionen bezeichnet. Somit ist die Anzahl gemeinsamer Pixelpositionen in der Regel kleiner als die Kameraauflösung.

Abbildung 2.5 zeigt zwei übereinandergelegte Pixpos-Bilder, die durch die selbe Geometrie, jedoch mit unterschiedlichen Transformationen erzeugt wurden. Die Vektordifferenzen sind mit Linien eingezeichnet. Pixelpositionen, die durch den selben Sichtstrahl erzeugt wurden, sind also durch diese Linie miteinander verbunden.

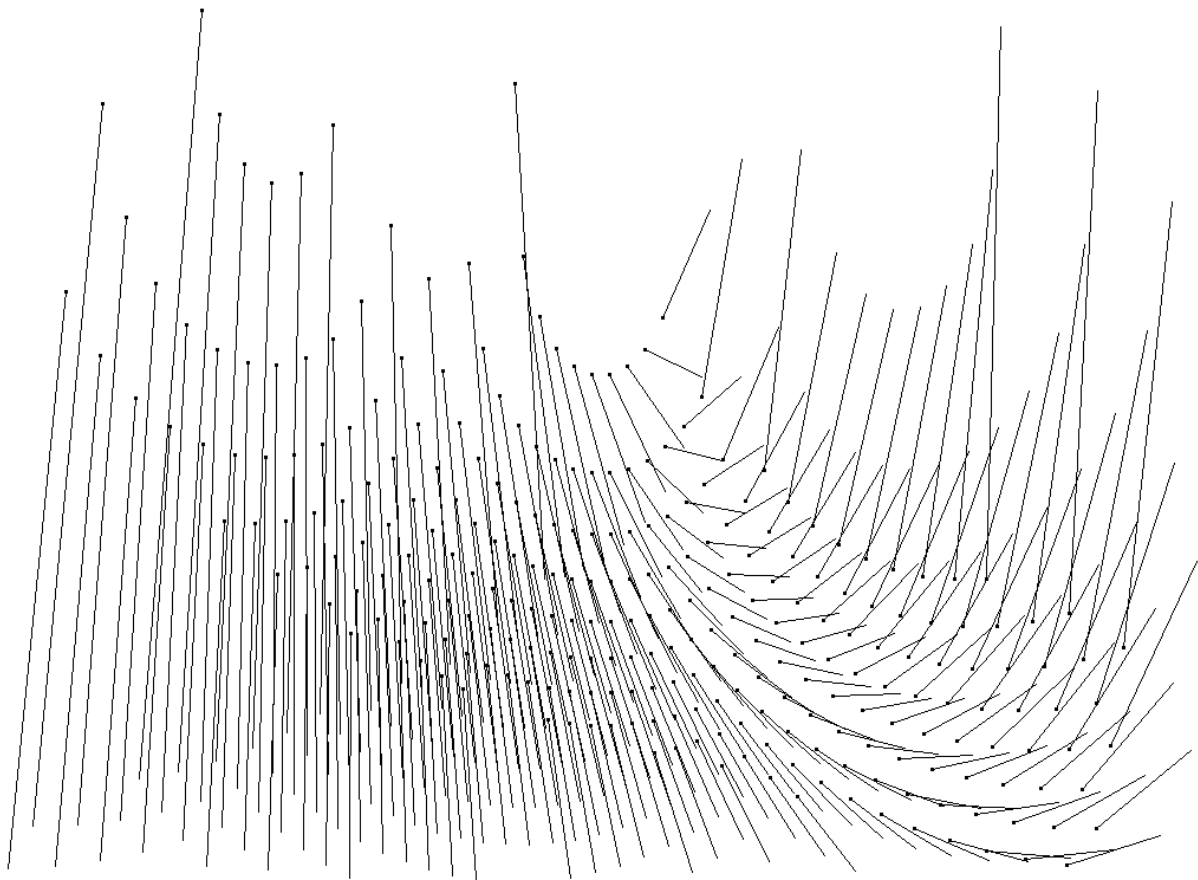


Abbildung 2.5: Die Linien verbinden jeweils zwei Pixelpositionen, die durch ein und denselben Sichtstrahl der Kamera erzeugt wurden. Diejenigen Enden der Linien, die mit einem kleinen Quadrat markiert sind, entsprechen denjenigen Koordinaten im S-KoSy, die die Pixelpositionen im real aufgenommenen Pixpos-Bild haben. Im Gegensatz dazu entsprechen die nicht markierten Enden der Linien solchen Pixelpositionen, die die simulierte Geometrie mit dem zufällig gewählten Transformationsparametersatz erzeugt. Somit entspricht eine Linie einer Vektordifferenz.

2.2.3 Nichtlineare Optimierung

Um nun in der Simulation das gleiche Pixpos-Bild wie in der Realität zu erzeugen, muss die normierte Summe aller quadratisch gewichteten, euklidischen Abstände von gemeinsamen realen und simulierten Pixelpositionen minimiert werden. Dabei handelt es sich um ein sechsdimensionales, nichtlineares Optimierungsproblem. Sechsdimensional deswegen, weil die Pixelpositionsabstände von den sechs Transformationsparametern abhängen.

Es existieren zahlreiche Methoden, nichtlineare Zielfunktionen zu minimieren. Allen Verfahren gemein ist, dass sie iterativ arbeiten, sie sich also schrittweise einem Optimum nähern. Abbildung 2.6 zeigt die Zwischenschritte einer solchen nichtlinearen Optimierung (kurz: NLO). Gut zu erkennen sind die Abstände der Pixelpositionen, die von Iterationsschritt zu Iterationsschritt kleiner werden.

Wichtig ist zu erwähnen, dass die Summe der Vektordifferenzen normiert werden muss. Hierbei wird die Summe aller quadratisch gewichteten, euklidischen Abstände durch die Anzahl der Pixelpositionen geteilt, die sowohl im simulierten als auch im realen Pixpos-Bild vorkommen. Auf diese Weise verhindert man, dass das nichtlineare Optimierungsverfahren eine Transformation bestimmt, die das Bauteil in eine Position überführt, in der keinerlei gültige Pixelpositionen erzeugt werden. Befände sich die Komponente hinter der Kamera, wäre dies ein solcher Fall, in der die Vektordifferenzsumme aufgrund keines Summanden zwar 0 wäre, die rekonstruierte Transformation aber keinesfalls der realen entspricht.

Somit lautet die zu minimierende Zielfunktion $f^{obj} : \mathbb{R}^6 \longrightarrow \mathbb{R}_0^+$

$$f^{obj}(\phi_1, \phi_2, \phi_3, t_1, t_2, t_3) = \begin{cases} \frac{\sum_{i=0}^{c_{\text{horiz}}-1} \sum_{j=0}^{c_{\text{vert}}-1} \mathbb{1}_{(i,j) \in A} \|f_{\text{simul}}^{img}(i,j) - f_{\text{real}}^{img}(i,j)\|^2}{|A|} & , \text{ für } |A| > 0 \\ \infty & , \text{ für } |A| = 0 \end{cases} \quad (2.2)$$

wobei c_{horiz} die horizontale und c_{vert} die vertikale Auflösung der Kamera ist. $f_{\text{simul}}^{img}(i,j)$ entspricht der simulierten und $f_{\text{real}}^{img}(i,j)$ der realen Pixelposition des Sichtstrahls mit Index (i,j) . $A = \{(i,j) | f_{\text{simul}}^{img}(i,j) \neq \perp, f_{\text{real}}^{img}(i,j) \neq \perp, (i,j) \in \mathbb{Z}_{[0, c_{\text{horiz}}-1]} \times \mathbb{Z}_{[0, c_{\text{vert}}-1]}\}$ enthält die Indizes aller gültigen, gemeinsamen Pixelpositionen. $\mathbb{1}_{(i,j) \in A}$ ist die Indikatorfunktion

$$\text{mit } \mathbb{1}_{(i,j) \in A} = \begin{cases} 1 & , \text{ falls } (i,j) \in A \\ 0 & , \text{ falls } (i,j) \notin A \end{cases}.$$

Um die Funktion 2.2 zu minimieren wird, also ein nichtlineares Optimierungsverfahren benötigt. Die Theorie nichtlinearer Optimierung soll nicht Gegenstand dieser Arbeit sein. Stattdessen sei auf [Spe93] verwiesen. Während der Implementierung des Rekonstruktionsalgorithmus, hat sich das nichtlineare Optimierungsverfahren von Nelder und Mead [NM65] als das zuverlässigste herausgestellt. Dabei handelt es sich um einen Simplexalgorithmus. Zwar benötigt es vergleichsweise viele Iterationsschritte, bis ein Optimum der Zielfunktion gefunden wird, dafür wird im hier verwendeten Szenario das globale Optimum in deutlich mehr Fällen gefunden als von anderen Verfahren. Zu diesen alternativen Optimierungsprozeduren zählen die Methode von Broyden, Fletcher, Goldfarb und Shanno [Bro70, Fle70, Gol70, Sha70] (kurz: BFGS), die Nonlinear Conjugate Gradient-Methode [FP63, FR64] (kurz: NCG) sowie eine Modifikation der Sattelpunktnäherungs-Methode [DZ93] und der Gauß-Newton-Algorithmus.

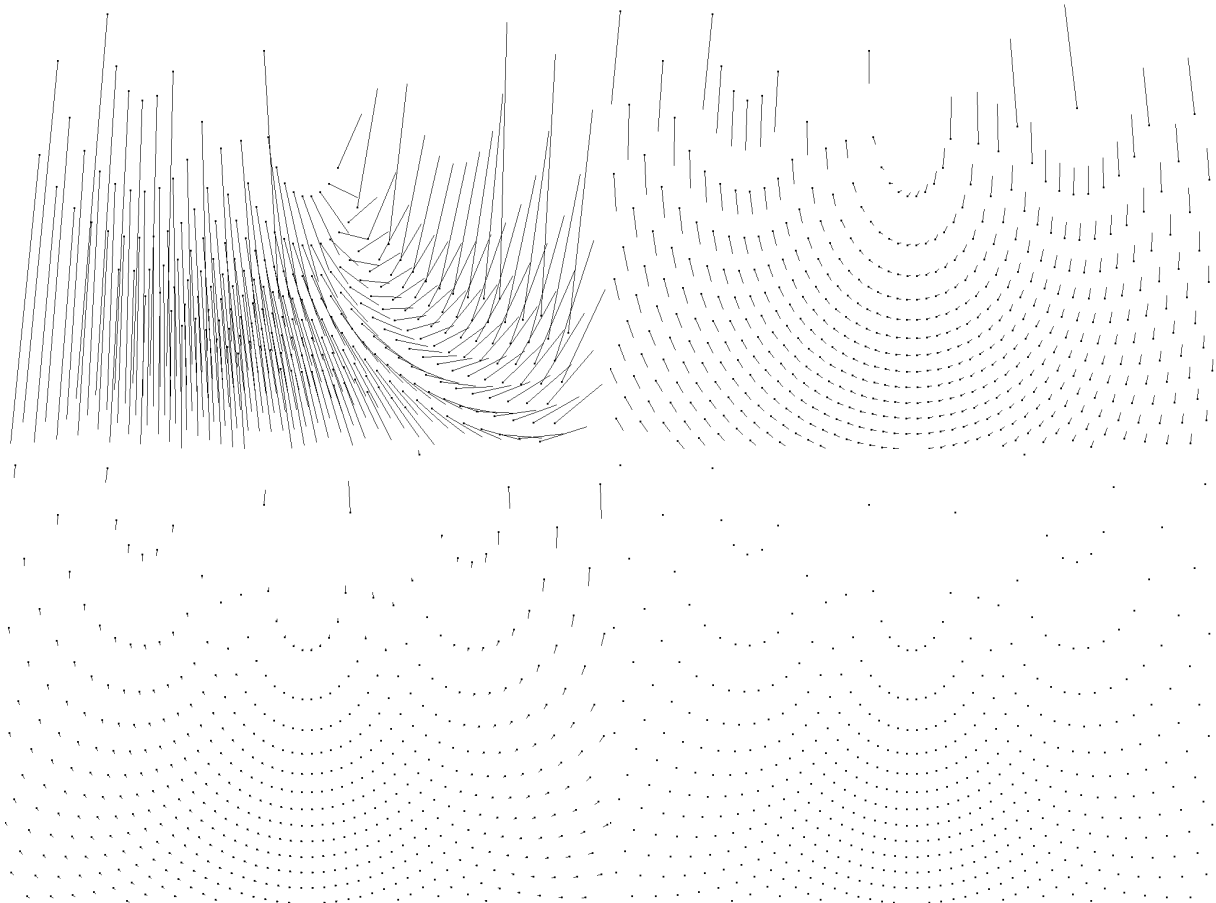


Abbildung 2.6: Diese vier Bilder zeigen allesamt die Vektordifferenzen zweier Pixelpositionsbilder. Die mit einem kleinen Quadrat markierten Enden der Linien sind die Koordinaten der Pixelpositionen, die derjenige Parametersatz erzeugt, der durch Optimierung gefunden werden soll. Im Gegensatz dazu werden die anderen Enden der Linien durch diejenigen Pixelpositionen vorgegeben, die sich aus dem aktuellen Zwischenergebnis der Optimierung ergeben. Man kann erkennen, dass die Linien schrumpfen, die mittlere Vektordifferenz also immer kleiner wird. Im letzten Bild rechts unten stimmt das durch Optimierung erhaltene Pixpos-Bild mit dem Zielbild exakt überein.

2.3 Modifikation des Algorithmus zur Verbesserung der Rekonstruktionsgüte

2.3.1 Kritik am bisherigen Rekonstruktionsalgorithmus

Im vorherigen Abschnitt wurde die Vermutung geäußert: Je ähnlicher zwei Pixpos-Bilder werden, desto ähnlicher werden auch die Transformationsparameter. Diese Annahme kann in der Praxis im Allgemeinen nicht bestätigt werden. Abbildung 2.7 zeigt dafür ein Gegenbeispiel. Somit bleibt nur die Aussage bestehen, dass zwei gleiche Parametersätze zwei gleiche Pixpos-Bilder liefern.

Zwar bedeutet das nicht automatisch, dass eine kleinere normierte Summe von quadra-

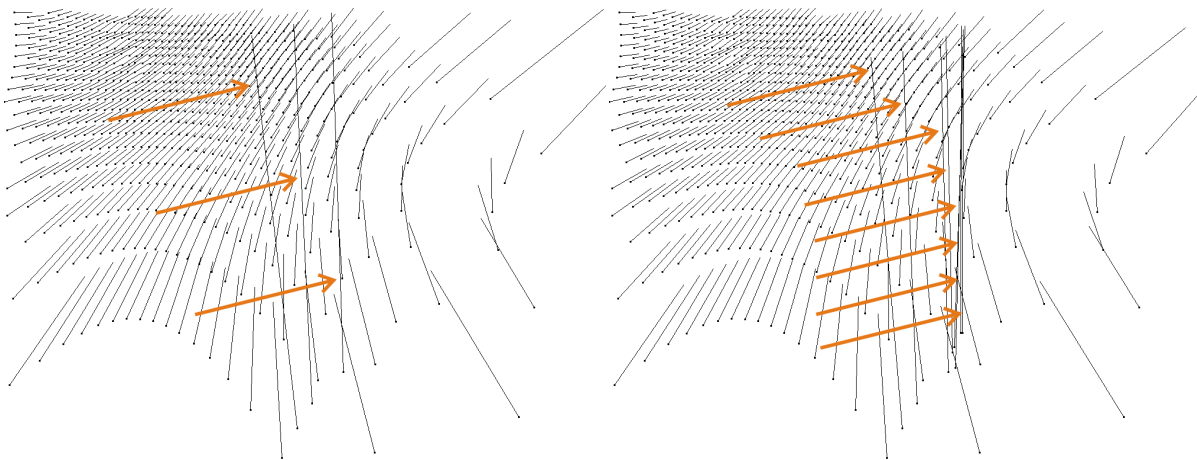


Abbildung 2.7: Zu sehen sind zwei Vektordifferenzbilder. Die gesuchten Transformationsparameter lauten in diesem Fall $T_0 = (\phi_1, \phi_2, \phi_3, t_1, t_2, t_3) = (0, 0, 0, 0, 0, 100)$, das CAD-KoSy entspricht also hier bis auf eine Translation in z-Richtung dem W-KoSy.

Im linken Bild sieht man dasjenige Vektordifferenzbild, das sich aus den Parametersätzen T_0 (wieder markiert durch kleine Quadrate) und $T_1 = (10, 5, 15, -10, -5, 85)$ ergibt. Die normierte Summe der quadratisch gewichteten Abstände der Pixelpositionen liegt gerundet bei 4259.

Mit den Parametersätzen T_0 und $T_2 = (9.2, 4.6, 13.8, -9.2, -4.6, 86.2)$ ergibt sich das rechte Vektordifferenzbild. Der Fehler beträgt gerundet 4300. Obwohl $\|T_2 - T_0\|^2 < \|T_1 - T_0\|^2$ ($\|\cdot\|$ entspricht der euklidischen Norm) ist $f^{obj}(T_2) > f^{obj}(T_1)$.

Die für die Erstellung der Pixpos-Bilder gewählte Geometrie ist übrigens die Utah Teekanne (siehe Abbildung 2.9). Die mit den orangen Pfeilen markierten „Ausreißer“ in den Vektordifferenzbildern stammen von Sichtstrahlen der Kamera, die im Szenario von T_0 den Körper der Teekanne, in den Szenarien T_1 und T_2 jedoch deren Henkel treffen.

tisch gewichteten Vektordifferenzen nicht auch zu näher an den gesuchten Werten liegen Transformationsparametern führen kann, jedoch lässt eine solche Beobachtung die Befürchtung aufkommen, dass die Zielfunktion 2.2 zahlreiche insignifikante Nebenoptima

besitzt. Allen nichtlinearen Optimierungsverfahren gemein ist, dass sie ein Optimum finden, man jedoch keine Garantie hat, dass es sich bei diesem Optimum um das globale Minimum oder Maximum der Zielfunktion handelt, oder doch nur um ein lokales Extremum. Gesucht ist aber das globale Optimum, weshalb eine Zielfunktion möglichst wenige Nebenoptima besitzen sollte.

Untermauert wird die Vermutung der zahlreichen insignifikanten Nebenoptima von praktischen Testläufen des in Abschnitt 2.2 präsentierten Algorithmus. Dieser terminiert zwar, jedoch ist der berechnete Fehler zwischen gesuchtem und durch Simulation und nichtlineare Optimierung gefundenem Parametersatz oft weit von dem bekannten Minimum 0 entfernt. Lediglich in etwa 10% der Fälle kann die reale Position rekonstruiert werden.

Um eine befriedigendere Rate erfolgreicher Rekonstruktionen zu erhalten, gibt es zwei Möglichkeiten. Entweder es lässt sich eine andere Zielfunktion als die normierte Summe der quadratisch gewichteten Vektordifferenzen in den Pixpos-Bildern finden, die weniger lokale Extrema besitzt, oder das bisherige Verfahren lässt sich so modifizieren, dass die lokalen Minima „ausgebügelt“ werden können. Durch eine Modifikation kann das bisherige Verfahren wesentlich verbessert werden. Bevor jedoch diese Änderung erläutert werden kann, ist es sinnvoll, sich Gedanken darüber zu machen, woher die Nebenminima stammen könnten.

2.3.2 Ursache der mangelhaften Rekonstruktionsgüte

Will man vermeiden, dass ein nichtlineares Optimierungsverfahren, anstatt beim globalen, in einem lokalen Extremum terminiert, muss dies explizit behandelt werden. Dafür gibt es Verfahren wie Simulated Annealing [KGV83, Cer85], die temporär einen größeren Fehler der Zielfunktion zulassen. Das Studium der Zielfunktion beziehungsweise die Analyse der Simulation kann jedoch ebenfalls befriedigende Ergebnisse liefern.

Man betrachtet nun immer zwei unterschiedliche Transformationsparametersätze, die dazu führen, dass der Sichtstrahl der Kamera von der transformierten Geometrie in zwei unterschiedliche Richtungen reflektiert wird und somit, falls er den Monitor schneidet, zwei unterschiedliche Pixelpositionen generiert. Bei den beiden betrachteten Transformationsparametersätzen handelt es sich einerseits um die zu rekonstruierenden Werte und andererseits um diejenigen sechs Zahlen der Simulation, die die NLO gerade berechnet. Eine Vektordifferenz zwischen zwei Pixelpositionen kann nur dann berechnet werden, wenn es sich bei den beiden Pixelpositionen um gemeinsame Pixelpositionen handelt. Je nach gewähltem Startwert der nichtlinearen Optimierung, erzeugt dieser Startwert ein Pixpos-Bild, in dem es Pixelpositionen gibt, die im auf der gesuchten Transformationsspezifikation aufbauenden Pixpos-Bild keine gemeinsame Entsprechung haben. Umgekehrt kann ein aus den gesuchten Parametern erstelltes Pixpos-Bild Pixelpositionen enthalten, die sich im auf den Startwerten basierenden Pixpos-Bild nicht wiederfinden.

Je ähnlicher sich die beiden Pixpos-Bilder aus realer Messung und Simulation im Zuge der NLO werden, desto mehr gemeinsame Pixelpositionen gibt es. Interessant ist der Fall zwischen zwei Iterationsschritten, wenn im zweiten Iterationsschritt eine gemeinsame Pixelposition hinzu kommt, die es im ersten Iterationsschritt noch nicht gab. Abbildung 2.8 illustriert einen solchen Fall.

Es fällt auf, dass neu hinzukommende gemeinsame Pixelpositionen einen relativ großen Beitrag zur Summe der quadratisch gewichteten Vektordifferenzen beitragen. Eine Be-

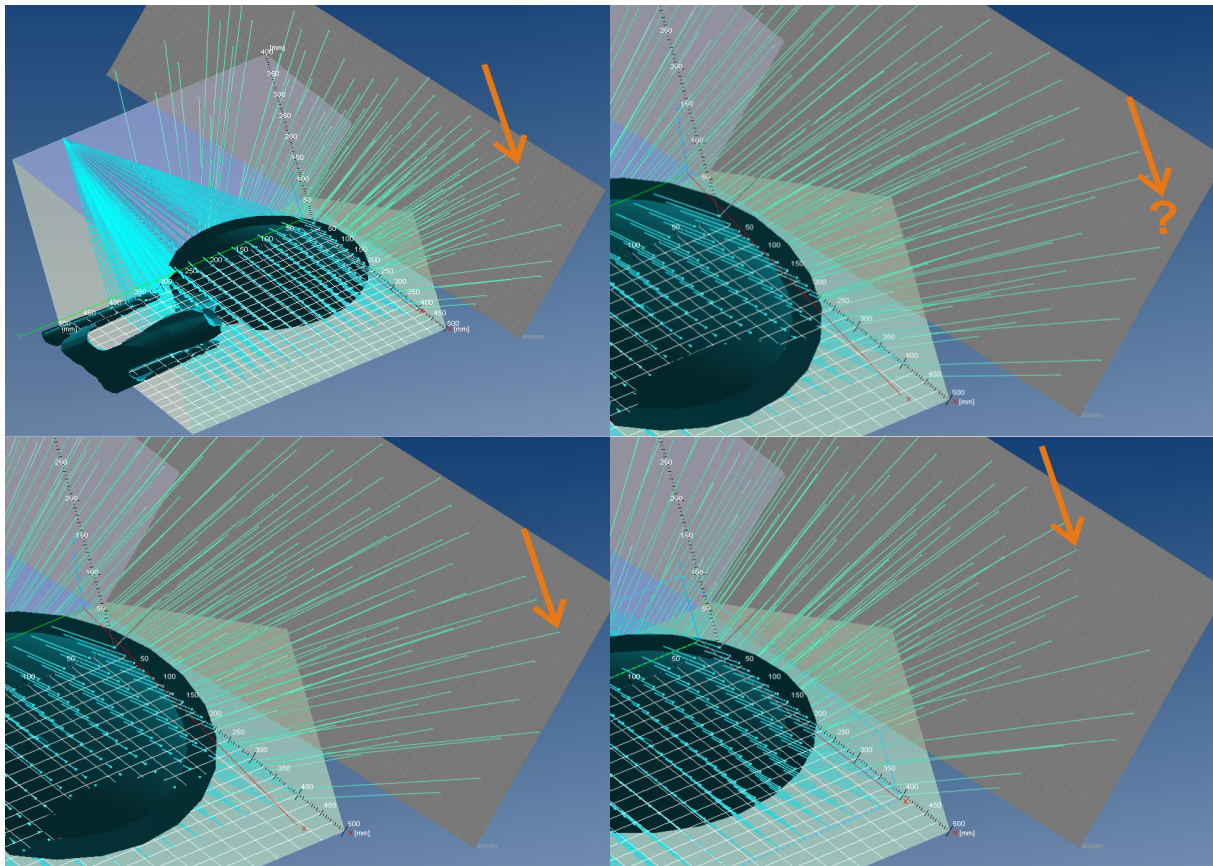


Abbildung 2.8: Links oben sieht man die Konstellation, die rekonstruiert werden soll. Der durch den orangenen Pfeil markierte Sichtstrahl wird genauer betrachtet. Im Bild rechts oben befindet sich die Geometrie in einer Lage, in der sie zwar den gekennzeichneten Sichtstrahl reflektiert, dieser den Bildschirm aber nicht trifft. Rotiert man die CAD-Vorlage nur minimal, so kann man in der Grafik links unten sehen, dass der betrachtete Sichtstrahl nun zu einer Pixelposition führt, die jedoch noch weit von der Pixelposition im Bild rechts unten entfernt ist. Es zeigt vergrößert die Ausgangssituation links oben, die rekonstruiert werden soll.

gründung dafür findet sich bereits in Abbildung 2.7. Dort wird die bekannte Utah-Teekanne (siehe Abbildung 2.9) als Vorlage für die verwendete Geometrie eingesetzt. Aufgebaut ist diese Teekanne aus dem runden Bauch, einem Henkel und dem Ausgießer. Die letzteren beiden Teile haben keinen glatt gekrümmten Übergang zum Kannenbauch, die Grenze ist also eine harte Kante. Wählt man zwei unterschiedliche Transformationsparametersätze, so ist es denkbar, dass ein und der selbe Sichtstrahl einmal den Bauch und einmal den Henkel der Kanne trifft, an ihnen reflektiert wird, den Bildschirm schneidet und man somit zu zwei unterschiedlichen Pixelpositionen gelangt. Aufgrund des abrupten Übergangs von Henkel, Kannenbauch und Ausgießer ist davon auszugehen, dass die beiden Reflexionen des Sichtstrahls sehr unterschiedlich ausfallen, die Vektordifferenz der gemeinsamen Pixelpositionen also sehr groß wird. Bezogen auf die zu minimierende Zielfunktion bedeutet das, dass der Abstand dieser beiden gemeinsamen Pixelpositionen einen großen

Beitrag zum Gesamtfehler leistet. Die Auswertung der Zielfunktion kann also größer werden als im vorherigen Iterationsschritt, obwohl man nun einen Sichtstrahl miteinbezieht, der auch in demjenigen Szenario eine Pixelposition erzeugt, dessen Transformationsparametersatz gesucht ist.

Würde man es schaffen, den Einfluss der eben beschriebenen Sichtstrahlen zu verhindern oder zumindest zu reduzieren, so dürfte dies auch einen positiven Einfluss auf die Zahl und Ausprägung der insignifikanten Nebenoptima haben.



Abbildung 2.9: Die Utah-Teekanne ist ein beliebtes Beispiel in der 3D-Computergrafik, da sie zahlreiche Eigenschaften aufweist, die in Tests von Interesse sind. Sie besitzt sowohl konvexe als auch konkave Abschnitte sowie harte und weiche Übergänge. Teilweise verdeckt sie sich selbst und der Henkel generiert ein Loch im Modell.

2.3.3 Elimination der Ursache

Gesucht ist also ein Verfahren, das den Einfluss von zusätzlichen gemeinsamen Pixelpositionen reduziert oder sogar eliminiert.

Dabei ist folgender Umstand sehr hilfreich. Vergleicht man im Kamerabild diejenigen Bildpunkte, die einerseits zu einer Pixelposition führen (im Folgenden als weiße Bildpunkte bezeichnet) und andererseits dies nicht tun (im Folgenden als schwarze Bildpunkte bezeichnet), stellt man fest, dass weiße und schwarze Bildpunkte Regionen bilden. Schwarze und weiße Bildpunkte sind also nicht fein gemischt, sondern bilden jeweils relativ große Flächen. Betrachtet man zwei Kamerabilder zwischen zwei Iterationsschritten, bei denen gemeinsame Pixelpositionen in den beiden Pixpos-Bildern dazukommen, sieht man, dass die neuen gemeinsamen Pixelpositionen durch Kamerapixel hervorgerufen werden, die genau im Randbereich zwischen weißen und schwarzen Flächen liegen.

Will man den überproportional hohen Einfluss erstmalig auftretender gemeinsamer Pixelpositionen verhindern, ist es sinnvoll, die Randbildpunkte im Kamerabild und die durch sie hervorgerufenen Pixelpositionen zu ignorieren. Zu diesem Zweck erzeugt man eine Bildmaske, in der sich zu Anfang alle schwarzen Kamerapixel befinden, die ohnehin keine Pixelpositionen erzeugen. Im nächsten Schritt nimmt man zu dieser Maske all diejenigen weißen Punkte der Kamera hinzu, die genau am Rand zwischen schwarzen und weißen Regionen liegen. Abbildung 2.10 veranschaulicht die Wegnahme einer Randschicht von gültigen Kamerabildpunkten und deren Hinzufügen zur Maske. Dort wird auch der Begriff des Randes näher erläutert.

Der Algorithmus aus Abschnitt 2.2 wird nun unter Verwendung der neuen Erkenntnis-

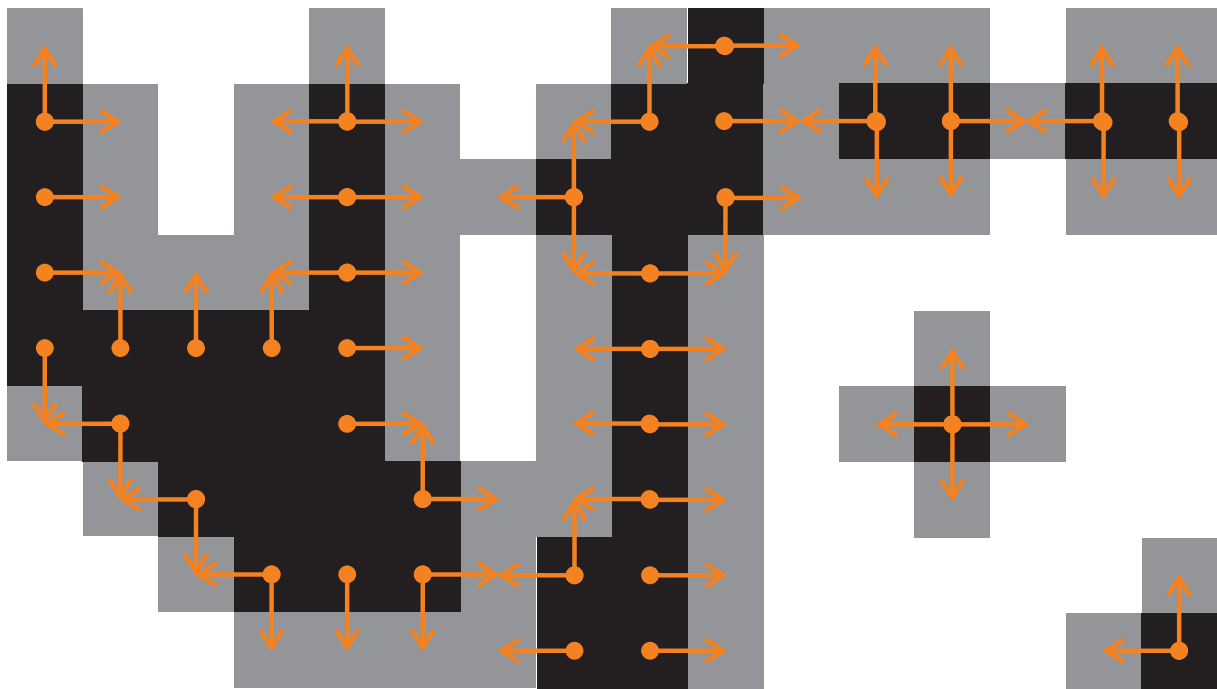


Abbildung 2.10: Dieses Bild zeigt eine Maske aus 16×9 Pixeln und visualisiert die Wegnahme von Randbildpunkten. Die weißen und grauen Quadrate sind im Gegensatz zu den schwarzen Quadraten zunächst keine Elemente der Maske. Es werden all diejenigen Elemente bestimmt, die in direkter Nachbarschaft ein Maskenelement besitzen und selbst noch nicht zur Maske gehören. Direkte Nachbarschaft bedeutet hier Vierer-Nachbarschaft, also dass die Element oben, unten, links und rechts miteinbezogen werden, jedoch keine diagonalen. Solche Maskenkandidaten sind grau eingefärbt. Die orangen Pfeile illustrieren die Nachbarschaften.

se angepasst. Zunächst wird wie gehabt eine nichtlineare Optimierung innerhalb einer Simulation zur Minimierung der normierten Summe der quadratisch gewichteten Vektordifferenzen durchgeführt. Sobald diese erste Optimierung endet und der berechnete Fehler nicht nahe an 0 liegt, muss man davon ausgehen, dass ein lokales Minimum gefunden wurde. Um das lokale Minimum zu verlassen und die Suche nach dem globalen Minimum fortzusetzen, muss zwischenzeitlich ein größerer Fehler der Zielfunktion zugelassen werden. Dies erreicht man durch Ignorieren von Randbildpunkten im Kamerabild,

die ansonsten gemeinsame Pixelpositionen zur Folge hätten, die einen besonders hohen Beitrag zum Ergebnis der Zielfunktion beisteuern. Dafür nimmt man das Kamerabild des letzten Iterationsschritts der ersten Optimierung und erstellt wie oben beschrieben die Maske. Zu ihr fügt man eine beliebige Anzahl von Randschichten hinzu und startet eine zweite NLO unter Verwendung der eben erzeugten Maske. Als Startwerte der zweiten NLO verwendet man das Ergebnis der ersten Minimierung. Solange man kein befriedigendes Ergebnis der NLO erhält, wiederholt man die Bestimmung einer Maske, Wegnahme von Randschichten und Neustart der Optimierung. Somit ergibt sich der folgende Rekonstruktionsalgorithmus:

1. Führe eine reale deflektometrische Messung der Komponente durch, deren Position bestimmt werden soll. Ergebnis dieser Messung ist ein Pixpos-Bild.
2. Übernehme sämtliche Parameter des Deflektometriesystems in eine Simulation. Nur die sechs gesuchten Transformationsparameter sind variabel.
3. Versuche mittels nichtlinearer Optimierung die mittlere Summe der quadratisch gewichteten Abstände zwischen gemeinsamen Pixelpositionen im realen und simulierten Pixpos-Bilder auf 0 zu minimieren.
4. Hat die NLO anstatt des globalen ein lokales Minimum gefunden, so erstelle eine Maske und füge zu ihr eine beliebige Menge von Randschichten hinzu.
5. Starte unter Verwendung dieser Maske und des Ergebnisses der vorherigen Minimierung die NLO erneut.
6. Wiederhole ab Schritt 4 so lange, bis der Fehler der Optimierung mit Ausnahme von numerischen Instabilitäten dem Wert 0 entspricht.

Es hat sich gezeigt, dass zwei bis fünf Wiederholungen der NLO reichen, um eine akzeptable Rekonstruktionsgüte zu erhalten. Der Wert zwei für die Anzahl der weggenommenen Randschichten hat sich als optimal herausgestellt. Mehr ignorierte Randschichten haben sogar einen negativen Einfluss auf die Rekonstruktionsgüte.

Abbildung 2.11 zeigt eine reale Maske und die Wegnahme von Randschichten weißer Kamerapixel. Sie präsentiert außerdem Effekte, die in der Praxis auftreten können.

Vorerst bleibt noch zu erwähnen, dass die Modifikation mit Masken zwar die Rekonstruktionsgüte deutlich erhöht, die Rate erfolgreicher Rekonstruktionen aber nur in Ausnahmefällen Werte nahe 100% erreicht. Die größere Anzahl erfolgreich bestimmter Positionen erkauft man sich außerdem mit einer längeren Laufzeit, da mehrere nichtlineare Optimierungen durchgeführt werden müssen. Wäre dies noch nicht schlimm genug, muss zuletzt auch noch eingeräumt werden, dass es wenige Konstellationen gibt, in denen eine einfache NLO das korrekte Ergebnis liefern würde, die Modifikation mit Masken bei der Rekonstruktion jedoch scheitert. Nichtsdestotrotz verbessert die Erweiterung mit ignorierten Randbildpunkten den ersten Algorithmus signifikant.

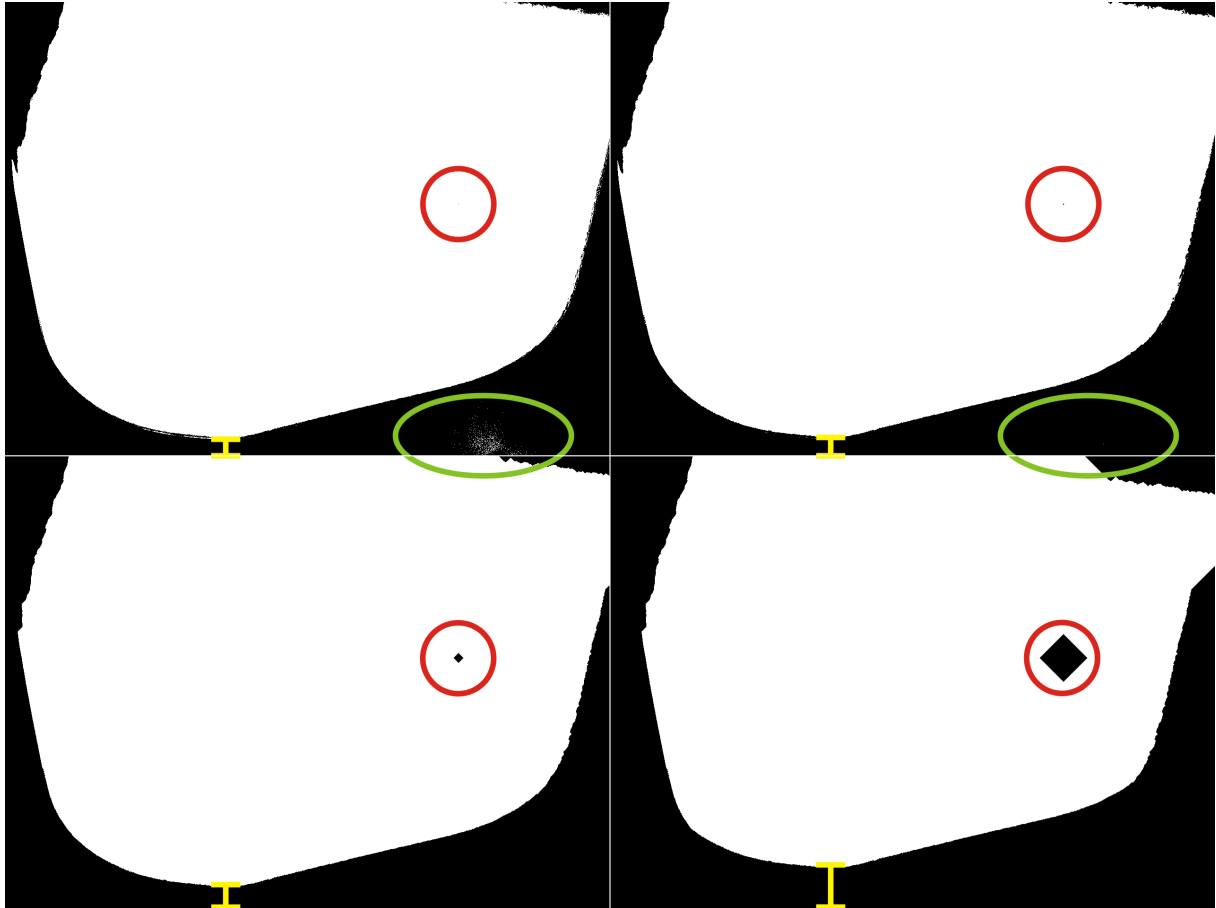


Abbildung 2.11: Anhand dieser vier Masken lässt sich die Wegnahme von Randschichten zur Maske erkennen. Im ersten Bild sieht man eine Maske, die aus einem realen deflektometrischen Messprozess stammt. Die grün markierten weißen Bildpunkte, die zunächst nicht zur Maske gehören und offensichtlich Ergebnis eines Messfehlers sind, verschwinden allesamt, sobald eine Schicht zur Maske hinzugefügt wird. Wie sich die Form der Maske entwickelt, wenn 20 beziehungsweise 50 Schichten zur Maske hinzugefügt werden, zeigen die unteren beiden Bilder. Dies lässt sich auch an den gelben Markierungen erkennen. Ein weiterer Messfehler ist rot eingezeichnet. Während dieser irrtümlich zur Maske hinzugefügte Fehler im ersten Bild nur aus einem einzigen Pixel besteht, wächst er bei 50 hinzugenommenen Schichten zu einem sofort erkennbaren Quadrat.

2.4 Beschreibung des Simulationaufbaus

Entscheidend für die Arbeit des Rekonstruktionsalgorithmus ist die Präzision, mit der das deflektometrische Messsystem in die Simulation übernommen wird. Man muss sich darüber im Klaren sein, dass eine Simulation die Wirklichkeit niemals absolut exakt nachmodellieren kann. Ziel ist es dennoch, den Fehler zwischen Realität und Virtualität so klein wie möglich so halten. Aus diesem Grund befasst sich dieser Abschnitt mit der Modellierung des Deflektometriesystems.

2.4.1 Konventionen für Transformationen

Bevor mit der Beschreibung der Eigenschaften der einzelnen Teile des Deflektometriesystems begonnen wird, muss eine Konvention über die Handhabung der Transformationen festgelegt werden.

Rotation und Translation, die zusammen die gesuchte Transformation bilden, sind nicht

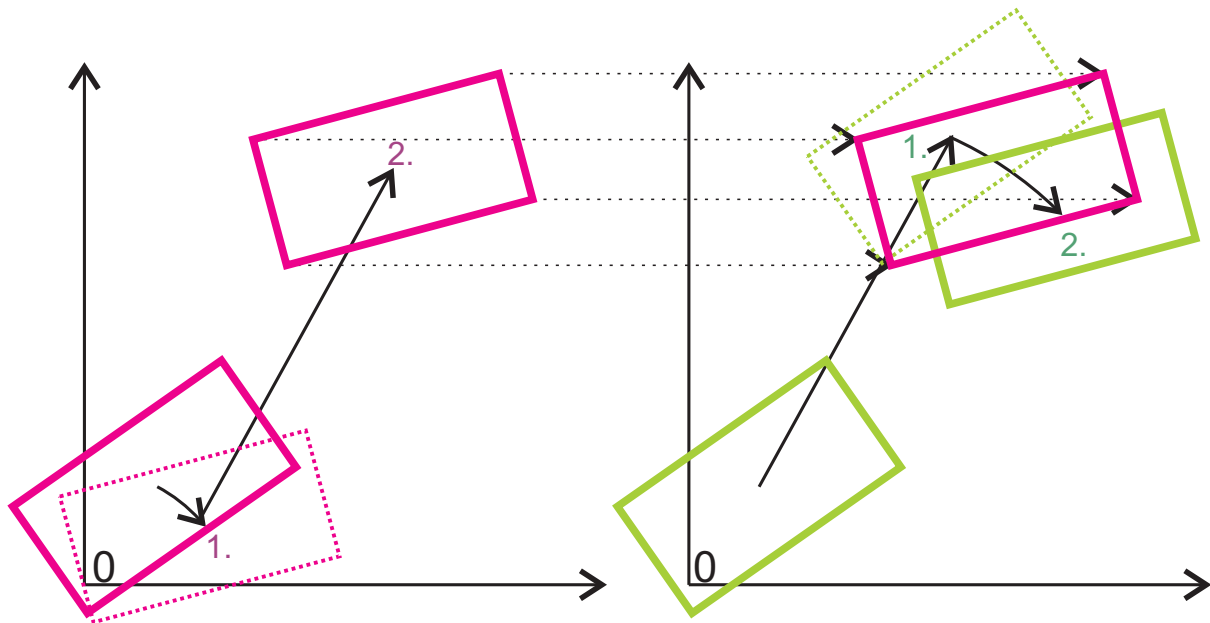


Abbildung 2.12: Im linken Bild (violett) wird eine Transformation durchgeführt. Zunächst wird das Rechteck um 20° um den Ursprung rotiert (1.) und im Anschluss um den eingezeichneten Vektor translatiert (2.). Im rechten Bild (grün) wurde die Reihenfolge von Rotation (2.) und Translation (1.) vertauscht. Sowohl Rotationswinkel als auch Translationsvektor sind gleich geblieben. Der direkte Vergleich zwischen violetterm und grünem Endrechteck verdeutlicht, dass die beiden Endrechtecke nicht gleich sind.

kommutativ. Dies veranschaulicht auf einfache Weise Abbildung 2.12. Das bedeutet, dass eine unterschiedliche Reihenfolge bei der Ausführung von Rotation und Translation zu zwei unterschiedlichen Transformationen führt. Im Rahmen dieser Arbeit wird festgelegt, dass bei Transformationen zuerst die Rotation und anschließend die Translation angewandt wird. Die Rotation um die drei Eulerwinkel ist ebenfalls nicht kommutativ, weshalb

verlangt wird, dass als erstes die Rotation um die x-Achse, im Anschluss um die y-Achse und zuletzt um die z-Achse geleistet wird. Eine derartige Festlegung der Reihenfolge bei der Translation ist nicht notwendig. Komponentenweise Translation ist kommutativ und somit ist es unerheblich, entlang welcher Koordinatenachse zuerst transliert wird.

In [Pis08] und [Owe99] finden sich alle Matrizen, die die einzelnen und hintereinander ausgeführten Rotationen, Translationen und Transformationen ausdrücken.

Rotationen werden wie üblich im Grad- oder Bogenmaß angegeben. Blickt man entlang einer Koordinatensystemachse in Richtung Ursprung, so werden Drehungen im Uhrzeigersinn vollzogen.

Die Einheit für Verschiebungen ist Millimeter.

2.4.2 Modellierung der Kamera

Am anspruchsvollsten ist die Kamera zu simulieren. Die Herausforderung liegt im Finden eines Modells, das es ermöglicht, das reale Linsensystem mit dem Lochkameramodell zu beschreiben. Ein solches Modell muss drei Sorten von Parametern unterstützen:

- Extrinsische Kameraparameter
- Intrinsische Kameraparameter
- Parameter, die die Verzeichnungen des Linsensystems beschreiben

Extrinsische und intrinsische Parameter wurden bereits in Unterabschnitt 2.2.1 umrissen. Eine gesonderte Behandlung von Verzeichnungen ist notwendig, da Kameralinsen in der Realität nie so perfekt gefertigt sind, dass die Beschreibung durch das bloße Lochkameramodell haltbar ist.

Wie schon bei der Theorie der Deflektometrie und der nichtlinearen Optimierung, werden die Grundlagen von Kameramodellen in dieser Arbeit nicht behandelt. Stattdessen sei auf [RPR87] und [Sla80] verwiesen. [Don06] und [Don09] behandeln die Thematik der Kamerakalibrierung, die in diesem Zusammenhang eine wichtige Rolle spielt. Da das Hauptanliegen dieser Arbeit in der Präsentation eines Rekonstruktionsalgorithmus und in dessen Beschleunigung durch programmierbare Grafikkarten liegt, sei anstatt eines tieferen Studiums von Kameramodellen die erwähnte Literatur empfohlen.

2.4.3 Sonstige Komponenten des Deflektometriesystems

Neben der Kamera besteht ein Deflektometriesystem noch aus dem Bildschirm und der Geometrie, die geprüft werden soll. Die Modellierung des Bildschirms ist sehr einfach und wurde bereits in Unterabschnitt 2.2.1 behandelt. Im Gegensatz dazu ist die Modellierung des zu inspizierenden Objekts wesentlich komplexer. Aus diesem Grund beschäftigt sich ein ganzes Kapitel mit dieser Herausforderung. Kapitel 4 behandelt Dreiecksmengen und das STL-Modell.

2.4.4 Inverse Transformationen

Je nach verwendetem Modell zur Speicherung der Oberfläche des zu testenden Bauteils kann es sehr aufwändig und zeitintensiv sein, dessen Position und Lage zu verändern. Bei

den verwendeten Dreiecksmengen ist dies beispielsweise der Fall, da jedes einzelne Dreieck transformiert werden muss.

Anstatt den CAD-Entwurf zu transformieren, kann es sinnvoller sein, den restlichen Messaufbau, also die Kamera und den Bildschirm, invers zu manipulieren. Dabei müssen immer nur zwei Rotationen und Translationen durchgeführt werden, je eine für Kamera und Monitor, unabhängig von der eingesetzten Beschreibung für das Objektideal. Eine solche inverse Transformation bedeutet, dass zunächst die Translation $t^{-1} = -(t_1, t_2, t_3)$ angewandt wird. Anschließend wird die Rotation in umgekehrter Reihenfolge $r^{-1} = -(\phi_3, \phi_2, \phi_1)$ bewerkstelligt.

Es ist zu beachten, dass bei der Kamera nicht nur das optische Zentrum sondern auch die Sichtrichtung angepasst werden muss, also sämtliche extrinsischen Parameter. Da es sich bei der Sichtrichtung nur um einen Vektor handelt, muss sie nur rotiert, nicht jedoch transliert werden.

Selbiges gilt für den Bildschirm. Nicht nur der Ursprung des S-KoSy muss invers transformiert werden, sondern auch das S-KoSy selbst und der Normalenvektor der Monitorebene.

Kapitel 3

Grafikkartenprogrammierung mit CUDA

Nachdem ein zufriedenstellender Algorithmus für die Positionsbestimmung des zu prüfenden Bauteils gefunden wurde, kann damit begonnen werden, dieses Verfahren unter Verwendung programmierbarer Grafikprozessoren zu beschleunigen. Dafür stehen derzeit im Prinzip vier Technologien zur Verfügung:

- Das CUDA-Framework der Firma NVidia.
- Das Stream-Framework der Firma AMD/ATI.
- OpenCL der Khronos Group.
- DirectCompute der Firma Microsoft.

Die vorliegende Arbeit stützt sich voll und ganz auf das CUDA-Framework und behandelt sowohl dessen theoretisches Programmiermodell, als auch dessen Anwendung und Eigenschaften. Auf keine der drei anderen Programmierschnittstellen wird im weiteren Verlauf eingegangen.

3.1 Einführung und Kontext

Ziel dieses Abschnitts ist eine noch wenig technische Einführung in die Grafikkartenprogrammierung. An dieser Stelle soll neben einer kurzen historischen Zusammenfassung über die Entwicklung von Grafikprozessoren auch beschrieben werden, inwieweit sich GPUs von klassischen Hauptprozessoren unterscheiden und welche Möglichkeiten sich daraus ergeben.

3.1.1 Historie der Grafikprozessoren

Etwa zu Beginn der 80er Jahre des 20. Jahrhunderts kann erstmals von eigenständigen Grafikprozessoren auf separaten Steckkarten gesprochen werden. Ihre Leistungsfähigkeit war noch äußerst beschränkt. So lag die Hauptaufgabe damaliger Grafikkarten in der Konvertierung des Inhalts des Videospeichers, der die auf dem Monitor darzustellenden Informationen in digitaler Form enthält, in ein elektrisches Ausgabesignal, das vom Bildschirm verarbeitet werden konnte.

Zehn Jahre später konnte erstmals von GPUs geredet werden, da sie neben der Manipulation einzelner Pixel auch Funktionen zur Verfügung stellten, die das Zeichnen zweidimensionaler, geometrischer Primitive wie Linien ermöglichten. Dadurch war es erstmals möglich, dem Anwender grafische Benutzeroberflächen, wie wir sie auch heute noch kennen, zur Verfügung zu stellen.

Der nächste wichtige Evolutionsschritt vollzog sich Mitte der 90er Jahre. Während bis dahin lediglich 2D-Operationen von GPUs durchgeführt werden konnten, ermöglichte eine Erweiterungskarte der Firma 3dfx, die mittlerweile von der Firma AMD/ATI aufgekauft wurde, auch das Rendering dreidimensionaler Szenen. Eine Darstellung von räumlichen Primitiven, vor allem Dreiecken und deren farbliche Verarbeitung, war zuvor nur durch die CPU möglich. Aufgrund der extrem rechenaufwändigen Operationen, die beim Rendering auftreten, mussten die ersten 3D-Szenen daher noch sehr einfach gehalten werden und wirkten nicht sehr realistisch. Dies änderte sich erstmals durch die Auslagerungen der Berechnungen auf einen speziellen Chip. Später wurden die Rechenwerke für 2D und 3D in einer einzelnen GPU verschmolzen.

Von da an ist die Entwicklung der Grafikprozessoren eng mit der Entwicklung der Schnittstelle DirectX und im Speziellen mit Direct3D verbunden. Diese Schnittstelle ermöglicht es dem Programmierer, die virtuelle Szene zu spezifizieren, von der Grafikkarte berechnen und an den Monitor ausgeben zu lassen. Zwar gibt es neben Direct3D noch weitere Schnittstellen wie OpenGL, die dies ermöglichen, technisch aber hinken diese Alternativen Direct3D meist mehrere Monate hinterher, vor allem im Bereich der 3D-Spiele. Jede neue Version von Direct3D fordert GPUs ein erweitertes Funktionsspektrum ab, was immer realistischere Visualisierungen erlaubt.

Entscheidend für die Grafikkartenprogrammierung war die Einführung von Vertex- und Pixel-Shadern mit DirectX 8. Dabei handelt es sich um kleine Programme, die die bis dahin starre Rendering-Pipeline von 3D-Beschleunigern aufbrachen. Vertex-Shader erlauben die Manipulation von Polygonen in einer flexiblen Weise, die vorher unmöglich war. Selbiges gilt für Pixel-Shader, die verwendet werden können, um Beleuchtungen realistischer zu verwirklichen. Vorher konnte ein Programmierer nur auf die starren Sprachfunktionen von Direct3D zurück greifen und überließ den kompletten Verarbeitungsprozess (siehe

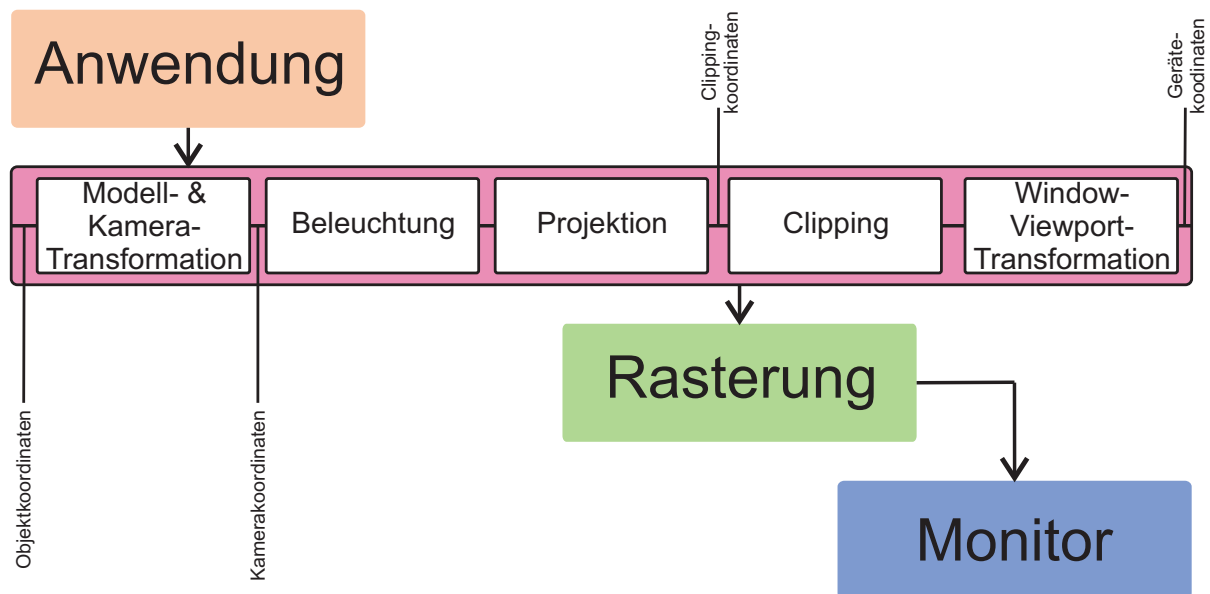


Abbildung 3.1: Die Grafik stellt schematisch die einzelnen Stufen der Rendering-Pipeline dar. Eine einheitliche Definition existiert nicht, da je nach Stand der Technik die Pipeline variieren kann. Diese Vorlage stammt aus [Wik10c].

Abbildung 3.1) der GPU. Mit Vertex- und Pixel-Shadern war es nun möglich, eigene Programme zu schreiben, die über den Sprachumfang von Direct3D hinaus gehen. Die Grundstruktur der Rendering-Pipeline existiert zwar nach wie vor, der Entwickler hat aber nun an zahlreichen Stellen die Möglichkeit, den Ablauf der einzelnen Stufen seinen Erfordernissen anzupassen.

Diese neue Flexibilität wurde mit der Einführung der High Level Shading Language (kurz: HLSL) in DirectX 9 untermauert. Sie macht es dem Programmierer einfacher, Vertex- und Pixel-Shader zu schreiben, als es noch unter DirectX 8 der Fall war, indem sie eine C-ähnliche Syntax zur Verfügung stellt. Wie [PBMH05] zeigt, war es ab diesem Zeitpunkt auch möglich, Grafikkarten für die Abarbeitung von Programmen einzusetzen, die nicht primär für die Darstellung von 3D-Szenen gedacht sind.

Jedoch ist man bei beliebigen Berechnungen darauf angewiesen, dass man auf die durch die Grafikschnittstelle zur Verfügung gestellten Ein- und Ausgabe-Puffer zurückgreifen muss, um die GPU mit zu verarbeitenden Daten zu versorgen und um auf Berechnungsergebnisse zuzugreifen. Bei OpenGL muss die Eingabe über Vertices und Texturen erfolgen. Ergebnisse stünden im Color und Depth Buffer. Man muss sein Problem also „visuell beschreiben“. Diese Speicher unterstützen nur primitivste Datentypen wie `float` oder Vektoren bestehend aus zwei oder vier `float`-Elementen. Selbst definierte Datentypen können über diese Puffer nicht ausgetauscht werden. Erschwerend kommt hinzu, dass Gleitkommaarithmetik auf GPUs bis vor kurzem nicht dem IEEE 754 Standard [IEE85] entsprach.

Den Durchbruch programmierbarer GPUs ebnete die im Februar 2007 von NVidia veröffentlichte CUDA-Architektur. Sie stellt dem Entwickler ein Programmiermodell zur Verfügung, das die Grafikhardware soweit abstrahiert, dass ein Programmieren der Grafikkarte in der Programmiersprache C möglich ist, ohne auf die Besonderheiten der Rendering-

Pipeline achten zu müssen. Hardwaredetails werden hinter klar strukturierten Abstraktionen, die in Abschnitt 3.2 detailliert vorgestellt werden, verborgen. Eine solche Abstraktion ermöglicht die effektive Ausnutzung der Besonderheiten von GPUs.

Dadurch wird aus einer Graphics Processing Unit eine General Purpose Computation on Graphics Processing Unit (kurz: GPGPU).

3.1.2 Eigenschaften von Grafikkarten und Abgrenzung zur CPU

Zweck dieses Abschnitts ist die Beantwortung der Frage, warum die Ausführung von Programmen auf der GPU sinnvoller sein kann, als deren Ausführung auf der CPU. Schließlich muss sich der Programmierer trotz abstrahierenden Programmiermodells erst in dieses einarbeiten, was mit Aufwand verbunden ist. Somit müssen Grafikprozessoren, zumindest in einigen Fällen, einen Mehrwert gegenüber Hauptprozessoren haben.

Um das wesentliche Charakteristikum eines Grafikchips zu erkennen, ist ein Blick auf die klassischen Aufgaben, die dieser bewältigen muss, sehr hilfreich. Beim Durchlaufen der Rendering-Pipeline muss praktisch in jeder Stufe eine große Anzahl von Elementen auf die selbe Art und Weise manipuliert werden. Modelltransformation und Projektion bedeutet, dass auf jeden Vertex der Polygone, aus denen die 3D-Szene in der Regel aufgebaut ist, die selbe Operation angewendet wird. Durch Post-Processing Effekte werden während der Rasterung alle Pixel des Ausgabebildes mit der selben Funktion manipuliert. Dies schlägt sich in der Architektur von GPUs nieder. Sie bestehen aus zahlreichen, gleich aufgebauten Rechenkernen, die zeitgleich eine Operation auf mehreren unterschiedlichen aber gleichförmigen Datensätzen ausführen. Es handelt sich also um ein Single Instruction Multiple Data-Prinzip (kurz: SIMD). Während heutige CPUs aus zwei, vier oder sechs Kernen bestehen, besitzen GPUs 32 bis 512 Rechenkerne in Hardware.

GPU-Rechenkerne sind aber nicht ohne Weiteres vergleichbar mit CPU-Kernen. Mehrkernige Hauptprozessoren folgen eher dem Prinzip Multiple Instruction Multiple Data (kurz: MIMD). Jeder CPU-Kern kann ein eigenes Programm oder einen beliebigen Thread ausführen. Dabei spielt es keine Rolle, ob die Threads zu ein und dem selben Prozess gehören oder aus mehreren Programmstarts stammen. Deshalb können CPU-Kerne zu einem Zeitpunkt auch unterschiedliche Befehle auf unterschiedlichen Daten durchführen. Ein derartig flexibles Multithreading ist mit GPUs nicht möglich.

Aus diesem Grund müssen die Algorithmen, die auf GPUs ausgeführt werden, eine bestimmte Struktur haben, um von den zahlreichen Rechenkernen zu profitieren. Sie müssen sozusagen „SIMD-kompatibel“ sein. Das einfachste Beispiel, das in diesem Zusammenhang genannt wird, sind Matrix-Addition und -Multiplikation. Alle Elemente der Ergebnismatrix werden gleich berechnet (Single Instruction), jedoch aus unterschiedlichen Einträgen der Eingabematrizen (Multiple Data). Aber auch komplexere Algorithmen wie Matrixtransponierungen [RM09], Sortieren [SHG09], Präfixsummen [HSO07], Standardbildverarbeitungsalgorithmen [Sto09] oder Zufallszahlengeneratoren [Pod07] sind geeignet.

Im Zusammenhang mit GPUs fällt oft der Begriff „massiv parallel“. Selbst mit 512 Rechenkernen stehen bei weitem nicht so viele Hardwarekerne zur Verfügung, wie Daten manipuliert werden müssen. Die Wiedergabe eines Full-HD Videos mit $1920 \times 1080 = 2\,073\,600$ Bildpunkten bräuchte wesentlich mehr Kerne, würde man jedem einzelnen Pixel oder auch nur jedem Makroblock einen physischen Kern zuweisen. Selbiges gilt für die Transformation einer großen Vertexmenge. 10^7 Vertices oder mehr in einer solchen Menge sind

realistisch. Darum müssen die Einzelaufgaben effektiv auf die Rechenkerne aufgeteilt werden, was ein separates Hardwareschedulersystem hocheffizient bewerkstelligt. Erst bei derart großen Aufgaben können GPUs ihre Stärken gegenüber CPUs demonstrieren. Die Hauptaufgabe des Scheduling liegt darin, die extremen Datenmengen, die von derart vielen parallelen Threads verarbeitet werden, zu transferieren.

3.1.3 Komponenten von CUDA

CUDA ist eine aus mehreren Komponenten bestehende Technologie.

Die größte Errungenschaft ist die in 3.1.1 erwähnte Abstraktion der Hardware. Als Entwickler benötigt man kein vertieftes Verständnis der Grafikkhardware mehr. Für die massiv parallele Nutzung der Rechenkerne steht ein einfaches Programmiermodell zur Verfügung. Mit der Verteilung der Threads muss man sich nicht auseinandersetzen. Dies übernimmt CUDA. Das gilt auch für die Speichernutzung, die CUDA nach außen als Hierarchie präsentiert, in der jede Komponente individuelle Stärken und Schwächen hat. Somit kann für jeden Zweck die passende Speicherart gewählt werden.

All diese Abstraktionen können durch wenige Erweiterungen der Programmiersprache C angesprochen werden. Es muss also keine neue Programmiersprache erlernt werden, sondern nur die korrekte Nutzung des CUDA-Modells. Wesentliche Konzepte von C wie imperative/prozedurale Programmierung und Pointer finden sich auch bei CUDA wieder. Es ist sogar möglich, auf zwei Ebenen zu programmieren. C for CUDA kann als High Level Modell gesehen werden, während die Driver API (kurz für: Application Programming Interface) Assembler-ähnliche Züge hat. Man ist nicht nur auf C als Programmiersprache beschränkt. Eine Erweiterung für Fortran existiert ebenfalls. Spätere Versionen sollen auch objektorientierte Ansätze aus C++ beherrschen (siehe Abschnitt 6.3).

Zu CUDA zählen darüber hinaus diverse Softwarewerkzeuge. Das wichtigste ist der Compiler `nvcc` [NVi08]. Zu Beginn des Übersetzungsprozesses trennt er den Code für die GPU (Device-Code) vom Quelltext für die CPU (Host-Code) und übersetzt den Device-Code entweder in die Assembler-Sprache PTX (kurz für: Parallel Thread Execution) oder in Binärform (cubin object). Host-Code wird wie gewohnt von einem C/C++-Compiler übersetzt. Im Anschluss fügt `nvcc` Device- und Host-Code zusammen. Um seine Programme zu testen stellt NVidia den CUDA Visual Profiler zur Verfügung, der detaillierte Benchmarks durchführen kann. Zahlreiche Beispiele, Bibliotheksfunktionen [NVi09b] und die dafür nötigen Header und Dynamic Link Libraries (kurz: DLL) finden sich im CUDA-Software Development Kit (kurz: SDK).

3.2 Das CUDA-Modell

In diesem Abschnitt werden sukzessiv alle wesentlichen Paradigmen und Begriffe des CUDA-Modells eingeführt. Sie ermöglichen die Grafikkartenprogrammierung und ihre Bewertung.

3.2.1 Compute Capabilities

Compute Capabilities spezifizieren auf der einen Seite die Grafikkhardware, auf der anderen Seite Größen, die in das CUDA-Modell einfließen. Auf Hardwareseite ist dies beispielsweise die Fähigkeit, Gleitkommazahlen mit doppelter Genauigkeit vom Typ `double` zu verarbeiten oder die Anzahl der Register, die jedem Rechenwerk zur Verfügung stehen. Ein Kennwert einer Compute Capability, der das CUDA-Modell betrifft, wäre zum Beispiel die maximale Anzahl von Threads gleichen Typs, die eine GPU ausführen kann. Zahlen, wie man sie von Versionsnummern her kennt, drücken die Compute Capability aus. Die Nomenklatur folgt dabei dem Schema „Major.Minor“. So war die erste Compute Capability 1.0, die zum Zeitpunkt der Erstellung dieser Arbeit aktuelle ist 1.3 und kurz vor Abschluss wurde Compute Capability 2.0 veröffentlicht.

Zu den Eigenschaften von Compute Capabilities gehört es, dass sie abwärtskompatibel sind. So besitzt eine GPU einer bestimmten Compute Capability alle Eigenschaften, die auch die Vorgänger Compute Capabilities aufwiesen. Ein Programm, das mit einer konkreten Compute Capability kompiliert wurde, läuft auch auf allen Karten mit einer jüngeren Compute Capability.

Dadurch entsteht für den Entwickler eine gewisse Sicherheit. Ebenso rasant wie die Entwicklung der Leistungssteigerung von Grafikkarten verändert sich auch die zu Grunde liegende Architektur. Compute Capabilities stellen sicher, dass in der Vergangenheit geschriebene Software auch in Zukunft auf neuerer Hardware korrekt läuft.

In [NVi09a] beziehungsweise [NVi10b] findet sich eine vollständige Auflistung aller Werte zu den einzelnen Version der Compute Capabilities.

3.2.2 Kernel, Threads, Blocks und Grids

CUDA erweitert die Programmiersprache C um wenige Schlüsselwörter. Dadurch können Programme geschrieben werden, die von der GPU ausgeführt werden. Eines dieser Schlüsselwörter ist `__global__`. Mit ihm markiert man klassische C-Funktionen, die dann anstatt auf der CPU von der Grafikkarte abgearbeitet werden. Solche Funktionen heißen Kernel. Führt man einen Kernel aus, so werden mehrere Threads gestartet und auf die einzelnen Rechenwerke, im CUDA-Jargon Stream-Processor oder CUDA-Core genannt, verteilt. Jedem Thread wird dabei eine eindeutige Identifikationsnummer (kurz: ID) zugeordnet, die über die implizit zur Laufzeit vorhandene Variable `threadIdx` abgerufen werden kann. Ein Minimalbeispiel für einen Kernel findet sich in Listing 3.1, der zwei $N \times N$ -Matrizen miteinander addiert.

```
1 __global__ void matAdd(  
2     const float A[N][N], const float B[N][N], float C[N][N]) {  
3     int i = threadIdx.x;  
4     int j = threadIdx.y;
```

```

5
6     C[i][j] = A[i][j] + B[i][j];
7 }

```

Listing 3.1: Matrixaddition als Minimalbeispiel eines Kernel.

Kernels, die mit dem Schlüsselwort `__global__` versehen sind, müssen vom Host-Code aufgerufen werden. Im Gegensatz dazu können Funktionen mit dem Schlüsselwort `__device__` nur von Device-Code gestartet werden. Ihre Ausführung findet auf der GPU statt. Allen Funktionen, die Host-Code enthalten, wird implizit das Schlüsselwort `__host__` vorangestellt. Tabelle 3.1 liefert einen Überblick über die drei Modifikatoren. Kernel besitzen keinen impliziten Stack. Dadurch ist auch keine Rekursion möglich. Möch-

Modifikator	aufgerufen von	ausgeführt auf	Anmerkungen
<code>__host__</code>	Host-Code	Host	implizit, klassischer C-Code
<code>__global__</code>	Host-Code	Device	heißt Kernel, Rückgabotyp <code>void</code>
<code>__device__</code>	Device-Code	Device	kann beliebigen Rückgabewert haben

Tabelle 3.1: Diese Tabelle listet die Modifikatoren auf, mit denen Funktionen versehen werden können um zu kennzeichnen, ob sie von der CPU (Host) oder von der GPU (Device) ausgeführt werden sollen.

te man sich selbst aufrufende Kernel realisieren, so müssen diese dahingehend modifiziert werden, dass der Programmierer selbst den Funktionsstack schreibt und verwaltet.

Threads werden in eine zweistufige Hierarchie unterteilt. Die untere Hierarchiestufe sind die sogenannten Blocks, auch Thread Blocks genannt. Bei ihnen handelt es sich um einen ein-, zwei- oder dreidimensionalen, arrayförmigen Zusammenschluss der Threads. Jeder Block kann insgesamt bis zu 512 Threads enthalten beziehungsweise maximal 512, 512 und 64 Threads für die x-, y- und z-Richtung des Blocks. So wären (512), (16, 16) und (8, 8, 4) gültige Konfigurationen für Blockgrößen, (1024), (32, 16) und (2, 2, 128) jedoch nicht.

Neben den Blocks stellt das Grid die zweite Hierarchiestufe für die Organisation von Threads dar. Analog den Blocks, die Threads ein-, zwei- oder dreidimensional zusammenfassen, bündelt das Grid Blocks von Threads. Wieder kann dies ein-, zwei- oder dreidimensional geschehen, wobei ein Grid aus maximal $(65535, 65535, 65535) = (2^{16} - 1, 2^{16} - 1, 2^{16} - 1)$ Blocks bestehen kann. (32768), (65535, 32) und (16, 16, 1024) wären also gültige Konfigurationen für ein Grid, (2^{17}) , und $(2^{17}, 2, 4)$ wiederum nicht.

Somit kann ein Kernel maximal $(2^{16} - 1) \cdot (2^{16} - 1) \cdot (2^{16} - 1) \cdot 2^9 \cdot 2^9 \cdot 2^6 \approx 4,72 \cdot 10^{21}$ mal gestartet werden. Abbildung 3.2 skizziert exemplarisch den Aufbau eines zweidimensionalen Grids, das dreidimensionale Blocks enthält.

Zu einem Zeitpunkt ist immer nur die Ausführung eines Grids, das aus Threads eines Kernels besteht, möglich. Möchte man mehrere Grids starten, die jeweils Threads zweier unterschiedlicher Kernel enthalten, ist dies im Moment nur durch die Verwendung zweier CUDA-fähiger Chips möglich. Eine der zukünftigen Compute Capabilities soll es allerdings erlauben, auch auf einem einzelnen CUDA-Gerät unterschiedliche Grids parallel zu starten.

Man sieht also, dass die Anzahl der Threads die Anzahl der zur Verfügung stehen CUDA-Cores beinahe um den Faktor 2^{63} übersteigen kann. Auch wenn in Zukunft die Menge an

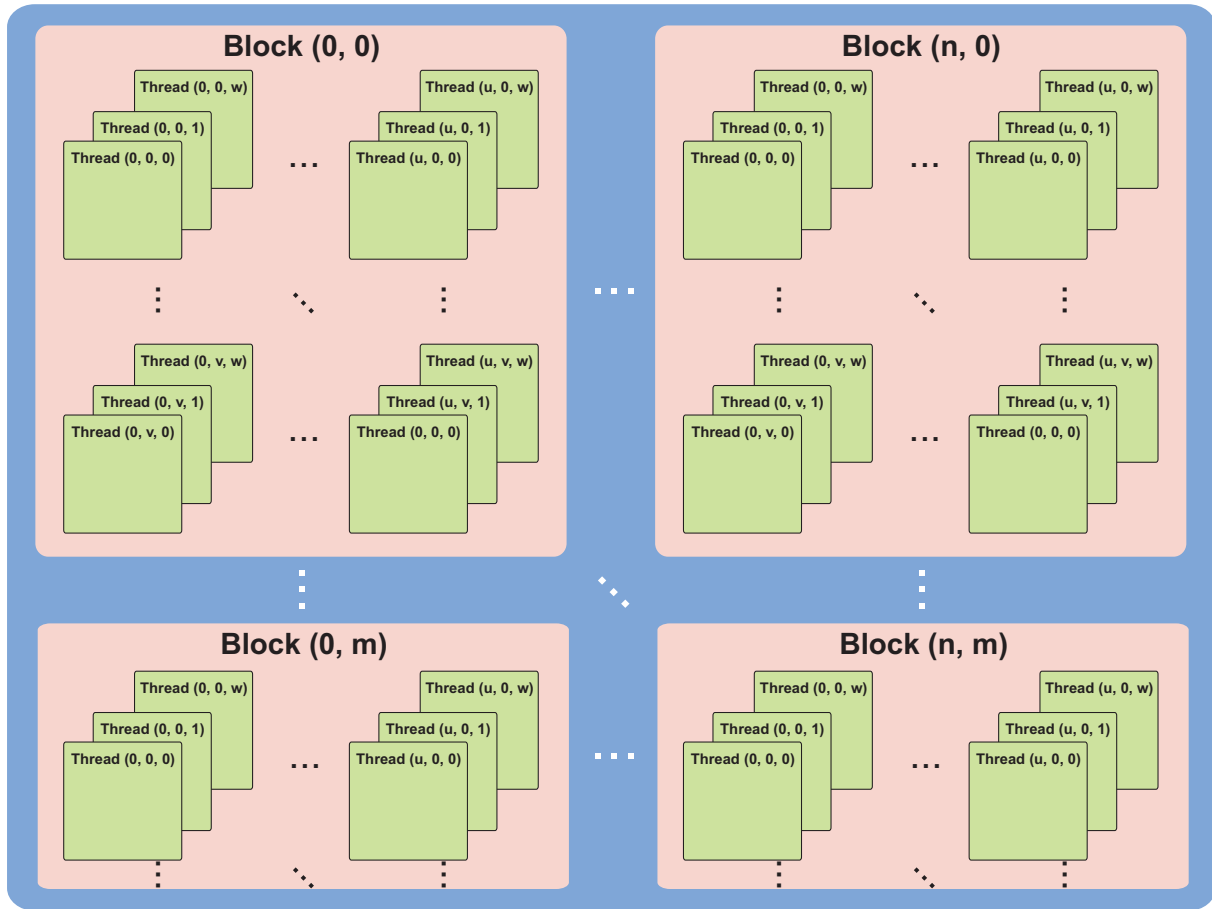


Abbildung 3.2: Diese Abbildung zeigt schematisch das Zusammenspiel von Threads, Blocks und Grids. Das Grid ist zweidimensional und enthält $N \times M$ Blocks. Jeder Block ist dreidimensional und enthält $U \times V \times W$ Threads. Die Vorlage für diese Grafik stammt aus [Sto09].

$$m = M - 1, n = N - 1, u = U - 1, v = V - 1, w = W - 1$$

Stream-Processors im Zuge des technischen Fortschritts kontinuierlich steigen wird, verändert das das Ungleichgewicht nicht wesentlich. Das HardwareSchedulersystem muss also eine äußerst zentrale Aufgabe lösen: Kontextwechsel der einzelnen Threads dürfen nur sehr wenig Zeit benötigen.

Wie oft ein Kernel auf dem Device gestartet wird, muss beim Aufruf des Kernels durch den Host-Code konfiguriert werden. Dabei wird mittels der speziellen Syntax `<<< Gridkonfiguration, Blockkonfiguration >>>` die Block- und Gridgröße spezifiziert. Listing 3.2 zeigt ein Beispiel für den Aufruf des in Listing 3.1 definierten Kernel `matAdd()`.

```

1 int main() {
2     ...
3     dim3 dimBlock(16, 16);
4     dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,
5                 (N + dimBlock.y - 1) / dimBlock.y);
6 
```

```

7   matAdd<<<dimGrid, dimBlock>>>(A, B, C);
8 }

```

Listing 3.2: Aufruf eines Kernel.

Zur Laufzeit stehen implizit Informationen über die Konfiguration der Blocks und des Grid sowie über die ID eines Thread in den Werten `blockDim`, `gridDim`, `threadIdx` und `blockIdx` bereit. Tabelle 3.2 bietet einen Überblick über diese reservierten Bezeichner und erläutert ihre Bedeutung.

Blocks und Grids sind die programmiertechnische Abstraktion der realen CUDA-Cores.

Bezeichner	Typ	Gespeicherte Information
<code>threadIdx</code>	<code>dim3</code>	Gibt dreidimensionale Thread-ID innerhalb eines Blocks an.
<code>blockIdx</code>	<code>dim3</code>	Gibt dreidimensionale Block-ID innerhalb des Grid an.
<code>blockDim</code>	<code>dim3</code>	Gibt die Größe eines Blocks in jeder Dimension an.
<code>gridDim</code>	<code>dim3</code>	Gibt die Größe des Grids in jeder Dimension an.

Tabelle 3.2: Variablen, die implizit zur Laufzeit zur Verfügung stehen und die Informationen über die Thread- und Grid-ID eines Threads liefern sowie über die Konfiguration der Blöcke und des Grid.

Als Entwickler muss man sich keine Gedanken über die Verteilung der Threads auf die Stream-Processors machen. Dazu dient der Block/Grid-Mechanismus. Hardwareseitig werden jeweils acht CUDA-Cores in einem sogenannten Multiprocessor (kurz: MP) zusammengefasst. Dies wird in Unterabschnitt 3.2.4 noch eine wichtige Rolle spielen.

3.2.3 Speicherorganisation

Neben einem Mechanismus für die Nutzung der Stream-Processors, bietet CUDA auch einen Mechanismus für die Verwendung der unterschiedlichen Speicher auf einer Grafikkarte. Zu diesem Zweck gibt es grundsätzlich zwei Arten von Speicher. Einerseits ist das der sogenannte Device Memory, der prinzipiell dem auf der Grafikkarte verbauten Arbeitsspeicher entspricht. Andererseits sind das Speicher, die sich auf dem selben Chip befinden, wie die GPU.

Zunächst sollen Charakteristika und die verschiedenen Arten des Device Memory vorgestellt werden. Grundsätzlich ist zu sagen, dass Device Memory viel Kapazität besitzt, die Latenzen für Lesen und Schreiben jedoch recht hoch sind.

Wichtigster Vertreter des Device Memory ist der Global Memory (kurz: GMem). Er entspricht in seiner Funktionsweise dem, was der Arbeitsspeicher für CPUs ist, und kann beliebige Datentypen, auch selbst definierte und rekursive, aufnehmen. Mittels der Befehle `cudaMalloc()` und `cudaFree()` lässt sich GMem allozieren und wieder deallozieren. Die GPU hat keinen direkten Zugriff auf den Hauptarbeitsspeicher der CPU, sondern nur auf den Device Memory und die schnellen Speicher auf dem GPU-Chip. Deshalb müssen zu verarbeitende Daten vom Hauptarbeitsspeicher in den Device Memory kopiert und Ergebnisse umgekehrt vom Device Memory in den Hauptarbeitsspeicher zurückgebracht werden. Der Befehl `cudaMemcpy()` ermöglicht dies in beide Richtungen. Listing 3.3 führt ein Beispiel für die Verwendung von GMem auf.

```
1 // Device-Code
2 __global__ void vecAdd(float* A, float* B, float* C) {
3     int i = blockDim.x * blockIdx.x + threadIdx.x;
4
5     if (i < N) {
6         C[i] = A[i] + B[i];
7     }
8 }
9
10 // Host-Code
11 int main() {
12     int N = ...;
13     size_t size = N * sizeof(float);
14
15     // Allokierung der Inputdaten im Hauptarbeitsspeicher
16     float* h_A = malloc(size);
17     float* h_B = malloc(size);
18     float* h_C = malloc(size);
19
20     // Allokierung des Device Memory
21     float* d_A;
22     cudaMalloc((void**) &d_A, size);
23     float* d_B;
24     cudaMalloc((void**) &d_B, size);
25     float* d_C;
26     cudaMalloc((void**) &d_C, size);
27
28     // Kopieren vom Host auf das Device
29     cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
30     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
31
32     // Aufruf des Kernel
33     int threadsPerBlock = 256;
34     int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
35
36     vecAdd<<<blocksPerGrid, threadsPerBlock>>>>(d_A, d_B, d_C);
37
38     // Zurückkopieren des Ergebnisses auf den Host
39     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
40
41     // Freigabe des Device Memory
42     cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
43
44     // Freigabe des Hauptarbeitsspeicher
45     free(h_A); free(h_B); free(h_C);
46 }
```

Listing 3.3: Speichertransfers zwischen Hauptarbeitsspeicher und Device Memory.

An dieser Stelle offenbart sich der erste Flaschenhals der Grafikkartenprogrammierung. Während die Latenz zwischen Hauptarbeitsspeicher und CPU bzw. Device Memory und GPU zu den mittleren in der klassischen Speicherhierarchie zählt (100 bis 500 Prozessortaktzyklen pro Speicherzugriff), ist diese zwischen Hauptarbeitsspeicher und Device Memory über den PCI-Express-Bus extrem groß. Deshalb sollte die Anzahl der Transfers mit `cudaMemcpy()` klein gehalten werden. Die Menge der übertragenen Daten darf dabei groß sein, denn die Bandbreite des PCI-Express-Bus ist hoch. Am sinnvollsten ist es somit, zu Beginn des Programms alle Input-Daten für die GPU in wenigen großen Speichertransfers auf die Grafikkarte zu kopieren, dort so viele Berechnungen wie möglich auszuführen und erst am Ende die Ergebnisse in Form eines großen Transfers zurückzuholen.

Auf einen weiteren Flaschenhals der Grafikkartenprogrammierung wurde bereits in Unterabschnitt 3.1.2 hingewiesen. Alle parallel laufenden Threads müssen auch mit den Daten versorgt werden, die sie verarbeiten. Die massive Parallelität führt also zu gehobenen Anforderungen an die Speicheranbindung zwischen GPU und Device Memory. Daher bietet der GMem den Mechanismus des Coalescing. Er erlaubt es unter Berücksichtigung gewisser Bedingungen, mit nur einem einzigen Speichertransfer 16 Threads (die Größe eines Half-Warps, siehe nächster Unterabschnitt 3.2.4) mit Daten zu versorgen. Diese Voraussetzungen lauten:

- Die Threads müssen allesamt
 - entweder je 4-Byte große Wörter lesen (z.B. ein `float`), was zu einer 64-Byte Speichertransaktion führt,
 - oder je 8-Byte große Wörter lesen (z.B. ein `float2`), was zu einer 128-Byte Speichertransaktion führt,
 - oder je 16-Byte große Wörter lesen (z.B. ein `float4`), was zu zwei 128-Byte Speichertransaktionen führt.
- Alle 16 zu transferierenden Wörter müssen im selben Speicherabschnitt liegen, der so groß wie die Summe der 16 Wörter sein muss (oder doppelt so groß, falls ein Wort 16-Byte lang ist).
- Die Threads müssen in der richtigen Reihenfolge auf die Wörter zugreifen. Das i -te Wort muss für den i -ten Thread bestimmt sein.

Ab Compute Capability 1.2 fallen die letzten beiden Punkte weg.

In der Praxis ist man stets darum bemüht, Coalescing zu nutzen, da es die Speichertransfers zwischen GMem und GPU signifikant beschleunigt.

Ein dritter Flaschenhals programmierbarer Grafikkarten ist die Tatsache, dass es keine impliziten Caches gibt. Der Zugriff auf Elemente, die erst vor kurzem abgerufen wurden oder die hoch frequentiert sind, ist also genauso schnell, wie auf Elemente, die noch nie verwendet wurden. Um den GMem zu cachen, muss dieser in eine Textur (kurz: TMem) transformiert werden, was mit dem Befehl `cudaBindTexture()` vollzogen wird. Von da an steht jeder Textur ein Cache von 6 bis 8 KB pro Multiprocessor zur Verfügung. Texturen bieten jedoch noch weitere Vorteile. Sie können mit Gleitkommazahlen indiziert werden. Der zurückgegebene Wert entspricht dann einer bilinearen Interpolation

der betroffenen Nachbarwerte. Hier erkennt man deutlich die Hauptaufgabe von Grafikprozessoren, nämlich das Rendern von texturierten 3D-Szenen. Zusätzlich kann das Verhalten der Hardware bestimmt werden, falls eine Textur über ihre Grenzen hinaus angesprochen wird. Texturen zwingen den Entwickler aber auch zu Einschränkungen. Ihre Werte können nur gelesen, nicht jedoch geschrieben werden. Darüber hinaus dürfen sie nur ein-, zwei- oder vierdimensionale Vektoren von 8-, 16- oder 32-Bit Ganzzahlen (`char`, `unsigned char`, `char1`, `uchar1`, `char2`, `uchar2`, `char4`, `uchar4`, `short`, `unsigned short`, `short1`, `ushort1`, `short2`, `ushort2`, `short4`, `ushort4`, `int`, `unsigned int`, `int1`, `uint1`, `int2`, `uint2`, `int4`, `uint4`) beziehungsweise 16- oder 32-Bit Gleitkommazahlen (`float`, `float1`, `float2`, `float4`) enthalten.

Nachdem bisher verschiedene Arten und Ausprägungen des Device Memory erläutert wurden, soll nun der Blick auf diejenigen Speicher gerichtet werden, die sich auf dem selben Chip wie die GPU befinden. Das sind einerseits die Register (kurz: RMem) und andererseits der Shared Memory (kurz: SMem), auch Scratchpadspeicher genannt. Allen On-Chip Speichern gemein ist, dass sie sehr schnellen Zugriff ermöglichen. Pro Taktschritt kann ein Datum ausgetauscht werden. Zusätzlich erfolgen Zugriffe parallel, das heißt, alle Threads lesen beziehungsweise schreiben ihr Datum gleichzeitig. Jedoch sind sie in ihrer Größe stark limitiert. Jedem Multiprocessor stehen zwar 8192 32-Bit breite Register (ab Compute Capability 1.3 16384) zur Seite, jedoch nur 16 KB SMem.

Register auf CPUs sind vergleichbar mit den Registern klassischer Hauptprozessoren. Innerhalb eines Taktes kann ein Datum gelesen beziehungsweise geschrieben werden. Aus diesem Grund sollten möglichst viele Operationen mit Registern anstatt mit den langsameren Device Memories durchgeführt werden. Zwei Umstände erschweren dies jedoch. Zwar verfügt jeder Multiprocessor über mindestens 8192 Register, jedoch können pro MP 768 Threads aktiv sein, auch wenn immer nur acht gleichzeitig in der Hardware abgearbeitet werden. Somit kann ein einzelner Thread im Schnitt gerade einmal knapp elf Register nutzen, was sich in der Praxis als äußerst wenig erweist. Erschwerend kommt hinzu, dass die Zuteilung, welches Datum im Registerspeicher gehalten wird, von CUDA vollzogen wird. Als Programmierer hat man darauf keinerlei Einfluss. An dieser Stelle offenbart sich, dass CUDA noch eine sehr junge Technologie ist, denn die Registerzuweisung durch den Compiler ist oft wenig befriedigend und bietet noch sehr großes Optimierungspotential. Zweiter On-Chip Speicher ist der SMem. Seine Zugriffszeit entspricht der von Registern. Auf der anderen Seite hat man als Programmierer volle Kontrolle über den Shared Memory. Mittels des Modifiers `__shared__` deklariert man innerhalb des Host-Codes Speicher als SMem. Der Name „geteilt“ rührt daher, dass der komplette Shared Memory jedem Thread eines Blocks zugänglich ist. Registerinhalte konnten nur vom zugehörigen Thread bearbeitet werden. Somit ist der SMem auch das Medium für Inter-Thread-Kommunikation. Mit diesem Wissen erscheint die Tatsache, dass pro MP nur 16 KB Shared Memory verfügbar sind, in einem ganz anderen Licht. Denn diese 16 KB stehen einem einzelnen Block vollständig zur Verfügung, was nur in seltenen Szenarien nicht ausreichend ist. Jedoch sind auch hier zwei Dinge zu beachten. Der Shared Memory ist in 16 Bänke unterteilt. Diese kann man sich als Spalten vorstellen, die jeweils 32 Bit breit sind. In eine Zeile aller Bänke passen also $16 \text{ Bänke} \cdot 32 \text{ Bit} = 512 \text{ Bit}$ Daten. Greifen nun zwei Threads eines Half-Warps (siehe Unterabschnitt 3.2.4) auf die selbe Bank zu, so werden diese Zugriffe serialisiert, was die Laufzeit des Programms erheblich verlängert. Einzige Ausnahme ist der Fall, wenn alle Threads auf ein gemeinsames Datum zugreifen.

Device Memory					
Name	Abkürzung	gecached	Größe	Latenz	Lebensdauer
Global Memory	GMem	nein	MB bis GB	hoch bis niedrig	Ende des Programms
• Coalescing beschleunigt gemeinsame Speicherzugriffe					
Texture Memory	TMem	ja	MB bis GB	hoch bis mittel	Ende des Programms
• kann nur gelesen werden • Gleitkommazahlen als Indices zulässig $\Rightarrow$ bilineare Interpolation • enthält nur ein-, zwei oder vierdimensionale Vektoren von 8-, 16- oder 32-Bit Ganzzahlen beziehungsweise 16- oder 32-Bit Gleitkommazahlen					
Constants Memory	CMem	ja	64 KB total	hoch bis sehr niedrig	Ende des Programms
• kann nur gelesen werden • wird durch Modifier <code>__constant__</code> im Dateikopf des Host-Cods deklariert • ermöglicht bei gleichzeitigem Zugriff aller Threads eines Half-Warps auf die selbe Speicheradresse Zugriffszeit eines Registers					
Local Memory	LMem	nein	16 KB pro Thread	hoch	Ende des Kernel
• wird für große Structs und Arrays verwendet, falls diese nicht in die Register passen • Programmierer hat keinen Einfluss auf LMem $\Rightarrow$ implizite Verwaltung durch CUDA					
On-Chip Memory					
Name	Abkürzung	gecached	Größe	Latenz	Lebensdauer
Register Memory	RMem	unnötig	2^{13} bis 2^{14} pro MP	sehr niedrig	Ende des Kernel
Shared Memory	SMem	unnötig	16 KB pro MP	sehr niedrig	Ende des Kernel

Tabelle 3.3: Auflistung der unterschiedlichen Speicherarten mit ihren Eigenschaften.

Dann setzt ein Broadcastingmechanismus ein. Ebenfalls nachteilig ist die Tatsache, dass man keine Garantien hat, welche Threads bereits wie weit abgearbeitet wurden. Besonders bei der Inter-Thread-Kommunikation kann es hierbei zu Update-Problemen kommen. Für die Synchronisation aller Threads eines Blocks steht jedoch der Befehl `__syncThreads()` zur Verfügung.

Die intelligente Beherrschung des Shared Memory im Zusammenspiel mit dem Device Memory auf der einen Seite und den Threads auf der anderen ist ein Schlüssel zu effizienter Grafikkartenprogrammierung.

Tabelle 3.3 bietet abschließend einen Überblick über alle in CUDA vorkommenden Speicher. Sie enthält auch den nicht näher erwähnten Constants Memory (kurz: CMem) und Local Memory (kurz: LMem).

3.2.4 Warps, Half-Warps und Warp-Divergence

Neben den beiden Hierarchieebenen Blocks und Grids muss noch eine dritte Stufe, die sogenannten Warps, erläutert werden. Sie lassen sich nicht horizontal in die bisherige Ordnung einfügen, sondern müssen eher als eine vertikale Ergänzung gesehen werden. Ein Warp fasst 32 Threads zusammen, ein Half-Warp folglich 16 Threads. Sie beschreiben, wie ein Bündel von Threads auf der Hardware ausgeführt wird. Das SIMD-Prinzip führt dazu, dass der Befehlszähler für alle Threads eines Warps auf den selben Befehl verweist, den alle Threads parallel auf unterschiedlichen Daten ausführen sollen. Eine solche Parallelität wird also nicht für alle Threads im Grid gefordert, sondern nur für 32 Threads große Bündel.

Durch bedingte Anweisungen ist es möglich, dass einige Threads eines Warps Teile des Kernelcodes durchlaufen, die von den restlichen Threads nicht benötigt werden. Listing 3.4 zeigt ein sinnfreies Minimalbeispiel, bei dem dieses Phänomen auftritt. Alle Threads mit gerader ID nehmen einen anderen Ausführungsweg als Threads mit ungerader ID. Somit ist es also möglich, dass zu einem Zeitpunkt zwei Befehlszähler innerhalb eines Warps zwei unterschiedliche Adressen im Programmcode enthalten. In einem solchen Fall spricht man von Warp-Divergence.

```
1 // Device-Code
2 __global__ void kernel(const int *A, const int *B, int *C) {
3     if(threadIdx.x % 2 == 0) {
4         C[threadIdx.x] = B[threadIdx.x] << 1;
5     } else {
6         C[threadIdx.x] = A[threadIdx.x] * B[threadIdx.x] + 42;
7     }
8 }
9
10 // Host-Code
11 int main() {
12     ...
13     int N = ...;
14     int M = ...;
15
16     // Aufruf des Kernel
17     dim3 dimBlock(N);
```

```

18     dim3 dimGrid(M);
19
20     kernel<<<dimGrid, dimBlock>>>(A, B, C);
21     ...
22 }

```

Listing 3.4: Sinnfreies Minimalbeispiel für einen Kernel, der zu Warp-Divergence führt.

Warp-Divergence führt dazu, dass die Abarbeitung von Threads in einem Warp serialisiert wird, anstatt parallel ausgeführt zu werden. Dazu wird jeder einzelne Thread des Warps zu seinem eigenen Warp, in dem nur er sich befindet. Das ist gleich aus zweierlei Hinsicht negativ zu bewerten. Die Ausführung von 32 Threads hintereinander nimmt 32 mal mehr Zeit in Anspruch, als deren parallele Verarbeitung, was zu einer drastischen Laufzeitverschlechterung führt. Zusätzlich sinkt die Auslastung der auf massive Parallelität ausgelegten Hardware, wenn im Moment nur noch einer anstatt 32 Threads bearbeitet wird. In diesem Fall bleibt die Stärke von Grafikkhardware ungenutzt.

Deshalb ist Warp-Divergence unbedingt zu verhindern. Primär erreicht man dies durch die Vermeidung von bedingten Anweisungen, die abhängig von der ID eines Threads zu unterschiedlichen Programmverzweigungen führen. Kann man garantieren, dass für alle Threads eines Warps der selbe Programmzweig gewählt wird, so sind bedingte Anweisungen durchaus erlaubt. In der Praxis ist dies nicht immer möglich. Auch lassen sich bedingte Anweisungen nicht generell vermeiden. Das Problem der Warp-Divergence lässt sich jedoch mit dem schon bekannten Befehl `__syncThreads()` entschärfen. Setzt man ihn am Ende eines Bereichs, von dem man weiß, dass er zu Warp-Divergence führt, so findet an dieser Stelle eine Synchronisation der Threads statt. Der Rest des Kernel kann dadurch nach einer serialisierten Phase wieder parallel abgearbeitet werden.

3.2.5 Occupancy und Instruction Throughput

Um in CUDA geschriebene Programme für Grafikkarten bewerten zu können, stehen zwei Größen zur Verfügung, die eine Quantifizierung zulassen. Occupancy und Instruction Throughput. Mit ihnen lassen sich Aussagen treffen, wie gut ein Programm die vorhandene Grafikkhardware nutzt, und somit indirekt auch, wie gut ein Algorithmus für programmierbare GPUs geeignet ist.

Occupancy wird durch einen Wert im Intervall $[0, 1]$ angegeben und drückt den Umstand in Zahlen aus, dass Ressourcen auf der Grafikkarte limitiert sind. Sie gibt das Verhältnis aktiver Warps an, die maximal möglich sind (24 pro Multiprocessor, 32 seit Compute Capability 1.2) und die tatsächlich ausgeführt werden. Je höher die Occupancy, desto besser. Diese Diskrepanz ist durch Limitierungen der Grafikkarte bedingt, vor allem durch die Knappheit an Registern. Jedem Thread stehen im Schnitt knapp elf Register zur Verfügung, was aber nur in den seltensten Fällen ausreichend ist. Komplexere Kernel bedienen sich pro Thread durchaus 15 bis 25 Register. Aus diesem Grund muss die Anzahl der aktiven Warps, also Warps, die bereit zur Ausführung sind, reduziert werden, was zu einer Verminderung der Occupancy führt. Bevor ein Warp wieder aktiv gesetzt werden kann, müssen seine Operanden in die Register geladen werden, was die selben Nachteile bringt, wie ein Kontextwechsel auf der CPU.

Instruction Throughput ist ein Maß für die durchschnittliche Auslastung der CUDA-Cores. Ein Vergleich mit dem Verhältnis von Idle- zu Burst-Zeiten bei CPUs ist legitim.

Der Durchsatz ist definiert als die Anzahl von Operationen pro Taktzyklus und Multiprocessor. Drei Faktoren können zur Maximierung dieses Wertes beitragen.

- Anweisungen, die einen niedrigen Durchsatz besitzen, sollten vermieden werden.
- Die Nutzung von vorhandener Speicherbandbreite sollte forciert werden.
- Dem Scheduler sollte es möglich sein, Wartezeiten, in denen Daten aus dem Speicher geholt werden, mit arithmetischen Operationen zu verbringen.

Tabelle 3.4 listet eine Reihe von Gleitkommaanweisungen mit ihrem Durchsatz auf. Die Spannweite reicht hierbei von 8 bis 0,88 Operationen pro Takt.

Schlechte Organisation von Speicherzugriffen ist der Hauptgrund für niedrigen Instruction

Anweisung	Befehl	Operationen pro Taktzyklus
Addition, Multiplikation	<code>+, *</code>	8
Logarithmus	<code>__logf()</code>	2
inverse Wurzel $\frac{1}{\sqrt{x}}$	<code>rsqrtf()</code>	2
schnelle Division	<code>__fdividef()</code>	1,6
trigonometrische Funktionen	<code>__sinf()</code> , <code>__cosf()</code>	1
normale Division	<code>/</code>	0,88

Tabelle 3.4: Die Tabelle listet eine Auswahl von Operationen mit ihrem Instruction Throughput auf, die auf der Grafikkarte ausgeführt werden können. Alle Operationen beziehen sich auf Gleitkommazahlen mit einfacher Genauigkeit, also `float`.

Throughput. Unstrukturierter Zugriff auf den GMem führt zu hohen Latenzen. Stattdessen sollten Techniken wie Coalescing und schnelle Speicher wie SMem verwendet werden. Die Unterabschnitte 3.3.2 und 5.4.3 beschäftigen sich tiefer mit diesem Thema. Warp-Divergence ist ein weiterer Grund für unbefriedigenden Instruction Throughput. Dagegen sorgen rechenintensive Kernel mit wenigen Speicherzugriffen ebenso für einen wünschenswert hohen Instruction Throughput wie eine große Menge aktiver Threads. Dies führt mitunter zu scheinbar widersprüchlichen Programmierstechniken, wie Unterabschnitt 3.3.4 vorführt. In der Praxis gilt eine Auslastung $>0,5$ als gut, Werte $>0,8$ als ideal.

3.3 Klassische Probleme und Trade-Offs bei CUDA-Programmierung

Mit den in Abschnitt 3.2 gewonnenen Erkenntnissen lässt sich nun eine differenziertere Bewertung von Grafikkartenprogrammierung durchführen. An vielen Stellen fand sich bisher ein „aber“, was deutlich zeigt, dass Implementierungen für die GPU trotz zahlreicher Verbesserungen durch diverse Frameworks wie CUDA noch alles andere als intuitiv sind. In diesem Abschnitt sollen Entscheidungen für Probleme präsentiert werden, die dann auftreten, wenn zwei Aspekte der Grafikkartenprogrammierung sich diametral entgegenstehen. NVidia selbst stellt mit [NVi09c] ein Dokument zur Verfügung, in dem das effektive programmieren mit CUDA thematisiert wird.

3.3.1 Limitierung schneller Speicher vs. Occupancy

Aufgrund der schnellen Zugriffszeit von On-Chip Speichern sind diese grundsätzlich allen anderen Speicherarten auf einer Grafikkarte vorzuziehen. Mit Ausnahme von Bankkonflikten beträgt die Latenz dieser Speicher 0 Taktzyklen. Nur in Fällen, in denen nur ein einziges mal auf den Speicher zugegriffen muss, kann ein Zugriff auf den GMem mit Coalescing ähnlich performant sein. Liegt ein solches Szenario vor, dürfte aber die Transferzeit zwischen Host- und Device-Speicher eine entscheidende Rolle spielen, weshalb die Nutzung des Grafikchips wenig sinnvoll erscheint (siehe Unterabschnitt 3.3.6).

Shared Memory steht einem ganzen Block zur Verfügung. Da aufgrund des SIMD-Prinzips zumeist auf ein-, zwei- oder dreidimensionalen Arrays gearbeitet wird, sieht die Reihenfolge der Speichertransfers meistens wie folgt aus:

1. Kopieren aller Daten vom Host- in den Device-Speicher mit möglichst wenigen, möglichst großen Speichertransfers.
2. Kopieren der Daten für einen Block aus dem GMem in den SMem. Jeweils ein Thread kopiert ein Element. Dabei kann und sollte Coalescing genutzt werden.
3. Auf dem SMem können Threads mit Ausnahme von Bankkonflikten schnell beliebige Speicherzugriffe durchführen.

Da der SMem pro Multiprocessor 16 KB beträgt und ein Block diese 16 KB vollständig nutzen kann, ergibt das pro Thread mindestens 32 Byte (bei 512 Threads pro Block). Das wäre beispielsweise ein achtelementiger Vektor oder zwei vierelementige Vektoren mit Gleitkommaeinträgen einfacher Genauigkeit. In der Praxis ist diese Menge an Speicher in den meisten Fällen ausreichend und stellt somit selten einen Flaschenhals dar. Kritisch ist in diesem Zusammenhang eher, dass oftmals bei der Programmierung des Kernels noch nicht bekannt ist, welche Speicherlemente ein Thread-Block zur Laufzeit benötigen wird. Nutzt man mehr als die verfügbaren 16 KB SMem, so macht sich das erst nach dem Start des Programms mit einem Absturz bei der Ausführung des Device-Codes bemerkbar.

Limitierender Faktor bei den On-Chip Speichern sind für fast jeden Kernel die Register. Um eine Occupancy von 1 zu erreichen, darf ein Thread maximal $\left\lfloor \frac{\text{Anzahl der pro MP zur Verfügung stehenden Register}}{\text{Maximale Anzahl aktiver Threads pro MP}} \right\rfloor$ Register nutzen. Aufwändigere Kernel benötigen aber zumeist 15 bis 25 Register pro Thread, was bei den Compute Capabilities 1.0

und 1.1 eine hohe Occupancy verhindert. Darüber hinaus darf ein Thread unabhängig von der Compute Capability maximal 32 Register in Anspruch nehmen. Wird diese Grenze überschritten, so werden Daten auf den langsamen LMem ausgelagert. Mit dem Compiler-Flag `-maxregcount=N` lässt sich diese Grenze sogar noch absenken. Eine durch hohen Registerverbrauch geminderte Occupancy ist akzeptabler, wenn viele Threads ($> 10^6$) abgearbeitet werden müssen. Der Scheduler kann dann trotzdem für einen hohen Instruction Throughput und damit befriedigende Laufzeiten sorgen.

Abschließend lässt sich sagen, dass Occupancy mehr ein Maß für durchdachte Ressourcennutzung ist als ein Indikator für die Laufzeit eines CUDA-Programms. Eine niedrige Occupancy kann in Fällen vieler geschedulter Threads akzeptiert werden, da in solchen Fällen der Instruction Throughput trotzdem hoch sein kann. An dieser Stelle zeigt sich die Stärke des GPU-Hardwareschedulers.

3.3.2 Coalescing vs. komplexe Datenstrukturen

Hohe Zugriffszeiten auf den Device Memory machen ihn zu einem Flaschenhals bei der Abarbeitung von Programmen auf der Grafikkarte. Da in ihn aber der Inhalt des Hauptarbeitsspeichers kopiert werden um die Eingabedaten für die Kernel bereit zu stellen, ist er unverzichtbar. Auch ist seine Größe im Vergleich zu den On-Chip Speichern praktisch nicht limitiert, da er mehrere hundert Megabyte (kurz: MB) bis wenige Gigabyte (kurz: GB) groß ist. Coalescing und Texturcaches sind zwei Techniken, um die großen Latenzen des GMem zu reduzieren.

Der parallele Speicherzugriff ist aber nur unter den in 3.2.3 aufgelisteten Bedingung möglich. Somit ist Coalescing auf sehr einfache Datentypen wie Arrays beschränkt, deren Elemente auch noch einer wenig flexiblen Anordnung im Speicher entsprechen müssen. Komplexere Datenstrukturen wie Bäume oder Graphen können nicht von Coalescing profitieren, da ihre Elemente nur in den seltensten Fällen so hintereinander im Speicher angeordnet sind, wie sie von den Threads mit ihren aufsteigenden IDs angefordert werden, die parallel auf der Datenstruktur arbeiten. Verschärft wird dieser Missstand durch die Tatsache, dass die Knoten der Bäume und Graphen meistens selbst definierte Strukturen sind, die ebenfalls den Coalescingbedingungen widersprechen. Spätestens ab dem Zeitpunkt, an dem parallel Operationen wie Tiefen- oder Breitensuche durchgeführt werden sollen, findet ein unstrukturierter Zugriff auf den GMem statt.

Möchte man komplexe Datenstrukturen auf der Grafikkarte verwenden, ist die Nutzung von Caches unverzichtbar. Die Frage ist, ob der niedrige Instruction Throughput, der aufgrund der hohen Speicherzugriffszeiten entsteht, durch die massive Parallelität ausgeglichen werden kann, so dass die GPU das Ergebnis schneller berechnet als die CPU. Dies lässt sich nicht pauschal beantworten, sondern hängt vom Einzelfall ab. Viele Faktoren wie die Datenstruktur, die Nutzung des Caches und die Blockgröße haben darauf Einfluss. In Unterabschnitt 5.3.3 wird ein Beispiel näher beleuchtet und ausgewertet, das als Datenstruktur einen speziellen Baum, eine BVH, verwendet.

3.3.3 Kleine Kernel vs. Instruction Throughput

Wie in Unterabschnitt 3.3.1 beschrieben, ist der Mangel an Registern nicht nur ein Problem in Bezug auf die Occupancy, sondern auch eine technische Limitierung. Folglich stellt

sich die Frage, wie man den Registerverbrauch niedrig halten kann.

Eine direkte Manipulationsmöglichkeit der benötigten Register hat man als Programmierer nicht. Der CUDA-Compiler `nvcc` übernimmt zur Compilezeit die Zuordnung des RMem. Allerdings gibt es indirekte Einflussmöglichkeiten. Je kleiner ein Kernel ist und je weniger Variablen lokal in ihm definiert werden, desto weniger Register benötigt er. Darum lautet eine der Grundregeln der Grafikkartenprogrammierung, Algorithmen so gut wie möglich in kleinere Einzelteile zu zerlegen und für jedes der Teilstücke einen (mit `__global__`, nicht mit `__device__` gekennzeichneten) Kernel zu schreiben. Wie gut sich ein Algorithmus stückeln lässt, entscheidet also ebenfalls über dessen Eignung für GPUs.

Tendenziell lässt sich jedoch sagen: Je kürzer ein Kernel ist, desto weniger Rechenoperationen führt er durch. Die Vorbereitungen, die getroffen werden müssen, bevor ein Kernel effektiv zu rechnen beginnen kann, sind nicht unerheblich. Bis sich die entsprechenden Daten im Shared Memory befinden, vergeht viel Zeit. Stehen die Daten bereit, so ist es aufgrund der massiven Parallelität sinnvoll, so viele arithmetische Operationen wie möglich mit diesen Daten durchzuführen, da arithmetische Anweisungen zu einem hohen Instruction Throughput und damit einer akzeptablen Auslastung der Stream-Processors führen. Somit stehen kurze Kernel im Widerspruch zu hohem Instruction Throughput. Daraus ergibt sich die Begründung, warum sinnvolle Kernel einen Registerverbrauch von 15 bis 25 Stück haben.

Im Zweifel ist die Entscheidung für viele benutzte Register sinnvoller als ein niedriger Instruction Throughput, vor allem dann, wenn viele Threads geschedult sind. Die Stärke programmierbarer Grafikkarten liegt in ihren zahlreichen parallelen Rechenwerken. Nur ihre hohe Auslastung garantiert überlegene Laufzeit gegenüber einer klassischen CPU-Implementierung. Auch wenn wenige Warps aufgrund des hohen Registerverbrauchs aktiv sein können, schafft es der Hardwarescheduler wegen der zahlreichen arithmetischen Operationen im Kernel, die Auslastung aller CUDA-Cores hoch zu halten. Ein gutes Beispiel für diese These liefert Unterabschnitt 5.3.3.

3.3.4 Speicherzugriffe vs. Neuberechnung

An vielen Stellen wurde bereits thematisiert, dass Speicherzugriffe nur unter engen Voraussetzungen in ihrer Performanz mit den massiv parallelen Rechenwerken mithalten können. Der Flaschenhals zwischen GPU und Device Memory wird auch noch langfristig bestehen bleiben. Im Gegenzug wird die Leistungsfähigkeit und Zahl der CUDA-Cores überproportional ansteigen, was die unbefriedigende Latenz großer Speicher auf der Grafikkarte nur noch deutlicher unterstreicht. In dieser Hinsicht unterscheiden sich Grafikkarten nicht von CPUs und dem Hauptarbeitsspeicher. Dort steht die Leistungssteigerung der Hauptprozessoren in den letzten Jahren in keinem Verhältnis zu deren Anbindung an den Random Access Memory (kurz: RAM). Im Gegensatz dazu steht die bloße Rechenleistung in praktisch unlimitierter Fülle bereit, sofern man es schafft, die Aufträge gut zu parallelisieren. Dies führt zu einem Programmierparadigma, welches bei CPU-Programmierung kontraproduktiv wäre.

Als klassischer CPU-Programmierer ist man stets bemüht, jedes Zwischenergebnis, von dem man weiß, dass es mindestens ein weiteres Mal in der Zukunft benötigt wird, separat in einer Variable zu speichern. Große CPU-Caches ermöglichen den schnellen Zugriff auf

das Zwischenergebnis in der Zukunft. Da Caches auf der GPU, falls überhaupt vorhanden, in ihrer Größe sehr beschränkt sind und der ideale Speicher für Zwischenergebnisse, der RMem, ebenfalls nicht ausreichend üppig bemessen ist, ist diese klassische Vorgehensweise bei GPU-Programmierung nicht ratsam. Stattdessen ist die Neuberechnung von Zwischenergebnissen immer da, wo sie benötigt werden, deutlich klüger.

Auch die Speicherung von Daten, die sich implizit aus anderen Informationen ergeben, ist auf Grafikkarten unvernünftig. Beispielsweise wird im STL-Format (siehe Kapitel 4) für jede Fläche einer Dreiecksmenge ein separater Normalenvektor gespeichert, der sich auch durch das Kreuzprodukt von zwei Seiten des Dreiecks berechnen lassen würde. In solchen Fällen ist es sinnvoller, solche Daten nicht explizit zu speichern, sondern an gegebener Stelle berechnen zu lassen. Das reduziert nicht nur langsame Speichertransfers. Eine Minderung des Speicherverbrauchs sowie eine Erhöhung des Instruction Throughput ist ebenfalls gegeben.

3.3.5 Bedingte Anweisungen führen zu Warp-Divergence

Verzweigungen in der Ausführung des Programmcodes von Kernels führt zu Warp-Divergence. Neben den hohen Speicherlatenzen ist Warpdivergence die Hauptursache für die langsame Ausführung von Kernels, auch wenn sie Algorithmen implementieren, die grundsätzlich gut parallelisierbar sind. Aus diesem Grund kommt der Reduktion von Warp-Divergence eine ebenso große Rolle zu, wie ein durchdachtes Speichermanagement. Warp-Divergence lässt sich wesentlich schwerer handhaben als große Speicherlatenzen, da keine zusätzlichen Mechanismen wie etwa Coalescing verfügbar sind. Ratschläge, wie die Vermeidung von bedingten Anweisungen, sind ebenso logisch wie gut gemeint, in der Praxis lassen sie sich aber oftmals nicht umsetzen. Stattdessen kann man als Entwickler nur vom Umstand profitieren, dass der bool'sche Wert `false` als 0 und der bool'sche Wert `true` als 1 interpretiert werden kann.

Eine bedingte Anweisung wie in Listing 3.4 lässt sich mit diesem Mittel leicht durch eine Anweisung wie in Listing 3.5 ersetzen, die zu keiner Warp-Divergence führt. Zwar ist der arithmetische Aufwand für diese Alternative höher, dies ist aber in Anbetracht der Tatsache, dass Rechenkapazität normalerweise nicht der limitierende Faktor ist, unproblematisch. Natürlich lässt sich diese Technik nur dann einsetzen, wenn es sich im bedingten Programmzweig um arithmetische Operationen handelt. Andere Funktionalität wie Kopieraktionen lassen sich damit nicht modifizieren.

```
1 // Device-Code
2 __global__ void kernel(const int *A, const int *B, int *C) {
3     C[threadIdx.x] = !(threadIdx.x % 2) * (B[threadIdx.x] << 1) +
4         (threadIdx.x % 2) * (A[threadIdx.x] * B[threadIdx.x] + 42);
5 }
```

Listing 3.5: Vermeidung von Warp-Divergence durch arithmetischen Ersatz.

Oft dienen bedingte Anweisungen dem Ausschluss von Fällen, in denen eine Berechnung zu einem ungültigen Ergebnis führen würde. Division durch 0 ist ein solcher Fall oder Szenarien, in denen nur positive Ergebnisse zulässig sind. Anstatt solche unerwünschten Konstellationen mit einer bedingten Anweisung abzufangen, die zu Warp-Divergence

führt, kann neben dem Ergebnis auch ein zweiter, separater bool'scher Wert geschrieben werden, der angibt, ob das eigentliche Berechnungsergebnis gültig oder ungültig ist. Dieser bool'sche Indikator wird durch einen bool'schen Ausdruck bestimmt, in den alle Bedingungen einfließen, die sonst zur Vermeidung von fehlerhaften Berechnungen verwendet worden wären. Listing 3.6 ist ein reales Beispiel für die Verwendung eines solchen Indikators. Sinnvoll ist eine solche Technik nur dann, wenn zum Schreiben des Gültigkeitswertes Coalescing verwendet werden kann. Ansonsten erkauft man sich die fehlende Warp-Divergence durch eine erhöhte Speicherlatenz.

```

1  __global__ void cudaGeneratePixposImageKernel(
2      const float3 start, const float4 *directions,
3      const float a_200, const float a_020, const float a_002,
4      const float a_110, const float a_101, const float a_011,
5      const float a_100, const float a_010, const float a_001,
6      const float a_000,
7      const float3 screenNormal,
8      const float3 screenDirection1, const float3 screenDirection2,
9      const float screenDelta, const float2 originTranslation,
10     const float2 tftPixelSize,
11     const int screenHoriRes, const int screenVertRes,
12     PixposImageStruct pixposImage) {
13     ...
14     /*
15      * Neben dem bloßen Speichern des Ergebnisses in der ersten
16      * Anweisung wird mit der zweiten Anweisung der
17      * Gültigkeitsindikator geschrieben. In beiden Fällen kann
18      * Coalescing eingesetzt werden.
19      */
20     pixposImage.pixposArray[idxY * pixposImage.horiRes +
21         idxX] = pixpos;
22     pixposImage.pixposStatusArray[idxY * pixposImage.horiRes +
23         idxX] =
24         // Diskriminate > 0.
25         (SQR(linear) - quadratic * constant) > FLT_EPSILON &
26         // Vermeidung von Division durch 0
27         fabsf(quadratic) > FLT_EPSILON &
28         // Mindestens ein Schnitt in Sichtstrahlrichtung
29         (lambda1 > FLT_EPSILON | lambda2 > FLT_EPSILON) &
30         // Ist der gewählte Schnittpunkt in Sichtstrahlrichtung?
31         lambda > FLT_EPSILON &
32         /*
33          * Liegt die berechnete Pixelposition innerhalb der
34          * Grenzen des Monitors?
35          */
36         pixpos.x >= 0.0f & pixpos.x < (float) screenHoriRes &
37         pixpos.y >= 0.0f & pixpos.y < (float) screenVertRes;
38 }

```

Listing 3.6: Separates Setzen eines Gültigkeitsindikators.

Warp-Divergence kann auch an Stellen auftreten, an denen dies nicht auf Anhieb erkennbar ist. So kann sich die auslösende bedingte Anweisung in einer Hilfsfunktion, beispielsweise aus der CUDA-API, befinden. Bei den Hilfsfunktionen `fminf()` und `fmaxf()` ist dies der Fall. Außerdem sollten alle bool'schen Ausdrücke *eager* und nicht *lazy* ausgewertet werden. Tritt bei einer mehrstufigen bool'schen Anweisung der Fall ein, dass ein Thread das Ergebnis des Ausdrucks aufgrund einer *lazy*-Auswertung kennt, bevor das bei einem anderen Thread des selben Warps passiert, so kommt es zu Warp-Divergence. Wieder ist es von Vorteil, dass arithmetische Operationen keinen Flaschenhals darstellen, so dass anstatt der Operatoren `||` und `&&` die *eager*-Äquivalente `|` und `&` verwendet werden können.

3.3.6 Speichertransfers zwischen Host und Device

Bevor die GPU überhaupt Berechnungen durchführen kann, müssen zu Beginn die zu verarbeitenden Daten vom Hauptarbeitspeicher in den Device Memory unter Verwendung des Befehls `cudaMemcpy()` kopiert werden. Dieser Vorgang wird über den Peripheral Component Interconnect Express-Bus (kurz: PCIe) abgewickelt. Zwar bietet er Bandbreiten von $250 \text{ MB} \cdot \text{s}^{-1}$ bis $16 \text{ GB} \cdot \text{s}^{-1}$, jedoch kann mit Ausnahme des Mechanismus des Page-Locked Host Memory das Device nicht direkt auf den Host Speicher zugreifen. Page-Locked Host Memory unterstützt allerdings kein Coalescing und bietet auch keine Möglichkeit des Caching. Aufgrund der Latenz des PCIe-Bus ist Page-Locked Host Memory in jedem Fall langsamer als Device Memory. Er macht nur dann Sinn, wenn der Speicher der Grafikkarte nicht ausreicht und die GPU unbedingt Zugriff auf größere Datenmengen benötigt.

Konsequenz der Trennung des Speicher in Host und Device ist, dass zwei unterschiedliche

Transfergröße	Transfergröße in Bytes	Transferrichtung	Transferdauer in μs
10 KB	$10\,240 = 10^1 \cdot 2^{10}$	Host $\rightarrow$ Device	7,712
100 KB	$102\,400 = 10^2 \cdot 2^{10}$	Host $\rightarrow$ Device	42,274
1 MB	$1\,048\,576 = 10^0 \cdot 2^{20}$	Host $\rightarrow$ Device	397,344
10 MB	$10\,485\,760 = 10^1 \cdot 2^{20}$	Host $\rightarrow$ Device	8479,3
100 MB	$104\,857\,600 = 10^2 \cdot 2^{20}$	Host $\rightarrow$ Device	90668,1
10 KB	$10\,240 = 10^1 \cdot 2^{10}$	Device $\rightarrow$ Host	9,480
100 KB	$102\,400 = 10^2 \cdot 2^{10}$	Device $\rightarrow$ Host	54,656
1 MB	$1\,048\,576 = 10^0 \cdot 2^{20}$	Device $\rightarrow$ Host	526,944
10 MB	$10\,485\,760 = 10^1 \cdot 2^{20}$	Device $\rightarrow$ Host	6643,62
100 MB	$104\,857\,600 = 10^2 \cdot 2^{20}$	Device $\rightarrow$ Host	80924,6

Tabelle 3.5: Transferzeiten von Kopieroperationen zwischen Hauptarbeitspeicher und Device Memory und zurück. Zu diesem Zweck wird der Befehl `cudaMemcpy()` verwendet. Bei den kopierten Daten handelt es sich um zufällige Werte vom Typ `char`.

Adressräume verwaltet werden müssen. An dieser Stelle offenbart sich ein weiteres Hindernis der Nutzung von komplexen Datenstrukturen auf der Grafikkarte. Die Verweise der einzelnen Elemente eines Graphen auf Nachbarknoten sind durch Pointer realisiert, also

durch die Adressen im Speicher, an denen sich adjazente Knoten befinden. Möchte man nun eine derartige Datenstruktur in den Device Memory kopieren, so müssten aufgrund des geänderten Adressraums alle Pointer umgeschrieben werden, was sehr aufwändig und zeitraubend ist. Hier erweist sich wiederum die Nutzung von Arrays als die bessere Wahl, denn nur die Startadresse zu Beginn der betroffenen Speicherregion muss angepasst werden. Die Offsets innerhalb des Arrays sind für Host und Device gleich. Für komplexe Datenstrukturen bedeutet das, dass sie linearisiert werden müssen. Bäume lassen sich exemplarisch in Pre-, In- oder Post-Order bringen (siehe Unterabschnitt 4.4.4).

Einen Vergleich von unterschiedlich großen Datentransfers über den PCIe-Bus liefert Tabelle 3.5. Im KB-Bereich macht die Latenz des Bus einen Großteil der Gesamttransferzeit aus. Die überproportional großen Werte im MB-Bereich sind wohl in dem Umstand begründet, dass für den zusammenhängenden Datenblock im Device Memory erst ein zusammenhängender Speicherbereich frei gemacht werden muss.

Für Datentransfers zwischen Host und Device gilt grundsätzlich der bereits in 3.2.3 erläuterte Grundsatz, dass derartige Transaktionen so selten wie möglich durchgeführt werden sollten, am besten nur zu Beginn und ganz am Ende aller Berechnungen. Wenn eine solche Operation durchgeführt wird, dann ist es ratsam, große Datenmengen zu übertragen, am besten in Form eines Arrays.

Die in diesem Abschnitt geschilderte Problematik entfällt, wenn die benötigten Daten erst auf der Grafikkarte erzeugt werden oder nur eine einzelne Zahl das Ergebnis aller Berechnungen ist. Zum Beispiel können die Richtungen der Sichtstrahlen einer Kamera aus relativ wenigen Parametern, die zusammen weniger als 1 KB groß sind und die Kamera modellieren, erzeugt werden. Dass es sich beim Resultat aller Grafikkartenoperationen um ein einelementiges Ergebnis handelt, ist unterdessen unrealistisch, da es mit dem SIMD-Ansatz unvereinbar ist.

3.3.7 Größe von Blocks und dem Grid

Threads werden durch Blocks und das Grid hierarchisiert. Zum Abschluss soll folgender Unterabschnitt klären, wie die Konfiguration von Blocks und Grid optimal gewählt wird. Programmiertechnisch wird eine solche Konfiguration mit der `<<< Gridkonfiguration, Blockkonfiguration >>>`-Syntax umgesetzt. Die Größe eines Warp ist fest. Er enthält immer 32 Threads.

Grundsätzlich spielt die Größe eines Blocks eine eher untergeordnete Rolle. Da er in Warps eingeteilt wird, die dann tatsächlich auf der GPU ausgeführt werden, ist es sinnvoll, dass die Anzahl der Threads in einem Block ein Vielfaches von 32 ist. Ansonsten sollten so viele Threads wie möglich in einem Block zusammengefasst werden. Diesem Vorsatz entgegen wirkt der Mangel an Registern.

In einem Grid sollten sich so viele Blocks wie möglich befinden. Dies ist gleichbedeutend mit dem Wunsch, dass das zu lösende Problem gut parallelisierbar ist, also viele einzelne Datensätze bereitstellt, auf denen zur gleichen Zeit der selbe Befehl abgearbeitet wird. Dadurch erhält der Hardwarescheduler die Möglichkeit für hohen Instruction Throughput zu sorgen.

Ob Blocks und das Grid ein-, zwei- oder dreidimensional formuliert werden, spielt keine Rolle. Auch die Konfiguration der Anzahl der Threads pro Dimension ist irrelevant. Es ist aber darauf zu achten, dass es sich möglichst um eine Potenz von 2, also 2^n handelt.

3.4 Wann ist der Einsatz von CUDA sinnvoll?

Mit programmierbaren Grafikkarten steht ein neues Werkzeug zur Verfügung, um rechenintensive Probleme zeitnah zu lösen. Die Frage, ob eine GPU-Implementierung schneller ein Ergebnis berechnet als eine äquivalente CPU-Implementierung ist im Voraus oft sehr schwer zu beantworten. Dieser Abschnitt soll das Kapitel rund um CUDA abschließen, indem er eine Entscheidungshilfe an die Hand gibt, mit der tendenziell überprüft werden kann, ob ein Algorithmus profitiert, wenn er auf der GPU ausgeführt wird.

Zu diesem Zweck müssen die Abbildungen 3.3, 3.4 und 3.5 von oben nach unten durchlaufen werden. Auf jeder Stufe ist je nach Antwort auf die Frage in einen der beiden Äste abzustiegen. Auf der untersten Hierarchiestufe kann eine Aussage darüber getroffen werden, wie sinnvoll eine GPU-Implementierung für ein Verfahren ist. Je weiter links man endet, desto schlechter, je weiter rechts, desto besser. Alle drei Ergebnisse müssen in die Endentscheidung einfließen.

Zu beachten ist, dass das Ergebnis der drei Entscheidungshierarchien nur eine Tendenz

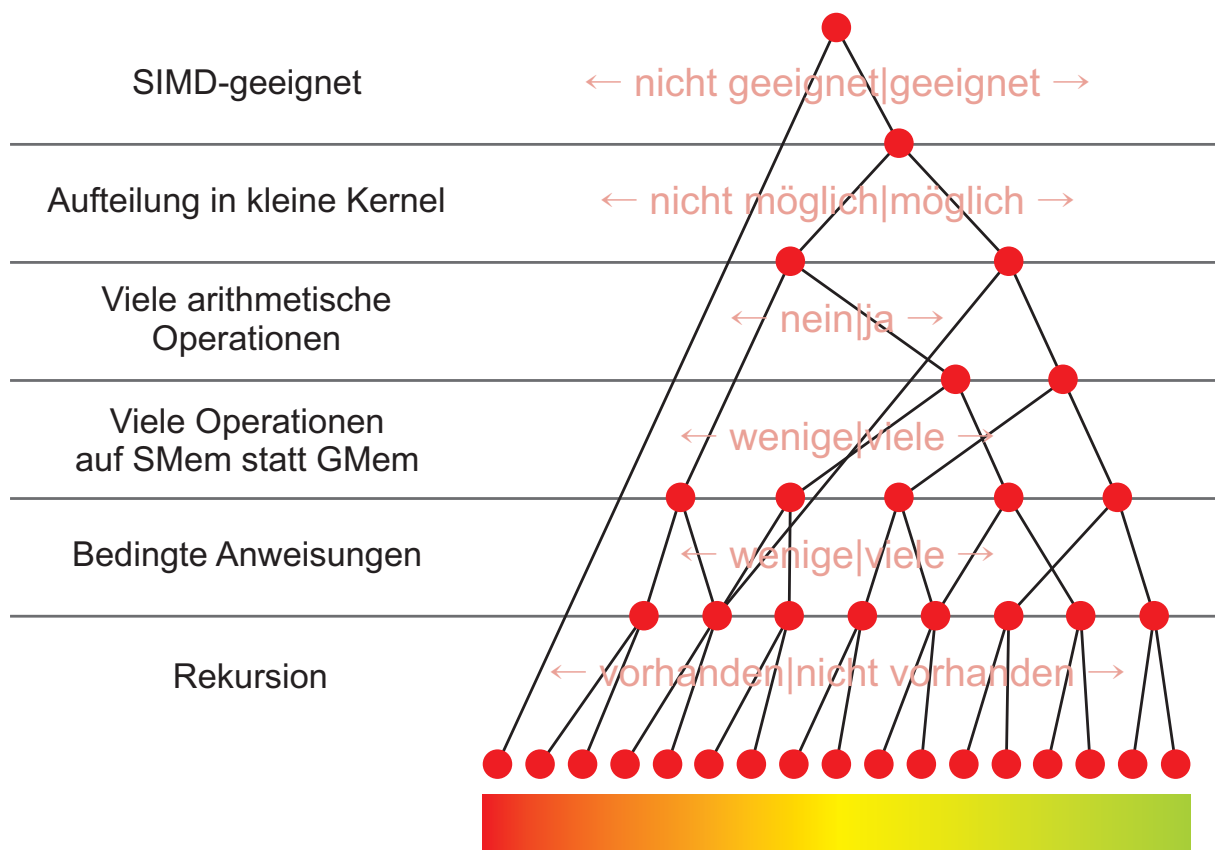


Abbildung 3.3: Entscheidungshilfe für den Einsatz von Grafikkarten zur Algorithmusbeschleunigung, die die Eignung des zu implementierenden Algorithmus allgemein beleuchtet.

vorgibt, wie sehr die Nutzung einer Grafikkarte ein Lösungsverfahren beschleunigen kann. Um absolute Sicherheit zu erhalten ist im Zweifelsfall eine konkrete GPU-Implementierung

und ihr direkter Vergleich mit einer äquivalenten CPU-Implementierung sinnvoller, auch wenn dies zumeist mit erheblichem Mehraufwand verbunden ist.

Abbildung 3.3 bewertet allgemein, ob sich ein Algorithmus gut parallelisieren und auf die Grafikkarte portieren lässt. Möchte man die Ausführungszeit eines kompletten Projekts durch programmierbare Grafikkarten reduzieren, so ist das Entscheidungsverfahren für jeden Algorithmus des Projekts einzeln durchzuführen.

Die Entscheidungsfragen aus Abbildung 3.3 sind wie folgt zu verstehen:

- *SIMD-geeignet*: Entspricht der Algorithmus dem SIMD-Prinzip? Nur wenn strukturell gegeben ist, dass ein und die selbe Operation auf unterschiedlichen Daten angewandt wird, ist der Einsatz von GPUs rentabel. Ist diese Voraussetzung nicht erfüllt, so kann pauschal gesagt werden, dass das zu implementierende Verfahren die Stärken von Grafikkarten nicht nutzen kann.
- *Aufteilung in kleine Kernel*: Um den Registerverbrauch niedrig zu halten, sind kleine Kernel wünschenswert. Lässt der Algorithmus sich also in kleine, hintereinander ausführbare Kernel stückeln, so profitiert er von GPUs.
- *Viele arithmetische Operationen*: Je mehr Berechnungen durchgeführt werden und je weniger Speicheroperationen ein CUDA-Programm enthält, desto höher ist der Instruction Throughput.
- *Viele Operationen auf SMem statt GMem*: Falls alle Threads ihre Operanden ausschließlich aus dem SMem laden können anstatt aus dem wesentlich langsameren GMem, so führt das zu einer signifikanten Beschleunigung der Berechnungen, da hohe Zugriffslatenzen entfallen.
- *Bedingte Anweisungen*: Bedingte Anweisungen haben zumeist Warp-Divergence zur Folge und sollten daher so selten wie möglich Verwendung finden. Wie groß der Einfluss von Maßnahmen gegen Warp-Divergence ist, zeigt Abbildung 3.4.
- *Rekursion*: Da Kernel keinen Stack besitzen, unterstützen sie normalerweise keine Rekursion. Falls diese dennoch unerlässlich ist, muss der Stack explizit vom Programmierer verwaltet werden, was zusätzlichen Aufwand bedeutet.

Nur weil ein Algorithmus gut parallelisierbar ist, heißt das nicht, dass es sich lohnt, ihn für die GPU zu implementieren. Die Entscheidungshierarchie 3.4 validiert die Kriterien „massiv parallel“ und die Robustheit des Algorithmus gegenüber Warp-Divergence.

Die Entscheidungsfragen aus Abbildung 3.4 sind wie folgt zu verstehen:

- *Anzahl der Threads*: Je mehr Threads von einem Kernel gestartet werden, desto besser kann der Hardwarescheduler die CUDA-Cores auslasten. Tritt Warp-Divergence nur in vereinzelt Warps auf, ist der Registerverbrauch hoch oder ist die Anzahl der arithmetischen Operationen eher niedrig, so kann dies oftmals durch sehr viele Threads ($> 10^7$) ausgeglichen werden.
- *Programmverzweigungen lassen sich durch Arithmetik ersetzen*: Treten Warp-Divergences auf, so sind deren Ursache, bedingte Anweisungen, unter allen Umständen zu reduzieren. Gelingt dies wie in Unterabschnitt 3.3.5 beschrieben, so ist diese Lösung eben so gut, wie wenn keine bedingte Anweisung vorhanden wäre.

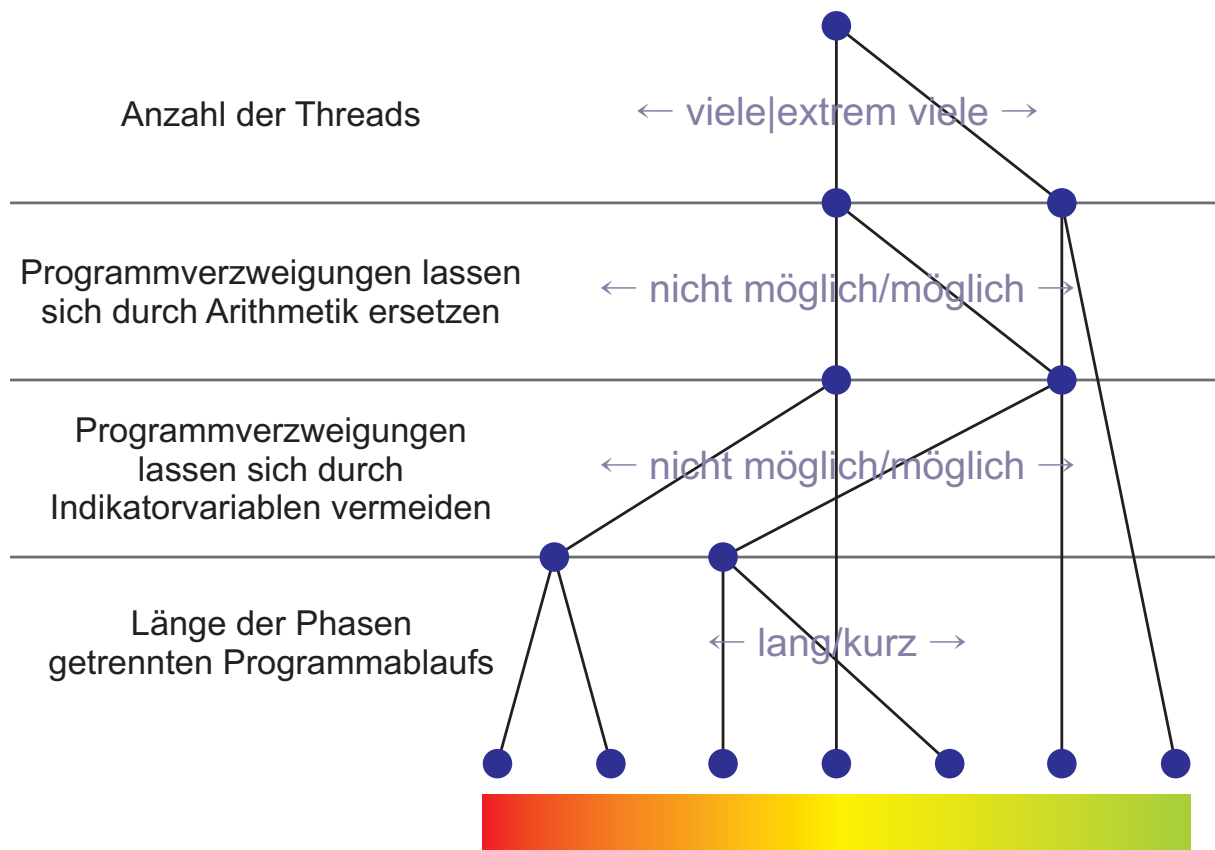


Abbildung 3.4: Entscheidungshilfe für den Einsatz von Grafikkarten zur Algorithmusbeschleunigung die überprüft, ob das Laufzeitverhalten gut mit den strukturellen Eigenschaften der Grafikkartenhardware harmonisiert.

- *Programmverzweigungen lassen sich durch Indikatorvariablen vermeiden:* Eine weitere Methode, um bedingte Anweisungen zu sparen sind Indikatorvariablen, die ein Kernel an den aufrufenden Host-Code zurückgibt. Für sie gilt bezüglich ihrer Bewertung das selbe wie in obigem Aufzählungspunkt.
- *Länge der Phasen getrennten Programmablaufs:* Sind die Phasen des getrennten Programmablaufs eher kurz und lassen sie sich an deren Ende synchronisieren, so hat dies einen positiven Effekt auf das Laufzeitverhalten.

Letztendlich muss ein Algorithmus den Speicher, der auf der Grafikkarte bereitsteht, in der korrekten Art und Weise nutzen. Das Schema in Abbildung 3.5 kontrolliert, wie gut ein Algorithmus den geforderten Kriterien entspricht.

Die Entscheidungsfragen aus Abbildung 3.5 sind wie folgt zu verstehen:

- *Wenige, große Speichertransfers zwischen Host und Device:* Aufgrund des Flaschenhalses PCIe-Bus zwischen Host und Device, sind wenige große Speichertransfers ökonomischer als viele kleine Transaktionen.
- *Lineare oder komplexe Datenstrukturen:* Lineare Datenstrukturen wie Arrays haben den Vorteil, dass sie automatisch zu Speichertransfers in einem Stück führen.

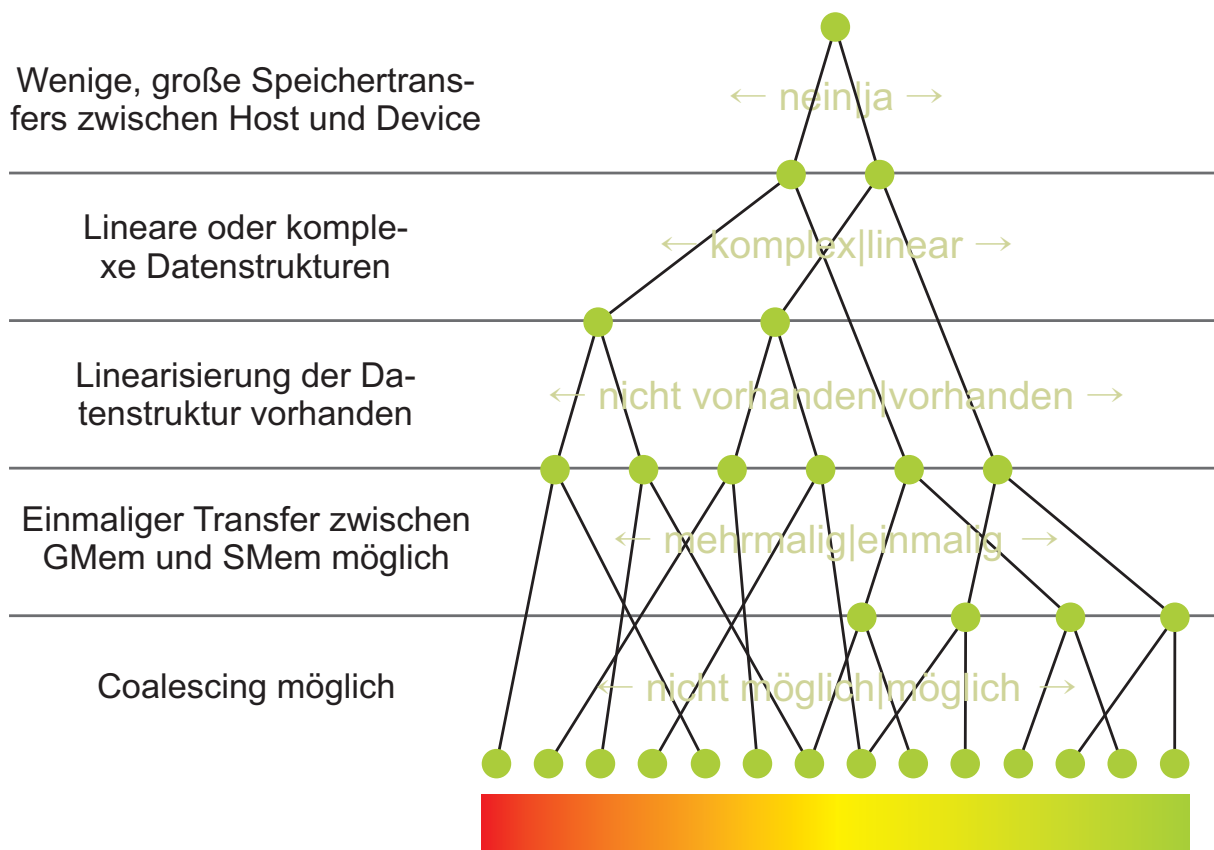


Abbildung 3.5: Entscheidungshilfe für den Einsatz von Grafikkarten zur Algorithmusbeschleunigung, die überprüft, ob das Speicherzugriffsverhalten gut mit der Speicherhierarchie der Grafikkarte vereinbar ist.

- *Linearisierung der Datenstruktur vorhanden:* Lässt sich eine Datenstruktur wie ein Baum oder allgemeiner ein Graph linearisieren, das heißt, in einen zusammenhängenden Speicherbereich schreiben, so können komplexe Datentypen ebenso schnell auf das Device kopiert werden wie lineare. Es gilt jedoch die Zeit, die für die Linearisierung benötigt wird, zu beachten.
- *Einmaliger Transfer zwischen GMem und SMem möglich:* Aufgrund der hohen Zugriffszeit auf den Device Memory ist man als Entwickler angewiesen, hauptsächlich mit dem deutlich schnelleren, aber limitierten SMem zu arbeiten. Passen alle benötigten Daten für einen Thread-Block in die 16 KB SMem, so profitiert die Anwendung davon deutlich.
- *Coalescing möglich:* Coalescing reduziert die Kopierzeit vom GMem in den SMem und umgekehrt. Passen nicht alle für einen Thread-Block benötigten Daten in den SMem oder lassen sich diese beim Kernelstart noch nicht sicher identifizieren, so kann der Zwangszugriff auf den GMem durch Coalescing dennoch einen deutlichen Geschwindigkeitsvorteil bringen.

Kapitel 4

Dreiecksmengen und STL

In dieser Arbeit werden Dreiecksmengen für die Modellierung des Ideals des deflektometrisch zu prüfenden Bauteils eingesetzt. Ihre Eigenschaften, Vor- und Nachteile präsentiert das folgende Kapitel. Dabei liegt ein besonderer Fokus auf der Beschleunigung des Raycastings durch hierarchische Strukturen, besonders in Hinblick auf die Verwendung von programmierbaren Grafikkarten.

4.1 Einführung und Herausforderungen

Bevor auf die algorithmische Optimierung von Dreiecksmengen eingegangen wird, soll dieser Abschnitt zunächst für ein grundlegendes Verständnis dieser sorgen. Dazu gehören sowohl die in diesem Zusammenhang auftretenden Termini als auch die prinzipiellen Eigenschaften von Triangulierungen von Geometrien.

4.1.1 Eigenschaften von Dreiecksmengen und des STL-Modells

Man kann die Aneinanderreihung von ebenen Polygonen im Raum dazu verwenden, um die Oberfläche von dreidimensionalen Objekten zu beschreiben. Dabei werden die Vielecke so aneinandergesetzt, dass ein geschlossenes Netz entsteht. Dies wird dadurch erreicht, dass an jede Seite eines Polygons die Seite eines anderen Vielecks angrenzt. Die Ecken eines Polygons, auch Vertex genannt, sind dadurch auch die Eckpunkte der angrenzenden Polygone. Alle Vertices eines Polygons liegen in der selben Ebene.

Zumeist werden Dreiecke als Polygone verwendet, da sie die einfachsten Polygone darstellen, die eine Ebene aufspannen. Sie haben an ihren Seiten immer insgesamt drei Nachbarn. Die Anzahl der Dreiecke, die an den selben Eckpunkt angrenzen, ist nicht fest. Durchschnittlich treffen sich fünf bis sechs Dreiecke in einem Vertex.

Um nun die Oberfläche einer Geometrie mit Dreiecken darzustellen, liegen deren Vertices genau auf der zu modellierenden Oberfläche. Oft beschreibt ein CAD-Entwurf eine Komponente mit Ausschnitten aus Kurven, die gekrümmte Oberflächen darstellen können. Um diese in eine Menge von Dreiecken zu überführen, muss eine sogenannte Triangulierung beziehungsweise Triangulation durchgeführt werden. Von ihr hängt es auch ab, wie viele Dreiecke sich in einem Vertex treffen. Eine Triangulierung kann unter den verschiedensten Gesichtspunkten stattfinden:

- Die Dreiecke der Triangulation sollen alle in etwa den selben Flächeninhalt besitzen.
- An Stellen großer Oberflächenkrümmung sollen mehr Dreiecke verwendet werden als für ebenere Regionen.
- Die Anzahl der sich in einem Vertex treffenden Dreiecke sollte für alle Vertices möglichst gleich sein.
- Keine Seitenlänge der Dreiecke darf einen bestimmten Schwellwert über- oder unterschreiten.
- ...

Abbildung 4.1 zeigt zwei verschiedene Arten der Darstellung einer Dreiecksmenge, die in diesem Fall die Oberfläche eines Hasen modelliert.

Zur konkreten Beschreibung der virtuellen Geometrie wird in dieser Arbeit das STL-Modell verwendet. Es setzt sich aus zwei Säulen zusammen. Zum einen, welche Werte der Dreiecksmenge gespeichert und verwaltet werden, und zum anderen, wie man diese Werte in eine Datei im STL-Format speichert.

Für jedes Dreieck, im STL-Jargon auch als Facette bezeichnet, wird neben den drei Eckpunkten in kartesischen Koordinaten auch jeweils ein Normalenvektor gespeichert. Er

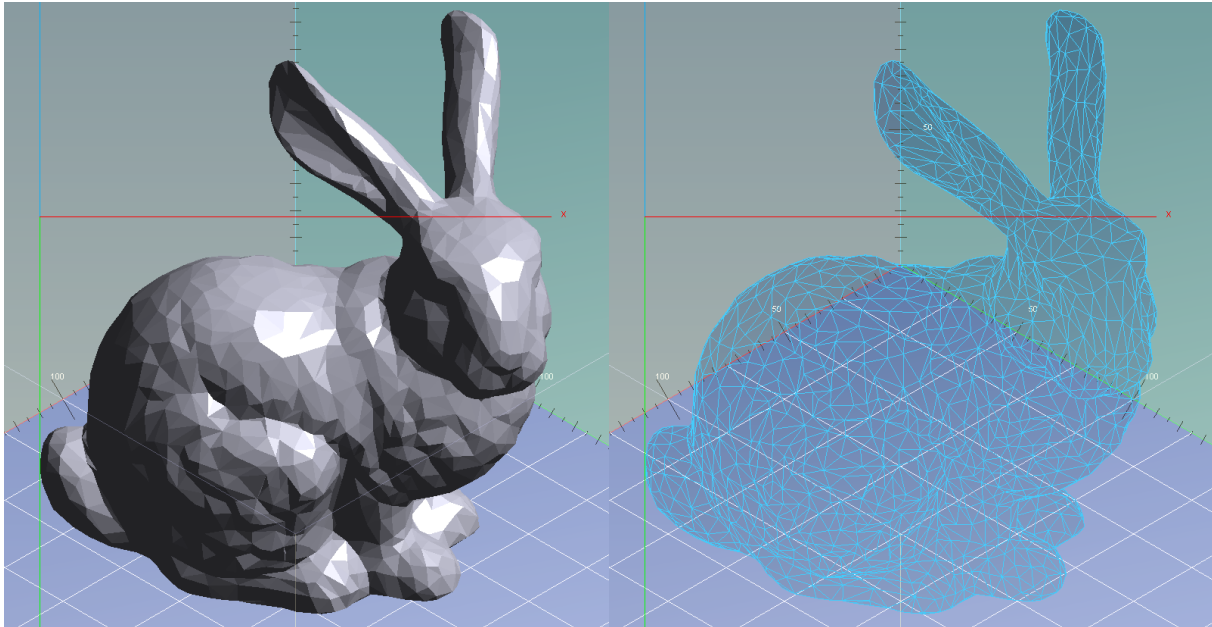


Abbildung 4.1: Das linke Bild zeigt die schattierte Oberfläche der Dreiecksmenge, die einen Hasen modellieren soll. Jedes Dreieck besitzt homogen über seiner Fläche die selbe Farbe. Dagegen werden im rechten Bild nur die Seitenlinien derjenigen Dreiecke angezeigt, die vom Betrachter aus sichtbar sind. Die Seitenlinien von Dreiecken auf der Rückseite des Hasen werden nicht dargestellt.

steht senkrecht auf der Ebene, die durch die drei Vertices der Facette vorgegeben ist. Dabei zeigt er weg von der darzustellenden Komponente, also nicht in die Geometrie hinein, sondern nach außen.

Gespeichert werden die Vertices entweder in einem Binärformat oder in einer ASCII-Darstellung. Der strukturelle Aufbau des ASCII-Formats kann Listing 4.1 entnommen werden. Er stammt aus [Wik09]. Dabei steht name für einen beliebigen Namen, der der Dreiecksmenge gegeben werden kann. Der Abschnitt von facet bis endfacet kann entsprechend der Anzahl der zu speichernden Dreiecke beliebig oft wiederholt werden und spezifiziert jeweils eine Facette. n_1 bis n_3 sind die Komponenten des Normalenvektors der Facette, p_{1x} bis p_{3z} die Komponenten der drei Eckpunkte des Dreiecks.

```

1 solid name
2     facet normal n1 n2 n3
3         outer loop
4             vertex p1x p1y p1z
5             vertex p2x p2y p2z
6             vertex p3x p3y p3z
7         endloop
8     endfacet
9     ...
10 endsolid name

```

Listing 4.1: Aufbau des ASCII-STL-Formats.

4.1.2 Vor- und Nachteile von Dreiecksmengen und des STL-Modells

Die Festlegung auf die Verwendung des STL-Modells zur Beschreibung des virtuellen Ideals der deflektometrisch zu inspizierenden Komponente birgt Vor- und Nachteile. Diese sind zum einen dem Umstand geschuldet, dass die Oberfläche aus Vielecken aufgebaut ist. Zum anderen hängen sie mit den speziellen Eigenschaften des STL-Formats zusammen. In der folgenden Auflistung finden sich alle wesentlichen Vorzüge der Beschreibung von Geometrien durch Dreiecksmengen.

- Viele der im Rekonstruktionsalgorithmus häufig auftretenden Operationen können mit Dreiecken besonders effizient durchgeführt werden. Vor allem der Schnitt eines Strahls mit einem Dreieck kann in wenigen Schritten unter ausschließlicher Verwendung von Addition, Multiplikation und Vergleich vollzogen werden. Näheres dazu findet sich in den Unterabschnitten 4.1.3 und 4.5.1.
- Auch Anfragen wie die Bestimmung eines Normalenvektors an einer beliebigen Stelle der Oberfläche können zeiteffizient abgearbeitet werden.
- Als Bounding Box (kurz: BB) wird derjenige Quader im Raum bezeichnet, der ein einzelnes Primitiv oder eine Menge von Primitiven komplett enthält. Solche BBs lassen sich bei der Verwendung von Facetten als Primitiven ebenfalls effizient bestimmen.
- Ein weiterer Vorteil von Dreiecken liegt in der Tatsache, dass sie immer konvex sind und ihre drei Eckpunkte immer in einer Ebene liegen.
- Die Speicherung von Facetten durch ihre Eckpunkte ist darüber hinaus sehr elegant. Andere Oberflächenbeschreibungen speichern zumeist die Koeffizienten von mathematischen Funktionen wie Polynomen. Diese müssen zuerst mit den entsprechenden Basisfunktionen verrechnet werden, was oftmals zu einem höheren arithmetischen Aufwand führt, als dies bei Polygonen der Fall ist.
- Aus Dreiecken lassen sich alle anderen planaren Polygone zusammenbauen. So ergeben zwei Facetten in der Ebene mit einer gemeinsamen Seite ein Viereck.

Demgegenüber stehen allerdings diverse Mängel, die Facettenmengen aufgrund ihrer Natur aufweisen.

- Die größte Stärke von Dreiecksmengen, dass sie aus einfachen, planaren Primitiven aufgebaut sind, mit denen sich viele Prozeduren schnell ausführen lassen, ist auch deren größte Schwäche. Da Dreiecke eben sind, lassen sich gekrümmte Oberflächen mit ihnen nur approximativ beschreiben. Echte Krümmungen, wie zum Beispiel bei einer Kugel, können mit Polygonen allgemein nicht exakt beschrieben werden.
- Je mehr Dreiecke verwendet werden dürfen, desto kleiner fällt der Approximationsfehler aus. Das führt zu dem klassischen Trade-Off, dass ein Mehr an Exaktheit mit einem Mehr an Speicher- und Rechenaufwand erkaufte werden kann.
- Eine Folge der Diskretisierung der Objektoberfläche mit planaren Primitiven ist, dass eben diese Diskretisierung auch für die Normalen der Dreiecke gilt. Da das STL-Format nur eine einzige Normale pro Oberfläche speichert, gilt diese auch für die gesamte Fläche der Facette. Während bei der Überschreitung der Grenze zwischen zwei Dreiecken die Normale „ruckartig umschlägt“, ist sie innerhalb eines Polygons konstant, auch wenn dieses Polygon Teil einer Krümmung ist. Abschnitt 4.6 beschäftigt sich zum Ende dieses Kapitels mit eben jener Problematik und stellt eine Palette von mehr oder weniger befriedigenden Abhilfen vor.

Zuletzt seien noch einige Mängel genannt, die der Struktur des STL-Modells geschuldet sind und die ebenfalls beachtet werden müssen, falls man mit dem STL-Modell arbeiten will.

- Einer dieser Nachteile ist in der Art und Weise begründet, wie das STL-Format die Dreiecksmenge speichert. Da die Oberfläche aus Dreiecken zusammenhängend sein soll, teilt sich eine Facette ihren Eckpunkt in der Regel mit mehreren anderen Facetten (siehe Abbildung 4.2). Aufgrund der facetten- und nicht vertexweisen Speicherung im STL-Format wird so ein gemeinsamer Vertex jedoch mehrfach gespeichert, nämlich für jede Facette, die an den Eckpunkt angrenzt. Solche redundanten Vertices werden im Folgenden auch als Vertexduplikate bezeichnet. Mit deren Elimination beschäftigt sich Abschnitt 4.2.
- Bei Triangulationen entsprechen in der Praxis nur noch die Vertices der Originaloberfläche. Die Seiten der Polygone fallen bei ursprünglich gekrümmten Formen nicht mehr mit der Originaloberfläche zusammen. Dieser Informationsverlust wird durch die Verwendung des STL-Modells verstärkt, denn die Information, welchen Normalenvektor die Originaloberfläche an der Stelle eines Vertex, die sogenannte Vertexnormale, hatte, wird bei STL nicht mitgespeichert. Benötigt man die Vertexnormalen, so müssen diese aus den vorhandenen Flächennormalen abgeleitet werden.

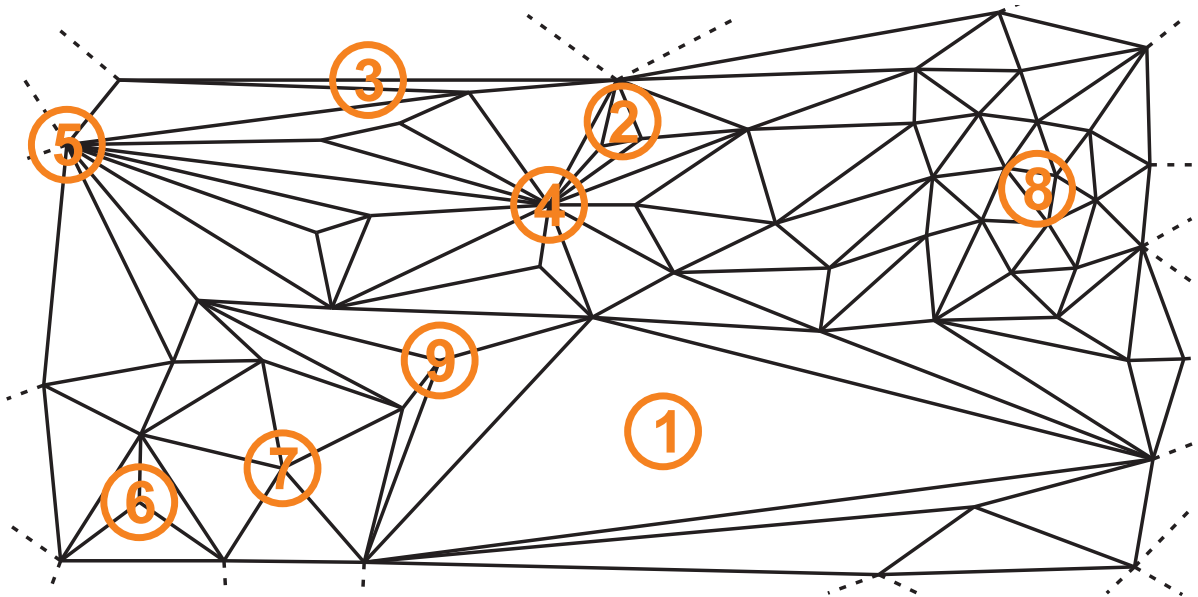


Abbildung 4.2: Die Abbildung zeigt ein zweidimensionales Dreiecksnetz mit unterschiedlichen Variationen von Facetten und Vertices. Während (1) ein relativ großes Dreieck ist, gehört (2) zu den kleinsten im Netz. (3) besitzt zwar eine relativ lange Außenkante, sein Flächeninhalt ist aber eher gering. An den Vertices bei (4) und (5) treffen sich sehr viele Facetten, bei (6) jedoch nur drei. Ließe man den Eckpunkt (7) weg, so würde sich ein Fünfeck ergeben. Im Gegensatz zur Region bei (8), die aus regelmäßigen Facetten besteht, ist (9) ein eher unharmonischer Bereich.

4.1.3 Raycasting

Die mit großem Abstand wichtigste, aber in der Summe auch aufwändigste Operation mit der Facettenmenge, ist das sogenannte Raycasting. Dabei handelt es sich im Falle des STL-Modells um die Beantwortung der Frage, welche Facette der Dreiecksmenge ein Strahl als erstes schneidet und wo dies der Fall ist. Ein solcher Strahl könnte von der Kamera des deflektometrischen Messaufbaus stammen. Alle Facetten zusammen modellieren das Ideal des zu prüfenden Bauteils. Unter der ersten Facette, die geschnitten wird, versteht man also diejenige, die vom Betrachter gesehen wird und die sich nicht auf der aktuellen Rückseite der Komponente befindet.

Raycasting wird für viele Anwendungen benötigt. Beim Raytracing wird mittels Raycasting immer derjenige Schnittpunkt mit dem STL-Modell berechnet, den ein (eventuell bereits mehrfach reflektierter) Sichtstrahl hat, um diesen, je nach Konfiguration, ein weiteres Mal zu reflektieren beziehungsweise zu streuen; oder um den Farbwert zu berechnen, den der zum Sichtstrahl gehörige Bildpunkt haben soll. Raytracing ist also mehrfach hintereinander ausgeführtes Raycasting. Um die Reflexion eines Strahls auf der Modelloberfläche zu bestimmen ist der Normalenvektor am Schnittpunkt notwendig. Dieser ist mit der getroffenen Facette verknüpft, so dass diese erst ermittelt werden muss. Eben dieser Reflexionsvorgang ist für den Rekonstruktionsalgorithmus und dessen Simulation essentiell. Das Raycasting ist diejenige Operation der Positionsbestimmung, die in der Summe mit Abstand die größte Laufzeit hat, da sie bei jeder Iteration der NLO für alle Sichtstrahlen der Kamera nötig ist.

Folgendes Zahlenbeispiel soll den Gesamtaufwand des Raycastings innerhalb des Rekonstruktionsalgorithmus verdeutlichen. Die Grundoperation beim Raycasting ist die Antwort auf die Frage, ob ein Strahl eine Facette schneidet und falls ja, wo. Das STL-Modell kann aus $15 \cdot 10^2$ bis $5 \cdot 10^5$ Facetten bestehen. Bei einer naiven Vorgehensweise würde man jeden Sichtstrahl der Kamera mit allen Facetten des Modells schneiden, um diejenigen Facetten zu bestimmen, die von den Sichtstrahlen als erstes getroffen werden. Bei einer üblichen Kameraauflösung von 1024×768 Bildpunkten, wären dies bis zu $1024 \cdot 768 \cdot 5 \cdot 10^5 \approx 4 \cdot 10^{11}$ Schnittprozeduren pro Simulationsbild. Für die vollständige Ausführung des Rekonstruktionsalgorithmus werden etwa $2 \cdot 10^3$ Simulationsbilder benötigt. Dies entspricht zwar nicht der Anzahl der Iterationen der NLO, aufgrund der Sechsdimensionalität der Zielfunktion und der Arbeitsweise des Verfahrens von Nelder und Mead, werden jedoch pro Iterationsschritt mehrere Pixpos-Bilder simuliert. Somit werden im angenommenen Szenario etwa $8 \cdot 10^{14}$ Schnittoperationen durchgeführt, was eine astronomische Zahl ist, zumindest unter der Bedingung, dass die Rekonstruktion innerhalb weniger Sekunden durchgeführt werden soll. Dabei werden bis hierher Schritte wie die Erzeugung der Sichtstrahlen, der Schnitt der reflektierten Sichtstrahlen mit dem Bildschirm, die Umrechnung der Pixelpositionen in das S-KoSy und die Vektordifferenzbildung noch überhaupt nicht beachtet.

Zwei Maßnahmen sollen trotzdem akzeptable Laufzeiten ermöglichen. Zum einen die massive Parallelität programmierbarer Grafikprozessoren und zum anderen die in Abschnitt 4.3 vorgestellten Strukturen, die es erlauben, das Raycasting eines einzelnen Strahls mit der Facettenmenge nicht in linearer, sondern logarithmischer Zeit im Verhältnis zur Anzahl der Facetten zu realisieren.

4.2 Elimination von Vertexduplikaten

Da jede Facette im STL-Format einzeln gespeichert wird, werden Vertices redundant hinterlegt. Hieraus ergeben sich Vertexduplikate. Je nach Triangulierung benötigt dadurch die Aufbewahrung der Vertices fünf bis sechs Mal mehr Kapazität, als minimal notwendig. Aufgrund des billigen Speicherplatzes auf Festplatten ist dies bei der persistenten Speicherung noch verkraftbar. Zur Laufzeit kann die Größe des RAM auf Host und vor allem Device jedoch ein limitierender Faktor sein. Aus diesem Grund ist es sinnvoll, jeden Vertex der Facettenmenge nur einmal abzulegen. Alle Eckpunkte von Dreiecken, die diesen Vertex gemein haben, verweisen dann mit einem Zeiger auf das Vertexunikat.

Darum wird beim Einlesen der STL-Datei für jeden einzelnen Vertex überprüft, ob ein anderer Eckpunkt mit den gleichen Koordinaten bereits zu einem früheren Zeitpunkt verarbeitet wurde. Dadurch wird eine Duplikatelimination durchgeführt. Hashing [Fre04] ermöglicht diesen Abgleich in $O(1)$ pro Vertex.

Damit Hashing durchgeführt werden kann, wird eine Hashfunktion benötigt. Im vorliegenden Fall weist sie einem Vertex einen eindeutigen Schlüssel zu. Jeder Vertex ist aus drei Komponenten zusammengesetzt, jeweils eine Koordinatenkomponente pro Dimension im $\mathbb{R}^3$. Somit muss eine Hashfunktion aus den drei Komponenten einen Schlüssel generieren. Erschwerend kommt hinzu, dass eine Vertexkoordinate in Gleitkommazahlen ausgedrückt wird. Vergleiche auf Gleichheit von Gleitkommazahlen sind strengstens zu unterlassen, da hierbei Rundungs- und Darstellungsfehler [IEE85] die Auswertung verfälschen können. Nur exakt übereinstimmende Vertices sollen als gleich identifiziert werden. Für ähnliche Vertices, die nur einen geringen Abstand voneinander haben, darf dies nicht gelten.

Folgende Hash-Funktion erfüllt die geforderte Eigenschaft. Eine Gleitkommazahl einfacher Genauigkeit, ein `float`, ist 32-Bit lang. Das Prinzip der Interpretation jener 32 Bit für eine Gleitkommazahl kann in [Don05] nachgeschlagen werden. Anstatt diese 32 Bit als Gleitkommazahl zu verstehen, können sie auch eine vorzeichenlose Ganzzahl, einen `unsigned int`, ausdrücken, der ebenfalls 32-Bit lang ist. Tabelle 4.1 veranschaulicht die alternative Deutung. Für jede Komponente der Vertexkoordinate wird die alternative Interpretation angewandt. Um aus den drei resultierenden `unsigned int` einen einzelnen Schlüssel zu bekommen, werden sie bitweise mit der XOR-Operation verknüpft. Listing 4.2 zeigt den C-Code, der diese Hashfunktion implementiert.

```

1 unsigned int hash(float3 *vertex) {
2     unsigned int result;
3     unsigned int *pointer;
4
5     pointer = (unsigned int*) vertex;
6     result = *pointer++;
7
8     for (int i = 1; i < 3; i++) {
9         result ^= *pointer++;
10    }
11
12    return result;
13 }
```

Listing 4.2: Code einer Hashfunktion für Vertices.

Zwar ist die vorgestellte Hashfunktion sehr schnell, allerdings erzeugt sie Kollisionen. Eine Kollision tritt dann auf, wenn die Hashfunktion zwei unterschiedlichen Eingabewerten den selben Schlüssel zuweist. Da der Schlüsselraum 2^{32} Elemente besitzt, der Eingaberaum theoretisch jedoch $(2^{32})^3$, sind Kollisionen unvermeidbar. Deshalb müssen diese gesondert betrachtet werden, denn durch die Kollisionen können zwei unterschiedliche Vertices den gleichen Schlüssel besitzen, was für die Duplikatelimination kontraproduktiv ist.

Folgendes Beispiel zeigt eine Kollision. Drückt man die beiden Koordinaten (126,8701171875; 123,82183837890625; 35,885826110839844) und (125,70004272460937; 122,89067077636719; 34,808792114257813) binär in `float`-Schreibweise aus, so ergeben sich (01000010111110110111101100000000; 01000010111101111010010011001000; 01000010000011111000101100010110) und (01000010111110110110011001100110100; 01000010111101011100100000000110; 01000010000010110011110000110100). Wird nun die bitweise XOR-Operation durchgeführt, erhält man die beiden Schlüssel $01000010111110110111101100000000 \oplus 01000010111101111010010011001000 \oplus 01000010000011111000101100010110 = 01000010000001011001001001011110$ und $010000101111101101100110011001100 \oplus 01000010111101011100100000000110 \oplus 01000010000010110011110000110100 = 01000010000001011001001001011110$ als Ergebnis der Hashfunktion. Diese beiden Schlüssel sind gleich, obwohl die Eingabepunkte unterschiedlich sind.

4.3 Hierarchisierung der Facettenmenge zur Laufzeitverbesserung

4.3.1 Notwendigkeit der Hierarchisierung

Bereits in Unterabschnitt 4.1.3 wird anhand eines Zahlenbeispiels gezeigt, dass die Anzahl der Strahl/Dreiecks-Schnitte im Rekonstruktionalgorithmus unpraktikabel hoch ist. Da die dortige Vorgehensweise sehr naiv ist, bietet sie ein großes Optimierungspotential. Bisher wird angenommen, dass für jeden einzelnen Sichtstrahl der Kamera ein Test mit jeder Facette des Modells durchgeführt werden muss, um dasjenige Primitiv zu identifizieren, das vom Strahl als erstes geschnitten wird. Anstatt wie bisher die gesuchte Facette in $O(n)$ bezüglich der Anzahl der Dreiecke zu bestimmen, ist es wünschenswert, diesen Vorgang in logarithmischer Zeit, also in $O(\log n)$, zu bewerkstelligen. Übertragen in die Praxis bedeutet das, dass ein Verfahren gesucht wird, das in jedem Schritt für einen bestimmten Anteil der Facettenmenge, zum Beispiel die Hälfte oder ein Drittel, die Aussage treffen kann, ob die in der Teilmenge enthaltenen Dreiecke definitiv nicht von einem gegebenen Strahl geschnitten werden.

Hierarchisierungen der Dreiecksmenge ermöglichen ein derart effizientes Raycasting. Im Folgenden sollen drei Arten der Hierarchisierung vorgestellt werden. Bei diesen Datenstrukturen handelt es sich in allen drei Fällen um spezielle Bäume. Octrees werden nur der Vollständigkeit wegen und als Einstieg in die gleichmäßige Gliederung von würfelförmigen Datensätzen vorgestellt. In der Praxis werden wesentlich häufiger k-dimensional Trees (kurz: kd-Trees) und BVHs eingesetzt. Je nach Anwendungsszenario ist die eine Methode der anderen überlegen. BVHs eignen sich besser für dynamische Szenen, in denen Teile der Hierarchisierung aktualisiert werden müssen [WMG⁺07], während kd-Trees effizienter gespeichert und traversiert werden können [Hav00]. Die in dieser Arbeit gewählte Hierarchisierungsart sind BVHs, weshalb BVHs in 4.4 und 4.5 vertieft behandelt werden. Aufgrund der praktischen Relevanz von kd-Trees wird an geeigneter Stelle auf weiterführende Literatur verwiesen, die die für BVHs gezeigten Schritte auch an kd-Trees erläutern.

4.3.2 Octree

Octrees sind Erweiterungen von Binärbäumen und Quadrees. Während Binärbäume eindimensionale und Quadrees zweidimensionale Daten verwalten, sind Octrees in der Lage, dreidimensionale Daten zu strukturieren. Es handelt sich um einen unbalancierten Suchbaum, in dem jeder innere Knoten acht Söhne besitzt.

Sämtliche zu verwaltenden Daten, zum Beispiel die Facetten der Dreiecksmenge, sind zu Anfang in einem Würfel lokalisiert. Er bildet die Wurzel des Octrees. Für die nächste Hierarchiestufe wird der Würfel in acht Unterwürfel zerteilt, deren Vereinigung den ursprünglichen Kubus ergibt. Jede der drei Dimensionen wird hierzu in zwei gleich große Teile zerlegt. Somit ergeben sich die acht Oktanten des Ausgangswürfels, die die Söhne der Wurzel werden. Diese Zerlegung wird nun so lange rekursiv für jeden Sohn fortgesetzt, bis sich in einem Unterwürfel maximal ein zu verwaltendes Objekt wiederfindet. Am Ende findet eine Verknüpfung der Primitive mit den Blättern des Octrees statt. Innere Knoten des Octrees verwalten also nur Verweise auf Unterbäume, die zu verwaltenden Objekte

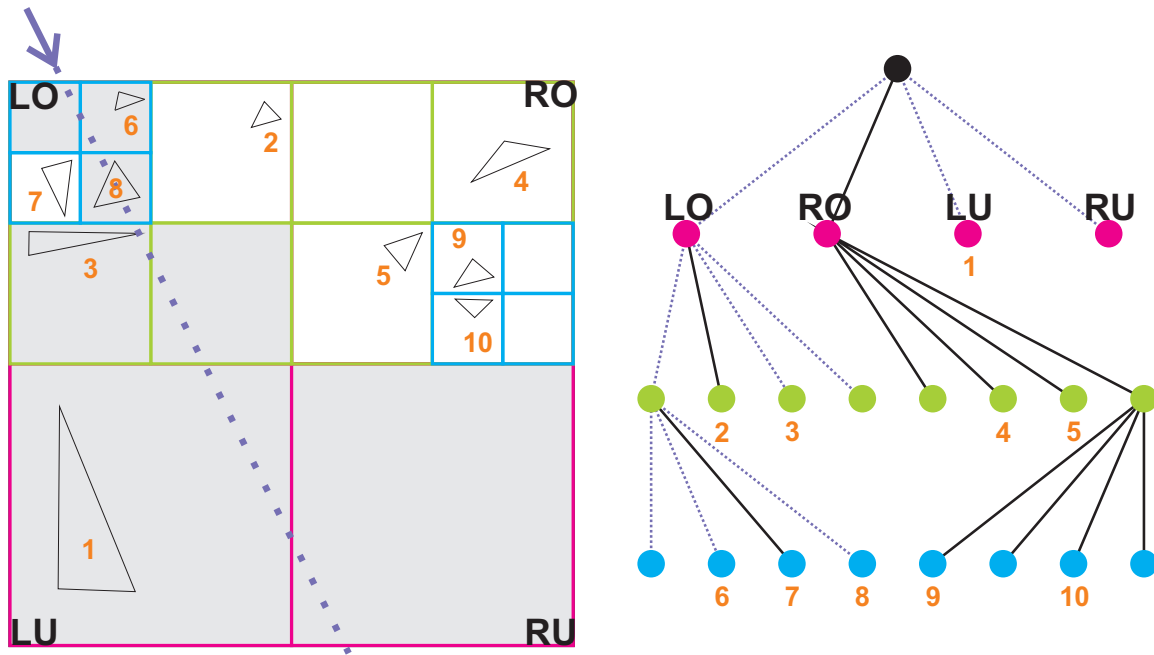


Abbildung 4.3: Anhand eines Quad- soll die Arbeitsweise eines Octrees gezeigt werden. Das Quadrat links wird in seine vier magentafarbene Quadranten zerlegt. Die zweite Hierarchiestufe bilden grüne, die dritte blaue Quadrate. Ein regelmäßiges Viereck wird nur dann unterteilt, falls sich mehr als eines der insgesamt zehn Dreiecke darin befindet. Mit violett ist der Sichtstrahl eingezeichnet, für den das Raycasting durchgeführt werden soll. Schneidet er ein Quadrat (grau hinterlegt), so muss im Quadtree rechts in den entsprechenden Unterbaum abgestiegen werden. Dies wird durch violett gepunktete Kanten in der Hierarchie angezeigt. Somit kommen die Facetten (1), (3), (6) und (8) als Kandidaten für einen Schnitt mit dem Strahl in Frage.

werden von den Blättern referenziert.

Um ein Raycasting durchzuführen, wird an der Wurzel des Octrees begonnen. Dort wird überprüft, welche der acht Oktanten, die von den Söhnen der Wurzel verwaltet werden, einen Schnittpunkt mit dem gegebenen Strahl besitzen. Für all diejenigen Oktanten, auf die das zutrifft, muss im nächsten Schritt für deren Söhne getestet werden, ob die Sohnräume wiederum vom Strahl geschnitten werden. Diejenigen Unterbäume, deren Wurzeloktante nicht vom Strahl getroffen wird, müssen nicht tiefer traversiert werden. Dieses rekursive Vorgehen wird solange praktiziert, bis man die Blätter des Octrees erreicht hat. Verwaltet ein erreichtes Blatt eine Facette, so ist die gefundene Facette ein Kandidat für den Schnitt mit dem Strahl. Dafür können durchaus mehrere Facetten in Frage kommen, die dann einzeln auf Schnitt überprüft werden müssen.

Abbildung 4.3 zeigt anstatt eines Octrees einen Quadtree, da sich die Arbeitsweise besser in der Ebene als im Raum illustrieren lässt. Anstatt einen Würfels in Oktanten wird ein Quadrat in Quadranten zerlegt. Das Grundprinzip von Aufbau und Traversierung des Baums ist das selbe. Links ist der unterteilte Raum mit den zu verwaltenden Objekten zu sehen, rechts der resultierende Octree.

Will man Octrees zum Management von Dreiecksmengen und vor allem Dreiecksnetzen verwenden, stellt man fest, dass diese Bäume dafür eher ungeeignet sind. Zum einen ist die Frage zu klären, wie man eine Facette einem Oktanten zuordnet. Verwendet man dafür das Zentrum des Dreiecks, so ist es möglich, dass die Vertices der Facette in benachbarten Würfeln liegen, was gesondert beachtet werden müsste. Benutzt man stattdessen die drei Vertices selbst als Entscheidungsgrundlage, ergibt sich wiederum das Problem, dass diese nicht innerhalb ein und desselben Würfels liegen müssen, da die Unterteilung in Oktanten keinerlei Rücksicht auf die Facettenmenge nimmt sondern die Würfel pro Dimension immer halbiert. Zum anderen sind die Seitenlinien der Dreiecke im Regelfall nicht in Richtung der Koordinatenachsen ausgerichtet. Somit wird es bei einem Dreiecksnetz unmöglich zu garantieren, dass sich maximal eine Facette innerhalb eines Blattwürfels befindet.

Darum sind andere Strukturen wie BVHs bei der Hierarchisierung von Dreiecksmengen besser geeignet. Octrees eignen sich stattdessen besser zur Raumindizierung, zur Entfernung von verdeckten Oberflächen oder zur effizienten Speicherung von Binärbildern.

4.3.3 kd-Tree

Ein kd-Tree ist grundsätzlich ein unbalancierter, binärer Suchbaum zur Speicherung und Verwaltung von Punkten aus dem $\mathbb{R}^k$. Während bei homogenen kd-Trees jeder Knoten des Baums einen Datensatz speichert, ist dies bei inhomogenen kd-Trees nur in den Blättern der Fall. Die inneren Knoten sind dann lediglich Verweise auf andere Knoten und dienen der Navigation zu den Blättern.

Bei den inhomogenen kd-Trees sei $H_i(t) = \{x = (x_1, \dots, x_i, \dots, x_k) | x_i = t\}$, $1 \leq i \leq k$ die $(k - 1)$ -dimensionale Hyperebene an der Stelle t .

Um einen kd-Baum aufzubauen, wird im ersten Schritt ein $t_1 \in \mathbb{R}$ so gewählt, dass die Hyperebene $H_1(t_1)$ die zu verwaltende Punktemenge X in zwei möglichst gleichmächtige Untermengen zerteilt. In die „linke“ Menge X_l kommen dabei diejenigen $x \in X$, für die $x_1 < t_1$ gilt, in die „rechte“ Menge X_r stattdessen all jene Elemente, für die $x_1 > t_1$ gilt. Folglich ist $X = X_l \cup X_r$. Im nächsten Schritt muss sowohl für X_l als auch für X_r ein separates $t_{21}, t_{22} \in \mathbb{R}$ für zwei Hyperebenen $H_2(t_{21})$ und $H_2(t_{22})$ gefunden werden, die aufgrund der Definition von $H_i(t)$ orthogonal zu $H_1(t)$ sind. $H_2(t_{2j})$, $j \in \mathbb{Z}_{[1,2]}$ soll je eine der beiden Mengen X_l und X_r ebenfalls in zwei möglichst gleichmächtige Untermengen zerteilen. Das Finden von Hyperbenen $H_i(t_{ij})$, $i \in \mathbb{Z}_{[1,k]}$, $j \in \mathbb{Z}_{[1,k^2]}$ wird so lange rekursiv fortgesetzt, bis für jedes $x_a \in X$ gilt, dass es durch mindestens eine Hyperebene $H_i(t)$ von allen anderen Punkten $x_b \in X \setminus \{x_a\}$ getrennt ist. Dabei ist es möglich, dass für alle $1 \leq i \leq k$ bereits Hyperebenen bestimmt wurden, aber immer noch Punkte $x \in X$ vorliegen, die nicht durch eine Hyperebene voneinander getrennt sind. In diesem Fall beginnt die Bestimmung von Trennunterräumen der Dimension $(k - 1)$ wieder bei $i = 1$.

Möchte man die Dreiecke einer Facettenmenge mit einem kd-Tree verwalten, so liegt der Fall $k = 3$ vor. Die teilenden Hyperebenen sind dann Ebenen im Raum. Zu beachten ist jedoch, dass die bisher vorgestellte Variante des kd-Trees nur Punkte und keine Dreiecke verwaltet. Dafür müssen ähnlich wie bei den Octrees Anpassungen vorgenommen werden, auf die hier nicht näher eingegangen wird. Stattdessen sei abermals auf BVHs verwiesen, die in dieser Arbeit zur Hierarchisierung der Facettenmenge eingesetzt werden.

Abbildung 4.4 zeigt für den Fall $k = 2$ den Aufbau eines kd-Trees. Die zu verwaltenden

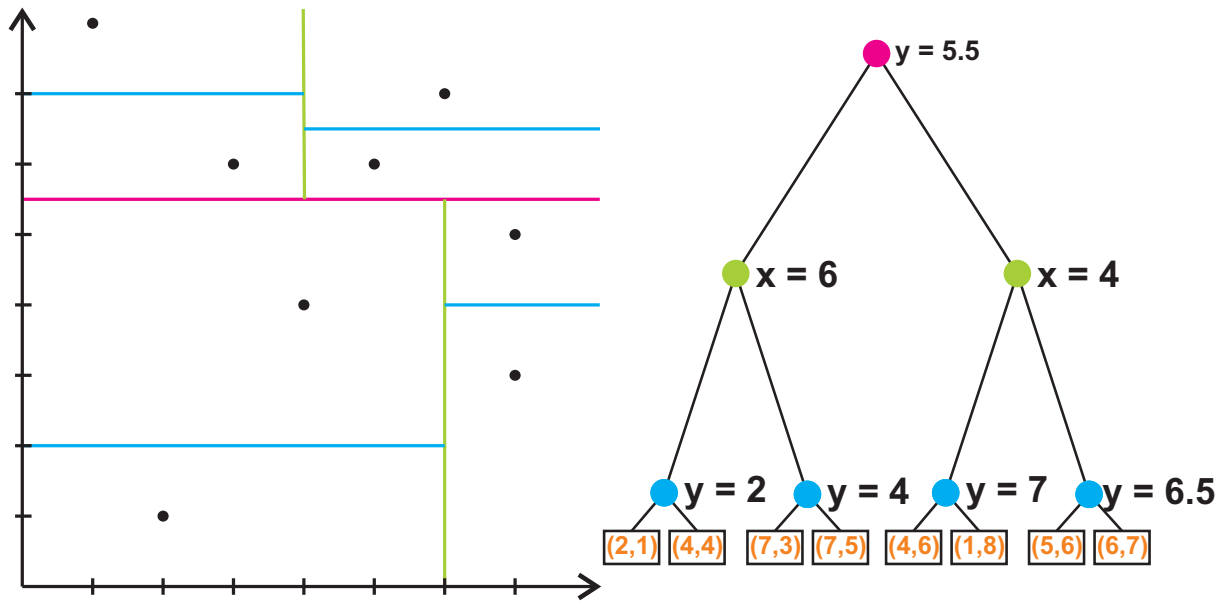


Abbildung 4.4: Auf der linken Seite befinden sich acht Punkte in der Ebene, die durch einen kd-Tree verwaltet werden sollen. Die erste Hyperebene $H_1(t_1)$ ist die rot eingezeichnete $y = 5,5$ -Gerade und zerteilt die Punktemenge in je vierelementige Teilmengen „oben“ und „unten“. Jede der beiden Teilmengen wird wiederum unterteilt. Dies geschieht durch zwei zur y -Achse parallele Geraden $H_2(t_{21})$ und $H_2(t_{22})$, die grün eingezeichnet sind. Eine Hierarchiestufe tiefer zerteilen die vier blauen Geraden $H_3(t_{3j}), j \in \mathbb{Z}_{[1,4]}$ die entstandenen Teilmengen derart, dass jedes Paar von Punkten durch mindestens eine Hyperebene voneinander getrennt ist. An dieser Stelle terminiert der Aufbau des kd-Trees. Für die Hyperebenen gilt $H_1(t_1) \perp H_2(t_{2j})$ und $H_2(t_{2j}) \perp H_3(t_{3j})$. Da $k = 2$ ist $H_1(t_1) \parallel H_3(t_{3j})$.

Punkte liegen also in der Ebene und bei den $(k - 1)$ -dimensionalen Hyperebenen handelt es sich um Geraden.

kd-Bäume werden trotzdem erwähnt, da sie in der Praxis oft zum Einsatz kommen, wenn es um die Strukturierung von Objektmengen geht, die Elemente im Raum enthalten. Einer ihrer Vorteile liegt im niedrigen Speicherverbrauch, der in die Komplexitätsklasse $O(k \cdot n)$ fällt. Darüber hinaus können kd-Trees besonders schnell und effizient traversiert werden.

4.3.4 BVH

Bei einer Bounding Volume Hierarchy handelt es sich ebenso wie bei einem kd-Tree um einen unbalancierten, binären Suchbaum, der k -dimensionale Objekte indexieren kann. Mit einem inhomogenen kd-Tree teilt eine BVH ebenfalls die Eigenschaft, dass die zu verwaltenden Primitive (hier die Facetten) durch die Blätter der BVH referenziert werden. Innere Knoten verweisen nur auf andere BVH-Knoten. Jedes Blatt verweist auf genau ein zu verwaltendes Objekt.

BVHs sind besonders gut geeignet, Facetten zu verwalten. Im Gegensatz zu Octrees und

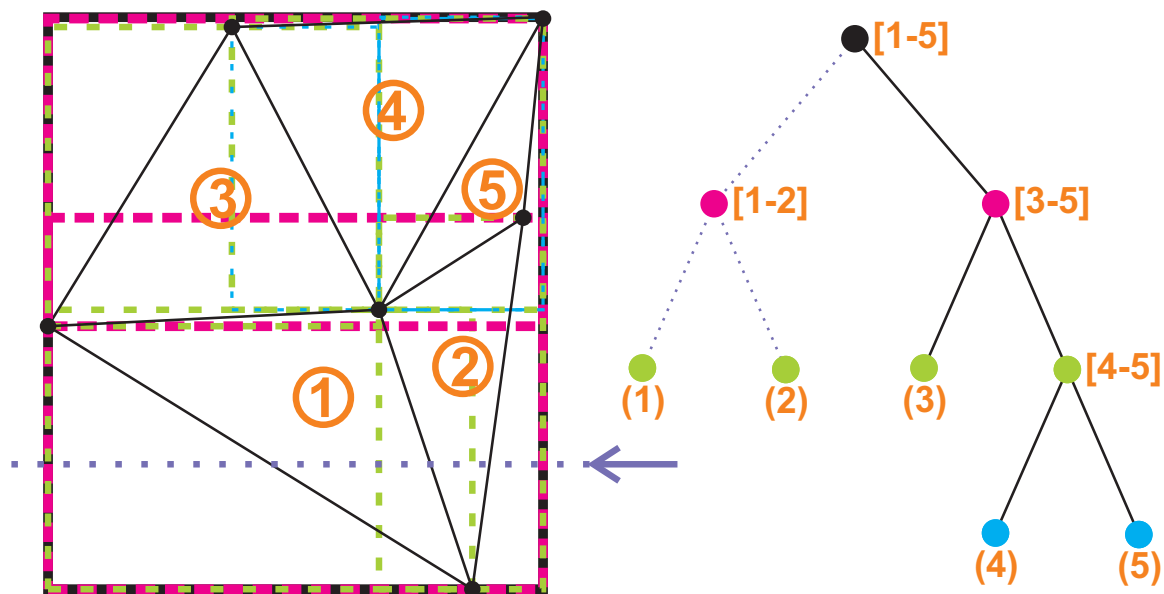


Abbildung 4.5: Links werden fünf Dreiecke in der Ebene, nummeriert von (1) bis (5), gezeigt. Diese werden allesamt von einem schwarzen, begrenzenden Volumen, einer AABB, umschlossen. Im nächsten Schritt werden die fünf Facetten in zwei Mengen unterteilt. Die eine Menge enthält die Facetten (1) und (2), die andere Menge die Objekte (3) bis (5). Für beide Mengen wird die kleinste umschließende AABB bestimmt, die jeweils rot eingezeichnet ist. Für den nächsten Rekursionsschritt werden grüne und den übernächsten blaue AABBs verwendet.

kd-Trees werden BVHs explizit mit dem Hintergrund der Handhabung von Dreiecksmengen besprochen.

Ähnlich einem Octree werden die zu hierarchisierenden Objekte von einem Volumen umgeben. Ein derartiges Volumen muss die selbe Dimension wie die zu hierarchisierenden Gegenstände besitzen. Dabei ist die Art des Volumens generisch. Man ist nicht nur auf Würfel beschränkt. Kugeln sind ebenfalls denkbar. Öfter verwendet werden BBs oder Axis Aligned Bounding Boxes (kurz: AABB). Letztere unterscheiden sich von BBs, indem sämtliche Quaderflächen parallel zu den Ebenen des Koordinatensystems sind. Das vereinfacht zahlreiche Operationen wie das Raycasting. Generell gilt für die Wahl des Volumentyps, dass dieser die zu verwaltenden Primitive möglichst eng umschließen soll, es also keine Regionen im Volumen geben soll, in denen sich keine Primitive befinden.

Abschnitt 4.4 beschäftigt sich gesondert mit dem Aufbau von BVHs. An dieser Stelle wird klar, warum sich BVHs besonders gut für die Verwaltung von Dreiecksmengen im Raum eignen. Außerdem wird dort erläutert, wie die begrenzenden Volumen (daher der Name Bounding Volume) gewählt werden.

Vorwegnehmend zeigt Abbildung 4.5 ein einfaches Beispiel mit fünf zusammenhängenden Dreiecken in der Ebene, für die eine BVH aufgebaut wird. Auf die Traversierung von BVHs wird in Abschnitt 4.5 gesondert eingegangen. Ebenso wie in Abschnitt 4.4 liegt dort ein besonderer Fokus auf der Beschleunigung der Algorithmik durch GPUs.

4.4 Aufbau einer BVH

4.4.1 Aufbau durch die CPU

In diesem Abschnitt soll der klassische Aufbau einer BVH erklärt werden. Dadurch entsteht ein Verständnis dieser Hierarchisierungsmethode sowie eine Erklärung für die logarithmische Laufzeit des Raycastings bei Verwendung von BVHs. Als zu strukturierende Objekte werden nun explizit Dreiecke im Raum verwendet.

Um eine Facettenmenge zu hierarchisieren, wird, wie bereits erwähnt, ein kleinstmögliches Volumen gesucht, das alle Dreiecke umschließt. Wählt man hierfür eine AABB, so muss die komplette Vertexmenge $V = \{v_i\}_{i \in \mathbb{Z}_{[1,n]}} = \{(x_i, y_i, z_i)\}_{i \in \mathbb{Z}_{[1,n]}}$ durchlaufen und jeweils das Minimum und Maximum der x-, y- und z-Koordinate bestimmt werden. n ist die Kardinalität der Facettenmenge. Mathematisch ergibt sich dann die AABB aus den sechs Seiten $\{(x, y, z) | x = x_{links}\}$, $\{(x, y, z) | x = x_{rechts}\}$, $\{(x, y, z) | y = y_{oben}\}$, $\{(x, y, z) | y = y_{unten}\}$, $\{(x, y, z) | z = z_{vorne}\}$ und $\{(x, y, z) | z = z_{hinten}\}$ des Quaders mit $x_{links} = \min(x_i)_{i \in \mathbb{Z}_{[1,n]}}$, $x_{rechts} = \max(x_i)_{i \in \mathbb{Z}_{[1,n]}}$, $y_{oben} = \min(y_i)_{i \in \mathbb{Z}_{[1,n]}}$, $y_{unten} = \max(y_i)_{i \in \mathbb{Z}_{[1,n]}}$, $z_{vorne} = \min(z_i)_{i \in \mathbb{Z}_{[1,n]}}$ und $z_{hinten} = \max(z_i)_{i \in \mathbb{Z}_{[1,n]}}$. Die eben ermittelte AABB wird der Wurzel der BVH zugeordnet.

Im nächsten Schritt wird die Ausgangsfacettenmenge in zwei Mengen zerlegt. Mögliche Vorgehensweisen dafür finden sich in Unterabschnitt 4.4.3. Dabei muss es nicht unbedingt von vorrangigem Interesse sein, dass die beiden Teilmengen gleich viele Elemente enthalten. Aus den beiden Teilmengen ergeben sich die zwei Söhne der Wurzel. Für jede der Teilmengen wird nun wiederum das kleinstmögliche umgebende Volumen bestimmt. Dabei geht man genauso vor wie beim Finden der AABB für die gesamte Dreiecksmenge. Diese neuen AABBs der beiden Teilmengen sind maximal so groß wie das Ausgangsvolumen. Normalerweise sind sie aber kleiner. Sie werden den im aktuellen Schritt erzeugten Söhnen zugeordnet. Eine Facette befindet sich immer nur in einer einzigen AABB pro Hierarchiestufe.

Jetzt wird klar, warum BVHs auch Objekte verwalten können, die nicht punktförmig sind, da sich die Größe der begrenzenden Volumen dynamisch den Facetten anpasst. Bei Octrees ist aufgrund der Aufteilung eines Würfels in Oktanten die Unterwürfelgröße fest. Gefundene umgebende Volumen der beiden Teilmengen dürfen sich darüber hinaus überlappen.

Rekursiv spaltet man nun immer weiter eine Facettenteilmenge in zwei Teilmengen auf und bildet deren kleinstmögliche AABBs. Jeder innere Knoten der BVH speichert dabei das umgebende Volumen, das alle Primitive umschließt, die durch die Blätter des Unterbaums des inneren Knotens verwaltet werden. Hat man die Objektmenge soweit rekursiv aufgeteilt, dass sie nur noch aus einem einzigen Dreieck besteht, so wird für dieses Objekt kein umschließendes Volumen mehr bestimmt. Stattdessen verweist der Baumknoten direkt auf das Primitiv. Die Rekursion endet an dieser Stelle und es entsteht ein Blatt der BVH.

Die Idee, die hinter den umschließenden Volumen steht, kann wie folgt erläutert werden. Ziel ist die Beschleunigung des Raycastings. Schneidet ein Strahl eine AABB nicht, so kann man sicher sein, dass er auch die im Quader enthaltenen Facetten nicht schneidet. Diese Eigenschaft spielt die wesentliche Rolle bei der Traversierung der BVH, die in Abschnitt 4.5 näher beleuchtet wird.

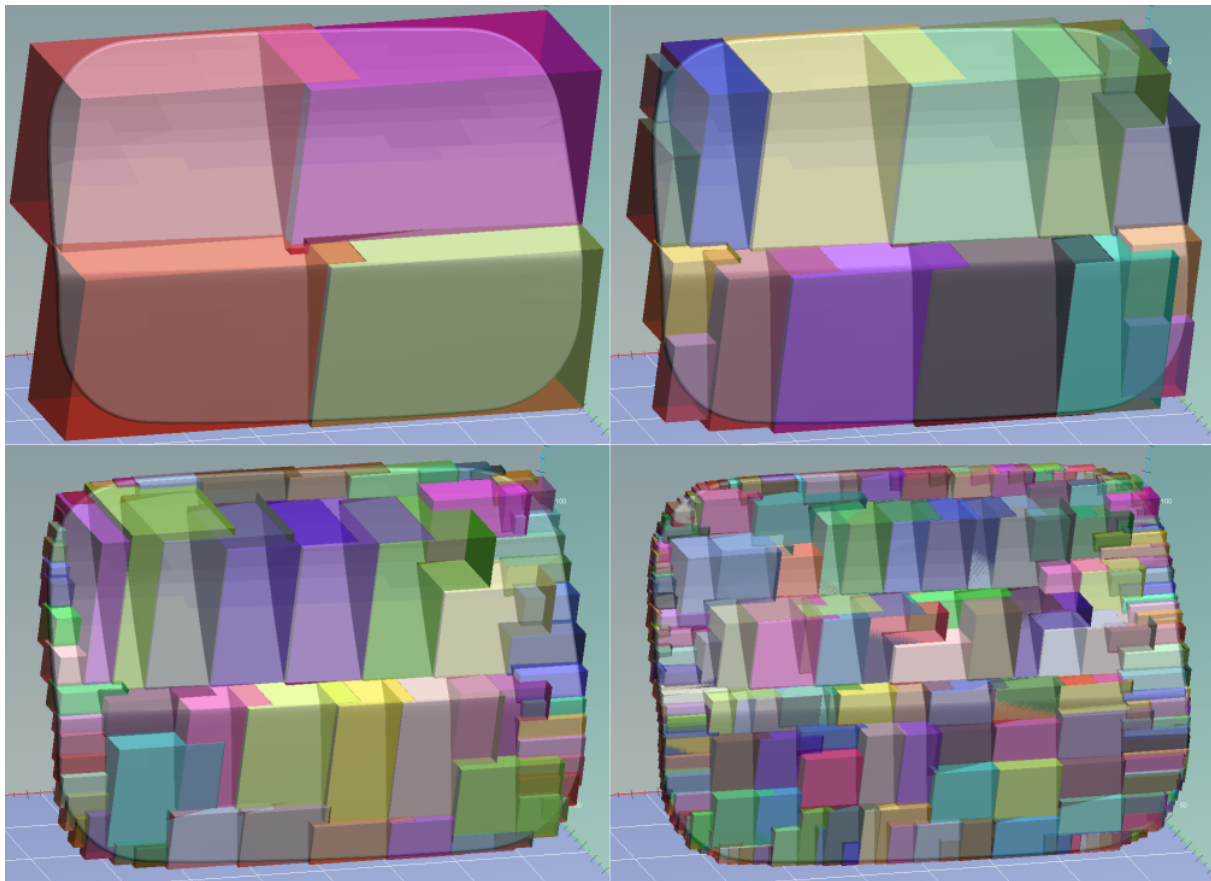


Abbildung 4.6: Im Bild links oben sind die vier AABBs der zweiten Hierarchiestufe einer BVH unterhalb der Wurzel eingezeichnet. Die Überlappung der einzelnen Volumen ist gut zu erkennen. Das Bild rechts daneben zeigt die vierte Hierarchiestufe, also $2^4 = 16$ Boxen. $2^6 = 64$ beziehungsweise $2^8 = 256$ AABBs zeigen die Bilder der unteren Reihe, die die sechste beziehungsweise achte Hierarchiestufe visualisieren.

Somit ergibt sich folgender rekursiver Algorithmus für den Aufbau einer BVH, die Facetten im $\mathbb{R}^3$ verwaltet und AABBs als begrenzendes Volumen verwendet:

1. Bestimme die kleinste umschließende AABB um die komplette Vertexmenge V . Weise die gefundene AABB der Wurzel der BVH zu.
2. Teile V in zwei Mengen V_l und V_r auf. Ermittle für sie jeweils ein umgebendes Volumen und verknüpfe die AABBs mit zwei Knoten, die die Söhne des Knotens der Menge V werden.
3. Wende Schritt 2 sowohl auf V_l als auch auf V_r an, solange bis sich in der Teilmenge nur noch eine Facette befindet.
4. Jedes Mal, wenn ein einzelnes Dreieck erreicht wurde, erzeuge ein Blatt in der BVH und setze eine Referenz auf das einzelne Primitiv.

Zur besseren Veranschaulichung zeigt Abbildung 4.6 die AABBs unterschiedlicher Hierarchiestufen der BVH für den bereits zuvor verwendeten Tankdeckel.

4.4.2 Aufbau durch die GPU

Der Aufbau von BVHs kann von Grafikkartenbeschleunigung profitieren, wie [WBS07] zeigt. Lauterbach et al. [LGS⁺09] verwenden hierzu die raumfüllenden Morton-Kurven [AGCA08]. Damit kann jeder Facette ein Morton-Code zugeordnet werden. Dabei handelt es sich um eine binäre Zahl. Um die Strukturierung der Dreiecke in der BVH zu erhalten, müssen die Morton-Codes sortiert werden. Dafür kann ein massiv paralleles Verfahren auf der Grafikkarte [SHG09] eingesetzt werden. Die sortierten Morton-Codes werden im Anschluss binärstellenweise, beginnend mit der höchstwertigen Stelle, eingelesen. Jede Binärstelle steht dabei für eine Hierarchiestufe in der BVH. Stimmen zwei Ziffern an der selben Binärstelle überein, so befinden sich die assoziierten Facetten in der selben AABB. Unterscheiden sie sich, so werden die Dreiecke von zwei unterschiedlichen umschließenden Volumen und damit Knoten im Baum verwaltet.

Die im Rahmen dieser Diplomarbeit programmierte Bibliothek baut BVHs nur unter Nutzung der CPU auf. Auch für sehr facettenreiche Dreiecksmengen ist eine derartige Implementierung performant genug. Da für den Rekonstruktionsalgorithmus die STL-Datei nur einmal geladen werden muss, ist die Ausführungszeit des BVH-Aufbaus im Vergleich zu den Raycastings während der Positionsbestimmung kurz. Bei anderen Anwendungen von BVHs, zum Beispiel bei dynamischen Szenen, in denen sich die Struktur der Facettenmenge mehrmals ändert, kann eine GPU-Beschleunigung aber durchaus sinnvoll sein und die Gesamtausführungszeit wesentlich verkürzen.

Auch der Aufbau von kd-Trees kann durch programmierbare Grafikkarten beschleunigt werden, wie [PGSS06], [HMS06] und [ZHWG08] zeigen.

4.4.3 Heuristiken für performante Hierarchien

In der Art und Weise, wie die Teilmengen von zu strukturierenden Objekten von einer Hierarchiestufe auf eine tieferliegende aufgeteilt werden, liegt der Schlüssel für effektives Arbeiten mit BVHs. Grundsätzlich gilt die Aussage, dass man bestrebt ist, die umschließenden Volumen der beiden Teilmengen so klein wie möglich zu wählen. Klein bezieht sich in diesem Fall auf die Oberfläche der begrenzenden Struktur, nicht auf deren Volumen. Bei AABBs bedeutet dies konkret, dass die sechs Flächen des Quaders und somit die Seitenkanten zu minimieren sind.

Sehr oft wird für die Entscheidung, welches Primitiv in welche Teilmenge getrennt werden soll, die Surface Area Heuristic (kurz: SAH) [GS87, MB90] verwendet, da sie sehr befriedigende Ergebnisse liefert. Der hohen Trenngüte steht eine nicht zu verachtende Komplexität und ein mittlerer bis hoher Aufwand gegenüber. Stattdessen wird an dieser Stelle ein Vorgehen gewählt, das ebenfalls ein zufriedenstellendes Verhalten zeigt.

Für alle Facetten $v_i \in V, i \in \mathbb{Z}_{[1,n]}$ wird deren Schwerpunkt $s_i = (s_{xi}, s_{yi}, s_{zi})$ berechnet. Mit Hilfe aller Schwerpunkte $S = \bigcup_{i \in \mathbb{Z}_{[1,n]}} s_i$ lässt sich der Schwerpunkt $s =$

$(s_x, s_y, s_z) = \frac{\sum_{s_i \in S} s_i}{|S|}$ der Facettenschwerpunkte bestimmen. Von diesem zentralen Schwerpunkt aus werden nun die komponentenweisen Betragsabstände $s_x^{dist} = \sum_{i=1}^n |s_x - s_{xi}|$, $s_y^{dist} = \sum_{i=1}^n |s_y - s_{yi}|$ und $s_z^{dist} = \sum_{i=1}^n |s_z - s_{zi}|$ zu den Dreiecksschwerpunkten berechnet. Da hier im $\mathbb{R}_3$ gearbeitet wird, gibt es drei dieser Betragsabstände. Verwendet man einen Raum anderer Dimension, so ist die Anzahl der Werte entsprechend anzupassen. s_x^{dist} , s_y^{dist} und s_z^{dist} sind Indikatoren, in welcher Raumdimension einzelne Facetten am

stärksten derart abweichen, so dass sie einen überproportionalen Anteil an der Länge einer Seitenkante der umgebenden AABB haben. Überproportional bedeutet, dass sich die meisten Facetten eigentlich in einer anderen Region des begrenzenden Volumens befinden. Da aber einzelne, abseits liegende Dreiecke trotzdem mit beachtet werden müssen, spannen sie den achsparallelen Quader unnötig weit auf. Möchte man den Indikatoreffekt zusätzlich verstärken, so kann man auch die quadratisch gewichteten Betragsabstände $s_x^{dist} = \sum_{i=1}^n (s_x - s_{xi})^2$, $s_y^{dist} = \sum_{i=1}^n (s_y - s_{yi})^2$ und $s_z^{dist} = \sum_{i=1}^n (s_z - s_{zi})^2$ einsetzen. Nun berechnet man $\max \{s_x^{dist}, s_y^{dist}, s_z^{dist}\}$. Je nachdem, welcher Indikator am größten ist, wird eine zu den Koordinatensystemebenen parallele Ebene eingezogen, an der die Facettenschwerpunkte s_i „links“ und „rechts“ sortiert werden. Ist beispielsweise $\max \{s_x^{dist}, s_y^{dist}, s_z^{dist}\} = s_x^{dist}$, so trennt die Ebene $\{(x, y, z) | x = \frac{x_{rechts} - x_{links}}{2}\}$, die parallel zur yz-Ebene ist, die Dreiecksschwerpunkte. Alle s_i mit $s_{xi} < \frac{x_{rechts} - x_{links}}{2}$ fallen in die eine Teilmenge, alle s_i mit $s_{xi} > \frac{x_{rechts} - x_{links}}{2}$ in die andere.

Die Trennung der Facetten findet also durch eine Raumebene statt. Dies muss aber nicht unbedingt der Fall sein. Grundsätzlich ist die Art und Weise der Ermittlung von zwei Teilmengen generisch. Zu beachten ist ebenfalls, dass im eben präsentierten Algorithmus die Wahl der Dimension, in der die Trennebene eingezogen wird, nicht unbedingt derjenigen Raumrichtung entsprechen muss, in der die AABB der Obermenge die größte Ausdehnung hat. Obiges Verfahren eignet sich in der Praxis besser um AABBs zu finden, deren Seitenkanten relativ kurz sind und dadurch die Oberfläche der Quader klein ausfällt.

Abbildung 4.7 zeigt schematisch die eben beschriebene Vorgehensweise, allerdings in der Ebene, nicht im Raum. Das Prinzip bleibt aber gleich.

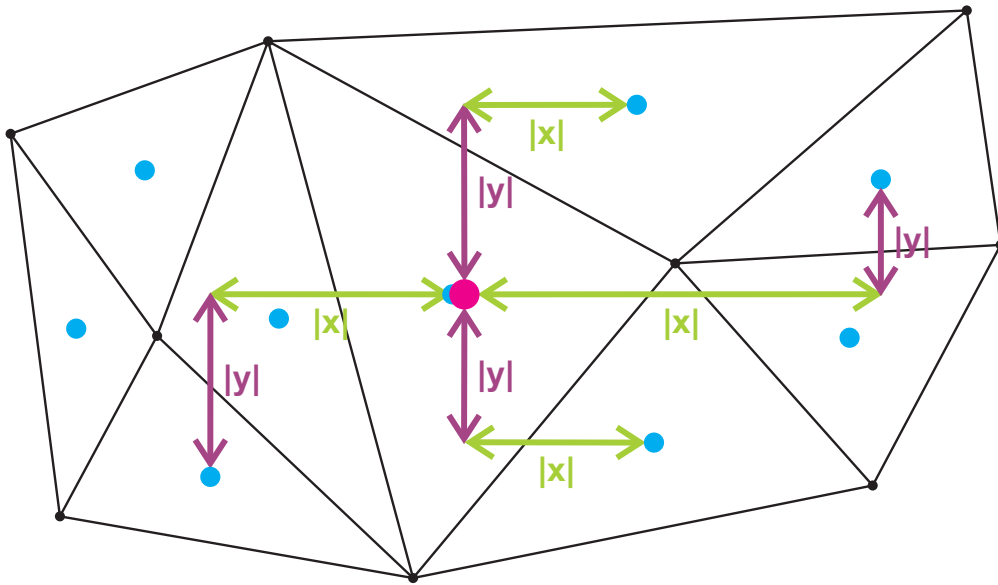


Abbildung 4.7: Mit blau sind die Zentren der Facetten eingezeichnet. Der rote Punkt ist der Mittelpunkt aller Dreieckszentren. Von dort aus werden die Abstände zu den Facettenschwerpunkten eingezeichnet. Grün sind die Distanzen in x-Richtung, violett in y-Richtung. Aus Gründen der Übersichtlichkeit sind nur vier der insgesamt neun Distanzen vermerkt.

4.4.4 Konvertierung einer Hierarchie für die Grafikkarte

Es ist nicht sinnvoll, die BVH mit ihren zahlreichen Pointern, die der Referenzierung der Baumknoten dienen, auf die Grafikkarte zu kopieren. Erstens müssten alle Pointer auf den Adressbereich des Device-Speichers geändert werden, zweitens müsste jeder Baumknoten einzeln übertragen werden, was zu vielen Speichertransfers führen würde. Aus diesem Grund ist eine Linearisierung der BVH wünschenswert, sodass sich alle Knoten des Baums in einem Array hintereinander im Speicher befinden. Die Offsets in diesem Array wären sowohl im Host- als auch im Device-Speicher gleich, nur die Startadresse im Grafikspeicher müsste verwaltet werden.

Dazu wird die BVH Preorder sortiert. Das bedeutet, dass die Hierarchie von der Wurzel her durchlaufen wird. Jedes Mal, wenn ein Knoten erreicht wird, wird dieser in die nächste freie Position des Arrays geschrieben. Anschließend findet ein rekursiver Aufruf des linken Sohnes des aktuellen Knotens statt. Ist dieser rekursive Aufruf abgearbeitet, findet ein rekursiver Aufruf des rechten Sohnes statt. Listing 4.3 gibt das Preorderverfahren in Pseudocode wieder.

```

1 void preOrder(node) {
2     writeToArray(node);
3
4     if(node.left != NULL) {
5         preOrder(node.left);
6     }
7
8     if(node.right != NULL){
9         preOrder(node.right);
10    }
11 }
```

Listing 4.3: Pseudocode Preorder.

Der Vorteil von Preorder gegenüber Inorder und Postorder liegt darin, dass der linke Sohn eines Knotens im Array immer in der nächsten Position nach seinem Vater liegt. Somit müssen nur die Verweise auf die rechten Söhne in einem separaten Array gespeichert werden.

In der Praxis werden folgende Arrays angelegt, die die Linearisierung der BVH darstellen:

- `char isLeaf[]`: Dieses Array speichert, ob es sich beim Eintrag mit einem konkreten Index um ein Blatt der BVH oder um einen inneren Knoten handelt.
- `float4 boundingBoxes[]`: In je zwei aufeinander folgenden Einträgen wird eine AABB der Hierarchie gespeichert. Dabei werden im ersten `float4` x_{links} , y_{oben} und z_{vorne} abgelegt, im zweiten `float4` x_{rechts} , y_{unten} und z_{hinten} .
- `int rightChildren_facetIndices[]`: Zwei unterschiedliche Informationen werden in diesem Array verwaltet. Handelt es sich am entsprechenden Index um einen inneren Knoten, wird der Index der Position des rechten Sohnknotens gespeichert. Liegt dagegen ein Blatt vor, so handelt es sich beim vorliegenden Wert um den Index der referenzierten Facette in der Dreiecksmenge. Ob es sich um einen inneren Knoten oder ein Blatt handelt, kann über `isLeaf` entschieden werden.

Die eher ungewöhnlichen Datentypen sind der Tatsache geschuldet, dass auf der Grafikkarte so oft wie möglich Coalescing genutzt werden soll. Dafür müssen die Daten einem gewissen Alignment entsprechen. Darum ist auch ein erhöhter Speicherverbrauch zulässig. Abbildung 4.8 zeigt die Umwandlung eines Baumes in ein Array mittels Preorder.

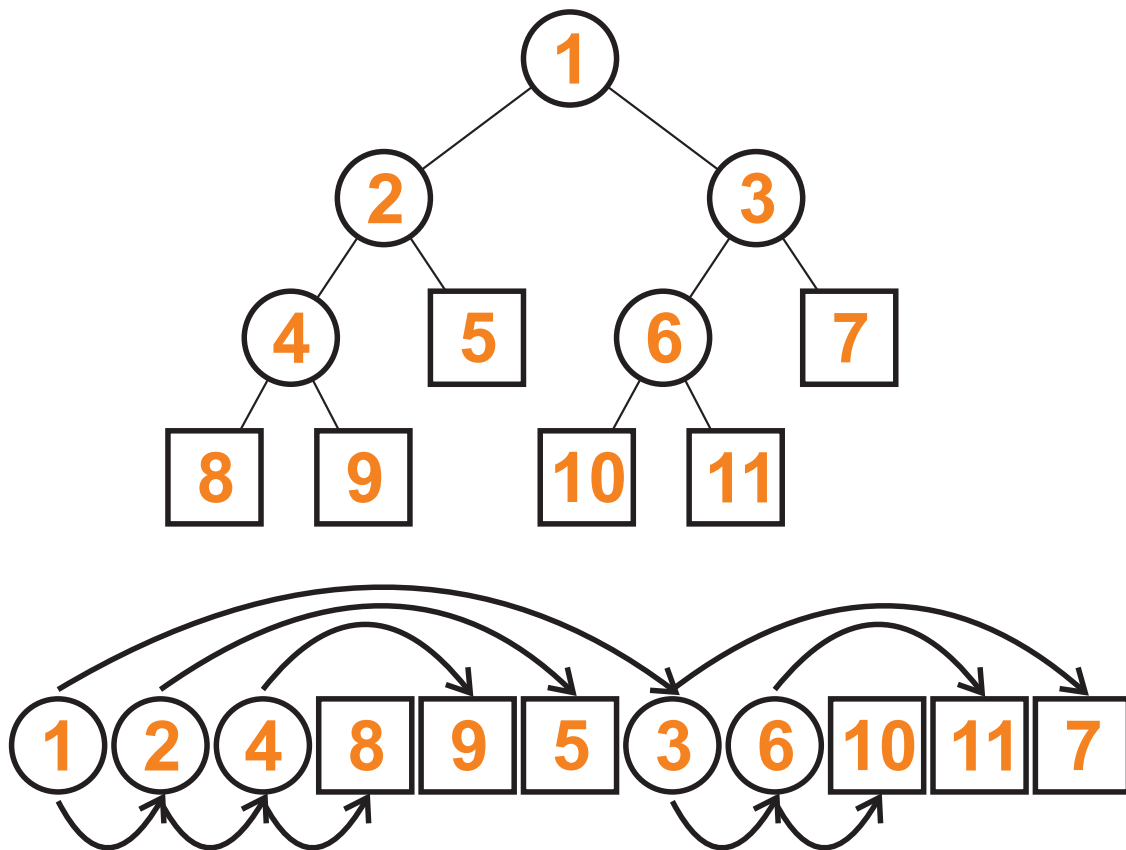


Abbildung 4.8: Diese Abbildung zeigt einen Binärbaum (oben), der mittels Preorder in ein Array (unten) linearisiert wird. Innere Knoten sind als Kreise, Blätter als Quadrate gekennzeichnet. Die Zeiger, die im Array oben vermerkt sind, sind Verweise auf den rechten Sohn. Dagegen zeigen die unten eingetragenen Pointer auf den linken Sohn. Sie referenzieren immer auf den direkt folgenden Knoten im Array.

4.5 Traversierung einer BVH

4.5.1 Notwendige Berechnungen

Der Aufbau einer BVH muss nur einmal zu Beginn des Algorithmus beim Einlesen der STL-Datei durchgeführt werden. Auch wenn von Iterationsschritt zu Iterationsschritt im Rahmen der NLO die Position und Lage des Modells verändert wird, so wird immer die selbe BVH eingesetzt. Anstatt den CAD-Entwurf zu transformieren, werden, wie in 2.4.4 beschrieben, Kamera und Bildschirm invers angepasst. An der Form der Facettenmenge selbst ändert sich nichts, weshalb die berechnete Hierarchie wiederverwendet werden kann. Der Großteil der Gesamtlaufzeit fällt auf die wiederholte Generierung simulierter Pixpos-Bilder und somit auf das Raycasting. Deshalb wird für die Erstellung der BVH eine CPU-Implementierung verwendet, das Durchlaufen der Hierarchie wird hingegen auf der Grafikkarte durchgeführt.

Traversiert man eine BVH, so fallen grundsätzlich drei unterschiedliche Operationen an. Während das bloße Folgen der Verzeigerung sehr schnell erfolgt, ist die Berechnung der Schnittpunkte von einem Strahl mit einer AABB beziehungsweise einem Dreieck schon aufwändiger. Vor allem in Hinblick auf eine GPU-Implementierung für das Durchlaufen der Hierarchie ist es von Interesse, dass der Registerverbrauch der Schnittpunktbestimmungen niedrig ist.

Ein Strahl schneidet eine AABB oder allgemeiner einen Quader entweder in zwei oder keinem Punkt. Eine Ausnahme ist möglich, wenn der Strahl das Objekt nur tangiert. In solchen Fällen kann es aufgrund numerischer Instabilitäten zu Problemen kommen, die gesondert behandelt werden müssen. Geimer et al. [GM03] stellen ein Verfahren zur Lösung des Strahl/AABB-Schnittproblems vor, das sowohl im CPU- als auch im GPU-Bereich die bevorzugt angewandte Berechnung ist.

Listing 4.4 zeigt den CUDA-Code der Schnittpunktermittlung. `start` und `direction` spezifizieren den Sichtstrahl. `xMin`, `yMin`, `zMin`, `xMax`, `yMax` und `zMax` beschreiben die AABB. Am Ende der Berechnung stehen in `tMin` und `tMax` Vielfache von `direction`, die den Ein- und Austrittspunkt des Strahls angeben.

```

1  __device__ inline bool intersectBox(
2      const float3 start, const float4 direction,
3      const float xMin, const float yMin, const float zMin,
4      const float xMax, const float yMax, const float zMax,
5      float *tMin, float *tMax) {
6      const float3 directionInv = make_float3(
7          __fdividef(1.0f, direction.x),
8          __fdividef(1.0f, direction.y),
9          __fdividef(1.0f, direction.z));
10     float l1 = (xMin - start.x) * directionInv.x;
11     float l2 = (xMax - start.x) * directionInv.x;
12     *tMin = fminf(l1, l2);
13     *tMax = fmaxf(l1, l2);
14
15     l1 = (yMin - start.y) * directionInv.y;
16     l2 = (yMax - start.y) * directionInv.y;
17     *tMin = fmaxf(fminf(l1, l2), *tMin);

```

```
18     *tMax = fminf(fmaxf(l1, l2), *tMax);
19
20     l1 = (zMin - start.z) * directionInv.z;
21     l2 = (zMax - start.z) * directionInv.z;
22     *tMin = fmaxf(fminf(l1, l2), *tMin);
23     *tMax = fminf(fmaxf(l1, l2), *tMax);
24
25     return ((*tMax >= *tMin) &
26            (*tMax >= -FLT_EPSILON));
27 }
```

Listing 4.4: Berechnung der beiden Schnittpunkte eines Sichtstrahls mit einem Quader.

Als besonders registerschonend bei der Schnittpunktberechnung eines Strahls mit einer Facette im $\mathbb{R}^3$ hat sich das Verfahren von Möller et al. [MT97] herausgestellt. Dies gilt sowohl für die CPU als auch für die GPU. Zwar gibt es Berechnungsmethoden, die mit weniger arithmetischen Befehlen auskommen, in Hinblick auf eine GPU-Implementierung ist ein niedriger Registerverbrauch jedoch vorzuziehen.

Listing 4.5 zeigt den CUDA-Code der Methode von Möller. Wieder beschreiben `start` und `direction` Aufpunkt und Richtung des Strahls. `vertex1`, `vertex2` und `vertex3` sind die Vertices der Facette, die auf einen gemeinsamen Punkt mit dem Strahl überprüft werden soll. `t` ist ein Vielfaches von `direction` und gibt den gefundenen Schnittpunkt an.

```
1  __device__ inline bool intersectFacet (
2      const float3 start, const float4 direction,
3      const float4 vertex1, const float4 vertex2, const float4 vertex3,
4      float *t) {
5      float3 leg1, leg2;
6      float3 a, b;
7      float d;
8      float u, v;
9
10     SUB_FPOINT3(&vertex2, &vertex1, &leg1);
11     SUB_FPOINT3(&vertex3, &vertex1, &leg2);
12     PRODUCT_FPOINT3_FAST(&direction, &leg2, &a);
13     d = __fdivdef(1.0f, SCP_FPOINT3(&leg1, &a));
14     SUB_FPOINT3(&start, &vertex1, &b);
15     u = d * SCP_FPOINT3(&b, &a);
16     PRODUCT_FPOINT3_FAST(&b, &leg1, &a);
17     v = d * SCP_FPOINT3(&direction, &a);
18     *t = d * SCP_FPOINT3(&leg2, &a);
19     PRODUCT_FPOINT3_FAST(&leg1, &leg2, &a);
20
21     return ((u > -FLT_EPSILON) &
22            (v > -FLT_EPSILON) & (u < 1.0f + FLT_EPSILON) &
23            (u + v < 1.0f + FLT_EPSILON) & (*t > 0.0f) &
24            (SCP_FPOINT3(&a, &direction) < -FLT_EPSILON));
25 }
```

Listing 4.5: Berechnung des Schnittpunkts eines Sichtstrahls mit einem Dreieck im Raum.

Oft wird im Zusammenhang mit Strahl/Facette-Schnittpunktberechnungen das Verfahren von Shevtsov et al. [SSK07] erwähnt. Es kommt mit wesentlich weniger Anweisungen aus und nutzt vor allem die Befehlssatzerweiterungen moderner CPUs wie Streaming SIMD Extensions (kurz: SSE). Im Kontext von CPUs ist diese Methode konkurrenzfähig. Der relativ hohe Registerverbrauch gegenüber dem Möller-Test disqualifiziert sie jedoch für den Grafikkarteneinsatz.

4.5.2 Traversierung auf der CPU

Wieder wird hier explizit auf die Nutzung von Facetten und AABBs eingegangen. Die Vorgehensweise lässt sich jedoch auch ohne Einschränkung auf beliebige Primitive und begrenzende Volumen erweitern.

Um zu überprüfen, welches Dreieck einer Objektmenge ein Strahl zuerst schneidet, wird die BVH von der Wurzel her durchlaufen. Dabei wird an jedem Knoten der Hierarchie gleich verfahren. Für die beiden Söhne N_l und N_r des aktuell betrachteten Knotens N wird überprüft, ob der Strahl gemeinsame Punkte mit den von N_l und N_r assoziierten umgebenden Volumen hat. Hierbei können drei Fälle auftreten:

- Der Strahl trifft weder das begrenzende Objekt von N_l noch von N_r . In diesem Fall muss nicht weiter in der BVH abgestiegen werden.
- Entweder der begrenzende Quader von N_l oder N_r wird getroffen. Dann ist je nach getroffenem Objekt in N_l beziehungsweise N_r abzustiegen.
- Sowohl die AABB von N_l als auch die von N_r werden vom Strahl durchlaufen. Ist dies der Fall, müssen beide Söhne von N traversiert werden. In der Praxis wird für gewöhnlich zuerst der eine Ast komplett abgearbeitet, bevor der andere durchlaufen wird. Behandeln kann man eine solche Verzweigung entweder rekursiv mit einem Funktionsaufruf oder iterativ mit einem expliziten Stack, in dem die noch zu traversierenden Äste gespeichert werden. Letzteres kennt man auch als Depth First Search (kurz: DFS).

Dieses Verhalten basiert auf der Tatsache, dass, falls ein Strahl die AABB nicht trifft, er auch keines der im Quader befindlichen Dreiecke treffen kann. So lassen sich schon früh große Teilmengen ausschließen, deren enthaltenen Facetten nicht einzeln auf Schnitt mit dem Strahl getestet werden müssen. Trifft der Strahl nicht einmal die AABB der Wurzel, so trifft er auch keine einzige Facette und die BVH muss nicht abgearbeitet werden.

Wird ein Blatt in der Hierarchie erreicht, so referenziert dieses eine einzelne Facette, die ein Kandidat für einen erfolgreichen Schnitt mit dem Strahl ist. Trifft der Strahl die Facette tatsächlich, so wird die Entfernung des Schnittpunkts vom Aufpunkt des Sichtstrahls d explizit gespeichert.

Aufgrund des dritten Falls in obiger Aufzählung ist es möglich, dass noch andere Äste der BVH Facetten verwalten, die gemeinsame Punkte mit dem Strahl haben können. Sie müssen in der Folge ebenfalls traversiert werden. Dabei kann man die Abarbeitung wesentlich beschleunigen. Liegt der Schnittpunkt des Strahls mit der AABB eines weiteren Astes weiter entfernt vom Aufpunkt des Sichtstrahls als das bisher gefundene d , so muss man den Ast nicht durchlaufen. Alle von der AABB des Astes umschlossenen Dreiecke sind dann garantiert weiter vom Aufpunkt entfernt, als der bisher ermittelte Schnittpunkt.

Aus diesem Grund speichert man immer den Abstand derjenigen Facette zum Startpunkt des Strahls, die die geringste Entfernung zu ihm hat. d muss angepasst werden, sobald eine Facette identifiziert wurde, die näher am Betrachter liegt als alle bisherigen.

Nun wird auch klar, warum man beim Aufbau einer BVH interessiert ist, Facettenmengen so zu teilen, dass die umgebenden AABBs der Teilmengen eine möglichst kleine Oberfläche besitzen. Solche Quader werden seltener von einem Strahl getroffen, was dazu führt, dass sich schneller Facettenteilmengen für das Raycasting verwerfen lassen.

Ein einfaches Beispiel für die Traversierung einer BVH wurde bereits mit der Abbildung 4.5 gegeben.

4.5.3 Traversierung auf der GPU

Neben der Positionsbestimmung liegt die algorithmische Herausforderung dieser Arbeit im Finden eines Verfahrens zum effektiven Abarbeiten einer BVH auf der Grafikkarte. Dabei gilt es den Widerspruch zu lösen, dass klassisches Traversieren eines Baumes viele bedingte Anweisungen enthält, auf Rekursion aufbaut, Speicherzugriffe verstreut sind und einzelne Strahlen die BVH beim Raycasting auf unterschiedlichen Wegen durchlaufen können.

Das Fehlen eines impliziten Aufrufstacks bei Kernels führt zu zwei unterschiedlichen Ansätzen. [TS05] stellt ein Verfahren vor, das komplett auf einen Stack verzichtet. Alternativ kann ein geteilter Stack verwendet werden, wie es [GPSS07] vorschlägt, der explizit vom Programmierer verwaltet werden muss. Letztere Variante wird hier verwendet und genauer vorgestellt.

Grundsätzlich arbeitet dieser Algorithmus genau so wie serielles Traversieren auf der CPU. Anstatt eines einzelnen Strahls wird jedoch ein ganzes rechteckiges Paket von Strahlen parallel durch die Hierarchie bewegt. Jedem Thread wird ein Sichtstrahl der Kamera zugeordnet, für den das Raycasting durchgeführt werden soll. Diese werden in Paketen zu Thread Blocks zusammengefasst. 8×8 große Bündel haben sich am sinnvollsten dafür herausgestellt.

Ein solches Paket wird nun synchron durch die BVH bewegt. Dabei operiert zu einem Zeitpunkt das komplette Bündel auf einem Knoten des Baumes. Handelt es sich bei dem gerade besuchten Knoten um ein Blatt, so werden alle Strahlen des Pakets mit der referenzierten Facette geschnitten. Haben der Sichtstrahl eines Threads und die erreichte Facette tatsächlich einen gemeinsamen Punkt, so wird pro Thread, wie gehabt, in einer lokalen Variable der Abstand zur Facette und deren Index gespeichert.

Befindet sich dagegen das Paket gegenwärtig an einem inneren Knoten der Hierarchie, so wird das komplette Strahlenbündel parallel mit den AABBs des linken und rechten Sohnes auf Schnitt überprüft. N ist wieder der aktuelle Knoten, N_l und N_r wieder linker und rechter Sohn. Ähnlich der CPU-Variante können nun unterschiedliche Fälle auftreten:

- Keiner der Strahlen schneidet die mit N_l und N_r assoziierten Boxen. In diesem Fall wird die Traversierung des kompletten Pakets im aktuellen Ast beendet.
- Alle Strahlen, die einen der beiden Sohnquader schneiden, schneiden entweder die linke oder die rechte AABB. Dann wandert das komplette Paket in den entsprechenden Zweig.

- Ein paar Strahlen des Pakets treffen die AABBs der Söhne. Einige Strahlen schneiden die von N_l verwaltete AABB, andere treffen die von N_r verwaltete, wiederum andere durchlaufen beide begrenzende Volumen. In diesem Fall muss entschieden werden, in welchen Ast abgestiegen wird.

Für die letzte der drei Möglichkeiten wird im SMem ein Array angelegt, das für jeden Thread des Blocks einen Eintrag enthält. Jeder einzelne Thread schreibt nun in dieses Array an die für ihn vorgesehene Position einen Wert, der sich danach richtet, ob der einzelne Thread in den linken, rechten oder beide Zweige absteigen will. Hat jeder Thread parallel seinen Wert in das gemeinsame Array eingetragen, so findet eine Auswertung statt. Sie ermittelt, welcher Ast der bevorzugte ist, also von mehr Threads mit ihren Strahlen besucht werden will. Mit ihm wird dann fortgefahren. Der andere Zweig, in den mindestens ein Thread des Pakets absteigen will, wird auf einem gemeinsamen Stack im SMem gespeichert.

Um das gemeinsame Entscheidungsarray auszuwerten, kommt die parallele Präfixsummenbildung aus [HSO07] zum Einsatz. Will ein Thread in N_l absteigen, so schreibt es den Wert -1 in seine Position des gemeinsamen Entscheidungsarrays, auf das das komplette Paket Zugriff hat. Für den Wunsch, mit Knoten N_r weiterzuarbeiten, wird 1 geschrieben. Müssen beide Söhne von N abgearbeitet werden, so wird eine 0 eingetragen. Je nach Implementierung des Präfixsummenverfahrens steht nach dessen Ausführung entweder im ersten oder im letzten Element des Entscheidungsarrays die Summe aller von den Threads eingetragenen Werte. Ist das Ergebnis negativ, so bedeutet das, dass mehr Threads in N_l absteigen wollen. Ist es positiv, so haben mehr Threads für N_r „gestimmt“. Falls das Ergebnis 0 ist, spielt es keine Rolle, mit welchem der Söhne zuerst weitergearbeitet wird.

Jedes Mal, wenn in einen Ast abgestiegen wird, obwohl ein Teil der Threads des Pakets (auch) einen anderen Zweig traversieren muss, wird dies in einem gemeinsamen Stack vermerkt. Alle Threads des Blocks lesen diesen Stack im SMem parallel an der selben Stelle, falls ein Element aus ihm entfernt wird, und schreiben in diesen Stack parallel an die selbe Stelle, falls ein Element in den Stack eingefügt werden soll. Lese- und Schreiboperationen finden in den folgenden Situationen statt:

- `push()`: Letzter Fall der obigen Auflistung, in dem das Paket in beide Söhne absteigen muss. Diejenige Alternative, die noch nicht verarbeitet wird, wird auf dem Stack abgespeichert.
- `pop()`: Hat das Paket ein Blatt im Baum erreicht und dieses abgearbeitet, so kann im nächsten Schritt an einer Stelle in der BVH weitergearbeitet werden, die zuvor aufgeschoben werden musste.
- `pop()`: Tritt die Situation ein, dass kein Strahl des gesamten Pakets tiefer in der BVH absteigen möchte, so kann im nächsten Schritt an einer anderen Stelle der Hierarchie weitergearbeitet werden. Hierbei handelt es sich um den ersten Fall in obiger Auflistung.

Es handelt sich also um einen gemeinsamen Stack.

Die parallele Schnittoperation eines ganzen Bündels von Sichtstrahlen mit einer AABB beziehungsweise einem Dreieck führt dazu, dass bei den meisten Threads kein Schnittpunkt zustande kommt. Dies könnte meistens auch schon vorzeitig ermittelt werden. Dennoch

ist der Grad an Nebenläufigkeit immer noch deutlich höher, als bei einer seriellen Verarbeitung.

Listing 4.6 zeigt in Pseudocode des eben vorgestellten Algorithmus. Es demonstriert, welche Variablen lokal pro Thread gehalten werden und welche den gemeinsam nutzbaren SMem verwenden. Erstere werden bestenfalls in Registern gehalten.

```
1 ray = (start, direction); // Ein Strahl (pro Thread)
2 d = INFINITY;             // Abstand zum nächsten
3                             // Schnittpunkt (pro Thread)
4 *node_p;                  // Pointer auf die Wurzel (pro Thread)
5 id;                       // ID dieses Threads (pro Thread)
6
7 __shared__ node_l[];       // Array, das die linken Söhne aller
8                             // Threads verwaltet (pro Block)
9 __shared__ node_r[];       // Array, das die rechten Söhne aller
10                             // Threads verwaltet (pro Block)
11 __shared__ decision[];     // Array zur Bestimmung der
12                             // Abstiegsrichtung (pro Block)
13 __shared__ stack[];        // Stack für noch abzuarbeitende
14                             // Äste (pro Block)
15
16 loop {
17     if(isLeaf(node_p)) {
18         // Schneide ray mit von node_p verwalteter Facette
19         lambda = intersect(ray, node_p);
20         // Aktualisiere d falls näherer Schnittpunkt gefunden
21         d = min(d, lambda);
22
23         if(stack.isEmpty()) {
24             break;
25         } else {
26             node_p = stack.pop();
27         }
28     } else {
29         node_l[id] = node_p.left(); // Paralleles Laden
30         node_r[id] = node_p.right(); // Paralleles Laden
31
32         // Schneide ray mit von node_l[id] verwalteten AABB
33         lambda = intersect(ray, node_l[id]);
34         // Schneide ray mit von node_r[id] verwalteten AABB
35         mue     = intersect(ray, node_r[id]);
36
37         b_1 = lambda < d & lambda >= 0;
38         b_2 = mue    < d & mue    >= 0;
39
40         if(id < 4) {
41             decision[id] = false;
42         }
43         decision[2 * b_1 + b_2] = true;
44     }
```

```

45     if(decision[3] | (decision[1] & decision[2])) {
46         // Beide Söhne müssen bearbeitet werden
47         // Jeder Thread gibt seine "Stimme" ab
48         decision[id] =
49             (b_1 | b_2) * (2 * (b_2 & (mu < lambda)) - 1);
50
51         // "Zählung" der Stimmen
52         parallelScan(decision);
53
54         // "Auswertung" der Abstimmung
55         if(decision < 0) {
56             // Falls mehr Threads links absteigen wollen
57             node_p = node_l[id];
58             stack.push(node_r[id]);
59         } else {
60             // Falls mehr Threads rechts absteigen wollen
61             node_p = node_r[id];
62             stack.push(node_l[id]);
63         }
64     } else if(decision[2]) {
65         // Nur der linke Sohn muss bearbeitet werden
66         node_p = node_l[id];
67     } else if(decision[1]) {
68         // Nur der rechte Sohn muss bearbeitet werden
69         node_p = node_r[id];
70     } else {
71         // Kein Sohn muss bearbeitet werden
72         if(stack.isEmpty()) {
73             break;
74         } else {
75             node_p = stack.pop();
76         }
77     }
78 }
79

```

Listing 4.6: Pseudocode für das parallele, paketbasierte Durchlaufen einer BVH.

Für kd-Trees existieren ebenfalls Ansätze zur parallelen Traversierung. Foley et al. [FS05] und Popov et al. [PGSS07] stellen jeweils ein Verfahren ohne Stack vor. Horn et al. [HSHH07] präsentieren einen Algorithmus mit einem kleinen Stack fester Größe.

4.6 Glättung der Facetten

4.6.1 Notwendigkeit von Vertexnormalen

Bereits in 4.1.2 wurde angesprochen, dass Dreiecksmengen gekrümmte Oberflächen nur annäherungsweise darstellen, jedoch nicht exakt nachbilden können. Eine weitere Einschränkung ist dem STL-Modell geschuldet, das nur eine Normale pro Facette und nicht pro Dreieckseckpunkt speichert.

Die Normalen werden benötigt, um Reflexionen von Sichtstrahlen an der Dreiecksmenge zu simulieren. Auch wenn pro Fläche nur eine Normale zur Verfügung steht, ist es wünschenswert, an jeder Stelle einer Facette so auf eine Senkrechte zurückgreifen zu können, als handle es sich um eine gekrümmte Oberfläche, die an jeder Position eine veränderte Normale besitzt.

Im Folgenden wird von glatten Normalen die Rede sein. Dabei handelt es sich um Normalenvektoren, die sich so verhalten, wie es im letzten Absatz gewünscht wird.

Unter Verwendung der Facettennormalen kann heuristisch in zwei Schritten für jeden

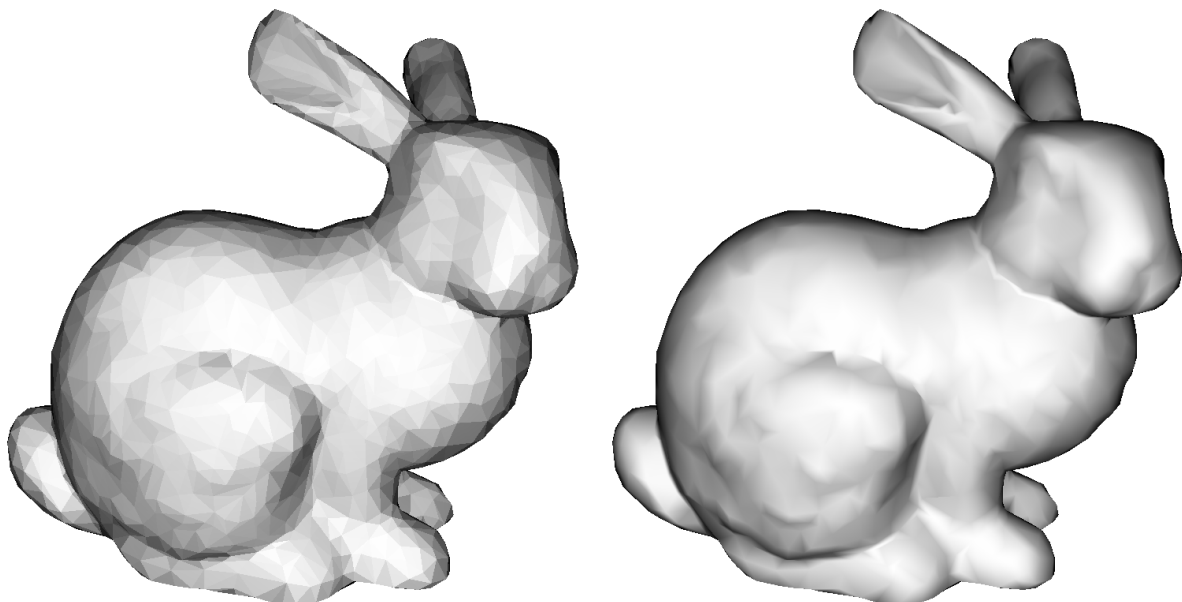


Abbildung 4.9: Das linke Bild zeigt den bereits zuvor verwendeten Hasen. Die Facetten, aus denen er aufgebaut ist, lassen sich leicht erkennen, da eine Facette über ihre ganze Fläche die selbe Farbe besitzt. Im Gegensatz dazu werden für das rechte Bild geglättete Normalen überall dort verwendet, wo ein Kerasichtstrahl auf die Modelloberfläche trifft. Dies macht sich in einer glatteren Schattierung bemerkbar.

Punkt der Dreiecksoberfläche eine glatte Normale approximiert werden. Im ersten Schritt bestimmt man aus den Normalen der Facetten, die an einen Vertex angrenzen, die Vertexnormale des Eckpunktes. Anschließend lässt sich unter Verwendung baryzentrischer Koordinaten [Wik10a] aus den drei Vertexnormalen eines Dreiecks für jeden Punkt der Facette eine sich kontinuierlich verändernde Normale interpolieren.

Einen Vergleich zwischen ungeglätteter und geglätteter Oberfläche liefert Abbildung 4.9.

Da die Schattierung von Oberflächen auch von den Normalenvektoren abhängt, lässt sich der Unterschied zwischen reinen Flächennormalen und approximierten glatten Normalen leicht an der Darstellung der Dreiecksmenge demonstrieren. Hier wird bewusst ein facettenarmes Beispiel gewählt, da so die Diskretisierung gekrümmter Oberflächen durch planare Primitive stärker auffällt.

Die Berechnung der Vertexnormalen ist eine große Herausforderung. In den folgenden beiden Abschnitten werden Heuristiken vorgestellt, die Werte berechnen, mit denen die Facettennormalen gewichtet werden können, um aus ihnen eine Vertexnormale zu erhalten. Einen Überblick über derartige Heuristiken kann man sich auch mit [Sch04a] verschaffen. Solche Gewichte sind notwendig, da ein Dreiecksnetz sehr heterogen sein kann, wie Abbildung 4.2 illustriert.

Problematisch am Gebrauch derartiger Heuristiken ist, dass sie aus dem Bereich der Computergrafik stammen, wo „schönes Aussehen“ wichtiger ist als Realismus. Ebenfalls muss beachtet werden, dass zwar die Normalen approximiert werden, die Reflexion aber immer noch an der Facettenoberfläche stattfindet. Eine Rekonstruktion der ursprünglichen, gekrümmten Oberfläche findet nicht statt.

4.6.2 Heuristiken unter Berücksichtigung der gesamten Facette

Wieder sei $V = \{v_i\}_{i \in \mathbb{Z}_{[1,n]}} = \{(x_i, y_i, z_i)\}_{i \in \mathbb{Z}_{[1,n]}}$ die Menge aller Vertices. Darüber hinaus sei $F = \{f_i\}_{i \in \mathbb{Z}_{[1,m]}}$, $F \subseteq (V \times V \times V)$ die Menge aller Facetten, wobei m die Größe dieser Menge ist. Dabei gilt natürlich, dass ein Vertex nur einmal in einer Facette vorkommen darf. Die Vertices einer Facette f_i werden als $f_i = (v_{xi}, v_{yi}, v_{zi})$, $v_{xi} \in V, v_{yi} \in V \setminus \{v_{xi}\}, v_{zi} \in V \setminus \{v_{xi}, v_{yi}\}$ bezeichnet.

Um eine Vertexnormale zu bestimmen, werden die Flächennormalen der an den Vertex angrenzenden Facetten linearkombiniert. Die Koeffizienten der Linearkombination ergeben sich aus den Eigenschaften der Dreiecke, die den Vertex gemeinsam haben. Alle folgenden Berechnungsmethoden für Gewichte basieren nur auf den Eigenschaften der Facette f_i , unabhängig vom Vertex v_{xi} , v_{yi} oder v_{zi} , für den die Vertexnormale berechnet werden soll. Man kann ein Gewicht umso größer wählen, je größer der Flächeninhalt des Dreiecks ist, für das es ermittelt werden soll. Hinter diesem Ansatz steckt die Idee: Je größer eine Facette ist, desto größer ist der Anteil an der Modelloberfläche, den das Dreieck einnimmt. In einem solchen Fall handelt es sich um eine „vorherrschende“ Region der Oberfläche, deren überproportional große Fläche sich auch überproportional in der Gewichtung widerspiegeln soll. Umgekehrt könnte man das Gewicht umso größer wählen, je kleiner eine Facette ist. Dies ist dann sinnvoll, wenn kleine Facetten wichtige Nuancen der Oberfläche modellieren.

Ein anderer Ansatz bei der Gewichtung von Facetten ist die Einbeziehung der Kanten $[v_{xi}, v_{yi}]$, $[v_{yi}, v_{zi}]$ und $[v_{zi}, v_{xi}]$ von f_i , um die Relevanz eines Dreiecks zu beurteilen. Analog der Verwendung des Flächeninhalts einer Facette, kann diese umso wichtiger betrachtet werden, je größer deren längste Kante $\max\{\|v_{xi} - v_{yi}\|, \|v_{yi} - v_{zi}\|, \|v_{zi} - v_{xi}\|\}$ ist. Je länger eine Dreieckskante ist, desto mehr Regionen des Modells werden vom Dreieck involviert. Umgekehrt kann es aber wieder von Interesse sein, möglichst kurze längste Seiten höher zu gewichten, damit diese nicht unterrepräsentiert sind.

Die Abbildungen 4.10 und 4.11 zeigen die schattierten Bilder eines Hasen, die geglättete Normalen einsetzen. Die dafür notwendigen Vertexnormalen werden aus den Flächennor-

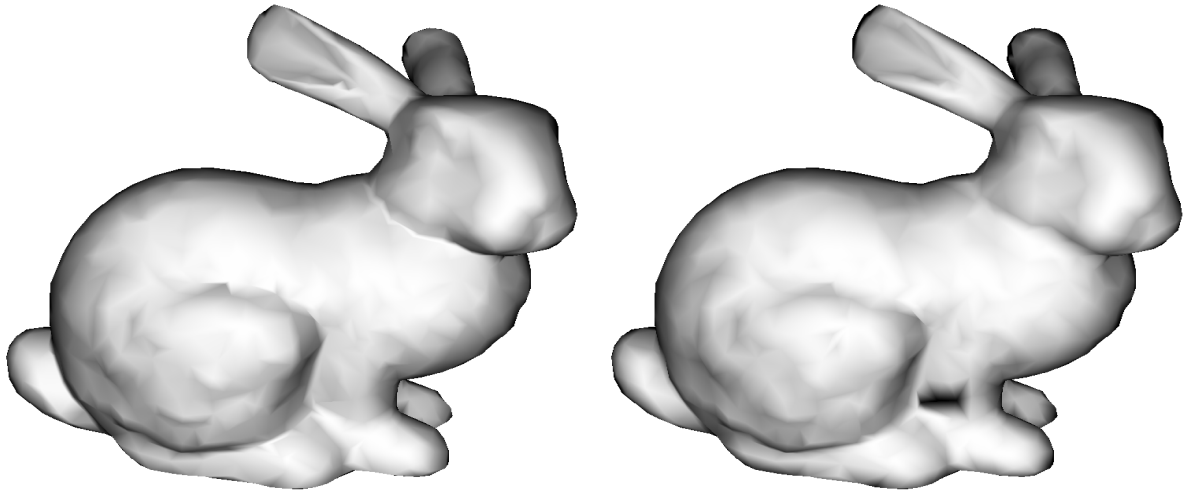


Abbildung 4.10: Die Abbildung zeigt unterschiedliche Schattierungen des gleichen STL-Modells. Das abweichende Erscheinungsbild ist den verschiedenen Heuristiken für die Gewichtung von Flächennormalen geschuldet.

links: je größer die Vertexfläche, desto größer das Gewicht

rechts: je größer die Vertexfläche, desto kleiner das Gewicht

malen berechnet, wobei die dafür nötigen Gewichte aus je einer der vier eben präsentierten Heuristiken stammen.

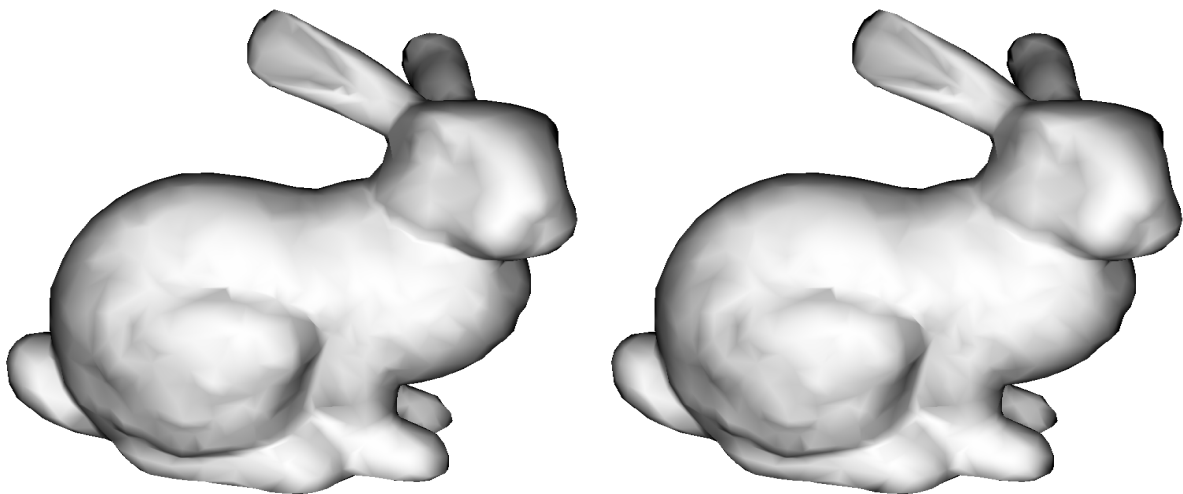


Abbildung 4.11: *links*: je länger die längste Vertexkante, desto größer das Gewicht

rechts: je länger die längste Vertexkante, desto kleiner das Gewicht

4.6.3 Heuristiken unter Berücksichtigung des Vertex

Im Gegensatz zu den Heuristiken in 4.6.2 kann auch die Position des Vertex im Dreieck, für den die Normale berechnet werden soll, eine Rolle spielen.

So kann man das Gewicht für eine Flächennormale umso größer wählen, je weiter der Winkel zwischen den Strecken $[v_{ri}, v_{si}]$ und $[v_{ri}, v_{ti}]$ ist. $r, s, t \in \{x, y, z\}, r \neq s \neq t$. Große

Winkel bei einem Eckpunkt bedeuten, dass die Region rund um den Eckpunkt von der entsprechenden Facette dominiert wird. Dementsprechend hoch wird sie gewichtet. Näheres dazu findet sich in [TW98].

Alternativ kann die Distanz des längeren der beiden Schenkel $[v_{ri}, v_{si}]$ und $[v_{ri}, v_{ti}]$ als Gewicht herangezogen werden. Der Wichtungsfaktor ist dann umso höher, je größer $\max\{\|v_{ri} - v_{si}\|, \|v_{ri} - v_{ti}\|\}$ ist. Die Begründung für einen solchen Ansatz ist die selbe wie im Szenario von 4.6.2, in dem das Gewicht umso höher ist, je weiter die längste Seite im Dreieck ist. Der Unterschied liegt nun darin, dass nur zwei Dreieckskanten betrachtet werden, nämlich jene, die am betrachteten Eckpunkt ansetzen.

Ein in der Computergrafik häufig eingesetztes Verfahren berechnet die Gewichte nach

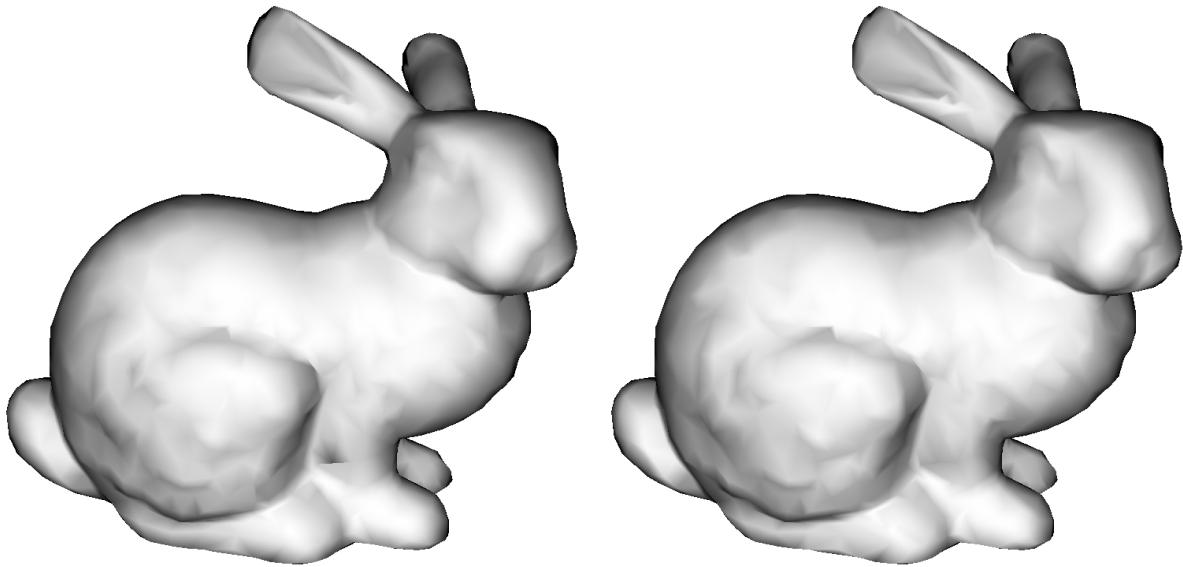


Abbildung 4.12: Die Abbildung zeigt unterschiedliche Schattierungen des gleichen STL-Modells. Das abweichende Erscheinungsbild ist den verschiedenen Heuristiken für die Gewichtung von Flächennormalen geschuldet.

links: je größer der Winkel an einem Vertex, desto größer das Gewicht

rechts: je größer der längere Schenkel an einem Vertex, desto größer das Gewicht

dem Produkt der inversen Seitenlängen am Vertex v_{ri} . Mathematisch lässt sich das mit $\frac{1}{\|v_{ri} - v_{si}\|} \cdot \frac{1}{\|v_{ri} - v_{ti}\|}$ ausdrücken. Bei der Visualisierung von zusammenhängenden Dreiecksmengen führt dies zumeist zu sehr anschaulichen und natürlich wirkenden Ergebnissen. Die Eignung für eine realitätsnahe Approximation von Normalenvektoren muss von Fall zu Fall entschieden werden.

Ebenfalls aus dem Bereich der Computergrafik stammt der Ansatz, ein Gewicht umso größer zu wählen, je größer die inverse Summe der Schenkellängen $\frac{1}{\|v_{ri} - v_{si}\| + \|v_{ri} - v_{ti}\|}$ ist. Es hat mit der vorigen Heuristik gemeinsam, dass sich diese Berechnungsmethode nicht durch reale Gegebenheiten begründen lässt. Lediglich der ästhetische Aspekt von Schattierungen rechtfertigt dieses Verfahren.

Abermals dient in den Abbildungen 4.12 und 4.13 das schattierte Bild eines Hasen zur Demonstration von geglätteten Normalen. Die Vertexnormalen, die dafür notwendig sind, werden durch Gewichte bestimmt, die aus den vier eben präsentierten Heuristiken stam-

men.

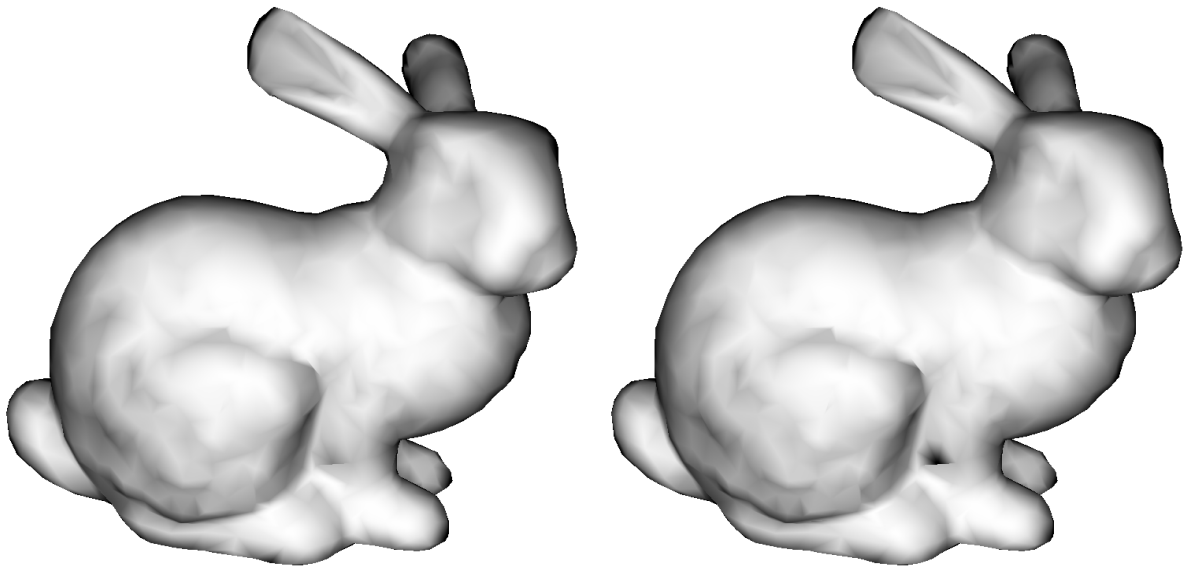


Abbildung 4.13: *links*: je größer das Produkt der inversen Schenkellängen an einem Vertex, desto größer das Gewicht
rechts: je größer die inverse Summe der Schenkellängen, desto größer das Gewicht

4.6.4 Reapproximation mit der Taylorreihe

Alle bisher vorgestellten Verfahren zur Vertexnormalenberechnung basieren auf einer mehr oder weniger realitätsnahen Gewichtung der Facettennormalen der an den Vertex angrenzenden Dreiecke. Diese Gewichte können bisher auf relativ einfache Weise berechnet werden, denn sie benutzen ausschließlich Informationen wie Seitenlängen oder Flächeninhalte, die implizit vorhanden sind oder sich schnell und einfach berechnen lassen.

Max [Max99] verfolgt einen weiterführenden Ansatz. Auch er gewichtet die angrenzenden Facettennormalen. Dazu kommt ein Taylorreihenpolynom [Wik10d] zum Einsatz, mit dem die Modelloberfläche lokal um den Vertex approximiert wird.

Damit lassen sich vor allem in Fällen, in denen die angrenzenden Dreiecke stark in Größe und Form variieren, gute Ergebnisse erzielen. Diese Aussage bezieht sich allerdings abermals auf das optische Erscheinungsbild bei der Verwendung von glatten Normalen zur Schattierung von Dreiecksmengen.

Für eine ausführlichere Betrachtung dieser Idee wird auf die eingangs erwähnte Literatur verwiesen.

4.6.5 Probleme bei der Bestimmung von Vertexnormalen

Nicht nur der Umstand, dass die hier vorgestellten Verfahren zur Berechnung von Vertexnormalen aus der Computergrafik stammen, ist problematisch. Selbst falls sich perfekte glatte Normalen finden lassen würden, gilt es zwei Einschränkungen zu beachten.

Die Normalen dienen nur dazu, die Richtung der Reflexion korrekt zu bestimmen. Den

exakten Ort, wo diese Strahlspiegelung stattfindet, können sie nicht liefern. Schließlich wird für den reflektierten Sichtstrahl nicht nur eine Richtung, sondern auch ein Aufpunkt benötigt. Eigentlich sollte dieser Aufpunkt durch das Raycasting bestimmt werden. Als Grundlage dafür dient jedoch die diskretisierte Dreiecksmenge aus planaren Objekten. War die ursprüngliche Form gekrümmt, so besitzen nur noch die Vertices die selbe Position wie im gekrümmten Original. Die Facettenflächen verbinden nur die Vertices, entsprechen aber nicht der ursprünglichen Oberfläche.

Die Konsequenz daraus veranschaulicht Abbildung 4.14. Ein Sichtstrahl trifft die Drei-

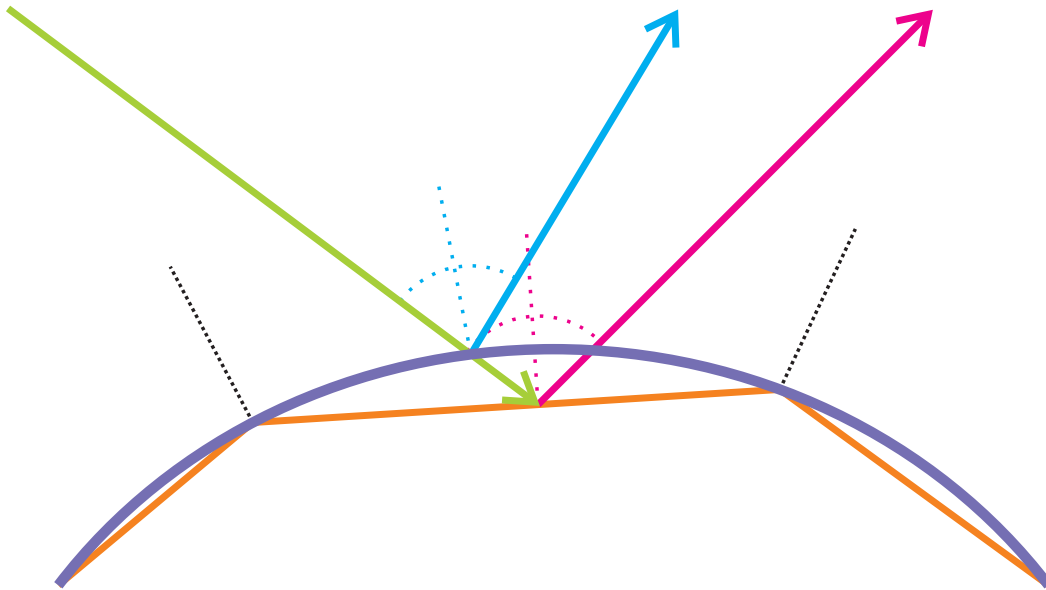


Abbildung 4.14: Violett ist die gekrümmte Oberfläche dargestellt, die durch die orangefarbenen Flächen nachgebildet wird. Der grüne Strahl soll reflektiert werden. Würde dies auf der gekrümmten Oberfläche geschehen, so würde sich die blaue Reflexion ergeben. Die rote Spiegelung an der Diskretisierung weicht davon deutlich ab.

ecksmenge an einer anderen Position, als er das bei der gekrümmten Originaloberfläche getan hätte. Dadurch ist es möglich, dass, obwohl die approximierte Normale auf der Facette realistisch ist, sie in eine andere Richtung zeigt als die Normale am Schnittpunkt mit der runden Fläche. Somit unterscheidet sich nicht nur der Ort, an dem die Reflexion stattfindet, sondern auch deren Richtung.

Nicht jede Oberfläche muss geglättet werden. Mitunter sind Stellen im Modell gewünscht, die eine harte Kante besitzen. Ein Würfel beispielsweise, mit seinen sechs Seiten, würde durch gemeinsame Vertexnormalen und baryzentrische Koordinaten glatte Normalen besitzen, die sich ähnlich der auf einer Kugel verhalten. Stattdessen müssen in einer solchen Konstellation explizit die unveränderten Flächennormalen verwendet werden.

Das STL-Format bietet keine Möglichkeit zu speichern, in welchen Regionen glatte Normalen angebracht sind, oder ob alternativ Flächennormalen Verwendung finden sollen. Diese Frage nur anhand der Dreiecksmenge zu klären, ist kritisch. Ein Schwellwertwinkel, ab dem eine Kante als hart gilt und nicht geglättet wird, ist zu wenig flexibel, denn so würden konkave Flächen niemals eine Glättung erfahren.

Die Konsequenzen, die auftreten, wenn harte Kanten ignoriert werden und stattdessen jeder Bereich eines Modells auf die selbe Art und Weise geglättet wird, zeigt Abbildung 4.15.

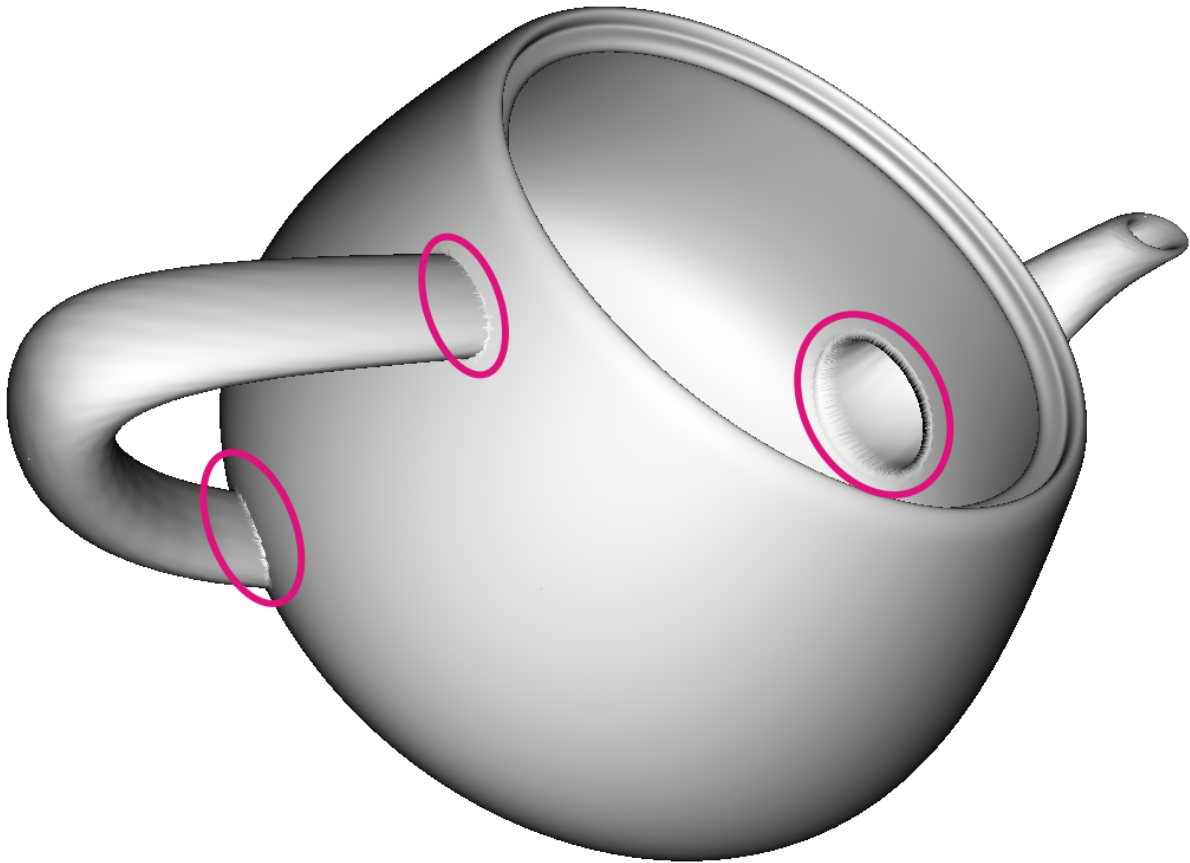


Abbildung 4.15: Harte Kanten sind manchmal explizit erwünscht. An den rot markierten Stellen treten Schattierungsfehler auf, die sich aus der fehlerhaften Berechnung der glatten Normalen ergeben. Obwohl ein eckiger Übergang zwischen Kannenbauch und Henkel beziehungsweise Ausgießer gewünscht ist, findet fälschlicherweise eine Glättung statt.

Kapitel 5

Ergebnisse und Leistungsmessungen

In diesem Kapitel soll eine Auswertung der in dieser Arbeit verwendeten Technologien beziehungsweise Techniken stattfinden. Dabei liegt ein besonderer Schwerpunkt auf dem Mehrwert, der durch die Grafikkartenimplementierung erzielt wird. Neben dem Nutzen der Grafikkartenprogrammierung für das vorliegende Szenario wird die erfolgreiche Rekonstruktionsrate des Algorithmus für die Positionsbestimmung bewertet und ein Vergleich zu der Variante ohne die in Abschnitt 2.3 erläuterten Modifikation gezogen.

Vorausgreifend kann man sagen, dass sich der GPU-Einsatz für die Positionsrekonstruktion lohnt. Er beschleunigt die Algorithmik wesentlich. Zwar profitieren die einzelnen Stufen des Verfahrens unterschiedlich stark von der massiven Parallelität, jedoch ist insgesamt eine deutliche Reduktion der Laufzeit zu verzeichnen.

Die an den Rekonstruktionsalgorithmus selbst gestellten Anforderungen, werden jedoch nur zum Teil erfüllt. Einerseits führt die Modifikation dazu, dass zwischenzeitlich gefundene, nichtsignifikante Nebenoptima wieder verlassen werden können und hierdurch die Anzahl erfolgreicher Rekonstruktionen deutlich zunimmt, die Rate erfolgloser Rekonstruktionen kann allerdings trotzdem nur mit sehr viel Vorwissen unter 20% gedrückt werden. Vor allem die Rekonstruktion von Pixpos-Bildern, die aus realen deflektometrischen Messprozessen stammen, leidet unter zahlreichen Einflüssen.

Trotz Grafikkartenbeschleunigung und der Hierarchisierung der Facettenmenge dauert eine Positionsbestimmung je nach Parametrisierung des Rekonstruktionsalgorithmus mehrere Sekunden bis zu einer Minute, was unbefriedigend ist. Dies lässt sich jedoch durch eine Reduktion der Auflösung der Pixpos-Bilder in den Griff bekommen.

Eine genauere quantitative Auswertung findet auf den folgenden Seiten statt.

5.1 Spezifikation des Testsystems

Bevor mit der Auswertung des Rekonstruktionsalgorithmus und dem Benchmarking begonnen werden kann, wird zunächst das Testsystem spezifiziert, auf dem die Messungen stattfinden. Dies ist notwendig, um Vergleichbarkeit sicherzustellen.

Aufgrund der unterschiedlichen Charakteristika der einzelnen Stufen des Rekonstruktionsalgorithmus und dessen Anforderungen an die Soft- und Hardware können unterschiedliche Komponenten der limitierende Faktor sein. So leiden einige Berechnungen auf der Grafikkarte massiv an Warp-Divergence, sodass eine höhere Anzahl von Multiprocessors und CUDA-Cores zu keiner wesentlichen Leistungssteigerung führen würde. Umgekehrt entsprechen einige Schritte so sehr dem Konzept programmierbarer Grafikprozessoren, dass sie linear mit der Anzahl der Stream-Processors skalieren würden.

Da CUDA noch eine relativ junge Technologie ist, ist eine erhöhte Performanz von Compilerversion zu Compilerversion zu verzeichnen, auch ohne den zu übersetzenden Quellcode anzupassen. Vor allem im Bereich des Registerverbrauchs herrscht noch großes Optimierungspotenzial. Auch die Compute Capabilities haben großen Einfluss auf das Laufzeitverhalten. Deshalb ist eine genaue und vollständige Angabe der Versionen der eingesetzten CUDA-Software vonnöten.

Um eine Aussage über die Beschleunigung der GPU-Version treffen zu können, wurde die komplette Simulationsumgebung sowohl klassisch für den Hauptprozessor als auch für die Grafikkarte programmiert. Dadurch lassen sich die beiden Versionen vergleichen. Deshalb spielt der eingesetzte C/C++-Compiler sowie die CPU selbst ebenfalls eine wichtige Rolle.

Obwohl der im Testsystem vorhandene Prozessor vier physische Kerne besitzt, wird von der geschriebenen Softwarebibliothek immer nur ein Kern benutzt. Sämtlicher Host-Code ist also singlethreaded. Das gilt einerseits für die Aufrufe der Kernel, bei denen Multithreading keinen Sinn machen würde, als auch für die Simulationsberechnungen auf der CPU.

Sämtliche Software wird 64-Bit kompiliert. Für deren Ausführung ist ein 64-Bit Betriebssystem notwendig. Auch die verwendeten Bibliotheken wie das CUDA-SDK müssen in einer 64-Bit Version vorliegen. Zu einer Leistungssteigerung führt diese Architekturentscheidung nicht.

Spezifikation der Grafikkarte/GPU	
Modellbezeichnung	NVidia GeForce 8800 GTS 512
Chipbezeichnung	G92
Speicher	512 MB GDDR3
Bus	PCIe x16 @ x16
Compute Capabilty	1.1
Anzahl der Multiprocessors	16

Tabelle 5.1: Diese Tabelle listet Bezeichnungen und Eckdaten der für die Tests eingesetzten Grafikhardware auf.

Spezifikation der restlichen Hardware	
CPU	Intel Core 2 Quad Q6600 @ 2,40 GHz
Arbeitsspeicher	4096 MB DDR2 @ 2×400 MHz Dual Channel

Tabelle 5.2: Diese Tabelle listet die restliche Hardware außer der Grafikkarte auf, die Einfluss auf die Leistungsmessungen hat.

Grafikkartenspezifische Software	
CUDA-Compiler	NVidia nvcc 2.3 build 0.2.1221
Grafikkartentreiber	NVidia ForceWare 190.62 build 8.16.11.9062
Analysetool	NVidia CUDA Visual Profiler 2.3.10
zusätzliche Compileroptionen	• <code>-ptxas-options=-v</code> • <code>-arch sm_11</code> • <code>-use_fast_math</code> • <code>-maxrregcount=32</code>

Tabelle 5.3: Diese Tabelle führt die auf dem Testsystem installierte CUDA-Software an.

Sonstige Software	
Betriebssystem	Microsoft Windows 7 x64 build 6.1.7600
C/C++-Compiler	Microsoft vc++ 9 build 9.0.30729.1
Entwicklungsumgebung	Microsoft Visual Studio 9 build 9.0.30729.1
zusätzliche Compileroptionen	• <code>/O2</code> • <code>/GL</code>

Tabelle 5.4: Diese Tabelle zeigt sämtliche auf dem Testsystem installierte Software, die für die Erstellung und Ausführung der getesteten Bibliothek notwendig ist.

5.2 Auswertung des Rekonstruktionsalgorithmus

5.2.1 Einführende Erläuterungen

Unter Rekonstruktionsrate beziehungsweise Rekonstruktionsgüte versteht man das Verhältnis von korrekt rekonstruierten Positionen zu gestarteten Rekonstruktionsversuchen, also einen Wert im Intervall $[0, 1]$. In diesem Abschnitt soll eine stichpunktartige Auswertung des in Kapitel 2 vorgestellten Algorithmus durchgeführt werden. Zahlreiche Größen wie die Anzahl der Iterationen der NLO, die Menge weggenommener beziehungsweise hinzugefügter Randschichten der Bildmaske oder die Auflösung der simulierten Pixpos-Bilder, haben Einfluss auf dessen Güte. Solche Werte bilden die sogenannte Konfiguration des Rekonstruktionsverfahrens. Ziel ist es, eine Rekonstruktionsrate $>80\%$ zu erhalten. Eine Rekonstruktion gilt als erfolgreich, falls der Fehler der normierten Summe der quadratisch gewichteten Vektordifferenzen <1 ist. Die Einheit für diesen Fehler sind Pixel des Bildschirms. Eine derart kleine Vektordifferenzsumme bedeutet in der Regel für die Transformationsparameter, dass Rotationen bis auf $0,01^\circ$ Gradmaß und Translationen bis auf $10\mu\text{m}$ den zu rekonstruierenden Werten entsprechen. Wurde ein derart kleiner Fehler erreicht, so lässt sich dieser fast immer ohne viel Aufwand durch wenige weitere Iterationen in den Bereich 10^{-4} bis 10^{-5} Monitorbildpunkte reduzieren.

Aufgrund der zahlreichen Kombinationsmöglichkeiten von Parametern, die Einfluss auf die Rekonstruktionsrate haben, präsentiert dieser Abschnitt nur auszugsweise die Ergebnisse und Erfahrungen, die während der Implementierung des Algorithmus zur Positionsbestimmung gemacht werden konnten.

Um die Rekonstruktionsrate für diverse Konfigurationen zu ermitteln, werden 100 unterschiedliche Rekonstruktionen durchlaufen. Die Startwerte weichen dabei um bis zu $\pm 15^\circ$ Rotation und $\pm 50\text{mm}$ Translation von der tatsächlichen zu bestimmenden Position und Lage ab. Eine Positionsbestimmung gilt als gescheitert, falls nach Ausführung des Algorithmus der Fehler der Zielfunktion >1 ist. In den Diagrammen 5.1 bis 5.5 werden solche Fälle durch den roten Anteil in den Balken dargestellt. Im Gegensatz dazu ist eine Rekonstruktion erfolgreich, wenn der Fehler am Ende <1 ist, was in den Diagrammen grün vermerkt ist. Während der Ausführung können aber auch Fehler auftreten. So ist es möglich, dass die Wegnahme von Randschichten in der Maske dazu führt, dass simuliertes und reales Pixpos-Bild keine gemeinsamen Pixelpositionen besitzen oder der aktuell von der NLO bestimmte Transformationsparametersatz ein Pixpos-Bild erzeugt, das keine gültigen Pixelpositionen enthält. Dann können auch keine Abstände gemeinsamer Pixelpositionen berechnet werden. Selbiges gilt, wenn die Startwerte der Optimierung so gewählt sind, dass das daraus resultierende Pixpos-Bild keine gemeinsame Pixelposition mit dem realen Pixpos-Bild besitzt. Die Rate solcher Fälle ist in den Diagrammen gelb eingezeichnet.

Die aus den zu ermittelnden Transformationsparametern resultierenden Pixpos-Bilder wurden allesamt ebenfalls mit der Simulationsumgebung erstellt. Dies geschah aus mehreren Gründen: Nur auf diese Weise kann sichergestellt werden, dass die zu rekonstruierende Transformation bis auf $0,01^\circ$ Rotation und $10\mu\text{m}$ Translation bekannt ist, da man sie selbst wählen kann und diese nicht erst durch einen alternativen Prozess gemessen werden muss. Außerdem lassen sich so leicht Szenarien konstruieren, die einen Sonderfall darstellen, um den Algorithmus in Extremfällen zu testen, beispielsweise für eine anfänglich niedrige An-

zahl von gemeinsamen Pixelpositionen. Übertragen auf eine Rekonstruktion, basierend auf einem real aufgenommenen Pixpos-Bild, bedeutet dies, dass die Rekonstruktionsrate dort niedriger ausfällt. Faktoren, die zur Verschlechterung beitragen, werden in 6.1 aufgezählt. Allgemein gilt, dass es nicht die optimale Konfiguration des Algorithmus gibt. Je nach zu vermessendem Bauteil und deflektometrischen Messaufbau kann sie variieren. Die im Folgenden behandelte Konfiguration kann jedoch als Ausgangspunkt und Richtwert für verfeinerte und spezifischere Werte dienen.

5.2.2 Einfluss der Anzahl der Iterationen

Bevor der Einfluss der Anzahl der Iterationen näher beleuchtet wird, muss zuvor geklärt werden, um welche Art von Iterationen es sich handelt. Der Algorithmus zur Positionsbestimmung kennt zwei Sorten von Iterationen.

Zum einen sind das die Iterationen für die nichtlineare Optimierung. Für die NLO wird das Verfahren von Nelder und Mead [NM65] eingesetzt. Die Iterationen einer NLO werden von nun an als „innere Iterationen“ bezeichnet. Sie definieren eine obere Schranke, wie viele Schritte das Optimierungsverfahren maximal durchführen darf. Wird ein Optimum gefunden, tritt keine signifikante Verbesserung mehr auf oder unterschreitet man einen Schwellwert, kann die Optimierung vorzeitig stoppen.

Zum anderen wird eine NLO durch die Modifikation des Rekonstruktionsalgorithmus (siehe Abschnitt 2.3) nicht nur einmal durchgeführt, sondern kann mit unterschiedlichen Masken mehrmals wiederholt werden. Solche Wiederholungen werden im Folgenden „äußere Iterationen“ genannt. Wird nur eine äußere Iteration durchgeführt, ohne die Verwendung einer Maske, so handelt es sich um das in 2.2 vorgestellte Ausgangsverfahren. Wird der geforderte Fehler <1 erreicht, bevor alle äußeren Iterationen durchlaufen sind, so stoppt der Algorithmus vorzeitig.

Zunächst gilt es, die optimale Anzahl äußerer Iterationen zu finden. Zu diesem Zweck

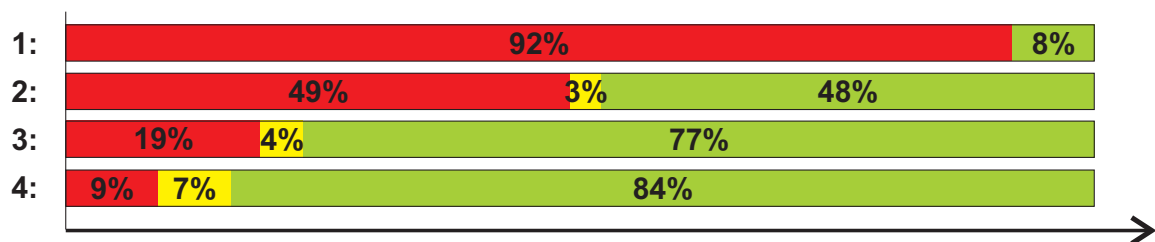


Abbildung 5.1: Einfluss äußerer Iterationen auf die erfolgreiche Rekonstruktionsrate. Die Zahlen am Anfang jeder Zeile geben die Wiederholungen äußerer Iterationen mit angepassten Masken wieder. Je mehr äußere Iterationen, desto besser die Rekonstruktionsgüte. Ab vier äußeren Iterationen liefert das Verfahren ausreichend befriedigende Ergebnisse $>80\%$.

werden 1280×960 Pixpos-Bilder verwendet. Pro äußerer Iteration werden zwei Randschichten von gültigen Pixelpositionen in der Bildmaske auf ungültig gesetzt. Die Anzahl innerer Iterationen ist mit 2000 sehr großzügig gewählt. So soll sichergestellt werden, dass pro äußerer Iteration immer ein (unter Umständen lokales) Minimum der Zielfunktion gefunden wird. Zumeist wird eine äußere Iteration jedoch beendet, bevor die 2000 inneren

Iterationen vollständig durchgeführt werden, da andere Abbruchkriterien vorzeitig erfüllt werden. Abbildung 5.1 zeigt das Ergebnis mit unterschiedlich vielen Durchläufen äußerer Iterationen. Wie zu erwarten, steigt die Rekonstruktionsrate mit der Anzahl äußerer Iterationen. Ab vier äußeren Iterationen wird die gewünschte Güte von $>80\%$ erreicht. Interessant ist der Vergleich der oberen beiden Fälle im Diagramm, bei denen nur eine beziehungsweise zwei äußere Iteration ausgeführt werden. Die Steigerung der Rekonstruktionsgüte zeigt ganz klar, dass die Modifikation aus 2.3 zu einer signifikanten Verbesserung des Algorithmus führt.

Die Wahl der Anzahl innerer Iterationen veranschaulicht Diagramm 5.2. Wieder werden

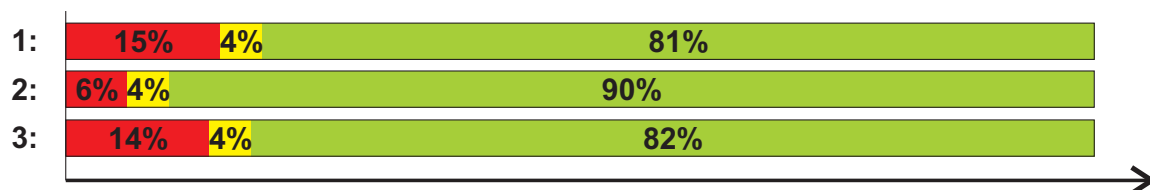


Abbildung 5.2: Einfluss innerer Iterationen auf die erfolgreiche Rekonstruktionsrate. (1) stellt das Ergebnis für vier äußere Wiederholungen à 500 innere Iterationen dar. Dem gegenüber verbessert sich die Rekonstruktionsgüte bei (2) für fünf äußere Wiederholungen bei je 500 inneren Iterationen. Den besten Kompromiss aus hoher Rekonstruktionsrate und wenigen Iterationen bietet (3): Fünf äußere Iterationen, die (500, 200, 200, 200, 500) innere Iterationen verwenden.

1280 × 960 Pixpos-Bilder verwendet. Die Anzahl weggenommener gültiger Randschichten zur Maske ist abermals zwei. Hier gilt es einen Kompromiss zu finden zwischen möglichst wenigen inneren Iterationen pro NLO und der gewünschten erfolgreichen Rekonstruktionsrate. Je weniger Iterationen, desto kürzer die Laufzeit. Als bester Kompromiss haben sich fünf äußere Wiederholungen herausgestellt, die (500, 200, 200, 200, 500) innere Iterationen verwenden. Die erste NLO darf mehr Durchläufe durchführen, um grob in die Region des globalen Minimums vorzustößen. Darauf folgen drei kürzere NLOs, um eventuelle insignifikante Nebenoptima zu erreichen und durch Anwendung der Maske wieder zu verlassen. Zuletzt dient die fünfte NLO der Verfeinerung des Endergebnis.

5.2.3 Relevanz weggenommener Randschichten

Um insignifikante Nebenoptima zu „überwinden“, wird zwischen zwei NLOs eine Maske bestimmt, die gültige Pixelpositionen im Randbereich ausschließt. Die Anzahl der Randschichten, die pro äußerer Iteration entfernt werden, hat entscheidenden Einfluss auf die positive Rekonstruktionsrate. Randschichten sind über Vierer-Nachbarschaften zu als ungültig gekennzeichneten Bereichen im Pixpos-Bild definiert.

Die Konfiguration der inneren und äußeren Iterationen entspricht bei der folgenden Auswertung denjenigen, die sich im vorausgehenden Unterabschnitt 5.2.2 als befriedigend herausgestellt haben. Die Auflösung der verwendeten Pixpos-Bilder ist 1280 × 960.

Wie Diagramm 5.3 zeigt, führen zwei entfernte Randschichten pro äußerer Wiederholung zu den besten Ergebnissen. Interessant ist dabei zu beobachten, dass bei zunehmender

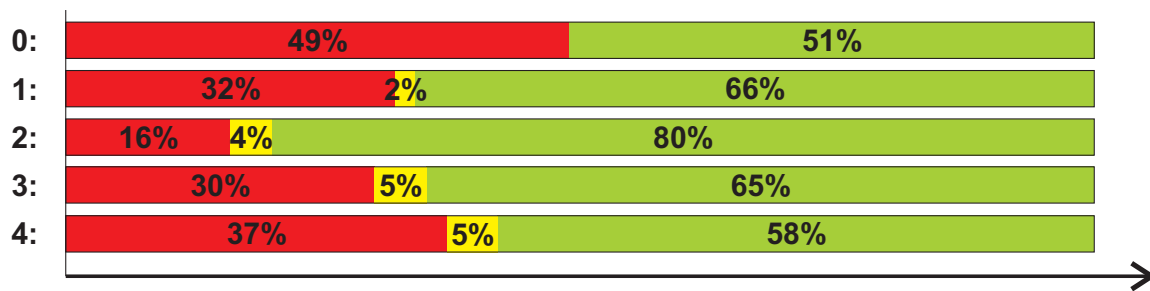


Abbildung 5.3: Abhängigkeit der Rekonstruktionsgüte von der Anzahl weggelassener Randschichten gültiger Pixelpositionen. Die Zahlen am Anfang jeder Zeile geben die Menge entfernter Randschichten an. Das Entfernen von zwei Randschichten pro äußerer Wiederholung liefert die besten Ergebnisse.

Zahl ignoriert Randschichten die Rekonstruktionsgüte wieder sinkt.

Je mehr Randschichten weggewonnen werden, desto größer wird der Anteil von Rekonstruktionen, die aufgrund eines internen Fehlers scheitern (gelber Bereich). Das ist logisch, denn je mehr Randschichten weggelassen werden, desto weniger gemeinsame Pixelpositionen im realen und simulierten Pixpos-Bild gibt es. Dadurch steigt die Möglichkeit, dass alle gemeinsamen Pixelpositionen ungültig gesetzt werden.

Überraschend ist die Beobachtung, dass 0 entfernte Randschichten (oberster Fall im Diagramm) immer noch eine Rekonstruktionsrate von 51% aufweist. Man würde eher eine Güte von 8% wie in Abbildung 5.1 erwarten. Der bloße Neustart der NLO führt zu einer deutlich höheren Qualität des Algorithmus. Eine Begründung hierfür liefert 5.2.5.

5.2.4 Auswirkungen der Pixpos-Bild-Auflösung

Bereits an mehreren Stellen wird erwähnt, dass die Simulation der Pixpos-Bilder sehr zeitaufwändig ist. Die Beschleunigung durch Grafikkarten, die Hierarchisierung der Facettenmenge und die Minimierung der Anzahl von Iterationen sind drei Maßnahmen, um die Laufzeit des Rekonstruktionsalgorithmus zu reduzieren. Eine weitere Möglichkeit der Laufzeitoptimierung liegt in der Reduktion der Auflösung der zu simulierenden Pixpos-Bilder. Je geringer diese ist, desto weniger Raycastings und Reflexionen müssen berechnet werden.

Natürlich machen verringerte Auflösungen nur Sinn, wenn darunter nicht die Rekonstruk-



Abbildung 5.4: Dieses Diagramm vergleicht die Rekonstruktionsgüte für unterschiedliche Pixpos-Bild-Auflösungen. Ansonsten gilt für beide Rekonstruktionsszenarien die selbe Konfiguration. Trotz der um den Faktor 100 reduzierten Auflösung vermindert sich die positive Rekonstruktionsrate nur um etwa 9%.

tionsrate leidet. Diagramm 5.4 vergleicht zwei Rekonstruktionen. Bei der oberen kommen Pixpos-Bilder mit einer Auflösung von 1280×960 Pixelpositionen zum Einsatz. Demgegenüber verwendet die zweite Rekonstruktion Auflösungen von 128×96 , also eine um einen Faktor 100 geringere Anzahl von Elementen. Sämtlich andere Größen der Konfiguration wie die Anzahl äußerer und innerer Iterationen oder die Menge entfernter Randschichten sind in beiden Fällen gleich. Sie entsprechen den in 5.2.2 und 5.2.3 gefundenen Werten. Aus dem Diagramm geht hervor, dass die Rekonstruktionsgüte zwar gemindert wird, der Unterschied bei den beiden Auflösungen jedoch eher gering ausfällt. Um eine ähnlich hohe Güte wie bei 1280×960 -Pixpos-Bildern zu erhalten, könnten bei 128×96 -Bildern eine leicht erhöhte Anzahl innerer Iterationen verwendet werden.

Trotz der veränderten Auflösung stellt sich heraus, dass die Anzahl ungültig zu setzender Randschichten nicht angepasst werden muss. Zwei entfernte Randschichten pro äußerer Iteration erweisen sich in beiden Fällen als die günstigste Wahl.

Bei einer verringerten Pixpos-Bild-Auflösung ist darauf zu achten, dass auch das real aufgenommene Pixpos-Bild, das durch wiederholte NLO rekonstruiert werden soll, entsprechend angepasst werden muss. Entweder wird bereits ein niedrig auflösendes Pixpos-Bild beim deflektometrischen Messprozess ertellt, oder es werden nur ausgewählte Sichtstrahlen der realen Kamera beachtet. In letzterem Fall ist darauf zu achten, dass diese selektierten Sichtstrahlen auch in der Simulation zum Einsatz kommen.

In der Praxis führt eine Reduktion der Pixpos-Bild-Auflösung um den Faktor 100 nicht automatisch zu einer Beschleunigung der Rekonstruktion um den selben Faktor. Für die GPU-Implementierung bedeutet eine derartige Auflösungs-minderung eine Beschleunigung um das Vierfache. Die CPU-Implementierung profitiert deutlicher, etwa um den Faktor 30. Für diese Diskrepanz gibt es mehrere Gründe. Während beim hochauflösenden Szenario ein Großteil der erfolgreichen Rekonstruktionen bereits nach zwei oder drei äußeren Iterationen feststehen, werden im niedrigauflösenden Fall öfter alle fünf Wiederholungen benötigt. Ein weiterer Grund liegt in der Tatsache, dass die massive Parallelität der GPU im Fall mit wenigen Pixelpositionen nicht so sehr zum Tragen kommt, wie im Fall mit 1280×960 Pixelpositionen. Abschnitt 5.3 liefert hierfür eine Begründung.

5.2.5 Größe des Startsimplex

Neben den bereits vorgestellten Größen gehören noch weitere Werte zur Konfiguration des Rekonstruktionsalgorithmus. Der letzte erwähnenswerte ist die Wahl der Größe des Startsimplex für die NLO. Das Verfahren von Nelder und Mead ist ein Simplexalgorithmus. Die Angabe der Ausdehnung eines Startsimplex kann helfen, das Erreichen insignifikanter Nebenoptima zu reduzieren und die Wahrscheinlichkeit des Findens des globalen Optimums zu erhöhen. Dementsprechend hängt auch die Rekonstruktionsgüte vom Startsimplex ab. Ist ein grobes Intervall bekannt, innerhalb dessen der Startwert der Rekonstruktion von den tatsächlichen Transformationsparametern abweicht, so kann diese Information für den Startsimplex verwendet werden. Bei der hier vorliegenden Auswertung der Rekonstruktionsgüte unterschiedlicher Konfigurationen des Algorithmus ist dies der Fall. Die Abweichung der Rotation liegt bei maximal $\pm 15^\circ$, die der Translation bei $\pm 50\text{mm}$.

Eine Strategie, die sich in der Praxis als befriedigend herausgestellt hat, wählt den Startsimplex für die erste äußere Iteration doppelt so groß wie die vermutete Abweichung von Start- und Zieltransformationsparametern. Für jede weitere äußere Iteration und den Neu-

start der NLO halbiert man die Größe des Startsimplex.

Diagramm 5.5 vergleicht drei unterschiedliche Taktiken für Startsimplices. Die restlichen Werte der Algorithmuskonfiguration werden entsprechend der vorausgehenden Punkte gewählt, die sich als praktikabel herausgestellt haben. Wählt man die Größe des Startsimplex zu groß oder reduziert ihn nicht von äußerer Iteration zu äußerer Iteration, so leidet darunter die Rekonstruktionsgüte.

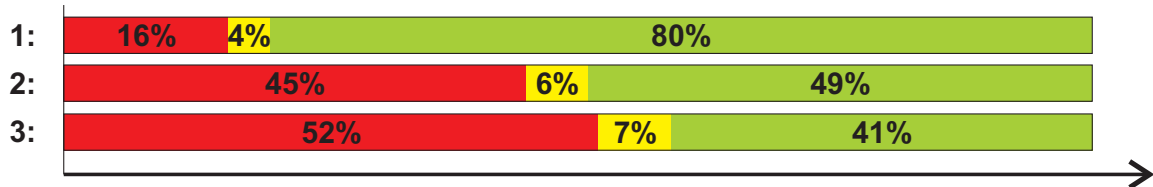


Abbildung 5.5: Dieses Diagramm vergleicht die positive Rekonstruktionsgüte in Abhängigkeit der Größe und Variation des Startsimplex für das Verfahren von Nelder und Mead. (1) wählt den Startsimplex doppelt so groß wie die vermutete maximale Differenz zwischen Start- und Zieltransforationsparametern und halbiert diese Größe für jede weitere äußere Iteration. Diese Taktik erweist sich am erfolgreichsten. (2) verfährt auf die selbe Weise, nur ohne die Halbierung der Simplexgröße. Wird der Startsimplex zu groß gewählt, steigt auch die Anzahl insignifikanter Nebenoptima, die von der NLO erreicht werden können. Dementsprechend sinkt die Rekonstruktionsgüte, wie (3) veranschaulicht.

5.3 Laufzeitvergleich zwischen CPU- und CUDA-Implementierung

5.3.1 Einführende Erläuterungen

Um die Grafikkartenhardware korrekt auszunutzen, ist es sinnvoll, Algorithmen in möglichst viele, sinnvoll kleine Kernel zu unterteilen. Die Begründung hierfür findet sich bereits in Unterabschnitt 3.3.3. Sowohl der Rekonstruktionsalgorithmus selbst (siehe Abschnitt 2.2), als auch dessen Modifikation (siehe Abschnitt 2.3) lassen sich gut in solche kleinen Kernel stückeln.

In diesem Abschnitt werden die Implementierungen für CPU und GPU Schritt für Schritt miteinander verglichen. Darüber hinaus wird auf typische Leistungsdaten von CUDA-Code wie Occupancy, Instruction Throughput und Warp Divergence eingegangen. Die einzelnen Schritte unterscheiden sich mitunter deutlich in Bezug auf ihre Eignung für die Grafikkarte, was sich mit den eben genannten Messwerten gut quantifizieren lässt.

Zu beachten ist, dass die Hauptprozessorimplementierung nicht in dem Maße optimiert ist, wie das für das Grafikkartenprogramm der Fall ist. Letzteres enthält deutlich mehr Know-How als der Host-Code. Dadurch bietet der CPU-Quelltext noch zahlreiche Möglichkeiten für Verbesserungen. Eine Reduktion der Ausführungszeit der Hauptprozessorvariante um den Faktor 2 ist daher nicht unrealistisch.

Das hat zwar Einfluss auf die Deutlichkeit, mit der die beiden Implementierungen auseinander liegen, an der grundsätzlichen Schlussfolgerung, welcher Chip die Berechnungen schneller abschließt, ändert das aber nichts.

Nicht nur Unterschiede in der eingesetzten Hard- und Software sowie unvermeidbare Unterschiede in der Qualität von CPU- und GPU-Implementierung sind Gründe, warum die Zahlen für sich alleine nur bedingt belastbar sind. Sie hängen zusätzlich von der Größe des eingesetzten STL-Modells, der Auflösung der Pixpos-Bilder und vielen weiteren Rahmenbedingungen wie der Kalibrierung des deflektometrischen Messsystems ab.

Nicht alle Arbeitsschritte der Simulation werden ausgewertet, sondern nur solche, die entscheidende Stationen der Positionsbestimmung darstellen. Hier ist es nicht von Interesse, den Rekonstruktionsalgorithmus exakt wiederzugeben, sondern die individuellen Herausforderungen bei den einzelnen Schritten in Bezug auf die Grafikkarte vorzustellen.

5.3.2 Sichtstrahlgenerierung

Der erste Schritt in der Simulation des deflektometrischen Messaufbaus ist die virtuelle Erzeugung der Sichtstrahlen der Kamera. Diese erzeugen, falls sie von der Geometrie reflektiert werden und den Bildschirm treffen, die Pixelpositionen eines Pixpos-Bildes. Um realitätsnahe Strahlen zu erzeugen, muss eine präzise Kalibrierung der realen Kamera vorausgegangen sein.

Sichtstrahlgenerierung ist ein Verfahren, das sich nahezu in idealer Weise für die Grafikkarte eignet. Die Kamera lässt sich mit wenigen Parametern beschreiben, die sich schnell in den Device Memory kopieren lassen. Sie passen allesamt in den RMem, auf den die Threads sehr schnell zugreifen können. Aus den Kameraparametern, die für jeden Bildpunkt gleich sind, werden im Anschluss unter ausschließlicher Verwendung arithmetischer Befehle und ohne Verzweigungen die Sichtstrahlen für jedes Pixel berechnet. Dabei benö-

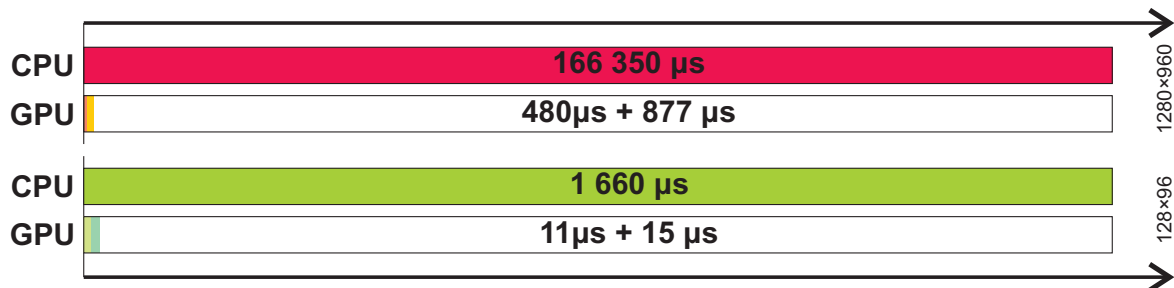


Abbildung 5.6: Während die oberen beiden Balken die Sichtstrahlerzeugung einer 1280×960 Bildpunkte großen Kamera vergleichen, stellen die unteren beiden Balken die Laufzeiten einer mit 128×96 wesentlich niedriger auflösenden Kamera gegenüber. Jeweils der erste der beiden Balken steht für die Ausführungszeit der CPU-Variante, der zweite für das GPU-Programm. Da die Generierung der Sichtstrahlen auf der Grafikkarte in zwei separaten Schritten erfolgt, müssen die beiden Zeiten aufsummiert werden.

tigt kein Sichtstrahl Informationen eines anderen Sichtstrahls. Die Bildpunktmatrix der Kamera lässt sich gut in rechteckige Blöcke unterteilen. Diese Bündel von Sichtstrahlen entsprechen zur Laufzeit den Thread Blocks. Somit kann die Speicherung der Sichtstrahlpakete Coalescing nutzen.

Um den Registerverbrauch pro Kernel weiter zu reduzieren, wird für die Grafikkartenvariante die Sichtstrahlerzeugung wiederum in zwei Einzelschritte unterteilt. Zunächst werden nur die intrinsischen Kameraparameter beachtet. Das Kamera-Koordinatensystem (kurz: K-KoSy) fällt mit dem W-KoSy zusammen. Im Anschluss werden die Strahlen normalisiert und der Transformation unterworfen, die das K-KoSy in das W-KoSy überführt.

Abbildung 5.6 vergleicht die Ausführungszeit der Sichtstrahlgenerierung von CPU und GPU. Dafür werden die Laufzeiten der Einzelschritte des Device-Codes aufaddiert, um sie dem Host-Code gegenüberzustellen.

5.3.3 Simulation der Pixpos-Bilder

Um ein virtuelles Pixpos-Bild zu erhalten, werden die Kamerasichtstrahlen zunächst mit der Geometrie geschnitten. Im Anschluss wird überprüft, ob die Reflexionen Pixelpositionen erzeugen, indem sie auf den Monitor treffen.

Das Raycasting findet in einem eigenen Kernel statt. Dieser arbeitet nach dem in 4.5.3 vorgestellten Prinzip. Bis hier hin muss nur diejenige Facette der Dreiecksmenge identifiziert werden, die von einem Strahl getroffen wird. Dabei handelt es sich um klassisches Raycasting. Erwartungsgemäß leidet es massiv unter Warp-Divergence, die der Baumstruktur der BVH und den zahlreichen bedingten Anweisungen geschuldet ist. Die Baumstruktur verhindert darüber hinaus Coalescing. Da sich der Kernel für die Facettenbestimmung nicht aufteilen lässt, ist der Registerverbrauch kritisch hoch. Dies alles führt dazu, dass der Instruction Throughput, vor allem bei niedrig auflösenden Pixpos-Bildern, extrem niedrig ist (siehe zusammenfassende Tabelle 5.6). An dieser Stelle lässt sich allerdings hervorragend demonstrieren, dass es eine große Menge von Threads dem Hardwareschedulersystem erlaubt, trotz dieser widrigen Umstände einen hohen Instruction Throughput

zu generieren. Erhöht man die Anzahl der Threads um den Faktor 100, so skaliert in diesem speziellen Fall der Instruction Throughput beinahe linear.

Hat man das Raycasting durchgeführt und für jeden Sichtstrahl diejenige Facette be-

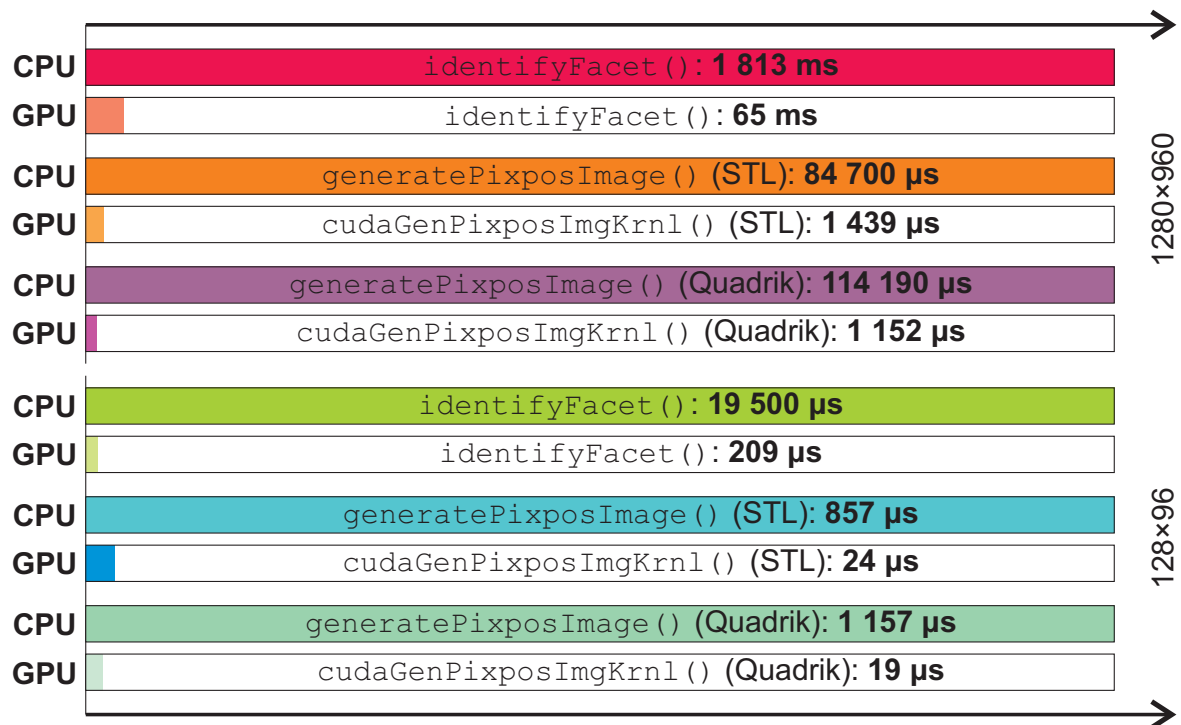


Abbildung 5.7: Wieder zeigt das obere Diagramm die Messungen an einem 1280×960 großen Bild, während die untere Grafik für ein um den Faktor 100 kleineres 128×96 Pixpos-Bild gilt. `identifyFacet()` ist diejenige Funktion/derjenige Kernel, der das Raycasting durchführt. Dabei wird das einzige Mal Millisekunden (kurz: ms) anstatt Mikrosekunden (kurz: μs) als Zeiteinheit verwendet. `generatePixposImage()` (STL) beziehungsweise `cudaGenPixposImgKrnl()` (STL) führen die Reflexion an der gefundenen Facette und die Pixelpositionsberechnung durch. `generatePixposImage()` (Quadrik) beziehungsweise `cudaGenPixposImgKrnl()` (Quadrik) fassen beide Schritte für Quadriken zusammen.

stimmt, die er als erstes schneidet, wird diese Information verwendet, um den Sichtstrahl an der Facette zu reflektieren und mit dem Monitor zu schneiden. Hierbei kann zwar wiederum nicht Coalescing eingesetzt werden, die Organisation der Vertex- und Normalenmenge in einer Textur erlaubt jedoch Caching. Wie groß der Leistungsgewinn dadurch ist, erläutert Unterabschnitt 5.4.2. Im Gegensatz zum Raycasting ist die Strahlreflexion und die Pixelpositionsberechnung besser für Grafikkarten geeignet. Der Monitor lässt sich ähnlich der Kamera mit nur sehr wenigen Größen beschreiben. Die Glättung der Normalen am Reflexionspunkt, wie in 4.6 beschrieben, sorgt für einen hohen arithmetischen Anteil. Diese positiven Voraussetzungen ermöglichen trotz eines ebenfalls hohen Registerverbrauchs einen akzeptablen Instruction Throughput.

Wie groß die Leistungseinbußen durch das Raycasting der Dreiecksmenge sind, zeigt ein

Vergleich mit Quadriken, die ebenfalls eine Oberfläche modellieren können. Quadriken sind die Nullstellenmengen von Polynomen vom Grad 2 in zwei Veränderlichen.

$$Q = \left\{ (x, y, z) \in \mathbb{R}^3 \mid a_0x^2 + a_1y^2 + a_2z^2 + a_3xy + a_4xz + a_5yz + a_6x + a_7y + a_8z + a_9 = 0 \right\} \quad (5.1)$$

Sie können gekrümmte Oberflächen darstellen, sind aber aufgrund ihrer Limitierung auf Grad 2 eher unflexibel. Unter Verwendung der zehn Koeffizienten $a_0, \dots, a_9 \in \mathbb{R}$, die eine Quadrik beschreiben, können Strahl/Quadrik-Schnitt, Normalen- und Reflexionsberechnungen in geschlossener Form und somit in $O(1)$ pro Quadrik vollzogen werden. Dies und der daraus resultierende hohe Rechenaufwand führen dazu, dass eine Verwendung einer Quadrik anstatt einer Dreiecksmenge zu äußerst schnellen Berechnungen auf der Grafikkarte führt, wie Abbildung 5.7 deutlich macht.

Das schlechte Abschneiden des Raycastings auf der CPU ist auch der mangelnden Optimierung des Hauptprozessorcodes geschuldet. Bei einer gewissenhaften Implementierung fällt der Unterschied deutlich niedriger aus. Nichtsdestotrotz ist die GPU-Version so schnell, dass sie sich für Echtzeitraytracing auch großer Dreiecksmengen eignet.

5.3.4 Wegnahme von Randschichten einer Maske

Die Wegnahme von gültigen Randschichten zu einer Maske eines Pixpos-Bildes wird zwischen jeder der hintereinander ausgeführten nichtlinearen Optimierungen vollzogen.

Dieser Vorgang profitiert relativ wenig von der Grafikkartenbeschleunigung. Zwar kann für jedes Element der Maske ein einzelner Thread benutzt werden und die rechteckige Maske lässt sich auch gut in Blöcke unterteilen, der Zugriff auf die Vierer-Nachbarschaft führt jedoch zu Problemen. Da ein Maskenelement die Daten seines oberen, unteren, linken und rechten Nachbarn in der Maskenmatrix benötigt um zu entscheiden, ob er gültig oder ungültig gesetzt wird, kann beim Laden eines Maskenblocks in den SMem nur bedingt Coalescing zum Einsatz kommen. Zusätzlich zu den eigentlichen Daten benötigen die Maskenpunkte im Randbereich des Blocks aufgrund der Vierer-Nachbarschaft auch Elemente der Maske, die eigentlich von anderen Blöcken bearbeitet werden. Somit kann also beim Laden der benötigten Maskendaten kein gemeinsamer Zugriff auf den Device Memory stattfinden, da die Elemente im Randbereich weitere Punkte laden müssen, die außerhalb des vom Thread-Block verwalteten Bereichs liegen. Für die Lese- und Schreiboperationen innerhalb eines Blocks kann Coalescing eingesetzt werden, die Einschränkung betrifft nur die Randbereiche des Blocks.

Nachteilig wirkend kommt hinzu, dass die benötigte Zeit für Speichertransfers wesentlich größer ist als die für die Berechnungen. Wie eben geschildert, gilt es beim Transfer vom GMem in den SMem das Nachbarschaftsproblem zu beachten, was Speicherzugriffe verlangsamt. Wurden die Daten geladen, so finden nur wenige Operationen mit ihnen statt, nämlich eine bool'sche Verknüpfung von oberem, unterem, linkem und rechtem Nachbarn, da nur diese Einfluss auf das betrachtete Element der Maske haben.

Hauptprozessoren eignen sich besser für derartige Anforderungen. Ihr Speicherzugriff ist flexibler und hängt nicht von Blöcken, die sich überlappen ab. Außerdem reagiert der Cache einer CPU flexibler, da er sich besser der Iteration durch die Maskenelemente anpasst und somit langsame Zugriffe auf den Hauptarbeitsspeicher reduziert werden können.

Abbildung 5.8 stellt die Messwerte für CPU- und GPU-Implementierung gegenüber. Die Hauptprozessorversion ließe sich beschleunigen, wenn, anstatt jede Zeile einzeln, mit Hilfe

einer ebenenfüllenden Kurve iteriert werden würde. Mehr Lokalität und dadurch bessere Ausnutzung des CPU-Cache wären die Folge. Im Gegenzug könnten bei der Grafikkartenvariante Block- und Gridkonfiguration so gewählt werden, dass sie größer als die zu bearbeitende Maske wären. Beim Speicherzugriff müssten die IDs der Threads und Blöcke zwar angepasst werden und es käme zu redundanten Berechnungen, aufgrund der massiven Parallelität wäre dies aber die bessere Wahl.

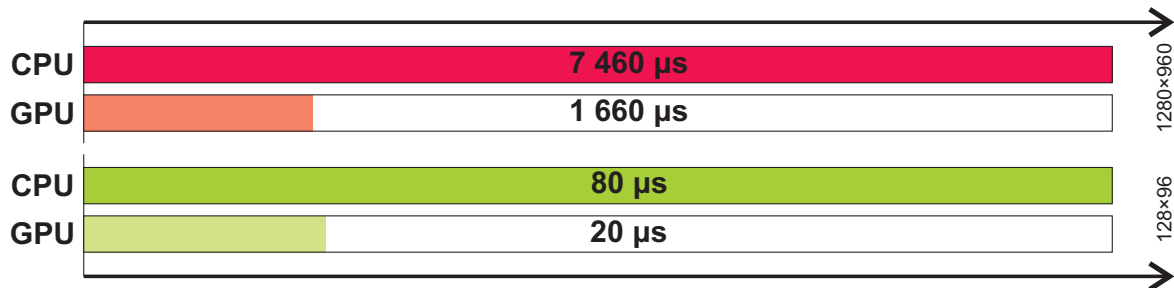


Abbildung 5.8: Oben ist der Leistungsvergleich zwischen Host- und Device-Code für eine 1280×960 Maske zu sehen, zu der eine Randschicht hinzugefügt wird, unten der zweier 128×96 Masken. Die Beschleunigung liegt diesmal nur bei einem Faktor von Vier, was verglichen mit anderen Funktionen/Kernels weit unterdurchschnittlich ist.

5.3.5 Berechnung der normierten Summe der Pixelpositions differenzen

Wie wichtig ein hohes Maß an arithmetischen Operationen in Kernels ist, zeigt das Beispiel der Berechnung der Vektordifferenzen. Sämtliche Voraussetzungen für die Grafikkarte sind erfüllt. Sowohl die Pixelpositionen aus dem realen wie auch aus dem simulierten Bild lassen sich durch Coalescing laden. Das Ergebnis, der quadratisch gewichtete Abstand zweier Pixelpositionen, kann unter Verwendung von gemeinsamen Speicherzugriffen gespeichert werden. Bis hierhin werden nur die Differenzen zweier Pixelpositionen berechnet. Die Aufsummierung aller Vektordifferenzen findet noch nicht statt. Auch der niedrige Registerverbrauch der einfachen Abstandsberechnung, deren Code in Listing 5.1 zu sehen ist, spricht für eine Programmierung in CUDA.

```

1  __device__ inline float pixposDifference(
2      const float2 pixpos1, const float2 pixpos2) {
3      return (pixpos1.x - pixpos2.x) * (pixpos1.x - pixpos2.x) +
4              (pixpos1.y - pixpos2.y) * (pixpos1.y - pixpos2.y);
5  }
```

Listing 5.1: Hilfskernel für quadratisch gewichtete Vektordifferenz.

Eben diese kurze Phase, in der arithmetische Operationen durchgeführt werden, zieht niedrigen Instruction Throughput nach sich. Zwar finden sämtliche Speicherzugriffe parallel statt, jedoch wird für sie wesentlich mehr Zeit benötigt als für die eigentliche Berechnung der Pixelpositionsabstände in gemessenem und virtuellem Pixpos-Bild.

Deshalb ist es auch nicht verwunderlich, dass der Zeitgewinn gegenüber dem Quelltext

für den Hauptprozessor relativ niedrig ausfällt, wie Abbildung 5.9 zeigt. Auch wenn viele Voraussetzungen für eine Grafikkartenimplementierung sprechen, können zu wenige Rechenvorgänge ein Verfahren für die GPU disqualifizieren.

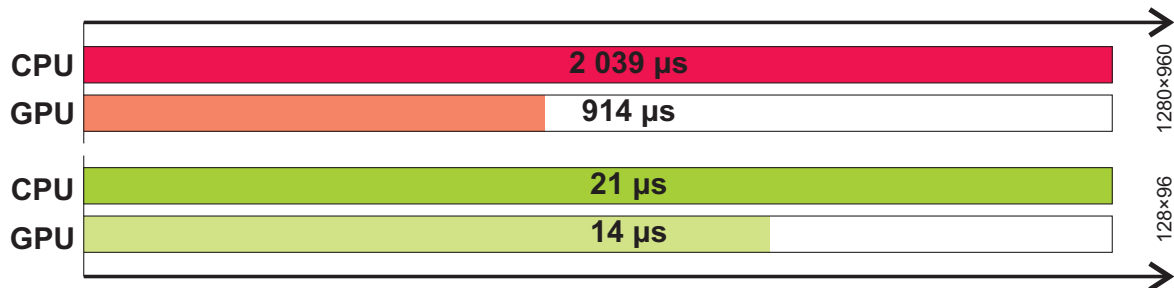


Abbildung 5.9: Aufgrund des niedrigen arithmetischen Aufwands des Kernels für die Vektordifferenzen fällt der Geschwindigkeitsvorteil sowohl für große (oben) als auch für kleine Masken (unten) nicht mehr so deutlich aus, wie bei den bisherigen Beispielen.

5.3.6 Beurteilung der Gesamtberechnung

Wie die bisherigen Einzelvergleiche vermuten lassen, profitiert die Simulation deutlich von Grafikkartenbeschleunigung. Mit der in den Unterabschnitten 5.2.2, 5.2.3 und 5.2.5 gefundenen Konfiguration und einer Pixpos-Bild-Auflösung von 128×96 dauert eine der 100 Testrekonstruktionen im Schnitt 14,556 Sekunden auf der GPU und 205,587 Sekunden auf der CPU. Der Einsatz von programmierbaren Grafikkarten beschleunigt die Ausführung des gesamten Algorithmus also etwa um den Faktor 15. Kommen Pixpos-Bilder mit einer Auflösung von 1280×960 Elementen zum Einsatz, so verlängert sich die GPU-Ausführung auf durchschnittlich 59,914 Sekunden pro Rekonstruktion. Eine Positionsbestimmung unter Verwendung des Hauptprozessors mit einer solchen Auflösung dauert mehrere Minuten, was unpraktikabel ist.

Ein Vergleich der Laufzeiten der Einzelschritte zeigt, dass das Raycasting, implementiert durch die Funktion/den Kernel `identifyFacet()`, mit weitem Abstand am zeitaufwändigsten ist. Dies ist aber nicht ausschließlich der eher mäßigen Eignung von Schnittpunktberechnungen mit Facettenmengen für GPUs geschuldet. Raycasting von Dreiecksmengen ist grundsätzlich aufwändig, und der passable Instruction Throughput bei höheren Auflösungen zeigt, dass es von Grafikkartenbeschleunigung profitieren kann. Ein Vergleich mit einer Version für den Hauptprozessor unterstreicht die Beobachtung.

Abschließend listen die Tabellen 5.5 und 5.6 die ermittelten Zahlen der Benchmarks auf. Während Tabelle 5.5 die Konfiguration des Testlaufs wie Block- und Gridgröße sowie Größen, die bereits zur Compilezeit feststehen, auflistet, enthält Tabelle 5.6 die Zahlen der Performanzanalysen.

Konfiguration der Kernel						
Kernelname	Gridgröße	Blockgröße	SMem/Block	CMem/Block	Register/Thread	
getMod24RawDirCCS ()	80 × 60 8 × 6	16 × 16 16 × 16	56+16 Bytes	4 Bytes	10	
getDirSCSFromRawDirCCS ()	80 × 60 8 × 6	16 × 16 16 × 16	56+16 Bytes	8 Bytes	7	
identifyFacet ()	160 × 120 16 × 12	8 × 8 8 × 8	488+16 Bytes	32 Bytes	25	
cudaGenPixposImgKrn1 () (STL)	80 × 60 8 × 6	16 × 16 16 × 16	120+16 Bytes	8 Bytes	25	
cudaGenPixposImgKrn1 () (Quadrk)	80 × 60 8 × 6	16 × 16 16 × 16	152+16 Bytes	8 Bytes	15	
cudaTrim ()	80 × 60 8 × 6	16 × 16 16 × 16	1320+16 Bytes	32 Bytes	29	
cudaNormedPixposDiffKrn1 ()	80 × 60 8 × 6	16 × 16 16 × 16	80+16 Bytes	4 Bytes	8	

Tabelle 5.5: Diese Tabelle listet die Konfigurationen der bisher in diesem Abschnitt absolvierten Performanceanalysen auf. Zusätzlich enthält die Tabelle Kernelinformationen, die bereits zur Compilezeit feststehen.

Erklärungen zu den Kernels `getMod24RawDirCCS ()` und `getDirSCSFromRawDirCCS ()` finden sich in Unterabschnitt 5.3.2. Eine Erläuterung zu den Kernels `identifyFacet ()` sowie `cudaGenPixposImgKrn1 ()` für das STL-Modell und Quadriken, die sich mit der Erstellung von Pixpos-Bildern beschäftigen, kann in Unterabschnitt 5.3.3 nachgelesen werden. `cudaTrim ()` passt eine Maske von NLO zu NLO an und wird in Unterabschnitt 5.3.4 behandelt. Zuletzt erläutert Unterabschnitt 5.3.5 das Verhalten von Kernel `cudaNormedPixposDiffKrn1 ()`.

Die Fortsetzung dieser Tabelle mit den gemessenen Zahlen ist Tabelle 5.6 auf der nächsten Seite.

Messergebnisse der Kernel				
Kernelname	Occ.	I. Throughput	Serial. Warps	Coalescing
getMod24RawDirCCS ()	1	0,55609 0,244779	0	ja. load: 153600 ja. load: 0, store: 1536
getDirSCSFromRawDirCCS ()	1	0,298313 0,179484	0	ja. load: 19200, store: 153600 ja. load: 192, store: 1536
identifyFacet ()	$\frac{1}{3}$	0,184908 0,0023589	5590213 2488	nein. read: 153600, store: 307200 nein. load: 1536, store: 3072
cudaGenPixposImgKrn1 () (STL)	$\frac{1}{3}$	0,73021 0,465487	0	ja. load: 28800, store: 115200 ja. load: 288, store: 1152
cudaGenPixposImgKrn1 () (Quadrik)	$\frac{2}{3}$	0,954725 0,576326	0	ja. load: 19200, store: 115200 ja. load: 192, store: 1152
cudaTrim ()	$\frac{1}{3}$	0,67294 0,324399	41985 2112	ja/nein. load: 21560/10780, store: 38400/0 ja/nein. load: 106/106, store: 384/0
cudaNormedPixposDiffKrn1 ()	1	0,273293 0,182073	0	ja. load: 48000, store: 76800 ja. load: 480, store: 768

Tabelle 5.6: An dieser Stelle werden die Messergebnisse der Benchmarks aufgelistet. Die Konfigurationen, die zu diesen Werten führen, lassen sich in Tabelle 5.5 auf der vorherigen Seite nachschlagen. Die Zeilen müssen dafür nur entsprechend auf dieser Seite fortgeführt werden.

Occ. ist die Abkürzung für Occupancy, I. Throughput die für Instruction Throughput und Serial. Warps ist die Kurzform für serialisierte Warps.

Die unter Coalescing angegebenen Zahlen beziehen sich auf die Lese- beziehungsweise Schreiboperationen, nicht auf verarbeitete Bytes.

5.4 Vergleich unterschiedlicher CUDA-Handhabungen

5.4.1 Einführende Erläuterungen

Dieser Abschnitt präsentiert und analysiert Mechanismen, mit denen sich die Ausführungszeit von CUDA-Programmen verkürzen lässt, ohne Anpassungen am Algorithmus oder sogar am Quellcode vornehmen zu müssen. Die Spanne reicht hierbei von wenigen Prozent bis hin zu Beschleunigungen um den Faktor 3 bis 4.

Abermals werden ausgewählte Beispiele verwendet, um den Geschwindigkeitsvorteil zu demonstrieren. Dabei wird Funktionalität getestet, die besonders stark profitiert.

5.4.2 Einfluss von gecachten Texturen

Ein bereits angemerktter Nachteil von programmierbaren Grafikkarten ist das Fehlen impliziter Caches. Damit Daten aus dem Global Memory zwischengespeichert werden, müssen diese in eine ein-, zwei- oder dreidimensionale Textur überführt werden. Um diesen Schritt durchführen zu können, müssen die Daten jedoch gewisse Voraussetzungen, wie ein gewisses Alignment und einen speziellen Typ, erfüllen.

Die Transformation in eine Textur ist immer dann sinnvoll, wenn beim Lesen aus dem GMem kein Coalescing möglich ist. Das ist zumeist dann der Fall, wenn die Speicherzugriffe keinem geordneten Muster entsprechen. Dieses Szenario tritt beim Raycasting der Facettenmenge auf, während unter Verwendung des Kernels `identifyFacet()` durch die BVH traversiert wird.

Tabelle 5.7 vergleicht den Fall, bei dem Speicherzugriffe gepuffert werden, mit einer Im-

Vergleich gecachter/ungecachter Speicherzugriff				
Auflösung	gecached/ungecached	Register	I. Throughput	Zeit
1280×960	gecached	25	0,184908	65ms
	ungecached	28	0,085796	221ms
128×96	gecached	25	0,0023589	209 μ s
	ungecached	28	0,00131888	7838 μ s

Tabelle 5.7: Diese Tabelle quantifiziert den Einfluss des Texturcaches von GPUs auf die Laufzeit beim Raycasting von Dreiecksmengen. Der Einsatz von Pufferspeichern reduziert in diesem Fall sowohl den Registerverbrauch (was jedoch hier keinen Einfluss auf die Occupancy hat) als auch die Ausführungszeit des Kernels `identifyFacet()`. Die Abkürzung I. Throughput steht für Instruction Throughput.

plementierung, bei der jede Leseoperation direkt auf dem GMem stattfindet. Wieder wird der Test für zwei unterschiedliche Auflösungen des Kamerabildes durchgeführt. Sowohl im Fall mit 1280×960 Raycastings, als auch mit einer Auflösung von 128×96 halbiert sich im ungecachten Fall der Instruktion Throughput. Beim hochauflösenden Bild führt das knapp zu einer Vervierfachung der Ausführungszeit. Noch stärker profitiert der niedrig auflösende Fall vom Texturcache. Der Größenunterschied liegt hier bei einem Faktor von etwa 40. Das hängt damit zusammen, dass im zweiten Fall ein größerer Anteil der BVH

in den Cache geladen werden kann.

Vom Texturcache profitieren nur Leseoperationen, denn Texturen können nur gelesen, nicht jedoch geschrieben werden. Jedoch beschleunigen die Caches von Hauptprozessoren ebenfalls nur die Lesezugriffe.

5.4.3 Relevanz des Shared Memory

Überall dort, wo es möglich ist, sollte Shared Memory eingesetzt werden. Er ermöglicht nicht nur den Zugriff aller Threads eines Blocks auf einen gemeinsamen Speicherbereich, sondern reduziert auch die Latenz von Speicherzugriffen. Bankkonflikte zu vermeiden ist wesentlich einfacher als Coalescing zu realisieren.

Wie stark sich der Verzicht des Shared Memory auswirkt, zeigt Tabelle 5.8. Der Kernel zum Hinzufügen von gültigen Randschichten eines Pixpos-Bildes zu einer Maske (`cudaTrim()`) verwendet SMem, da aufgrund der Auswertung der Vierer-Nachbarschaften auch auf Speicherbereiche zugegriffen wird, die nicht der ID des Threads entsprechen, der auf ihnen operiert. Interessant ist die Beobachtung, dass im niedrig auflösenden Fall, mit einer Maskengröße von 128×96 Elementen, die ausschließliche, ungecachte Nutzung des Global Memory deutlich schneller ist als die Verwendung mit Shared Memory. Hier dürfte der reduzierte Registerverbrauch eine Rolle spielen.

Um Shared Memory zu nutzen, können größere Änderungen am Quelltext vonnöten sein.

Vergleich Shared Memory gegen Global Memory				
Auflösung	SMem/GMem	Register	Instruction Throughput	Zeit
1280 × 960	SMem	29	0,67294	1152 μ s
Coalescing: ja/nein. load: 21560/10780, store: 38400/0				
1280 × 960	GMem	21	0,0936583	11687 μ s
Coalescing: ja/nein. load: 24008/1479500, store: 38400/0				
128 × 96	SMem	29	0,324399	20 μ s
Coalescing: ja/nein. load: 106/106, store: 384/0				
128 × 96	GMem	21	0,239447	8 μ s
Coalescing: ja/nein. load: 96/???, store: 384/0				

Tabelle 5.8: Vergleich des Kernels `cudaTrim()` mit und ohne SMem für unterschiedliche Auflösungen der Maske. Aus einem unbekannten Grund konnte der CUDA Visual Profiler im Fall mit 128×96 Maskenelementen die Anzahl der Leseoperationen ohne Coalescing nicht bestimmen.

Zu Beginn und am Ende des Kernels müssen die Speichertransfers zwischen SMem und GMem vollzogen werden. Die Indexierung der Speicherzugriffe innerhalb des Kernels bedarf einer entsprechenden Anpassung. Auch eignet sich nicht jeder Kernel für die Nutzung von SMem, wie beispielsweise `identifyFacet()`. Zur Compilezeit ist noch nicht bekannt, welche Teile der BVH ein Block des Raycastings benötigt. Die komplette BVH passt nur selten komplett in den SMem.

5.4.4 Verwendung schneller mathematischer Operationen

CUDA bietet die Möglichkeit, ausgewählte mathematische Operationen wie trigonometrische Funktionen und Logarithmusfunktionen direkt durch die Rechenwerke auf der GPU auszuführen. Eine Auflistung aller Funktionen, für die das möglich ist, findet sich in [NVi09a]. Nutzt man die nativen Rechenwerke der Grafikkarte, so benötigen die betroffenen Befehle weniger Taktschritte für ihre Ausführung. Diesen Geschwindigkeitszuwachs bezahlt man mit einer reduzierten Genauigkeit der Berechnungen, die nicht mehr dem IEEE 754 Standard [IEE85] entsprechen. Spielt Geschwindigkeit eine größere Rolle als Genauigkeit, so kann diese alternative Verarbeitungsmethodik eingesetzt werden. Andernfalls werden IEEE 754 konforme Berechnungen emuliert, die jedoch länger für ihre Ausführung benötigen.

Das Compiler-Flag `-use_fast_math` dient der Aktivierung des schnelleren Modus. Es bedarf keinerlei Anpassungen im Device-Code. Möchte der Programmierer selektiv entscheiden, an welchen Stellen er welche Variante verwendet, verzichtet er auf das Compiler-Flag `-use_fast_math`. Stattdessen lässt sich jede intrinsische Funktion, die nativ von der Grafikhardware unterstützt wird, durch einen eigenen Funktionsaufruf realisieren. Anstatt des Divisionsoperators `/` muss dann beispielsweise die Funktion `__fdividef()` aufgerufen werden. Die Tangensfunktion `tanf()` wird durch `__tanf()` ersetzt.

Jedoch profitiert kein Kernel des Rekonstruktionsalgorithmus von den schnel-

Vergleich schneller und langsamer mathematischer Operationen				
Auflösung	mit/ohne <code>-use_fast_math</code>	Register	I. Throughput	Zeit
1280 × 960	mit <code>-use_fast_math</code>	15	0,954725	1152 μ s
	ohne <code>-use_fast_math</code>		0,95408	1152 μ s
128 × 96	mit <code>-use_fast_math</code>	15	0,576326	19 μ s
	ohne <code>-use_fast_math</code>		0,576965	19 μ s

Tabelle 5.9: Vergleich der Laufzeit des Kernels `cudaGenPixposImgKrn1()` für Quadriken mit und ohne Verwendung des Compiler-Flags `-use_fast_math`. Für keine der beiden Auflösungen ist eine Verbesserung zu verzeichnen, weder beim Instruction Throughput, noch bei der Ausführungszeit.

leren, nativen Funktionen. Tabelle 5.9 zeigt dies exemplarisch am Kernel `cudaGenPixposImgKrn1()` für Quadriken, der in Unterabschnitt 5.3.3 vorgestellt wird. Er erstellt ein Pixpos-Bild, bei dem eine Quadrik als Modell des zu simulierenden Bauteils verwendet wird. Dieser Kernel führt sehr viele mathematische Operationen aus. Es zeigt sich jedoch, dass er nicht von `-use_fast_math` profitieren kann. Die Begründung findet sich in der Art der mathematischen Operationen, die `cudaGenPixposImgKrn1()` durchführt. Dabei handelt es sich fast ausschließlich um Additionen und Multiplikationen, die aus dem durch `-use_fast_math` aktivierten schnelleren Modus keinen Nutzen ziehen. Nur wenige Divisionen werden verwendet. Kein einziges Mal wird eine trigonometrische Funktion benötigt, weshalb die Berechnungen nicht schneller ausgeführt werden können.

Kapitel 6

Abschließendes Fazit und Ausblick

Zum Abschluss fasst das letzte Kapitel die Erkenntnisse und Erfahrungen dieser Arbeit zusammen. Sie beziehen sich im Kern auf die beiden bearbeiteten Schwerpunkte. Das ist zum einen der Algorithmus für die Positionsrekonstruktion und zum anderen der Einsatz programmierbarer Grafikkarten zur Beschleunigung des entwickelten Verfahrens.

Darüber hinaus wird noch ein kurzer Blick auf die mittelfristige Entwicklung des CUDA-Frameworks geworfen. Neben dem stetigen Fortschritt bei der Hardware, also den Grafikkarten selbst, ist für den Programmierer vor allem die Evolution der Programme zum Entwickeln, Testen und Verbessern von CUDA-Programmen von Interesse.

6.1 Bewertung des Rekonstruktionsalgorithmus

In Abschnitt 2.2 wird die Grundform eines Algorithmus präsentiert, der unter Verwendung von durch Deflektometrie gewonnenen Pixpos-Bildern die Position und Lage eines zu prüfenden Objekts rekonstruieren kann. Dabei handelt es sich im Detail um das Finden von sechs Transformationsparametern $(\phi_1, \phi_2, \phi_3, t_1, t_2, t_3)$, die das zu inspizierende Bauteil aus einer durch einen CAD-Entwurf spezifizierten Ausgangslage in die reale, zu ermittelnde Position überführen. Im Kern simuliert die Positionsbestimmung den deflektometrischen Messaufbau mit Kamera, Bildschirm und dem zu testenden Gegenstand. Das Ideal der Komponente wird durch das STL-Modell als Dreiecksmenge beschrieben, das in Kapitel 4 umfangreich besprochen wird. Erst durch eine Modifikation der Ausgangsform des Rekonstruktionsverfahrens liefert der Algorithmus befriedigende Resultate. Die Abänderung in Abschnitt 2.3 führt dazu, dass gefundene, nicht globale Extrema der Zielfunktion, wieder verlassen werden können. Erst dadurch sind befriedigende Rekonstruktionsraten überhaupt realisierbar.

Abschnitt 5.2 zeigt, dass die Güte des Rekonstruktionsalgorithmus darüber hinaus entscheidend von seiner Konfiguration abhängt. Deshalb ist es unerlässlich, vor dem Produktiveinsatz die Gegebenheiten der deflektometrischen Messung genau zu analysieren, um passende Werte für innere und äußere Iterationen, die Anzahl entfernter gültiger Randschichten und geeignete Startwerte, sowohl für die NLO, als auch für die Größe des Startsimplex zu finden. Weiter gilt es den Trade-Off zwischen niedrigauflösenden, aber schneller zu simulierenden und hochauflösenden, dafür aber aufwändiger zu berechnenden Pixpos-Bildern abzuwägen. Für jeden deflektometrischen Messaufbau und jede Form des zu vermessenden Bauteils variiert die optimale Konfiguration. Die zuvor besprochene Konfiguration aus 5.2 kann als Ausgangspunkt dienen.

Bei der Verwendung der aus den realen Messdaten stammenden Pixpos-Bildern und des STL-Modells treten jedoch zahlreiche unerwünschte Effekte auf, die die Rekonstruktion negativ beeinflussen.

- Dreiecksmengen bestehen aus planaren Primitiven, die gekrümmte Oberflächen nicht exakt wiedergeben können (siehe Unterabschnitt 4.1.2). Eine Erhöhung der Facettenanzahl lässt das Modell zwar runder erscheinen, löst allerdings das zugrundeliegende Problem nicht.
- Die Berechnung von glatten Normalen in Abschnitt 4.6 orientiert sich bisher nur an Verfahren, bei denen Ästhetik und nicht Realismus im Vordergrund steht. Ohne vertiefte Kenntnis der Oberfläche, die über die Informationen, die das STL-Modell liefert, hinausgeht, kann das Reflexionsverhalten der Oberfläche nur unzureichend simuliert werden.
- Um die Simulation so realistisch wie möglich zu gestalten, bedarf es einer exakten Kalibrierung der Kamera und des Monitors.
- Es wird vorausgesetzt, dass die Oberflächenform des real zu testenden Gegenstands der des CAD-Entwurfs entspricht. Vom Ideal darf sich das zu vermessende Objekt theoretisch nur in seiner Position und Lage unterscheiden. In der Praxis ist das freilich nur seltenst der Fall. Der Grund für die Verwendung von Deflektometrie liegt in dem Wunsch, Oberflächenmängel zu detektieren. Könnte man diese von

vornherein ausschließen, so wäre die deflektometrische Messung obsolet. Ungewollte Unzulänglichkeiten der Objektoberfläche führen zu einem Reflexionsverhalten, das in der Simulation bisher nicht berücksichtigt wird.

Welchen Effekt die unzureichende Approximation mit Dreiecken und das Fehlen wirklichkeitsnaher Normalen an beliebigen Punkten der Objektoberfläche hat, illustriert Abbildung 4.14.

Vergleicht man die Abbildungen 2.3 und 2.4, die ein simuliertes und ein reales Pixpos-Bild ein und der selben Geometrie wiedergeben, so lassen sich einfach strukturelle Unterschiede in den Bildern erkennen. Die Simulation zeigt Pixelpositionen, die bis auf numerische Instabilitäten perfekt sind, während das reale Bild unter der Aufsummierung kleinster Fehler bei Kalibrierung, Messprozess und unerwünschter Verformung des Bauteils leidet. All dies führt dazu, dass bei der Rekonstruktion, ausgehend von realen Pixpos-Bildern, nur sehr selten ein Fehler der aufgestellten Zielfunktion 2.2 in der Größenordnung <1 Bildschirmpixel auftritt. Ob es sich nun um das globale oder doch nur ein lokales Extremum handelt, lässt sich nicht einfach mit der Aussage überprüfen, dass eine optimale Rekonstruktion mit einem Fehler nahe bei 0 endet. Aus diesem Grund müsste das zu vermessende Objekt zunächst sowohl deflektometrisch, als auch mit einem zweiten Verfahren erfasst werden, um die vom Algorithmus berechnete Position und Lage zu validieren.

Trotz dieser Unzulänglichkeiten konnte gezeigt werden, dass der präsentierte Algorithmus mit Verwendung deflektometrischer Messdaten zur Positionsbestimmung geeignet ist. Je nach Konfiguration des Algorithmus liegt die erfolgreiche Rekonstruktionsrate bei über 80%, was in erster Linie durch das selektive Ignorieren von Pixelpositionen ermöglicht wird.

6.2 Nutzen der Grafikkartenprogrammierung

Unterschiedliche Firmen und Konsortien ermöglichen es mit Softwareschnittstellen und Entwicklungspaketen, die massiv parallele Architektur von Grafikprozessoren zu nutzen, um beliebige Programme auf ihnen auszuführen. Somit ist es möglich, auch Berechnungen durchzuführen, die grundsätzlich nichts mit der Visualisierung von 3D-Szenen zu tun haben.

Eines dieser Softwarepakete ist das CUDA-Framework des Grafikkartenherstellers NVidia. Es wird in dieser Arbeit verwendet, um die Ausführungszeit des Rekonstruktionsverfahren aus Kapitel 2 zu verkürzen. Eine Präsentation der elementaren Säulen und Komponenten von CUDA findet in Kapitel 3 statt.

Nicht jeder Algorithmus profitiert pauschal, wenn er auf GPUs anstatt auf CPUs ausgeführt wird. Er muss gewisse Anforderungen erfüllen, die in Abschnitt 3.4 aufgeführt sind. Wie sich in Abschnitt 5.3 herausstellt, profitieren alle Einzelschritte der Positionsrekonstruktion von der Grafikkartenbeschleunigung, wenn auch unterschiedlich stark. Nicht nur die Stationen, die sich auf den ersten Blick für SIMD eignen, sondern auch das Raycasting mit Facettenmengen (siehe Unterabschnitt 4.1.3), das durch traversieren eines Pakets von Sichtstrahlen durch eine BVH (siehe Unterabschnitt 4.5.3) vollzogen wird, kann auf GPUs schneller geleistet werden als auf CPUs. Der Unterschied der Laufzeiten des Gesamtalgorithmus fällt auf den beiden Hardwarearchitekturen zwar nicht so deutlich aus, wie das für die Einzelschritte des Verfahrens der Fall ist (siehe Unterabschnitte 5.3.3 bis 5.3.5), jedoch ist er mit dem Faktor 15 immer noch gravierend.

Eine derartige Erfahrung lässt sich allerdings nicht generell auf andere komplexe Berechnungen übertragen. Grafikkarten bleiben ein spezielles Stück Hardware, für die oftmals weitreichende Anpassungen im bereits existierenden Programmcode vorgenommen werden müssen. Auch das CUDA-Modell widerspricht oftmals etablierten Programmiergewohnheiten, wie beispielsweise Unterabschnitt 3.3.4 zeigt. Dabei ist nicht garantiert, dass der Leistungszuwachs durch eine Portierung für die GPU ähnlich deutlich ausfällt, wie das hier der Fall ist.

Man kann sich des Eindrucks nicht erwehren, dass vor allem CUDA einen Hype erlebt, dem es nicht immer gerecht werden kann. Dieser Hype ist nicht zuletzt einer massiven Werbekampagne geschuldet. NVidia ist ein kommerziell orientiertes Unternehmen, das Grafikkarten verkaufen will und das sich mit der CUDA-Technologie Märkte abseits von 3D-Spielen erschließen kann.

Letztendlich kommt man zu der Erkenntnis, dass CUDA (und allgemein programmierbare Grafikkarten) ein mächtiges Werkzeug ist, das in bestimmten Fällen erstaunliches leistet. Investiert man jedoch ebensoviel Know-How und Energie in die Programmentwicklung für klassische Hardware (also den Hauptprozessor), wie das für GPUs notwendig ist, erweist sich diese oftmals als ebenso performant. Dazu zählen beispielsweise die Nutzung von Befehlssatzerweiterungen wie SSE oder die intelligente Verwendung des CPU-Caches und dessen Besonderheiten.

6.3 Ausblick auf die zukünftige Entwicklung des CUDA-Frameworks

Interessant ist ein Blick auf die mittelfristige Zukunft von CUDA.

Sie hängt entscheidend von der Entwicklung der Grafikkhardware selbst ab. Der Trend immer höher getakteter Chips, größerer Speicher und zusätzlicher MPs (und damit Stream-Processors) wird auch zukünftig nicht abreißen. Davon profitiert auch in CUDA geschriebene Software.

Die Evolution von Grafikkhardware ist eng verknüpft mit DirectX. Jede neue Generation der Programmierschnittstelle stellt erhöhte Anforderungen an die GPU. So muss jede DirectX 11 kompatible Grafikkarte Tessellation unterstützen, eine Technologie, mit der sich der Detailgrad von 3D-Modellen implizit erhöhen lässt. Mit Ausnahme der Einführung von Vertex- und Pixel-Shadern hat diese Entwicklung jedoch keinen unmittelbaren Einfluss auf CUDA.

Wenige Tage vor Fertigstellung dieser Arbeit veröffentlichte NVidia im Februar 2010 die neueste Version 3.0 von CUDA [NVi10b]. Sie ist ein gutes Beispiel für die Fortführung der Technologie. Mit jeder weiteren Weiterentwicklung des Frameworks sind ähnliche Erweiterungen zu erwarten.

- Immer mehr Merkmale von C++ werden unterstützt. Waren bisher Features wie Polymorphismus und Funktionentemplates möglich, so führt CUDA 3.0 Klassen ein. Ihre Members werden automatisch im Device Memory gehalten, die Funktionen entsprechen Kernels mit dem Modifikator `__device__`.
- CUDA 3.0 führt mit Version 2.0 eine neue Compute Capability ein. Sie spezifiziert diverse Erweiterungen und Neuerungen für Hard- und Software.
 - Sämtliche Funktionen des SDK unterstützen nun Berechnungen in doppelter Genauigkeit, also von Typ `double`. Folglich müssen keine Einschränkungen der Präzision aufgrund der Verwendung von `float` mehr hingenommen werden.
 - Es steht mehr RMem und SMem zur Verfügung. Jeder MP kann nun auf $2^{15} = 32768$ Register und 48 KB SMem zurückgreifen. Vor allem die Erhöhung der Registerzahl entschärft zahlreiche Probleme.
 - Blöcke und Texturen dürfen nun größer sein. Jeder Block kann nun maximal 1024 anstatt bisher 512 Threads zusammenfassen. Die größtmögliche Auflösung für zweidimensionale Texturen erhöht sich auf 65536×65536 , für dreidimensionale auf $4096 \times 4096 \times 4096$ Elemente. Bisher lag die Beschränkung bei 65536×32768 beziehungsweise $2048 \times 2048 \times 2048$.
- Grundsätzlich bietet CUDA die Möglichkeit, auf unterschiedliche Verarbeitungsstufen von Direct3D und OpenGL zuzugreifen. Die Funktionalität dafür wurde in CUDA 3.0 vereinheitlicht.

Für CUDA-Entwickler spielt die technologische Entwicklung der Hardware selbst nur eine zweitrangige Rolle. Sie wünschen Softwarewerkzeuge, mit denen sie schnell und einfach CUDA-Code schreiben und debuggen können. Vor allem letzteres ist aufgrund der simultanen Ausführung zahlreicher Threads essentiell. Dafür bedarf es mächtiger, bisher

nur spärlich vorhandener Hilfssoftware. Nexus [NVi10a] wird erstmalig einen Debugger, einen Profiler und einen Analyseassistenten für CUDA-Code in die weit verbreitete Entwicklungsumgebung Visual Studio integrieren. Damit soll es möglich sein, zur Laufzeit eines Programms Speicherinhalte auf der Grafikkarte und Zustände einzelner Threads einzusehen. Die im März 2010 noch im Betastadium befindliche Erweiterung nimmt dem Programmierer sehr viel Arbeit ab und verkürzt das ansonsten kaum überschaubare Debugging enorm. Hiermit lassen sich auch komplizierte Algorithmen ähnlich ausführlich testen, wie man es von der Programmentwicklung für den Hauptprozessor her kennt.

Literaturverzeichnis

- [AGCA08] AJMERA, Prekshu ; GORADIA, Rhushabh ; CHANDRAN, Sharat ; ALURU, Srinivas: Fast, parallel, GPU-based Construction of Space Filling Curves and Octrees. In: *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*. New York, NY, USA : ACM, 2008. – ISBN 978–1–59593–983–8
- [Bro70] BROYDEN, C. G.: The Convergence of a Class of Double-rank Minimization Algorithms. In: *Journal of the Institute of Mathematics and its Applications* (1970), Nr. 6, S. 76–90
- [Bro09] BROMBA, Andrew: *Sensitivitätsoptimierung eines Deflektometriesystems*, Universität Passau, Fakultät Informatik und Mathematik, Diplomarbeit, September 2009
- [Cer85] CERNY, V.: A thermodynamical Approach to the Travelling Salesman Problem: An efficient Simulation Algorithm. In: *Journal of Optimization Theory and Applications* (1985), Nr. 45, S. 41–51
- [Don05] DONNER, Klaus: Einführung in die numerische Mathematik / Universität Passau, Fakultät Informatik und Mathematik. 2005. – Vorlesungsskript
- [Don06] DONNER, Klaus: Kamerakalibrierung und Stereorekonstruktion / Universität Passau, Fakultät Informatik und Mathematik. 2006. – Vorlesungsskript
- [Don09] DONNER, Klaus: Sensorfusion und Multisensorik / Universität Passau, Fakultät Informatik und Mathematik. 2009. – Vorlesungsskript
- [DZ93] DEIFT, P. ; ZHOU, X.: A Steepest Descent Method for oscillatory Riemann-Hilbert Problems. Asymptotics for the MKdV equation. In: *Annals of Mathematics* (1993), Nr. 137, S. 295–368
- [Fle70] FLETCHER, R.: A new Approach to variable Metric Algorithms. In: *Computer Journal* (1970), Nr. 13, S. 317–322
- [FP63] FLETCHER, R. ; POWELL, Michael J. D.: A rapidly Convergent Descent Method for Minimization. In: *Computer Journal* (1963), Nr. 6, S. 163–168
- [FR64] FLETCHER, R. ; REEVES, C. M.: Function Minimization by Conjugate Gradients. In: *Computer Journal* (1964), Nr. 7, S. 149–154

- [Fre04] FREITAG, Burkhard: Praktische Informatik 1 / Universität Passau, Fakultät Informatik und Mathematik. 2004. – Vorlesungsskript
- [FS05] FOLEY, Tim ; SUGERMAN, Jeremy: kd-Tree Acceleration Structures for a GPU Raytracer. In: *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. New York, NY, USA : ACM, 2005. – ISBN 1-59593-086-8, S. 15–22
- [GM03] GEIMER, Markus ; MÜLLER, Stefan: A CrossPlatform Framework for interactive Ray Tracing. In: *Proc. of GI Graphiktag*, 2003, S. 25–34
- [Gol70] GOLDFARB, Donald: A Family of variable Metric Updates derived by variational Means. In: *Mathematics of Computation* (1970), Nr. 24, S. 23–26
- [GP98] GHIGLIA, Dennis C. ; PRITT, Mark D.: *Two dimensional phase unwrapping: theory, algorithms & software*. Wiley InterScience, 1998. – ISBN 978-0471249351
- [GPSS07] GÜNTHER, Johannes ; POPOV, Stefan ; SEIDEL, Hans-Peter ; SLUSALLEK, Philipp: Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In: *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, 2007, S. 113–118
- [GS87] GOLDSMITH, Jeffrey ; SALMON, John: Automatic Creation of Object Hierarchies for Ray Tracing. In: *IEEE Comput. Graph. Appl.* 7 (1987), Nr. 5, S. 14–20. – ISSN 0272-1716
- [Hav00] HAVRAN, Vlastimil: *Heuristic Ray Shooting Algorithms*, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Dissertation, November 2000
- [HMS06] HUNT, Warren ; MARK, William R. ; STOLL, Gordon: Fast kd-Tree Construction with an adaptive error-bounded Heuristic. In: *2006 IEEE Symposium on Interactive Ray Tracing*, IEEE, September 2006
- [HSHH07] HORN, Daniel R. ; SUGERMAN, Jeremy ; HOUSTON, Mike ; HANRAHAN, Pat: Interactive kd-Tree GPU Ray Tracing. In: *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*. New York, NY, USA : ACM, 2007. – ISBN 978-1-59593-628-8, S. 167–174
- [HSO07] HARRIS, Mark ; SENGUPTA, Shubhabrata ; OWENS, John D.: Parallel Prefix Sum (Scan) with CUDA. In: NGUYEN, Hubert (Hrsg.): *GPU Gems 3*. Addison Wesley, August 2007
- [IEE85] IEEE: *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*, 1985
- [KGV83] KIRKPATRICK, S. ; GELATT, C. D. ; VECCHI, M. P.: Optimization by Simulated Annealing. In: *Science, New Series* 220 (1983), Mai, Nr. 4598, S. 671–680

- [Kic06] KICKINGEREDER, Reiner: *Optische Vermessung partiell reflektierender Oberflächen*, Universität Passau, Fakultät Informatik und Mathematik, Dissertation, November 2006
- [LGS⁺09] LAUTERBACH, Christian ; GARLAND, Michael ; SENGUPTA, Shubhabrata ; LUEBKE, David ; MANOCHA, Dinesh: Fast BVH Construction on GPUs. In: *Comput. Graph. Forum* 28 (2009), Nr. 2, S. 375–384
- [Max99] MAX, Nelson: Weights for Computing Vertex Normals from Facet Normals. In: *J. Graph. Tools* 4 (1999), Nr. 2, S. 1–6. – ISSN 1086–7651
- [MB90] MACDONALD, David J. ; BOOTH, Kellogg S.: Heuristics for Ray Tracing using Space Subdivision. In: *Vis. Comput.* 6 (1990), Nr. 3, S. 153–166. – ISSN 0178–2789
- [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
- [NM65] NELDER, J. A. ; MEAD, R.: A Simplex Method for Function Minimization. In: *Computer Journal* (1965), Nr. 7, S. 308–313
- [NVi08] NVIDIA CORPORATION (Hrsg.): *The CUDA Compiler Driver NVCC*. 2.0. http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/2.0/nvcc_2.0.pdf: NVidia Corporation, April 2008
- [NVi09a] NVIDIA CORPORATION (Hrsg.): *CUDA Programming Guide*. 2.3.1. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf: NVidia Corporation, August 2009
- [NVi09b] NVIDIA CORPORATION (Hrsg.): *CUDA Reference Manual*. 2.3.1. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/CUDA_Reference_Manual_2.3.pdf: NVidia Corporation, Juli 2009
- [NVi09c] NVIDIA CORPORATION (Hrsg.): *NVidia CUDA C Programming Best Practices Guide*. 2.3. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf: NVidia Corporation, Juli 2009
- [NVi10a] NVIDIA, Corporation: *NVidia Parallel Nsight („Nexus“)*. <http://developer.nvidia.com/object/nexus.html>, 2010
- [NVi10b] NVIDIA CORPORATION (Hrsg.): *CUDA Programming Guide*. 3.0. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf: NVidia Corporation, Februar 2010

- [Owe99] OWEN, Scott G.: *3-Dimensional Modeling Transformations*. http://www.siggraph.org/education/materials/HyperGraph/modeling/mod_tran/3d.htm, Juni 1999
- [PBMH05] PURCELL, Timothy J. ; BUCK, Ian ; MARK, William R. ; HANRAHAN, Pat: Ray Tracing on programmable Graphics Hardware. In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*. New York, NY, USA : ACM, 2005, S. 268
- [PGSS06] POPOV, Stefan ; GÜNTHER, Johannes ; SEIDEL, Hans-Peter ; SLUSALLEK, Philipp: Experiences with Streaming Construction of SAH kd-Trees. In: *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006, S. 89–94
- [PGSS07] POPOV, Stefan ; GÜNTHER, Johannes ; SEIDEL, Hans-Peter ; SLUSALLEK, Philipp: Stackless kd-Tree Traversal for High Performance GPU Ray Tracing. In: *Computer Graphics Forum* 26 (2007), September, Nr. 3, S. 415–424
- [Pis08] PISINGER, Georg: Computergrafik / Universität Passau, Fakultät Informatik und Mathematik. 2008. – Vorlesungsskript
- [Pod07] PODLOZHNYUK, Victor: *Parallel Mersenne Twister*. <http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MersenneTwister/doc/MersenneTwister.pdf>, 2007
- [RM09] RUETSCH, Greg ; MICIKEVICIUS, Paulius: *Optimizing Matrix Transpose in CUDA*. <http://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf>, 2009
- [RPR87] RÜGER, W. ; PIETSCHNER, J. ; REGENSBURGER, K.: *Photogrammetrie*. 5. Berlin : Verlag für Bauwesen, 1987. – ISBN 3–345–00196–9
- [Sch04a] SCHUNDLER, Ted: *The Problem of Calculating Vertex Normals for unevenly subdivided smooth Surfaces*. <http://epii.info/oss/blender/smooth/Calculating%20smooth%20vertex%20normals.pdf>, Dezember 2004
- [Sch04b] SCHWARZ, Oliver: *Hybrid Phase Unwrapping in Laser Speckle Interferometry with Overlapping Windows*, Universität Passau, Fakultät Informatik und Mathematik, Dissertation, Februar 2004
- [Sha70] SHANNO, D. F.: Conditioning of Quasi-Newton Methods for Function Minimization. In: *Mathematics of Computation* (1970), Nr. 24, S. 647–656
- [SHG09] SATISH, Nadathur ; HARRIS, Mark ; GARLAND, Michael: Designing efficient Sorting Algorithms for Manycore GPUs. In: *Parallel and Distributed Processing Symposium, International 0* (2009), S. 1–10. ISBN 978–1–4244–3751–1
- [Sla80] SLAMA, Chester C.: *Manual of Photogrammetry*. 5. Virginia : American Society of Photogrammetry & Remote Sensing, 1980. – ISBN 978–0937294017

- [Spe93] SPELLUCCI, Peter: *Numerische Verfahren der nichtlinearen Optimierung*. Basel : Birkhäuser, 1993. – ISBN 3–7643–2854–1
- [SSK07] SHEVTSOV, M. ; SOUPIKOV, A. ; KAPUSTIN, A.: Ray-Triangle Intersection Algorithm for modern CPU Architectures. In: *Proceedings of GraphiCon 2007*, 2007, S. 33–39
- [Sto09] STONE, Thomas C.: *Effiziente Bildverarbeitungsalgorithmen in CUDA*, Universität Passau, Fakultät Informatik und Mathematik, Bachelorarbeit, September 2009
- [TS05] THRANE, Niels ; SIMONSEN, Lars O.: *A Comparison of Acceleration Structures for GPU assisted Ray Tracing*, Masterarbeit, 2005
- [TW98] THÜRMER, Grit ; WÜTHRICH, Charles A.: Computing Vertex Normals from Polygonal Facets. In: *journal of graphics, gpu, and game tools* 3 (1998), Nr. 1, S. 43–46
- [WBS07] WALD, Ingo ; BOULOS, Solomon ; SHIRLEY, Peter: Ray Tracing deformable Scenes using dynamic Bounding Volume Hierarchies. In: *ACM Trans. Graph.* 26 (2007), Nr. 1, S. 6. – ISSN 0730–0301
- [Wik09] WIKIPEDIA: *STL-Schnittstelle* — *Wikipedia, Die freie Enzyklopädie*. <http://de.wikipedia.org/w/index.php?title=STL-Schnittstelle&oldid=59850473>, 2009. – [Online; Stand 12. März 2010]
- [Wik10a] WIKIPEDIA: *Baryzentrische Koordinaten* — *Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?title=Baryzentrische_Koordinaten&oldid=70088422, 2010. – [Online; Stand 18. März 2010]
- [Wik10b] WIKIPEDIA: *Eulersche Winkel* — *Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?title=Eulersche_Winkel&oldid=70413475, 2010. – [Online; Stand 12. März 2010]
- [Wik10c] WIKIPEDIA: *Grafikpipeline* — *Wikipedia, Die freie Enzyklopädie*. <http://de.wikipedia.org/w/index.php?title=Grafikpipeline&oldid=69982815>, 2010. – [Online; Stand 3. März 2010]
- [Wik10d] WIKIPEDIA: *Taylorreihe* — *Wikipedia, Die freie Enzyklopädie*. <http://de.wikipedia.org/w/index.php?title=Taylorreihe&oldid=71877811>, 2010. – [Online; Stand 18. März 2010]
- [WMG⁺07] WALD, Ingo ; MARK, William R. ; GÜNTHER, Johannes ; BOULOS, Solomon ; IZE, Thiago ; HUNT, Warren ; PARKER, Steven G. ; SHIRLEY, Peter: State of the Art in Ray Tracing animated Scenes. In: *STAR Proceedings of Eurographics 2007*. Prague, Czech Republic : Eurographics Association, September 2007. – ISSN 1017–4656, S. 89–116

- [ZHWG08] ZHOU, Kun ; HOU, Qiming ; WANG, Rui ; GUO, Baining: Realtime kd-Tree Construction on Graphics Hardware. In: *ACM Trans. Graph.* 27 (2008), Nr. 5, S. 126

Abbildungsverzeichnis

1.1	Abbildung zur Illustration von Rotation und Translation im $\mathbb{R}^3$	13
2.1	Schematischer Aufbau eines Deflektometriesystems	20
2.2	Realer Aufbau eines Deflektometriesystems	20
2.3	Schnittpunkte simulierter Sichtstrahlen mit dem virtuellen Bildschirm . . .	22
2.4	Schnittpunkte realer Sichtstrahlen mit dem realen Bildschirm	22
2.5	Vektordifferenz zweier Pixposbilder	25
2.6	Minimierung von Vektordifferenzen	27
2.7	Ähnlichere Pixpos-Bilder müssen nicht ähnliche Transformationsparame- ter bedeuten	28
2.8	Entstehung gemeinsamer Pixelpositionen	30
2.9	Utah-Teekanne	31
2.10	Beispiel einer Maske mit Wegnahme von Randbildpunkten	32
2.11	Reale Maske mit unterschiedlicher Zahl weggenommener Randbildpunkt- schichten	34
2.12	Abbildung zur Illustration der nicht vorhandenen Kommutativität bei Transformationen	35
3.1	Schema einer Rendering-Pipeline	41
3.2	Block- und Gridhierarchie	46
3.3	Entscheidungshierarchie für Grafikkartennutzung bezüglich Algorithmus- charakteristika	62
3.4	Entscheidungshierarchie für Grafikkartennutzung bezüglich Programmablauf	64
3.5	Entscheidungshierarchie für Grafikkartennutzung bezüglich Speicherver- halten	65
4.1	Beispiel für eine Dreiecksmenge	69
4.2	Illustration eines Netztes von zusammenhängenden Dreiecken	71
4.3	Funktionsweise eines Oc- beziehungsweise Quadrees	77
4.4	Funktionsweise eines kd-Trees	79
4.5	Funktionsweise einer BVH	80
4.6	AABBs unterschiedlicher Hierarchiestufen einer BVH	82
4.7	Einfache Heuristik für kleine AABBs	84
4.8	Linearisierung eines Baumes durch Preorder in ein Array	86
4.9	Vergleich nicht geglättete Oberfläche mit geglätteter Oberfläche	94
4.10	Glättungsheuristiken unter Berücksichtigung der gesamten Facette 1	96
4.11	Glättungsheuristiken unter Berücksichtigung der gesamten Facette 2	96

4.12	Glättungsheuristiken unter Berücksichtigung des Vertex 1	97
4.13	Glättungsheuristiken unter Berücksichtigung des Vertex 2	98
4.14	Approximation gilt nur für Normale, nicht für Oberfläche	99
4.15	Fälschliche Glättung gewünscht harter Kanten	100
5.1	Rekonstruktionsrate in Abhängigkeit von der Anzahl äußerer Iterationen .	105
5.2	Rekonstruktionsrate in Abhängigkeit von der Anzahl innerer Iterationen .	106
5.3	Rekonstruktionsrate in Abhängigkeit von der Anzahl weggenommener Mas- kenrandschichten	107
5.4	Rekonstruktionsrate in Abhängigkeit von der Pixpos-Bild-Auflösung	107
5.5	Rekonstruktionsrate in Abhängigkeit von der Startsimplexgröße	109
5.6	Leistungsvergleich Sichtstrahlgenerierung	111
5.7	Leistungsvergleich Pixpos-Bilder	112
5.8	Leistungsvergleich Maske	114
5.9	Leistungsvergleich Pixelpositionsabstände	115

Tabellenverzeichnis

3.1	CUDA-Modifikatoren für Funktionen	45
3.2	Automatisch vorhandene Werte zur Laufzeit in CUDA	47
3.3	Auflistung der unterschiedlichen Speicherarten mit ihren Eigenschaften. . .	51
3.4	Instruction Throughput für ausgewählte GPU-Anweisungen	54
3.5	Speichertransferzeiten zwischen Hauptarbeitsspeicher und Device Memory	60
4.1	Binärinterpretation von <code>float</code> -Zahlen als <code>unsigned int</code>	74
5.1	Spezifikation der Grafikkarte/CPU	102
5.2	Spezifikation der restlichen Hardware	103
5.3	Grafikkartenspezifische Software	103
5.4	Sonstige Software	103
5.5	Konfiguration der Kernel	116
5.6	Messergebnisse der Kernel	117
5.7	Vergleich gecachter/ungecachter Speicherzugriff	118
5.8	Vergleich Shared Memory oder Global Memory	119
5.9	Vergleich schneller und langsamer mathematischer Operationen	120

Listings

3.1	Matrixaddition als Minimalbeispiel eines Kernel.	44
3.2	Aufruf eines Kernel.	46
3.3	Speichertransfers zwischen Hauptarbeitsspeicher und Device Memory. . . .	48
3.4	Sinnfreies Minimalbeispiel für einen Kernel, der zu Warp-Divergence führt.	52
3.5	Vermeidung von Warp-Divergence durch arithmetischen Ersatz.	58
3.6	Separates Setzen eines Gültigkeitsindikators.	59
4.1	Aufbau des ASCII-STL-Formats.	69
4.2	Code einer Hashfunktion für Vertices.	73
4.3	Pseudocode Preorder.	85
4.4	Berechnung der beiden Schnittpunkte eines Sichtstrahls mit einem Quader.	87
4.5	Berechnung des Schnittpunkts eines Sichtstrahls mit einem Dreieck im Raum.	88
4.6	Pseudocode für das parallele, paketbasierte Durchlaufen einer BVH.	92
5.1	Hilfskernel für quadratisch gewichtete Vektordifferenz.	114

Abkürzungen

AABB	<u>A</u> xis <u>A</u> ligned <u>B</u> ounding <u>B</u> ox
API	<u>A</u> pplication <u>P</u> rogramming <u>I</u> nterface
BB	<u>B</u> ounding <u>B</u> ox
BFGS	<u>B</u> royden <u>F</u> letcher <u>G</u> oldfarb <u>S</u> hanno-Methode
BVH	<u>B</u> ounding <u>V</u> olume <u>H</u> ierarchy
CAD	<u>C</u> omputer <u>A</u> ided <u>D</u> esign
CAD-KoSy	<u>C</u> AD- <u>K</u> oordinatensystem
CMem	<u>C</u> onstants <u>M</u> emory
CPU	<u>C</u> entral <u>P</u> rocessing <u>U</u> nit
CUDA	<u>C</u> ompute <u>U</u> nified <u>D</u> evice <u>A</u> rchitecture
DFS	<u>D</u> epth <u>F</u> irst <u>S</u> earch
DLL	<u>D</u> ynamic <u>L</u> ink <u>L</u> ibrary
GB	<u>G</u> igabyte (2^{30} Bytes)
GHz	<u>G</u> igahertz (10^9 Hertz)
GMem	<u>G</u> lobal <u>M</u> emory
GPGPU	<u>G</u> eneral <u>P</u> urpose Computation on <u>G</u> raphics <u>P</u> rocessing <u>U</u> nit
GPU	<u>G</u> raphics <u>P</u> rocessing <u>U</u> nit
HLSL	<u>H</u> igh <u>L</u> evel <u>S</u> hading <u>L</u> anguage
ID	<u>I</u> dentifikationsnummer
K-KoSy	<u>K</u> amera- <u>K</u> oordinatensystem
KB	<u>K</u> ilobyte
kd-Tree	<u>k</u> - <u>d</u> imensional <u>T</u> ree
LMem	<u>L</u> ocal <u>M</u> emory
μ s	<u>M</u> ikrosekunde (10^{-6} Sekunden)
MB	<u>M</u> egabyte (2^{20} Bytes)
MHz	<u>M</u> egahertz (10^6 Hertz)
MIMD	<u>M</u> ultiple <u>I</u> nstruction <u>M</u> ultiple <u>D</u> ata
mm	<u>M</u> illimeter (10^{-3} Meter)
MP	<u>M</u> ultiprocessor
ms	<u>M</u> illisekunde (10^{-3} Sekunden)
NCG	<u>N</u> onlinear <u>C</u> onjugate <u>G</u> radient-Methode
NLO	<u>N</u> ichtlineare <u>O</u> ptimierung
NURBS	<u>N</u> on <u>U</u> niform <u>R</u> ational <u>B</u> - <u>S</u> plines
PCIe	<u>P</u> eripheral <u>C</u> omponent <u>I</u> nterconnect <u>E</u> xpress
Pixpos	<u>P</u> ixelposition
PTX	<u>P</u> arallel <u>T</u> hread <u>E</u> xecution

RAM	<u>R</u> andom <u>A</u> ccess <u>M</u> emory
RMem	<u>R</u> egister <u>M</u> emory
S-KoSy	<u>S</u> creen- <u>K</u> oordinatensystem
SAH	<u>S</u> urface <u>A</u> rea <u>H</u> euristic
SDK	<u>S</u> oftware <u>D</u> evelopment <u>K</u> it
SIMD	<u>S</u> ingle <u>I</u> nstruction <u>M</u> ultiple <u>D</u> ata
SMem	<u>S</u> hared <u>M</u> emory
SSE	<u>S</u> teaming <u>S</u> IMD <u>E</u> xtensions
STL	<u>S</u> urface <u>T</u> essellation <u>L</u> anguage
TMem	<u>T</u> exture <u>M</u> emory
W-KoSy	<u>W</u> elt- <u>K</u> oordinatensystem

Danksagung

An erster Stelle möchte ich mich bei meinem Erstkorrektor Prof. Klaus Donner bedanken, der mein Interesse für numerische Mathematik und vor allem die Bildverarbeitung geweckt hat. Er prägte in den vergangenen Jahren entscheidend meine Entwicklung als Informatiker. Darüber hinaus gilt mein Dank Prof. Paul Lukowicz für die Zweitkorrektur meiner Diplomarbeit.

Vorbildliche Unterstützung erhielt ich während meiner gesamten Arbeit von der FORWISS Passau. Allen voran möchte ich meinem Betreuer Dr. Alexander Zimmermann danken, der mir jederzeit interessiert, schnell und zuverlässig zur Seite stand. Einen besonderen Dank verdient Manfred Kuhnkie. Er stand mir oftmals und mit sehr viel Enthusiasmus zur Seite, vor allem bei Fragen zu Dreiecksmengen. Ebenfalls möchte ich mich bei Dr. Oleg Smirnow erwähnen, dem ich die entscheidenden Inspirationen bei der Entwicklung des Rekonstruktionsalgorithmus verdanke.

Zuletzt gilt mein Dank meinem Vater Karl Riesinger für die orthographische Korrektur dieser Arbeit.

Eidesstattliche Erklärung

Hiermit erkläre ich eidesstattlich, dass ich diese Diplomarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe, und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind. Diese Diplomarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

(Christoph Riesinger)