UNIVERSITÄT SIEGEN

Computergraphik und Multimediasysteme CG

Interaktive Rekonstruktion von Time-of-Flight Oberflächen

Interactive Reconstruction of Time-of-Flight Surfaces

Diplomarbeit im Fach Informatik

vorgelegt von

Markus Barth

geboren am 03. Dezember 1982 in Birkenfeld/Nahe

Angefertigt am

Lehrstuhl für Computergraphik und Multimediasysteme
Fachbereich 12
Universität Siegen

Betreuer:

Prof. Dr. A. Kolb, Lehrstuhl Computergraphik und Multimediasysteme
Dr. C. Rezk-Salama, Lehrstuhl Computergraphik und Multimediasysteme

Beginn der Arbeit: 01.07.2008
Abgabe der Arbeit: 20.12.2008

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Zusammenfassung

Mittels der in Siegen entwickelten PMD-Kamera lassen sich sogenannte Tiefenprofile einer Szene erstellen. Das heißt, es wird ein Bild ähnlich dem Bild einer 2D-Kamera aufgenommen, wobei zu jedem einzelnen Punkt zusätzliche Tiefeninformationen gemessen werden. Das Resultat einer solchen Aufnahme ist eine Punktwolke im Raum, ohne topologische Informationen zwischen den einzelnen Punkten. Die PMD-Kamera besitzt eine geringe Auflösung, so dass nur relativ wenige Punkte zur Wiedergabe der Szene zur Verfügung stehen. Ein weiteres Problem sind sogenannte *Ausreißer*, welche detektiert und entfernt werden müssen, um eine sinnvolle Oberflächenbeschreibung zu erhalten.
Die vorliegende Arbeit untersucht, ob die Berechnung von *Weighted Least Squares* (WLS) Oberflächen eine brauchbare Approximation der Oberfläche liefern. Zur Erhöhung der Auflösung werden weitere Punkte mittels eines punkbasierten Raycasting-Verfahrens zu der Punktwolke hinzugefügt. Um die Berechnung möglichst effizient zu realisieren, wird die Berechnung auf der GPU ausgeführt.

Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, die durch ihre fachliche und persönliche Unterstützung zum Gelingen dieser Diplomarbeit beigetragen haben. An erster Stelle möchte ich mich bei meinen Betreuern Prof. Dr. Andreas Kolb und Dr. Christof Rezk-Salama der Fachgruppe Computergraphik und Multimediasysteme der Universität Siegen für das Angebot eines interessanten Themas und die Betreuung der Diplomarbeit bedanken. Ganz besonders bedanke ich mich bei Dipl.-Inf. Marvin Lindner, der zu jeder Zeit bereit war, mir hilfreiche Ratschläge zu geben, welche zur Verbesserung der Arbeit beigetragen haben. Nicht zuletzt gilt mein Dank auch meinen Freunden und meiner Familie, deren Beistand mir während der Durchführung des Projektes viel bedeutet hat.

Markus Barth, Dezember 2008

Kapitel 1.

Einleitung

Die Entwicklung neuer Bilderfassungssysteme hat in den letzten Jahren rasant zugenommen. Die meisten solcher Systeme dienen lediglich der Erfassung von Helligkeitsinformationen und liefern somit ein zweidimensionales Bild. Doch wie sich an den Entwicklungen der letzten Jahre gezeigt hat, ist die Industrie zunehmend an der dritten Dimension interessiert. Mit 3D-Scannern ist es inzwischen möglich, dreidimensionale Szenen aufzunehmen. Allerdings gilt dies, aufgrund des zeitaufwendigen Scanvorganges, nur für statische Szenen und ist mit großem technischen und damit auch finanziellem Aufwand verbunden. Ein weiterer Nachteil solcher Scanner liegt in der Beschaffenheit der Abtastvorrichtung. Diese ist meist fest montiert und damit an einen Standort gebunden. Selbst eine leichte Verschiebung hat eine aufwändige Rekalibrierung der Apparatur zur Folge. Des Weiteren existieren bereits sogenannte Handscanner, welche nicht fest montiert sind. Diese haben den Nachteil, dass Referenzpunkte (sogenannte *Marker*) auf dem aufzunehmenden Objekt angebracht werden müssen, um die Daten von unterschiedlichen Aufnahme-Positionen aneinander fügen zu können.
2005 wurde das erste kommerzielle Produkt basierend auf *Photonic Mixer Device* (PMD)-Technologie vorgestellt. Eine solche Kamera ist relativ einfach aufgebaut und ermöglicht es, per Laufzeitmessung die Tiefeninformationen einer Szene parallel in Echtzeit aufzunehmen. Das Interessante an dem PMD-Sensor ist, dass sowohl die Detektion des Lichtsignals, als auch dessen Verarbeitung, zusammen auf einem Chip integriert ist. Durch die auf CMOS-Technologie basierte Konstruktion, sind die Anschaffungskosten für eine solche Kamera relativ gering, was sie für Industrie und Forschung interessant macht. Ein großer Nachteil besteht in der z.Z. geringen Auflösung der Kamera von maximal 160 (H) $\times$ 120 (V). Des weiteren unterliegen die Daten, die per Laufzeitmessung aufgenommen werden, Ungenauigkeiten bzgl. der Entfernungsinformationen der einzelnen Pixel. Diese machen sich zum einen in Form von *flying pixels* und zum anderen als leichte Abweichungen der tatsächlichen Pixelpositionen (*Rauschen*) bemerkbar.
Das Ziel der vorliegenden Arbeit besteht darin, die aufgenommene Szene, aus der dünn besetzten und verrauschten Punktwolke der PMD-Kamera, möglichst gut zu rekonstruieren. Dazu müssen die Nachteile der PMD-Kamera softwareseitig gelöst werden. D.h. es wird eine Verfeinerung der Daten, sowie ein interaktives Einfügen von sichtbaren

Punkten (auf nicht lineare Weise) in Echtzeit angestrebt. Die Verfeinerung wird durch eine Glättung, sowie die Reduktion von Ausreißern umgesetzt. Bei einfachen Geometrien, wäre es möglich eine globale Approximation zu bestimmen, allerdings ist dies im allgemeinen Fall aufgrund der Komplexität der Daten nicht möglich. Daher werden Approximationen für lokale Bereiche verwendet, durch deren Kombination die gesamte Oberflächenapproximation bestimmt werden kann. Mit dem Ziel einer möglichst glatten und gut approximierten Oberfläche wird ein *Weighted Least Squares* (WLS) Verfahren auf die resultierende Punktwolke der PMD-Kamera angewendet. Um die lokale Approximation zu bestimmen, wird ein lokales Referenzsystem sowie eine Mapping-Funktion berechnet. Die resultierende Oberflächenapproximation hängt nur von der Punktwolke und nicht von einem Mesh (Polygonnetz) ab. Außerdem müssen die Daten nicht uniform abgetastet sein, damit das Verfahren funktioniert, wodurch es sich gut eignet, um auf den Tiefendaten der PMD-Kamera zu arbeiten. Das Hauptziel der vorliegenden Arbeit ist die Darstellung der Oberfläche mittels einem punktbasierten Raycasting Verfahrens. Dabei soll auf aufwendige Vorverarbeitungsschritte verzichtet werden. Mit Hilfe des Raycastings in Kombination mit WLS kann die Auflösung erhöht werden, wodurch dem Nachteil der geringen Auflösung der PMD-Kamera entgegengewirkt wird. Um eine möglichst effiziente Berechnung gewährleisten zu können, werden die Berechnungen auf der GPU ausgeführt und somit die Vorteile der Graphikhardware ausgenutzt.
Zu Beginn werden einige Arbeiten angesprochen, die sich mit ähnlichen Problemstellungen wie die vorliegenden Arbeit beschäftigen. Im zweiten Kapitel wird dem Leser zunächst ein kurzer Überblick über ausgewählte Bilderfassungssystemen gegeben, bevor auf die verwendete PMD-Kamera eingegangen wird. Kapitel 4 gibt einen kurzen Einblick in die Graphikpipeline, sowie die GPU-Programmierung, gefolgt von einem Überblick über punktbasierte Renderverfahren. In Kapitel 6 werden dem Leser die wichtigsten mathematischen und computergraphischen Grundlagen vermittelt, um die vorliegende Arbeit nachvollziehen zu können. Im Hauptteil der Arbeit werden die Verfahren beschrieben, welche das Problem der geringen Auflösung der PMD-Kamera lösen. In Kapitel 7 wird das Verfahren erläutert, welches verwendet wird, um aus einer Punktwolke eine Approximation der Oberfläche zu gewinnen. Das folgende Kapitel baut darauf auf und erweitert den Algorithmus zu einem punktbasierten Raycasting Verfahren, wodurch zusätzliche Punkte erzeugt werden, um die Oberfläche zu verfeinern. Im Schlussteil der Arbeit wird auf Implementierungsdetails eingegangen, die Ergebnisse der Arbeit ausgewertet und diskutiert, sowie ein kurzer Überblick für zukünftige Ziele gegeben.

Kapitel 2.

Verwandte Arbeiten

Verfahren, die Entfernungsbestimmung per Laufzeitmessung realisieren, sind zunehmend attraktiver geworden. Die Visualisierung von *Time-of-Flight* (ToF)-Daten kann entweder einfach realisiert werden mit Hilfe von Punkten bzw. Quads, oder komplexer durch Triangulierung bzw. *Point Based Rendering* (PBR) (siehe Kapitel 5). Probleme, die mit dieser Art der Messung einhergehen sind u.a. sogenannte Ausreißer (*outlier*), sowie ein relativ starkes Rauschen der Daten, welches durch Messungenauigkeit verursacht wird. Die Arbeit von Huhle und Schairer et al. [HSJS08] stellen einen GPU-basierten Algorithmus vor, welcher beide Probleme löst. Das Verfahren basiert auf dem *Non-Local Means* (NL-Means) Filter (eine Erweiterung des Bilateralfilters) und führt ein zusätzliches Gewichtungsschema ein, welches sowohl Farbwerte, als auch *intra-patch* Ähnlichkeit (Ähnlichkeit von Pixeln innerhalb eines Suchfensters) berücksichtigt. Das Verfahren arbeitet auf Punktdatensätzen und liefert eine rauschreduzierte Punktwolke als Ergebnis. Lindner et al. [LLK08] kombinieren die PMD-Kamera mit einer RGB-Kamera, um unter anderem die niedrig aufgelösten *range images* zu verfeinern und damit die Rekonstruktion von Geometrien zu verbessern. Dazu muss zunächst eine Detektion ungültiger Pixel stattfinden, um fehlende Daten mit Hilfe von gültigen Pixeln ersetzen zu können. Um die zur Verfügung stehenden PMD-Daten in höherer Auflösung neu abzutasten, wird ein gradientenbasierter Ansatz verwendet. Huhle et al. [HF07] verwenden zusätzlich zu niedrig aufgelösten Intensitätsdaten und niedrig aufgelösten Tiefeninformationen eine hochaufgelöste Farbtextur, um die räumliche Auflösung der PMD-Daten zu erhöhen und das Rauschen zu verringern. Unter der Ausnutzung der Eigenschaft, dass Farbinformationen von den Tiefeninformationen abhängen, wird ein lernbasierter *Markov Random Field* (MRF) Ansatz verwendet. Yang et al. [YYDN07] verwenden ein iteratives Verfahren, um die Auflösung von niedrig aufgelösten *range images* in einem Nachverarbeitungsschritt zu erhöhen. Zunächst wird ein *cost volume* ausgehend von einer initialen Tiefenkarte aufgebaut. Aus den Annahmen, dass die Oberfläche stückweise glatt ist und Pixel gleicher Farbe eines Bereiches auch eine ähnliche Tiefe besitzen, wird durch die Anwendung eines Bilateralfilters auf jede Scheibe des Volumens ein neues *cost volume* aufgebaut. Aus dem neuen Volumen wird eine verfeinerte Tiefenkarte erstellt. Durch die iterative Anwendung dieser Schritte wird eine Verbesserung der räumlichen Auflösung von *range images* um

den Faktor 100 erreicht. Böhme et al. [BHMB08] nutzen ebenfalls die Tatsache aus, dass *range images* und Intensitätsbild der ToF-Daten nicht unabhängig voneinander sind, um die Genauigkeit von PMD-Messungen zu eröhen. Sind die Reflektionseigenschaften der aufgenommenen Oberfläche bekannt, so kann daraus das Intensitätsbild abgeleitet werden (*shape from shading*). Mit einem neuen PBR-Verfahren von Alexa et al. [ABCO+01] ist es möglich, ohne Vorverarbeitungsschritte ggf. neue benötigte Abtastpunkte zu generieren, um eine dichtere Punktwolke zu erzielen. Das Ergebnis dieser Methode ist ein sogenanntes *Point Set Surface* (PSS), welches für das Raytracing entscheidende Vorteile mit sich bringt. Dafür wird auf Basis der Ausgangspunktwolke eine glatte Oberfläche generiert, die möglichst nahe an der Originaloberfläche liegt. Um den Fehler so gering wie möglich zu halten wird die *Moving Least Squares* (MLS) Methode verwendet. Aufbauend auf [ABCO+01] haben Ertl et al. [TGN+01] 2006 eine Methode veröffentlicht, welche auf der Graphik-Hardware arbeitet und PSS effizient umsetzt.
Die Idee, Punkte als Renderprimitive zu benutzen, anstatt der traditionellen Dreiecksnetze, wird zunehmend populärer. Vor allem, wenn die Größe der Dreiecke kleiner wird, als ein Pixel, ist die Darstellung des entsprechenden Objekts als Punktwolke effizienter. Es muss weder eine Topologie gespeichert werden, noch muss eine aufwendige Vorverarbeitung in Form einer Triangulierung stattfinden. Ein Problem bei der Verwendung von Punkten als Renderprimitiven besteht darin, dass ab einer bestimmten Skalierung keine geschlossenen Oberflächen mehr dargestellt werden können. Dieses Problem liegt bei PMD-Datensätzen aufgrund der geringen Auflösung in verstärkter Form vor. Ein Ansatz zur Lösung dieses Problems, der unter den Begriff *point splatting* fällt, stellen Ren, Pfister und Zwicker in [RPZ02] vor. Das *object space EWA surface splatting* nimmt einen hohen Berechnungsaufwand in Kauf, um eine möglichst gute visuelle Darstellung zu erreichen, indem sichtbare Lücken in der Oberflächenrepräsentation vermieden werden. Das Verfahren besteht aus zwei Teilschritten: Zunächst wird für jeden Punk des Datensatzes ein Polygon gerendert und ein *object space EWA resampling Filter* generiert. Der Filter wird zur Kodierung einer Gaußfunktion verwendet, mit dessen Hilfe die Werte der Punkte gewichtet und damit die späteren Pixelwerte berechnet werden können. Auf verschiedene punktbasierten Renderverfahren wird in Kapitel 5 näher eingegangen.
Ein Ziel der vorliegenden Arbeit ist es, zu untersuchen, ob das Verfahren von Ertl et al. [TGN+01] auch geeignet ist, um PMD-Daten zu verfeinern. Ein wesentlicher Unterschied, den es zu beachten gilt, ist der, dass Ertl et al. statische Objekte verwenden und damit Informationen vorberechnen können. Das hier verwendete Verfahren nutzt einen Videostream, der kontinuierlich Daten liefert, so dass alle Berechnungen zur Laufzeit durchgeführt werden. Um die Informationen trotzdem effizient gewinnen zu können, wird versucht die Eigenschaft der PMD-Kamera auszunutzen, dass PMD-Punkte auf die Bildebene projiziert werden.

Kapitel 3.

Bilderfassungssysteme

Die möglichst genaue und schnelle Erfassung von dreidimensionalen Daten ist in vielen Bereichen inzwischen äußerst wichtig geworden. Z.B. können Sicherheitssysteme in Automobilen durch die Erfassung von Tiefeninformationen deutlich verbessert werden. Im Innenraum des Wagens kann z.B. das Auslösen des Airbags durch die räumliche Detektion der Fahrer- bzw. Beifahrerposition positiv beeinflusst werden. Aber auch Systeme, die auf eine Unfallvermeidung abzielen, sind auf die räumliche Erfassung der Umgebung angewiesen. Ein weiteres Anwendungsgebiet stellt die Medizintechnik dar. Bereits heute sind bei der Planung, Simulation und Durchführung von Operationen markerbasierte Systeme zur Positionserfassung im Einsatz. Es wird eine Entwicklung zu markerlosen Systemen und zu Systemen zur echtzeitnahen Objekterfassung angestrebt, wodurch neue Verfahren im Bereich der Operationstechnik ermöglicht werden. In der Industrie-Automatisierung können Bilderfassungssysteme für mehreren Aufgaben verwendet werden. Autonome Systeme können mit einer echtzeitfähigen Entfernungsmessung dynamisch auf ihr Umfeld reagieren und damit Unfallsituationen vermeiden. Des Weiteren sind Sortiersysteme in der Lage Volumina von zu sortierenden Objekten zu bestimmen [Kol].
Eine Kategorie von Bilderfassungssystemen stellen die 3D-Scanner dar, welche eine dreidimensionale Erfassung der Szene ermöglichen. Es gibt eine Vielzahl solcher Systeme, denen unterschiedliche Technologien zugrunde liegen und alle ihre spezifischen Vor- und Nachteile haben. 3D-Scanner lassen sich grob in die Kategorien *Contact-Scanner* sowie aktive und passive *Non-Contact-Scanner* einteilen.

3.1. Contact-Scanner

Ein solcher Scannertyp besteht zum Beispiel aus einem Stift, welcher an einem Gelenkarm befestigt ist. In jedem Gelenk befinden sich Sensoren, die jede Winkeländerung registrieren und speichern. Durch das Nachfahren der Konturen des zu digitalisierenden Objektes kann dieses später rekonstruiert werden [Hor02]. Nachteile ergeben sich durch

den aufwendigen physischen Abtastvorgang, da das Objekt (je nach Material und Beschaffenheit) verformt oder gar beschädigt werden kann. Auch die Geschwindigkeit stellt einen Nachteil dar, da die Abtasteinheit mechanisch an jeden Punkt der Oberfläche bewegt werden muss. Der zeitliche Aufwand sowie die Größe der Apparatur, steigen mit zunehmender Objektgröße, weshalb dieses Verfahren für große Objekte oder komplexe Szenen ungeeignet ist. Die Vorteile liegen zum einen in der Messgenauigkeit und zum anderen stellen die Reflektionseigenschaften der Objekte kein Hinderniss dar. Allerdings könnten, je nach Konsistenz des Objektes, Ungenauigkeiten beim Abtasten auftreten, die von der Kraft abhängt, welche auf die Abtastvorrichtung ausgeübt wird.

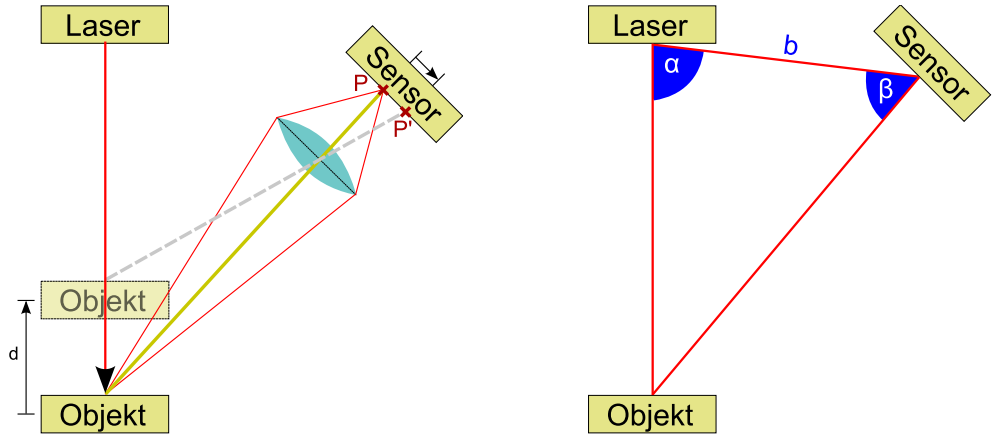
3.2. Aktive Non-Contact Scanner

Scanner dieses Typs nutzen eigene Lichtquellen, um einen Teil des Objektes zu beleuchten. Dabei hängt die benötige Leistung proportional von der Größe des zu scannenden Objektes ab. Während des Scanvorganges wird das Objekt von einem Sensor beobachtet, der die vom Objekt reflektierten Signale detektiert und verarbeitet. Die Hauptvorteile bei Systemen dieses Typs liegen darin, dass Mehrdeutigkeiten durch kontrollierte Beleuchtung ausgeschlossen werden können [Müh02] und dass eine direkte Abtastung umgangen wird. Die Nachteile liegen sowohl in Preis, Größe und Gewicht, als auch darin, dass das Objekt während des Scanvorganges nicht bewegt werden darf. Zur Kategorie der aktiven Non-Contact Scanner zählen z.B. Systeme, die folgende Techniken verwenden:

- Time-of-Flight: Der Aufbau besteht aus einer Lichtquelle und einem Sensor, die sich in unmittelbarer Nähe befinden. Die Entfernungsmessung geschieht über die Messung der Zeit, die ein Signal für die Strecke vom Sender zum Objekt und zurück benötigt (*round-trip-time* (RTT)). Dabei kann das ausgebende Signal entweder gepulst oder amplituden- bzw. frequenzmoduliert sein [Kol]. Durch geeignete Verfahren muss sichergestellt werden, dass nur Signale des Senders detektiert werden. Die ToF-Technik hat in den letzten Jahren stark von Entwicklungen auf dem Gebiet der Mikroelektronik profitiert. Die Systeme sind zunehmend kleiner, billiger und einfacher im Aufbau geworden. Neben der stetig zuverlässiger gewordenen Messgenauigkeit [LSBS99] bietet dieses Verfahren einen enormen Geschwindigkeitsvorteil. Der Nachteil bei dieser Methode liegt darin, dass in einem Raumwinkel jeder Punkt einzeln (also sequentiell) gescannt wird. Außerdem bestehen hohe Anforderungen an die signalverarbeitenden Komponenten [Müh02]. Ein Beispiel für ein System, welches das ToF-Verfahren verwendet, stellt die in dieser Arbeit verwendete PMD-Kamera dar (siehe Kapitel 3.4).

- Triangulation: Der Name dieses Verfahrens lässt sich durch die Anordnung des Sensors, eines Lasers und dem Objekt der Szene erklären. Abb. 3.1 (a) skizziert einen beispielhaften Aufbau der Komponenten. Der Laser sendet ein Signal in die

Szene, welches von dem Sensor auf dem Objekt erkannt wird. Aus der Kameraperspektive gesehen, hängt die Position des Laserpunktes auf dem Objekt von der Entfernung zwischen Objekt und Laser ab. Verschiebt sich das Objekt um die Distanz d, ändert sich auch die Position des Laserpunktes aus der Sicht der Kamera (P bzw. P'). Die Position des Objekts lässt sich aufgrund der bekannten Winkel α und β, sowie der Seitenlänge b (siehe Abb. 3.1 (b)) mit Hilfe von trigonometrischen Mitteln berechnen.



(a) Aufbau, Abb. aus [wik], Änderungen M.B.

(b) Mathematisches Modell

Abbildung 3.1.: Prinzip der Triangulation

Die Triangulation ist algorithmisch sehr einfach und liefert zudem sehr präzise Ergebnisse [Kol]. Allerdings lässt sich dieses Verfahren nicht dort einsetzen, wo Echtzeitfähigkeit gefordert wird, da nur eine Zeile oder Spalte pro Abtastvorgang gemessen werden kann. Einen weiteres Problem stellen Verschattungseffekte dar, welche durch den notwendigen Abstand zwischen Laser und Sensor (*Baseline*) bedingt sind. Bei Messungen für größere Entfernungen muss dieser Abstand entsprechend groß sein, wodurch die Größe des gesamten Aufbaus wächst [Kol].

- Strukturiertes Licht: Mit diesem Verfahren ist es möglich, gleichzeitig eine Messung aller Spalten durchzuführen. Die Vorgehensweise ist hier ähnlich der Triangulation, mit dem Unterschied, dass ein Projektor statt eines Lasers verwendet wird. Das Detektionsproblem, welches sich durch die zeitgleiche Messung ergibt, wird durch die, wie in Abb. 3.2 skizzierte, Projektion mehrerer unterschiedlicher Streifenmuster auf das zu scannende Objekt gelöst. Bei jedem weiteren Streifen halbiert sich die Breite, so dass eine Zuordnung durch eine binäre Nummerierung möglich ist. Im Vergleich zur Triangulation werden nur $log_2(n)$ Aufnahmen benötigt werden. Es kann eine vergleichbare Genauigkeit wie bei der Triangulation erreicht werden, wobei gleichzeitig die Geschwindigkeit bei der Entfernungsmessung erhöht wird.

Ein weiterer Vorteil ergibt sich aus der Tatsache, dass kein teurer Laser verwendet werden muss, der ggf. sogar für das menschliche Auge gefährlich werden kann [Müh02]. Mit zunehmender Auflösung, steigt die Anzahl der Muster, die verwendet werden muss und damit auch die Zeit, die für das Verfahren benötigt wird.

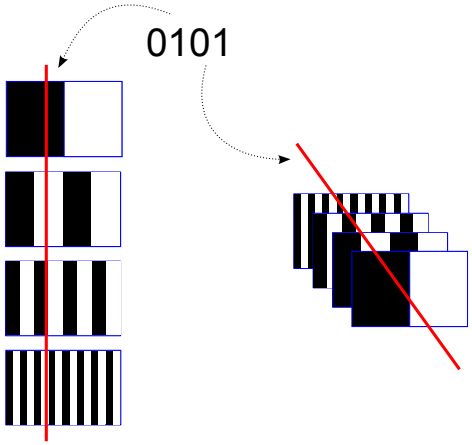


Abbildung 3.2.: Projektion unterschiedlicher Streifenmuster. Durch die Zuordnung von Schwarz/Weiß zu 0/1 (oder umgekehrt) ist eine binäre Nummerierung möglich.

3.3. Passive Non-Contact Scanner

Solche Systeme sind für gewöhnlich relativ kostengünstig, da keine Strahlung emittiert werden muss und somit die Projektionseinheit eingespart werden kann. Der Einsatz von passiven Verfahren ist dort angebracht, wo nicht unbedingt hohe Genauigkeit erforderlich ist, da die Genauigkeit der vorgestellten aktiven Scanner nicht erreicht werden kann. Neben dem Kostenfaktor bieten diese Verfahren auch den Vorteil, dass der zu scannende Bereich um ein Vielfaches größer sein kann, als es bei aktiven Verfahren möglich ist [Müh02].

- *Shape From Shading* (SFS): Es wird versucht, aus den Helligkeitsinformationen des Objektes die Normalen und damit die Oberfläche zu bestimmen. Zuerst ermittelt die Kamera die Intensitätswerte des Objektes, welche von der Kameraposition, der Position der Lichtquelle, Materialeigenschaften und dem Winkel zwischen Lichtvektor und Normalenvektor abhängen. Mit Hilfe mehrere Aufnahmen kann, über die Berechnung des Gradienten, das zum Normalenfeld gehörende skalare Höhenfeld berechnet werden. Für die Praxis ist das Verfahren nur in Sonderfällen geeignet, da viele Einschränkungen gemacht werden müssen, um plausible Ergebnisse zu erhalten. Denn da die Tiefenerfassung lediglich von den Intensitätswerten

abhängt, liefert die Berechnung ein falsches Ergebnis, sobald eine Intensitätsänderung *nicht* von einer Normalenänderung verursacht wird. Es müssen z.B. folgende Einschränkungen gemacht werden: Verwendung von nur einer Lichtquelle mit bekannter Position, sowie von rein diffus reflektierenden Oberflächen [Müh02].

- Stereo-Vision: Das räumliche Sehvermögen des Menschen wird nachgeahmt, indem dieselbe Szene mit zwei Kameras aufgenommen werden, deren Positionen leicht gegeneinander versetzt sind. Mit Hilfe dieser *stereoskopischen Halbbilder* kann die Entfernung eines Punktes zur Kamera mathematisch bestimmt werden. Zunächst müssen mit Hilfe eines (auf der Triangulation basierenden) Verfahrens Korrespondenzen in beiden Bildern gefunden werden. Da die Kamerapositionen bekannt sind, kann zu jedem Punkt eines Bildes der korrespondierende Punkt und damit die 3D-Position berechnet werden [Kol]. Die Schwierigkeit liegt hier bei der Korrespondenzsuche, die auch durch die Reduktion auf eine 1D-Suche (durch die Verwendung von *Epipolarlinien*) sehr zeitaufwendig ist. Außerdem ergeben sich Probleme bei Szenen mit stark homogenen Flächen, da in diesem Fall keine Korrespondenzen gefunden werden können [Kol].

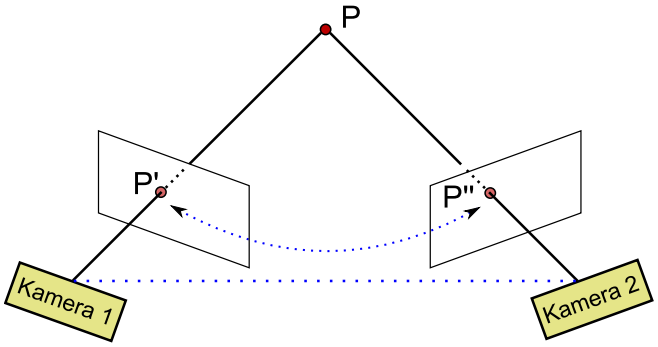


Abbildung 3.3.: Zuordnung korrespondierender Punkte.

3.4. PMD-Sensor

Das im Zusammenhang mit dieser Arbeit verwendete Bilderfassungssystem ist eine relativ neue Entwicklung und entspricht nicht den konventionellen optischen Systemen. Der wichtigste Bestandteil der PMD-Kamera ist ein Array von Sensoren, welche die Entfernung zu einem Objekt messen können. Das Besondere daran ist, dass die Szene nicht gescannt wird, sondern dass die Messung für jedes Pixel autonom und in allen Pixeln parallel stattfindet. Die Technologie der Kamera beruht auf Standard CMOS-Technologie (*complementary metal-oxide semiconductor*: Die dominierende Halbleitertechnologie für

Mikroprozessoren und Speicher) und hat dementsprechend deren Vorteile. Darunter zählen nach [Huß00] eine geringe Leistungsaufnahme, wahlfreier Zugriff auf alle Pixel, reduzierte *Blooming*- und *Smear*-Effekte, sowie verringerte Systemkosten bei der Integration weiterer Systemkomponenten. Im Folgenden wird der Aufbau und die Funktionsweise der PMD-Kamera näher beleuchtet.

3.4.1. Aufbau und Funktionsweise

Die PMD-Kamera verwendet das in Kapitel 3.2 diskutierte ToF-Verfahren, um die zusätzlichen Informationen der dritten Dimension erfassen zu können. Abb. 3.4 skizziert die grobe Funktionsweise der PMD-Kamera. Ein moduliertes Lichtsignal wird durch LEDs (*Light Emitting Diode*) im nahinfraroten (NIR) Bereich des Farbspektrums ausgesandt, um die Szene zu beleuchten. Da das Signal in einem Bereich gesendet wird, welches für das bloße menschliche Auge nicht sichtbar ist, muss keinerlei Schädigung für das Auge befürchtet werden.

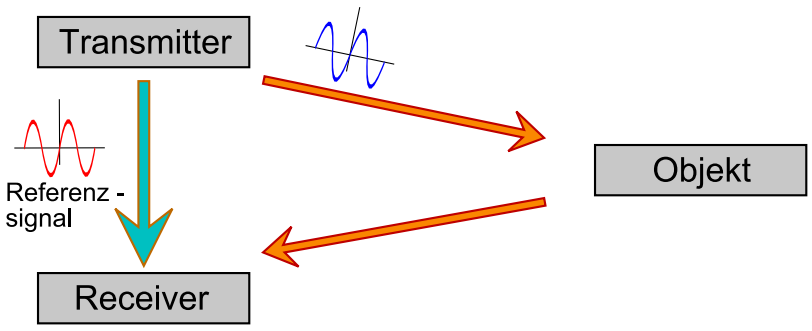


Abbildung 3.4.: Funktionsweise der PMD-Kamera.

Über die Stromstärke lässt sich die Lichtquelle direkt modulieren. Aufgrund der Demodulation wird meist eine sinusförmige Modulation verwendet, denkbar sind aber auch die Rechteckfunktion oder eine PN-Modulation (periodische Wiederholung eines zufälligen Rauschmusters). Die Modulation des Lichtsignals ist einerseits für die Entfernungsbestimmung notwendig, andererseits kann das Signal durch die Modulation auch von sonstiger Beleuchtung unterschieden werden (siehe Kapitel 3.2.2). LEDs sind billig und zudem relativ klein, daher entspricht deren Verwendung dem Wunsch nach einer Low-Budget-Realisierung. Außerdem haben LEDs den Vorteil, dass sie eine hohe Schaltgeschwindigkeit besitzen (Modulationsfrequenz bis 100 MHz). Das ausgesandte Signal wird an der Oberfläche des 3D-Objekts reflektiert und trifft auf den lichtempfindlichen Sensorbereich der PMD-Kamera, welcher aus einem Array von einzelnen PMD-Pixeln besteht. Jedes Pixel kann eigenständig das einfallende Lichtsignal detektieren, in ein elektrisches Signal umwandeln und demodulieren. Da diese komplexen Vorgänge in jedem einzelnen Pixel geschehen, spricht man auch von *smart pixels* [MKF+05].

Ein PMD Pixel besteht aus zwei lichtempfindlichen Modulationselektroden (*Photogates*), welche das eingehende Lichtsignal detektieren. Über Auslesedioden gelangt das Signal in den Ausleseschaltkreis des Pixels. Dort können die Ladungsträger über ein Referenzsignal kontrolliert werden, welches an den Gates anliegt. Abhängig von dem Referenzsignal werden die Ladungsträger entweder zur linken oder zur rechten Seite bewegt (*Ladungsträgerschaukel*), wodurch sich die Spannung am Ausgang der Dioden ändert. Die Ladungsträger aus den beiden Auslesedioden a und b fließen in die nachgeschaltete Ausleseschaltung und werden auf zwei eingefügte Speicherkapazitäten C_{PMD} akkumuliert [Huß00]. Die Summe der Ausgangsströme i_a und i_b, die an a bzw. b anliegen, werden so lange akkumuliert, bis t_{int} erreicht ist.

Die Summe

$$\sum U_{ab} = \frac{t_{int}}{C_{PMD}} \cdot (\bar{i_a} + \bar{i_b}) \tag{3.1}$$

liefert die Gesamtzahl aller Ladungsträger und steht für die Intensität des jeweiligen Pixels [Huß00]. Aus der Differenzspannung ΔU_{ab} beider Dioden a und b

$$\Delta U_{ab} = \frac{t_{int}}{C_{PMD}} \cdot (\bar{i_a} - \bar{i_b}) \tag{3.2}$$

lässt sich ein Wert bestimmen, aus dem die Entfernung berechnet werden kann. Durch Korrelation des detektierten optischen Signals im PMD-Pixel und des Referenzsignal (s. Abb. 3.5) und Sampling der Korrelationsfunktion, ist es möglich, den Phasenversatz und damit die Entfernung des Objekts zu bestimmen.

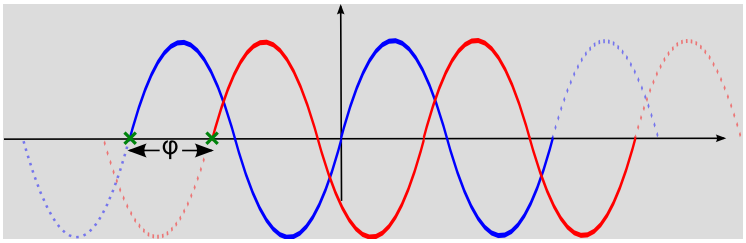


Abbildung 3.5.: Vergleich des Messsignals mit dem Referenzsignal zur Berechnung des Phasenversatzes.

Die Korrelationsfunktion $\varphi(\tau)$ kann aus dem gemessenen Signal $s(t)$ und dem Referenzsignal $g(t)$ berechnet werden [Lan00]:

$$\varphi(\tau) = s(t) \otimes g(t) = \lim_{T\to\infty} \frac{1}{T} \int_{-T/2}^{+T/2} s(t) \cdot g(t+\tau) dt \tag{3.3}$$

Mit dem Eingangssignal $s(t) = 1+a \cdot cos(\varpi t - \varphi)$ und dem Referenzsignal $g(t) = cos(\varpi t)$ ergibt sich die Korrelationsfunktion $C(\tau)$ (siehe [Lan00]).

$$C(\tau) = b + \frac{a}{2} \cdot cos(\varphi + \varpi\tau) \tag{3.4}$$

a ist dabei die Amplitude, φ der Phasenversatz, ϖ die Modulationsfrequenz und b der Offset der Korrelationsfunktion. Für ein Sinussignal reicht die Auswertung der Funktion an den Stellen $i = \frac{\pi}{2}$, wie z.B. $\varpi\tau_1 = 0°$, $\varpi\tau_2 = 90°$, $\varpi\tau_3 = 180°$ und $\varpi\tau_4 = 270°$, um die gesuchten Größen berechnen zu können [Huß00].

$$\varphi = atan\left(\frac{A_3 - A_1}{A_0 - A_2}\right) \qquad \text{mit} A_i = i \cdot \frac{\pi}{2} \tag{3.5}$$

Die Amplitude ist die Signalstärke und ein Maß für die Qualität der Messung [MKF+05].

$$a = \frac{\sqrt{(A_1 - A_3)^2 + (A_2 - A_4)^2}}{2} \tag{3.6}$$

Der Offset bestimmt den Grauwert eines Pixels und wird über das arithmetische Mittel der Samplingpunkte berechnet.

$$b = \frac{A_1 + A_2 + A_3 + A_4}{4} \tag{3.7}$$

Der Zusammenhang der Werte, sowie deren Bedeutung wird in Abb. 3.6 verdeutlicht.

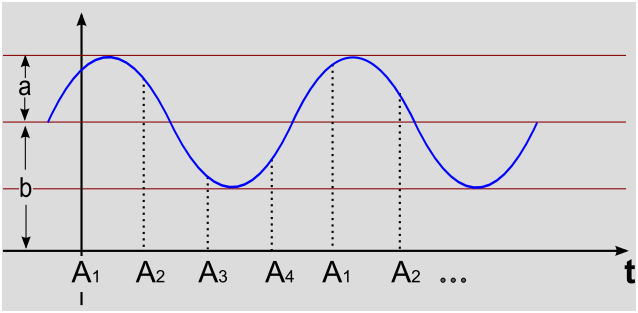
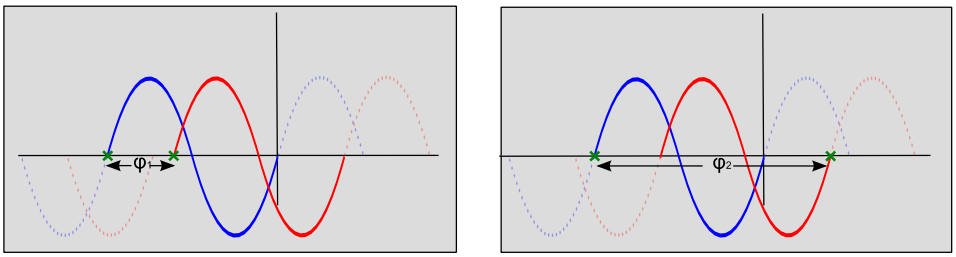


Abbildung 3.6.: Sampling des optischen Signals.

Durch Ausnutzung der Lichtgeschwindigkeit, lässt sich die Entfernung w des Objektes, an dem das Lichtsignal reflektiert wurde, durch folgende Formel bestimmen [Huß00]:

$$w = \frac{c \cdot \varphi}{4\pi \cdot f_{mod}} \tag{3.8}$$

Wobei c die Lichtgeschwindigkeit und f_{mod} die Modulationsfrequenz ist.



(a) (b)

Abbildung 3.7.: Beschränkung des Eindeutigkeitsbereiches.

Ein eindeutiger Messbereich muss gewährleistet sein, da es sonst zu Mehrdeutigkeiten kommt. In Abb. 3.7 (a) benötigt das Licht deutlich weniger Zeit, bis es vom Sensor detektiert wird, als es in 3.7 (b) der Fall ist. D.h. das Objekt in (a) ist näher am Sensor. Trotzdem ist der Phasenversatz in beiden Situationen gleich und somit liefert auch die Entfernungsberechnung das selbe Ergebnis. Der Grund für diese Situation liegt in dem sich wiederholenden sinusförmigen Signal, wodurch der Eindeutigkeitsbereich durch $d = \lambda/2$ (mit der Wellenlänge λ) beschränkt ist. Der Faktor $\frac{1}{2}$ muss berücksichtigt werden, da die Strahlen die Strecke doppelt zurücklegen müssen: Vom Sender zum Objekt und wieder zurück zum Sensor [MKF+05].
Das Prinzip der Laufzeitmessung ist keineswegs ein neuartiges Verfahren, allerdings geschieht der gesamte Prozess innerhalb des jeweiligen PMD-Pixels. Dadurch kann auf Verstärker des Signals verzichtet werden, wodurch im Mischprozess weniger Rauschen entsteht. Da Bauelemente eingespart werden können, reduziert sich auch der Preis und die Größe des Systems. Ein weiterer Vorteil, der durch den Fortschritt in der Mikroelektronik entstanden ist, ist durch die immer kleiner werdende Größe der Photogates zu erklären. Da die Zeit für den Ladungstransport von der Länge der Gates abhängt, wird hier immer mehr Zeit beim Transport eingespart. Da NIR die Szene nicht beeinflusst, lässt sich die PMD-Kamera relativ einfach mit anderen Kameras kombinieren. Die Vorteile des PMD-Arrays an sich lassen sich dadurch begründen, dass hier die CCD- bzw. CMOS-Technologie zugrunde liegt: Kompakte Bauweise, niedrige Herstellungskosten, geringer Stromverbrauch, geringes Rauschen, hohe Ausfallsicherheit sowie eine hohe geometrische Genauigkeit. Außerdem ist es empfindlich für ein breites Spektrum von Wellenlängen [XSHR05]. Einen Nachteil der PMD-Kamera stellt die geringe Auflösung dar. Z.B. besteht das PMD-Array der in dieser Arbeit verwendeten Kamera (PMD[VISION]C® 19K) aus 160 × 120 Pixeln, so dass nur 19.200 Punkte gemessen werden können. Ein weiterer Nachteil wird dadurch verursacht, dass bei der Messung des Phasenversatzes

zwischen dem empfangenen Signal und dem Referenzsignal drei (oder besser vier) Phasenwerte ausgelesen werden müssen. Dadurch entstehen zum einen Bewegungsartefakte und zum anderen schlägt sich dies auf die Geschwindigkeit der Entfernungsmessung nieder, die dennoch für Echtzeitfähigkeit ausreichend ist.

3.4.2. Hintergrundbeleuchtung

Ein Grund für die Messungenauigkeit bei Laufzeitmessungen besteht in unkorrelierter Beleuchtung, welche durch Hintergrundbeleuchtung verursacht wird. Ist keine Hintergrundbeleuchtung vorhanden, so muss jedes vom Detektor empfangene Lichtsignal von den LEDs der PMD-Kamera kommen. Allerdings soll die PMD-Kamera auch in nicht abgedunkelten Räumen (oder im Freien) genutzt werden können. Da z.B. Sonnenlicht ein sehr viel höheres Signal generieren kann, als die aktive Beleuchtung der Kamera, wird die Dynamik für das aktive Licht und damit die Sensorperformanz verringert [Rin07]. Daher muss die Hintergrundbeleuchtung von der aktiven Beleuchtung der Kamera getrennt werden. Findet keine Differenzierung statt, so sind die Auslesedioden sowohl mit dem Licht der Umgebung, als auch mit dem Licht der aktiven Beleuchtung der Kamera gefüllt. Da die Hintergrundbeleuchtung für beide Dioden gleich und der relevante Signalanteil sehr gering ist, werden auch bei beiden Auslesedioden fast die gleiche Anzahl an Ladungsträgern generiert. Dadurch wird abhängig von der Intensität der Hintergrundbeleuchtung das Entfernungsrauschen erhöht.
Eine Methode, mit der es möglich ist, zwischen dem modulierten Signal der Kamera und sonstiger Beleuchtung zu differenzieren, nennt sich *Suppression of Background Illumination* (SBI). Die SBI-Einheit ermittelt den Anteil des unkorrelierten Signals und steuert einen entsprechenden Gegenstrom, welcher die Beeinträchtigung schon während des Integrationszeitraums kompensiert. Dadurch hängt die Anzahl der erzeugten Ladungsträger nur noch von der aktiven Beleuchtung der Kamera ab und die Hintergrundbeleuchtung stellt keinen Störfaktor mehr dar. Weitere Störungen können durch erhöhte Temperaturen verursacht werden: Durch thermische Effekte werden spontan freie Ladungsträger in dem lichtempfindlichen Halbleiter generiert (*Dunkelstrom*), welche die Entfernungsmessung beeinträchtigen. Diese Störungen können ebenfalls durch SBI unterdrückt werden, wodurch die Kamera auch dort eingesetzt werden kann, wo die Abstandsmessung durch unkorrelierte Signale aufgrund von thermischen Effekten erschwert wird [MKF+05].
Zusammen mit weiteren Maßnahmen, wie optischen Filtern und dem *Burst*-Modus ist die PMD-Kamera quasi unempfindlich gegen Fremdlichteinflüsse [Rin07]: Durch einen optischen Filter (Bandpassfilter) kann der Frequenzbereich des Lichtes begrenzt werden, wodurch die Anzahl der Photonen, die den Sensor erreichen, reduziert wird. Ein Bandpassfilter lässt nur Signale eines bestimmten Frequenzbandes passieren und blockiert alle anderen Signale des Frequenzbereichs. Beim *bursting* einer Lichtquelle wird die selbe Anzahl von Lichtquellen (hier also LEDs) verwendet, allerdings strahlen diese anstatt kontinuierlichen Signalen, höhere Peaks aus. Dadurch verringert sich die Anzahl

der Rauschelektronen bei gleichbleibender Anzahl von Signalelektronen. Damit ist der Burst-Modus effektiv bei starker Hintergrundbeleuchtung und zur Bewegungsdetektion von schnellen Objekten [MKF+05].

3.5. Ähnliche Kameras

Neben der PMD-Kamera gibt es weitere ähnliche Techniken zur Entfernungsbestimmung von Objekten. Die wesentlichen Eigenschaften z.Z. aktueller Beispiele sollen hier kurz erläutert werden.

- SwissRanger: Die *SwissRanger* Kamera hat denselben Ursprung, wie die PMD-Kamera und wurde parallel dazu weiterentwickelt. Daher sind die beiden Kameras in Funktionsweise und Aufbau sehr ähnlich. Der *SwissRanger*^{TM} *SR4000* ist die neueste Entwicklung der Firma MESA Imaging. Im Vergleich zur hier verwendeten PMD-Kamera bietet diese Kamera den Vorteil eines größeren Pixel-Arrays (176×144) und eine höhere Bildrate (bis zu 54 FPS). Die Lichtquelle zur Beleuchtung der Szene besteht aus einem Array von 24 LEDs, die im infraroten Bereich des Farbspektrums ein amplituden moduliertes Signal senden [SR408].
- Canesta: Basiert auf der CMOS-Technologie und nutzt LEDs, die infrarotes Licht senden, um die Szene zu beleuchten. Der Sensorbereich besteht aus einem Array von „magic pixel“, welche u.a. einen *in phase receptor* und einem *out phase receptor* beinhalten. Der *in phase receptor* ist so lange aktiv, wie die LED Licht aussendet. Ist der Impuls zu Ende wird der *out phase receptor* aktiv. D.h. ist ein Objekt nahe an dem Sensor, trifft der überwiegende Teil des Lichtes auf den *in phase receptor*, ist es zu weit weg, trifft es auf den *out phase receptor*. Damit gibt das Verhältnis des eingefallenen Lichtes der beiden Rezeptoren ein Maß für die Distanz des Objektes an. Ein Nachteil des verwendeten Verfahrens liegt in der Verwendung von Lichtimpulsen, da eine minimale Zeitspanne vergeht, bis das Signal den Zustand gewechselt hat. Außerdem gibt es Probleme bei der Messgenauigkeit, da jedes Pixel nur eine gewisse Menge der eingehenden Lichtsignale akkumulieren kann. Wird diese Grenze überschritten, wird die Differenz und damit die Entfernungsmessung verfälscht [Can08].
- ZCam: Die *ZCam* der Firma 3DV Systems sendet kurze Lichtimpulse im infraroten Bereich des Lichtspektrums aus, deren Reflektionen an dem Objekt der Szene mit einem CCD-Sensor empfangen werden. Mit Hilfe einer Verschlusstechnik lässt sich die Relation zwischen der gesamten reflektierten Intensität und der beim *gating* gemessenen Intensität feststellen. Diese Relation liefert letztendlich ein Maß für die Laufzeit des Lichtes und damit auch für die Entfernung des Objektes zum

Sensor. Die Nachteile des Systems liegen in der geringen Genauigkeit, dem hohen Energiebedarf, sowie dem hohen Gewicht der Kamera [Kol].

Kapitel 4.

Die Graphikpipeline

Die Graphikpipeline (auch *Rendering Pipeline*) ist der Verarbeitungsweg einer vektoriellen, graphischen Beschreibung einer Szene bis hin zum gerasterten Bild, welches auf dem Monitor dargestellt wird. Die Berechnungen, die in den einzelnen Zwischenschritten durchgeführt werden, sind dabei unabhängig voneinander. In der Umsetzung der Graphikpipeline sind in den letzten Jahren enorme Fortschritte gemacht worden. Auf die einzelnen Entwicklungsstadien wird im Folgenden näher eingegangen.

4.1. Fixed Function Pipeline

In der ursprünglichen Form der Graphikpipeline (*Fixed Function Pipeline*) werden Vertices zur Verarbeitung an die Pipeline gereicht und durchlaufen fest definierte Verarbeitungsstufen. Die einzige Möglichkeit für den Programmierer Einfluss zu nehmen besteht in der Nutzung der *State Machine* (Menge aller Zustände). Z.B. kann der sogenannte Tiefentest zum Test auf Verdeckungen oder das Alphablending über den jeweiligen Zustand aktiviert bzw. deaktiviert werden.

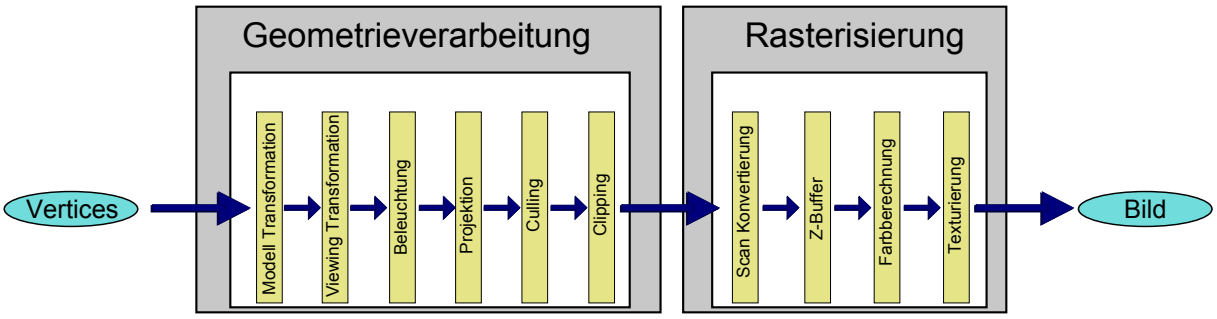


Abbildung 4.1.: Fixed Function Pipeline.

Die Graphikpipeline kann grob in drei Phasen unterteilt werden: Geometrieverarbeitung, Rasterisierung und Fragmentoperationen.

Bevor auf die erste Phase eingegangen wird, soll kurz erläutert werden, was unter dem Begriff *Vertex* zu verstehen ist. Bei einem Vertex (Mehrzahl: *Vertices*) handelt es sich um einen einfachen Geometriepunkt, der Primitive im Raum beschreibt. Er ist definiert durch seine Koordinate, optionale Farbwerte, Materialwerte, Normale, Texturkoordinate und Attribute.
Die Eingabe-Vertices liegen zunächst in Modellkoordinaten vor, d.h. die Koordinatensysteme einzelner Objekte müssen nicht zwingend übereinstimmen. Daher wird eine Modellierungstransformation angewendet, welche alle eingehenden Vertices in ein globales Koordinatensystem transformiert. Es folgt die Viewingtransformation, welche die vorliegenden Koordinaten in das Standard-Kamerakoordinatensystem transformiert, um spätere Berechnungsschritte zu vereinfachen. Beide Schritte werden meist zusammengefasst (z.B. in der OpenGL API), so dass nur eine einzige Matrixmultiplikation (*Model View Matrix*) für beide Transformationen notwendig ist. Ggf. können in der ersten Phase auch noch Texturkoordinaten und die Beleuchtung pro Vertex berechnet werden. Im zweiten Teil der ersten Phase findet die Projektion der Daten (die immer noch im dreidimensionalen Raum vorliegen) statt. Meist wird die perspektivische Projektion verwendet, bei welcher der Sichtbereich eine Pyramide (mit der Kamera an der Spitze) darstellt, wodurch das natürliche Sehen nachgebildet (*Lochkameramodell*) und Rechenaufwand gespart wird. Danach werden alle Flächen eliminiert, welche nicht zur Kamera zeigen (*Backface Culling*). Einen weiteren Schritt zur Effizienzsteigerung stellt das *Clipping* dar. Hier wird der Sichtbarkeitsbereich durch eine *near* und eine *far plane* begrenzt, wodurch die Pyramide zum Pyramidenstumpf (*Viewing Frustum*) wird. Alle Objekte, die sich außerhalb dieses Pyramidenstumpfes befinden, werden nicht gerendert. [WND97]
Die zweite Verarbeitungsstufe stellt der sogenannte *Rasterizer* dar, welcher eingehende Primitive in kleinere Einheiten (*Fragmente*) zerlegt. Ein Fragment entspricht einem Pixel im Framebuffer und beinhaltet verschiedene Daten, wie z.B. Koordinaten, Tiefenwert, Farbe und Texturkoordinate [WND97]. Die Werte dieser Attribute werden durch Interpolation zwischen den Vertex-Werten bestimmt.
Nachdem die Fragmente im Rasterisierungs-Prozess generiert wurden, können in der dritten Phase eine Reihe von optionalen Operationen ausgeführt werden, bevor die entsprechenden Werte im Framebuffer gespeichert werden. Darunter zählen Texturierung, Scissor-Test, Alpha-Test, Stencil-Test, Tiefentest (*Z-Test*) und Alphablending. Welche dieser Operationen tatsächlich ausgeführt wird hängt von der Aktivierung des jeweiligen Zustandes (*State*) ab. Am Ende der Pipeline findet die Bildkomposition statt [WND97].

4.2. Hardware Implementierung

Mit einer Softwareimplementierung der Graphikpipeline ist die Darstellung von 3D-Daten problemlos möglich. Mit dem Ziel immer mehr Details in höherer Auflösung durch Echtzeitrendering dartzstellen, wird Graphikhardware eingesetzt. Die darzustellende Szene wird in Dreiecke tesseliert und von einer Pipeline verarbeitet. Einzelne Stufen der Verarbeitung werden von verschiedenen Einheiten übernommen, die jeweils mehrfach in Hardware vorliegen, so dass die Verarbeitung parallel durchgeführt werden kann. 1997 wurde zum ersten mal ein 3D-Hardwarebeschleuniger auf einem Graphikchip umgesetzt. Durch die Implementierung in Hardware bleibt die Pipeline noch fix, allerdings wird die Verarbeitungsgeschwindigkeit deutlich erhöht. Die Stufe der Geometrieverarbeitung wurde nach wie vor auf der CPU erledigt und nur die Rasterisierung wurde auf die *Graphics Processing Unit* (GPU) verlagert und damit effizient in Hardware implementiert. Bei der GPU handelt es sich um einen speziellen Prozessor der Grafikkarte, mit welchem neben der CPU ein zweiter Prozessor zur Verfügung steht, wodurch eine Parallelisierung von Arbeitsschritten möglich ist. Durch Auslagerung von Berechnungen auf die GPU kann die CPU entlastet werden. Ein weiterer Vorteil besteht darin, dass der Output von GPU-Berechnungen direkt dort zur Verfügung steht, wo er benötigt wird und muss nicht erst von der CPU über den Graphikbus transportiert werden. Des Weiteren werden neue GPU-Generationen im Vergleich zu neuen CPU-Generationen schneller entwickelt, sind billig, leicht zu upgraden, sowie kompatibel mit vielen Hardware- Architekturen und Betriebssystemen. Diese Entwicklung der letzten Jahre liegt zum Teil an der Spieleindustrie, die wirtschaftlich gesehen sehr stark ist und die Entwicklung immer leistungsstärkerer GPUs vorantreibt. Die Daten wurden zunächst über den *peripherical component interconnect* (PCI) Bus auf die GPU transportiert. Als Besonderheit brachte die erste GPU *z-buffering* und einfaches *texture mapping* mit sich [FHWZ04].
1998 kamen Graphikkarten auf den Markt, die bereits Multitexturing unterstützten. Außerdem wurde der PCI Bus vom *Accelerated Graphics Port* (AGP) abgelöst, der eine deutlich höhere Bandbreite zwischen CPU und GPU zur Verfügung stellt. Zwischen 1999 und 2000 wurde Hardware entwickelt, auf der auch die Stufe der Geometrieverarbeitung durch Hardware umgesetzt wird. Außerdem wurden weitere Operationen wie z.B. *Bumpmapping* durch neue Hardwareeinheiten ermöglicht [FHWZ04].

4.3. Programmable Pipeline

2001 wurde eine neue Entwicklungsstufe der Graphikpipeline vorgestellt: Die *Programmable Pipeline*. Die GPU wird teilweise programmierbar, wodurch die starre Struktur der *Fixed Function Pipeline* aufgelöst wird. Als Folge daraus erhält der Programmierer mehr Möglichkeiten, um auf den Verarbeitungsprozess der Vertices Einfluss zu nehmen. Ein Programm, welches einen auf der GPU ausführbaren Shader implementiert, wird als

Shader-Programm bezeichnet. Diese entscheidende Entwicklung wurde mit dem Shader Model 1.1 ermöglicht. Der Nachteil bei der Programmierung mit Shader Model 1.1 ist zum einen der geringe Funktionsumfang, da auch einfache Konstrukte wie z.B. Schleifen und Verzweigungen nicht möglich sind. Zum anderen ist die Programmierung nur mit Hilfe von relativ schwer verständlichem Assemblercode möglich. Die nächsten größeren Verbesserungen brachte das Shader Model 2.0, wodurch Schleifen-Konstrukte ermöglichte wurden. Seit 2005 gibt es Graphikkarten, die das Shader Model 3 unterstützen, welches eine deutliche Performanzsteigerung mit sich bringt. Mit den neuen Shader Model Versionen ist es möglich, die GPU für die jeweilige Anwendung in Hochsprachen zu programmieren und damit als Programmierer einfacher Einfluss auf die Graphikpipeline zu nehmen. Außerdem stellt die Version 3.0 ein weiteres wichtiges Programmier-Konstrukt bereit: *dynamic branching* [FHWZ04]. Für die Programmierung der Pipeline existieren inzwischen verschiedene Hochsprachen, wie z.B. Cg, HLSL oder GLSL, die vergleichsweise leicht zu verstehen und anzuwenden sind. Bis zum Shader Model 3 können zwei Arten von Shadern verwendet werden: *Vertex Shader* und *Fragment Shader*. Sowohl der Vertex- als auch Fragmentprozessor arbeiten parallel, d.h. sie können zu einem Zeitpunkt mehrere Pixel bzw. Fragmente gleichzeitig bearbeiten (*SIMD, single-instruction, multiple-data*). Bis auf diese beiden programmierbaren Einheiten ist der restliche Teil der Graphikpipeline weiterhin starr. Mit dem Shader Model 4 kam, neben einigen anderen Verbesserungen, auch eine weitere programmierbare Stufe der Graphikpipeline hinzu: Der *Geometry Shader*, auf den in Kapitel 4.3.2 eingegangen wird. Die Graphikpipeline neuerer Graphikkarten unterscheidet sich also von der Fixed Function Pipeline durch die Programmierbarkeit des Vertex (, Geometry) und des Fragment Shaders (*Programmable Pipeline*) (siehe Abb. 4.2).

4.3.1. Vertex Shader

In der Fixed Function Pipeline wird in der Stufe der Vertexoperationen z.B. die Transformation der Vertices im Raum, deren Beleuchtungsberechnung, sowie die Normalentransformation vorgenommen. Aufgrund der Programmierbarkeit moderner Pipelines kann ein Programm geschrieben werden, welches beliebige Bearbeitungsschritte der einzelnen Vertices durchführt: Ein *Vertex Shader*. Dabei handelt es sich um ein Programm, welches vom Vertexprozessor (programmierbare Einheit auf der GPU) für jeden eingehenden Vertex ausgeführt wird. Ein Vertex Shader kann beliebige Berechnungen durchführen und ist nicht an die üblichen Berechnungen der Fixed Function Pipeline gebunden. Wird ein Shader benutzt, dann müssen dort alle nötigen Berechnungen implementiert sein. Es ist nicht möglich, einen Teil der Berechnungen selbst zu implementieren und einen anderen Teil von der Fixed Function Pipeline berechnen zu lassen. Des Weiteren besitzt ein Vertex keinerlei topologischen Kenntnisse oder kann auf Informationen anderer Vertices zugreifen. Ein weiteres neues Feature ist die Möglichkeit des wahlfreien Zugriffs auf den Texturspeicher, d.h. es können beliebige Werte ausgelesen werden, die vorher in einer

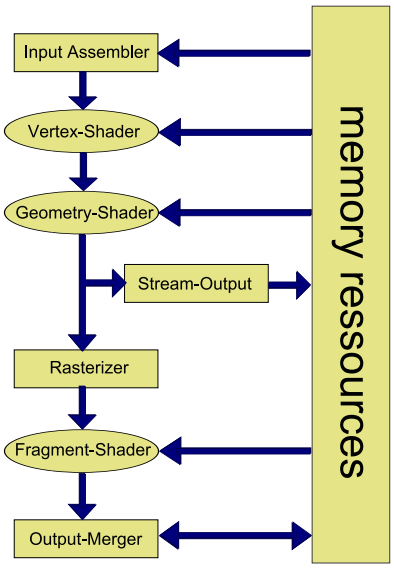


Abbildung 4.2.: Programmierbare Graphikpipeline (Shader Model 4). Abb. aus [Net08], Änderungen M.B.

Textur gespeichert worden sind. Im Vertex Shader können weder neue Punkte generiert, noch eingehende Vertices gelöscht werden. Am Ende der Bearbeitungsvorschrift wird genau ein Vertex an die nächste Stufe der Graphikpipeline weitergereicht.

4.3.2. Geometry Shader

Der Geometry Shader wird nach dem Vertex Shader aufgerufen und bearbeitet die ausgegebenen Vertices. Er wird für jedes Primitiv (Punkte, Linien, Dreiecke) einmal ausgeführt und kann daraus mehrere gleichartige Primitive erzeugen und erneut als Eingabe der Graphikpipeline verwenden. Dabei muss der Primitiv-Typ der Ausgabe nicht zwingend mit dem Primitiv-Typ der Eingabe übereinstimmen. Als Output kann der Geometry Shader jeglichen Input (Primitive, Texturkoordinaten, Farbe, Normale, etc.), sowie gespeicherte Daten aus Texturen generieren. Da der Geometry-Shader die einzige Station in der gesamten Pipeline ist, in der neue Vertices erzeugt werden können, nimmt er eine Sonderposition ein. Die vom Geometry Shader neu erzeugten Primitive werden wie jedes andere Primitiv behandelt, welches direkt von der Applikation (z.B. OpenGL) kommt.

4.3.3. Fragment Shader

Der *Fragmentprozessor* ist die dritte und letzte programmierbare Einheit auf der GPU. Der Fragment Prozessor arbeitet auf Fragment-Werten und deren Daten. Programme, die auf dem Fragment Prozessor ausgeführt werden heißen *Fragment Shader* (oder *Pixel Shader*). Analog zum Vertex Shader können hier auch Berechnungen durchgeführt werden, die den traditionellen Aufgabenbereich überschreiten. Der aktive Shader wird für jedes Fragment ausgeführt und kann weder neue Fragmente erzeugen, noch die Position bestehender Fragmente verändern. Auch hier ist es nicht möglich, nur einen Teil der benötigten Funktionalität zu implementieren und den übrigen Teil von der Fixed Function Pipeline berechnen zu lassen. Wird ein Fragment Shader verwendet, so müssen dort alle nötigen Berechnungen implementiert werden [Ros06].
Der Input des Fragment Prozessors besteht hauptsächlich aus den interpolierten Daten des Rasterisierungsprozesses. Der Shader wird für jedes Fragment ausgeführt, wobei während der Berechnung nicht auf die Daten anderer Fragmente zugegriffen werden kann. Auch hier ist ein wahlfreier Zugriff auf den Texturspeicher möglich. Es können z.B. Aufgaben wie Texturierung, echte Phong-Beleuchtung und Farbberechnung (pro Pixel) im Fragment Shader durchgeführt werden.

4.4. GPGPU und Shadersprachen

Die GPU lässt sich nicht für alle Probleme der Informatik ausnutzen. Es gibt Programme, die keinen Vorteil durch die GPU erzielen. Dies ist der Fall, wenn bei der Abarbeitung des Algorithmus kaum oder keine Parallelität möglich ist, oder die Anzahl der Speicherzugriffe, im Vergleich zu den arithmetischen Operationen, hoch ist. Außerdem unterliegt die Programmierung der GPU gewissen Einschränkungen, da z.B. programmiertechnische Konstrukte sowie die Genauigkeit von Gleitkommazahlen fehlen (auf der GPU wird eine Genauigkeit von 32 Bit verwendet, es ist aber keine doppelte Genauigkeit (double) möglich). Die GPU ist für Anwendungen in der Computergraphik sehr gut geeignet, da sie für dieses Einsatzgebiet entwickelt und optimiert wurde. Allerdings ist die Graphikhardware zu sehr spezialisiert, als dass sie flexibel für verschiedenste Anwendungen genutzt werden kann. *General Purpose computation on Graphics Processing Units* (GPGPU) bezeichnet die Nutzung der GPU für Berechnungen von nicht-graphischen Daten. Das Ziel liegt darin, die Vorteile des Graphikprozessors für allgemeine Anwendungen auszunutzen.
Bis vor wenigen Jahren gab es lediglich Assemblersprachen, um Programme für die programmierbaren Einheiten der GPU zu schreiben. Heute existieren einige Hochsprachen, welche die Programmierung erleichtern. Zur Entwicklung von 2D und 3D Computergraphik stehen zur Zeit zwei wichtige *Application Programming Interfaces* (API) zur Verfügung: Microsoft´s DirectX und OpenGL. Beide APIs unterstützen inzwischen ihre eigene höhere Programmiersprache. *OpenGL Shading Language* (GLSL, oft auch *GLSlang* genannt) ist seit OpenGL 2.0 im OpenGL Kern verfügbar, *High Level Shading Language* (HLSL) ist eine Direct3D-Komponente von Microsoft und *C for graphics* stammt aus dem Hause NVIDIA. Letztere kann sowohl für die OpenGL, als auch für die Direct3D API verwendet werden. Die Gemeinsamkeit jeder der drei Sprachen liegt darin, dass sie eng mit der Hochsprache C verwandt sind. Allerdings ist nur ein geringer Teil des Sprachumfangs in den Shadersprachen enthalten. Beispielsweise ist Folgendes nicht im Sprachumfang enthalten:

- Pointer und die dynamische Allokation von Speicher
- Parameterübergabe per Pointer
- Datentypen für Zeichen bzw. Zeichenketten
- Bit Operationen
- unions und enums

Einen entscheidenden Vorteil leisten hingegen eingebaute Matrix- und Vertex-Datentypen (für zwei-, drei- und vierdimensionale Vektoren bzw. Matrizen). Außerdem stehen häufig benötigte mathematische Funktionen zur Verfügung, wie z.B. Skalarprodukt, Kreuzprodukt und Norm [Ros06].

Die in den Hochsprachen geschriebenen Programme werden von einem Compiler in Maschinensprache übersetzt und zur Laufzeit in die GPU geladen. GPU-Programme sind keine eigenständigen Anwendungen, sondern bedürfen immer einem „Kontextprogramm“, in dem sie ausgeführt werden können [Müc07]. Außerdem muss bei Shaderprogrammen darauf geachtet werden, dass je nach verwendetem Shader Model bestimmte Einschränkungen (wie z.B. zulässige Anzahl der Instruktionen, Verfügbarkeit von Schleifen/Verzweigungen) gelten. In einem Kontextprogramm können mehrere Shader-Programme verwendet werden, von denen aber jeweils nur eines aktiv ist. Ein solches Programm besteht mindestens aus einer Main-Funktion als Einstiegspunkt.

Kapitel 5.

Point Based Rendering

Beim traditionellen Rendern wird die darzustellende Geometrie diskretisiert, um sie als Menge von Polygonflächen darzustellen. Dazu wird eine Menge von Vertices, sowie deren Topologie benötigt. Das in dieser Arbeit verwendete Verfahren unterscheidet sich von Standardverfahren zur Geometrierepräsentation und gewinnt in der Computergraphik immer mehr an Bedeutung: Das *Point Based Rendering* (PBR). Eine punktbasierte Darstellung einer Geometrie kann als Abtastung einer kontinuierlichen Oberfläche gesehen werden (z.B. mit Hilfe eines Scanners), welche eine Menge von 3D Punkten liefert. Bereits 1985 entwickelten Levoy und Whitted [LW85] die Idee, Punkte als universelle *Meta-Primitive* zu nutzen, um die Geometrie vom eigentlichen Rendervorgang zu trennen. Der Bedeutungsgewinn von PBR lässt sich folgendermaßen erklären: Die Komplexität von 3D-Modellen wächst stetig, durch z.B. immer besser werdende Scanner, welche immer mehr Daten liefern. Im Gegensatz dazu bleibt die Größe und Auflösung unserer Bildschirme stabil. Das führt bei der Wiedergabe von komplexen Geometrien dazu, dass viele kleine Primitive (z.B. Dreiecke) benötigt werden, welche bei der Darstellung auf weniger als ein Pixel abgebildet werden. D.h. es werden viele Daten berechnet und dargestellt, die keinen weiteren Qualitätsvorteil mit sich bringen. In diesen Fällen ist eine Darstellung der einzelnen Punkte (ohne deren Nachbarschaftsbeziehung) eine kostengünstige Möglichkeit, um die einzelnen Primitive und damit die gesamte Geometrie darzustellen [KB04]. Vor allem bei großen Datenmengen fällt die Speicherung der Topologie schwer ins Gewicht, die beim PBR nicht benötigt wird.
Bei der Darstellung mit herkömmlichen Verfahren muss beachtet werden, dass die polygonbasierte Pipeline zwar Punkte unterstützt, aber daraus keine Objekte darstellen kann. Eine wichtige Voraussetzung stellt eine polygonale Repräsentation dar, die durch eine schwierige und zeitaufwendige Triangulierung erreicht werden kann. Ein einfacher Weg wäre es, die Punktdaten direkt zu visualisieren. Um eine glatte Oberfläche darstellen zu können muss ggf. die Dichte der Abtastpunkte variiert werden. Im Folgenden wird auf verschiedene Verfahren eingegangen, mit denen es möglich ist, Geometrien aus Punktdatensätzen darzustellen.

- Triangulierung: Einer der einfachsten Ansätze, besteht darin, aus den vorliegenden Punkten wieder ein Dreiecksnetz zu konstruieren und dieses mit Polygon-Rendering Algorithmen darzustellen [Dac02]. Es handelt sich dabei nicht um ein PBR-Verfahren im eigentlichen Sinne, da zwar die Eingabedaten durch eine Punktwolke gegeben sind, aber trotzdem ein Polygon-Modell erzeugt wird. Die Triangulierung ist ein Standard-Problem in der Computergraphik, daher existieren auch viele verschiedene Algorithmen zur Erzeugung von Dreiecksnetzen aus einer gegebenen Punktwolke. Eine spezielle Triangulierung ist z.B. die Delaunay-Triangulierung, bei der die Innenwinkel der entstehenden Dreiecke optimiert werden [Kle06]. Allerdings bestehen die Nachteile bei diesem Verfahren darin, dass die Generierung des Meshes aufwendig ist und die Nachbarschaftsbeziehungen gespeichert werden müssen.

- Surfels und Visibility Splatting: Pfister und Zwicker et al. [PZvBG00] veröffentlichten 2000 eine Arbeit, welche ein Splatting-Verfahren auf der Basis sogenannter *Surfels* vorstellt. Ein Surfel besteht aus Tiefe, Texturfarbe, Normale und ggf. zusätzlichen Informationen. Die Daten, welche für das Rendering benötigt werden, werden in einem Vorverarbeitungsschritt generiert. Das geometrische Objekt wird von drei Seiten eines Würfels in drei aufeinander orthogonale *Layered Depth Images* (LDI) gesampled, welche zusammen als *layered depth cube* (LDC) bezeichnet werden. Per Raycasting wird die Oberfläche des Objekts abgetastet und an jedem Schnittpunkt mit den Strahlen ein Surfel mit den entsprechenden Eigenschaften erzeugt. Zur Speicherung der Surfels wird eine effiziente hierarchische Datenstruktur verwendet: Der LDC Baum. Der Baum wird während des Rendervorganges von der Wurzel abwärts traversiert und für jeden Block (Unterteilung des Baumes) zunächst ein *view frustum culling* durchgeführt. Im nächsten Schritt werden *visibility cones* verwendet, um eine Art backface culling zu erreichen. Die Surfels, die sich in einem sichtbaren Knoten des Baumes befinden, werden in den z-Buffer projiziert. Anschließend werden die sichtbaren Surfels, sowie Löcher im z-Buffer mit einer neuen Methode, dem *visibility splatting* bestimmt. Mit *Löchern* sind jene z-Buffer Pixel gemeint, die weder ein sichtbares Surfel, noch einen Hintergrund-Pixel beinhalten - sie müssen entsprechend markiert werden. Für jedes z-Buffer Pixel wird eine Referenz auf das nächstgelegene sichtbare Surfel, sowie der minimalen z-Wert gesetzt. Zur Vermeidung von Löchern in der Darstellung werden sogenannte *tangent discs* (siehe Abb. 5.1) verwendet. Dabei handelt es sich um Kreisscheiben, deren Normalen orthogonal auf der Objekt-Oberfläche stehen.

  Diese Scheiben werden um die Surfels angeordnet und in den z-Buffer gerendert, wodurch eventuelle Löcher in der Oberfläche geschlossen werden. Im letzten Schritt werden die Farbwerte für die Löcher berechnet, welche vorher im z-Buffer entsprechend markiert wurden.

- EWA Surface Splatting: Ren, Pfister und Zwicker [RPZ02] veröffentlichten 2001

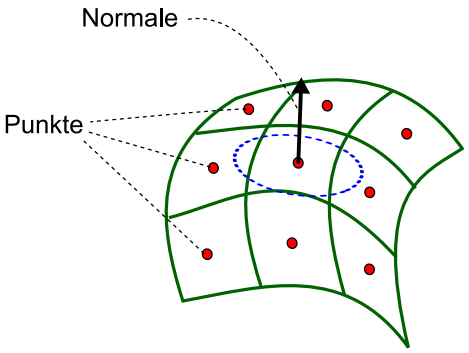


Abbildung 5.1.: Lückenlose Darstellung durch das Zeichnen von *tangent discs*.

ein Verfahren, mit dem es möglich ist, das sogenannte *Elliptical Weighted Average (EWA) Surface Splatting* auf moderner Graphikhardware mit Hilfe eines multipass Ansatzes zu nutzen: Das *object space EWA surface splatting*. Das Verfahren zielt auf eine möglichst gute Bildqualität bei interaktiver Nutzung ab. Der Rendervorgang besteht aus zwei Teilschritten:

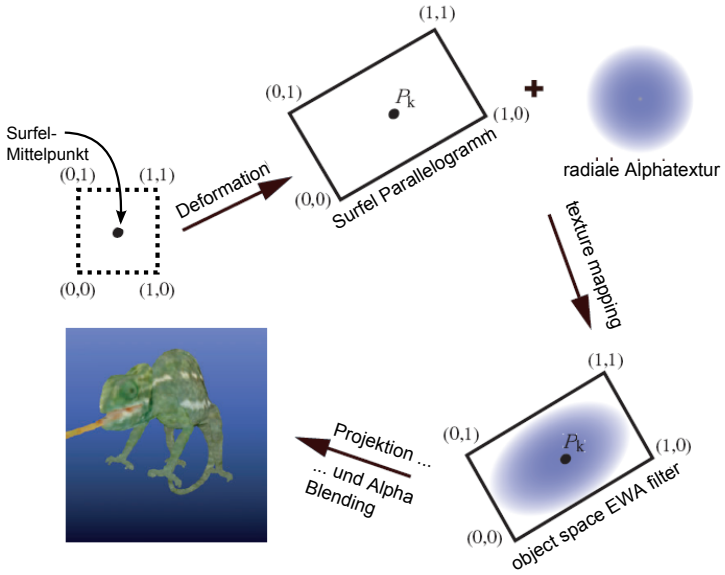


Abbildung 5.2.: EWA splatting. Abb. aus [RPZ02], Änderungen M.B.

Im ersten Durchlauf wird *visibility splatting* verwendet, wobei opake Polygone, mit dem Ziel einer lückenlosen Darstellung der Oberfläche, für jedes Surfel in den Z-Buffer gerendert werden. Um die Lücken im Tiefenbild zu vermeiden, wird die Kantenlänge der Quadrate als $2h$ gewählt, wobei h die maximale Distanz zwischen den Surfels in einer engen Nachbarschaft um das jeweilige Surfel beschreibt.

Im zweiten Renderpass wird zunächst der *object space EWA resampling filter* als Polygon mit semi-transparenter Alphatextur erstellt (siehe Abb. 5.2). Die Textur wird zur Kodierung einer Gaußfunktion, entsprechend dem *EWA resampling filter*, verwendet. Durch die Projektion des Polygons in den Bildraum erhält man den *screen space EWA resampling filter*, oder auch *EWA splat* genannt [RPZ02]. Die Auswertung der Splats geschieht durch die Multiplikation der Werte in den Splatmittelpunkten und der Farbe des aktuellen Surfels. Die Ergebniswerte werden in jedem Pixel aufaddiert. Durch einen Sichtbarkeitstest, der die Daten im Z-Buffer (aus dem ersten Schritt) verwendet, wird sichergestellt, dass nur solche Splats Einfluss auf das jeweilige Pixel haben, die dem Beobachter am nächsten gelegen sind. Alle anderen Splats werden verworfen.

- QSplats: [RL00] Ein weiteres PBR-Verfahren wurde durch das Digital-Michelangelo-Projekt populär: *QSplats*, welches von Rusienkiewicz und Levoy [RL00] entwickelt wurde. Bei dem Projekt ging es um die interaktive Darstellung von 3D-Scanning Datensätzen, die aus mehreren Millionen Knoten und Dreiecken bestehen [Dac02]. Ähnlich wie bei Pfister und Zwicker et al. [PZvBG00] wurde eine kompakte und effiziente Datenstruktur entworfen, welche hier zur Speicherung, *visibility culling*, *level-of-detail* (LOD) Kontrolle und zum Rendering verwendet wird. In einer Bounding Sphere Hierarchie werden jeweils Kugelmittelpunkt, Radius, Normale, Breite eines Normalenkegels, sowie optional ein Farbwert gespeichert. Die Daten werden quantisiert, um eine effiziente Speicherung zu ermöglichen. Rusienkiewicz und Levoy [RL00] verwenden zur Erstellung der Hierarchie ein Dreiecksgitter, welches aus den Scan-Daten berechnet wird, verwerfen dieses aber danach, da es beim Renderprozess keine Anwendung findet. Um die Daten zu rendern, wird die Datenstruktur von der Wurzel abwärts traversiert, wobei für jeden Knoten entschieden wird, ob er gerendert wird, oder nicht. Ist ein Knoten nicht sichtbar, so kann der ganze restliche Teilbaum ignoriert werden. Ist der aktuelle Knoten ein Blattknoten, so wird ein Splat gezeichnet. Bei den noch nicht behandelten Fällen muss entschieden werden, ob es sinnvoll ist, eine weitere Traversierung durchzuführen, oder ob der Nutzen zu gering wäre und die Traversierung abgebrochen wird. Zur Sichtbarkeitsprüfung wird *backface culling* mittels des gespeicherten Normalenkegels, sowie *view frustum culling* mit den Bounding Spheres verwendet [Dac02]. Als Splat-Form können sowohl Kreise, als auch Quadrate gewählt werden.

- LS-Oberflächen LS-Oberflächen stellen eine Möglichkeit dar, um eine Oberfläche aus punktbasierten Datensätzen implizit zu beschreiben. Als Sonderfall von LS-Oberflächen stellt Levin in [Lev03] die *Moving Least Squares* (MLS)-Oberflächen vor. Da in Kapitel 7 ein sehr ähnlicher Ansatz verwendet wird, soll hier nur auf die Kernidee eingegangen werden. MLS-Oberflächen sind nach [Lev03] durch alle Punkte definiert, die durch einen Projektionsoperator auf ich selbst abgebildet werden. Mit Hilfe von lokalen Referenzebenen wird versucht, lokale Polynome zu finden, die jeweils den geringsten Abstand zu einem Subset von Punkten haben.

MLS ermöglicht es einerseits Redundanz bei zu hoch abgetasteten Oberflächen aufzulösen, indem die Punkte entfernt werden, die am wenigsten Information beitragen (*downsampling*). Andererseits ist es möglich, weitere Punkte hinzuzufügen, wenn die Punktwolke nicht genügend dicht ist (*upsampling*). MLS-Oberflächen haben den Vorteil, dass Rauschen reduziert und die Punktwolke geglättet wird.

Kapitel 6.

Grundlagen

Das Ziel dieses Kapitel ist es, dem Leser eine Basis zu verschaffen, die dem Verständnis der vorliegenden Arbeit dient. Die einzelnen Abschnitte diskutieren nur die jeweiligen Grundzüge und sind daher relativ kurz gehalten.

6.1. Least Squares

Bei *Least Squares* (oder auch *Methode der kleinsten Quadrate*) handelt es sich um ein mathematisches Standardverfahren, das verwendet wird, um Modellparameter zu berechnen, welche gegebene Zieldaten möglichst gut repräsentieren. Die Kernidee, die hinter dem Verfahren steckt, ist die Minimierung der quadrierten Abstände zwischen den vorliegenden Daten und einem angestrebten Modell, welches die Daten möglichst genau beschreiben soll [Lev03, Nea04]. Die Eingabe für das Verfahren sind eine Reihe von Messdaten (Punktwolke) und als Ergebnis liefert es einen Vektor mit Koeffizienten. D.h. ausgehend von einer Menge von Punkten soll ein Polynom gefunden werden, das diesen Punkten möglichst nahe kommt, wobei eventuelle Messungenauigkeiten ausgeglichen werden. Die Koeffizienten des Polynoms werden numerisch bestimmt, indem die Summe über die quadratischen Abweichungen über alle Punkte minimiert wird. Dazu wird folgende Formel verwendet:

$$\min_{f \in \Pi_m^d} \sum_i \left(f(\mathbf{x_i}) - f_i\right)^2 \tag{6.1}$$

Wobei x_i der i-te Punkt der Punktwolke und f ein beliebiges Polynom aus dem Polynomraum Π mit der Dimension d und dem Grad m ist. Nach der Formel wird für jeden Punkt der quadratische Fehler berechnet und aufsummiert. Das hat den Vorteil, dass sich Fehler mit entgegengesetztem Vorzeichen nicht aufheben können.

In Matrixschreibweise formen die Messwerte ein lineares Gleichungssystem (GS) $Ax = b$ mit $A \in \mathbb{R}^{n \times k}$, $x \in \mathbb{R}^k$ und $b \in \mathbb{R}^n$. Bei Gleichungen dieser Form gibt es drei verschiedene Möglichkeiten [HZ04]:

- $n < k$: Das GS hat mehr Unbekannte als Gleichungen (*unterbestimmtes* GS), was zur Folge hat, dass keine eindeutige Lösung existiert, sondern ein ganzer Vektorraum von Lösungen.
- $n = k$: Sofern die Matrix A invertierbar ist, kann eine eindeutige Lösung durch $x = A^{-1}b$ berechnet werden.
- $n > k$: Das GS hat mehr Gleichungen als Unbekannte (*überbestimmtes* GS). Im Allgemeinen lässt sich ein solches GS nicht exakt lösen.

Wir betrachten den letzten Fall ($n > k$), d.h. die Matrix A hat mehr Zeilen, als Spalten. Da im Allgemeinen keine eindeutige Lösung existiert, muss ein Verfahren zur Approximation bemüht werden, um einen Lösungsvektor x zu berechnen, welcher eine möglichst gute Näherung der Lösung darstellt. Mit anderen Worten: Der Vektor x, der

$$\min \|\mathbf{Ax} - \mathbf{b}\|^2 \tag{6.2}$$

erfüllt, wobei $\|\cdot\|$ für den euklidischen Abstand steht, liegt sehr nahe an der Lösung. Allgemein lässt sich ein solches Problem lösen, indem man $f(x)$ zunächst als

$$f(\mathbf{x}) = \mathbf{b}(\mathbf{x})^T\mathbf{c} = \mathbf{b}(\mathbf{x}) \cdot \mathbf{c} \tag{6.3}$$

notiert [Nea04]. Hier ist $\mathbf{b}$ der Basisvektor, der die Vielfachen von Potenzen einer Variablen beinhaltet und $\mathbf{c}$ ist ein Vektor mit den zugehörigen Koeffizienten, welche gesucht sind. Die rechte Seite der Gleichung lässt sich nun in die Minimierungsgleichung 6.1 einsetzen. Anschließend werden die partiellen Ableitungen nach jedem c_i bestimmt und gleich 0 gesetzt. Das Ergebnis kann zu

$$\sum_i 2\mathbf{b}(\mathbf{x_i})[\mathbf{b}(\mathbf{x_i})^T\mathbf{c} - f_i] = 0 \tag{6.4}$$

zusammengefasst werden [Nea04]. Nach kleineren Umformungen von Gleichung 6.4 und der Auflösung nach c erhält man

$$\mathbf{c} = \left[\sum_i \mathbf{b}(\mathbf{x_i})\mathbf{b}(\mathbf{x_i})^T\right]^{-1} \sum_i \mathbf{b}(\mathbf{x_i})f_i. \tag{6.5}$$

In der Notation von Gleichung 6.2 ist das äquivalent zu

$$\mathbf{x} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}. \tag{6.6}$$

Damit hat man ein Gleichungssystem, das weder unter- noch überbestimmt ist und auf deren linker Seite der gesuchte Koeffizientenvektor steht. Die rechte Seite der Gleichung kann in Gleichung 6.3 eingesetzt werden, um das gesuchte Polynom direkt zu berechnen [Nea04].
Die *Weighted Least Squares* (WLS) Methode stellt eine Erweiterung zu LS dar und liefert ebenfalls eine Approximation für eine gegeben Punktwolke Der Unterschied besteht darin, dass die Terme der Fehlerfunktion mit einem Faktor gewichtet werden. Allgemein lässt sich die WLS-Fehlerfunktion durch

$$\min_{f \in \Pi_m^d} \sum_i \|f(\mathbf{x_i}) - f_i\|^2 \, \theta(x) \tag{6.7}$$

beschreiben. θ ist eine Gewichtungsfunktion, die nicht festgelegt ist. Häufig wird eine Gaußfunktion verwendet, bei der entfernte Punkte wenig Einfluss auf das Ergebnis haben. Es gilt: $\lim_{s\to\infty} \theta(s) = 0$. Die Berechnung des Polynoms mittels WLS verläuft analog zur Berechnung mittels LS, da der einzige Unterschied in den Gewichtungsfaktoren besteht.

6.2. Cholesky-Zerlegung

Die Cholesky-Zerlegung dient zur Lösung eines linearen GS, dessen zugrunde liegende Matrix sowohl positiv definit, als auch symmetrisch ist. Aus diesem Grund kann es auch verwendet werden, um die positive Definitheit einer Matrix zu überprüfen. Das Cholesky-Verfahren besteht aus drei Schritten:

(1) Zerlegung der Matrix A in ein Produkt zweier zueinander transponierter Dreiecksmatrizen: $A = LL^T$.

    Der folgende Algorithmus beschreibt die Zerlegung der Matrix A in die obere bzw.

untere Dreiecksmatrix [Car03]:

```
for k = 1 to n − 1 do
    if a_{k,k} ≤ 0 then
        return;
    else
        l_{k,k} = √a_{k,k};
        for s = k + 1 to n do
            l_{s,k} = a_{s,k} / l_{k,k}
        for j = k + 1 to n do
            for i = j to n do
                a_{i,j} = a_{i,j} − l_{i,k} l_{j,k}
```

(2) Vorwärtseinsetzen: $Ly = b \Longleftrightarrow Ly - b = 0$

Nach der Zerlegung kann man das Produkt der zwei Matrizen für A einsetzen und erhält damit $LL^T x = b$. Das Produkt $L^T x$ kann durch ein (bisher unbekanntes) y substituiert werden. Da L eine linke Dreiecksmatrix ist, ist die Berechnung von y durch Vorwärtseinsetzen leicht möglich.

(3) Rückwärtseinsetzen: $L^T x = y \Rightarrow L^T x - y = 0$

Analog lässt sich aus $L^T x = y$ das x berechnen, da y im vorangegangenen Schritt bestimmt wurde.

Das Verfahren hat den Vorteil, dass es numerisch sehr stabil ist, da bei der Zerlegung der Matrix die Quadratwurzel gezogen wird, wodurch der Wertebereich eingeschränkt wird.

6.3. Eigenwerte und Eigenvektoren

Das sogenannte *Eigenwertproblem* muss in vielen Gebieten, wie z.B. der Statik, dem Maschinenbau oder der numerischen Mathematik, gelöst werden. Nachfolgend wird nur auf die Grundgedanken der Eigenwerte bzw. Eigenvektoren von reellwertigen Matrizen eingegangen.
Bei der Eigenwertanalyse geht es um die Lösungen eines Gleichungssystems der Form:

$$Ax = \lambda x \tag{6.8}$$

Dabei ist $\lambda \in \mathbb{R}$, $A = (a_{ij})$ eine reelle $(n \times n)$-Matrix und x ein Spaltenvektor, der nicht der Nullvektor ist. Diese Einschränkung wird getroffen, da der Nullvektor sonst zu jedem reellen λ Eigenvektor wäre. Laut Definition ist jedes λ, das die obige Gleichung erfüllt ein *Eigenwert* und jeder Vektor x der zugehörige *Eigenvektor*. Die Eigenvektoren x_i werden unter A linear auf λx abgebildet, wobei die Abbildung eine Skalierung der Vektoren um den Faktor λ bewirken. Ist x_i Eigenvektor zu einem Eigenwert λ_i, so ist auch ax_i ein Eigenvektor zu λ_i, wobei a ein skalarer Wert ist [Dör88].
Zur Eigenwertberechnung stellt sich zunächst die Frage, ob Eigenwerte zur Matrix A existieren und wie diese berechnet werden. Diese Fragen lassen sich durch Umstellen von Gleichung 6.8 beantworten.

$$\begin{aligned} Id(x) = x \quad &\Rightarrow \quad Ax = Id(\lambda x) && (6.9)\\ &\Leftrightarrow \quad Ax - Id(\lambda x) = 0 && (6.10)\\ &\Leftrightarrow \quad (A - \lambda Id)x = 0 && (6.11) \end{aligned}$$

Sollte der Term $(A - \lambda Id)$ invertierbar sein, so ergibt sich daraus:

$$x = (A - \lambda Id)^{-1} 0 = 0 \tag{6.12}$$

Das bedeutet, dass λ (laut Definition) kein Eigenwert sein kann. Also darf $(A - \lambda Id)$ nicht invertierbar sein, was wiederum bedeutet, dass

$$P(\lambda) = det(A - \lambda Id) = \begin{vmatrix} a_{11} - \lambda & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} - \lambda & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} - \lambda \end{vmatrix} = 0 \tag{6.13}$$

sein muss, damit die Eigenwertanalyse auf der Matrix A durchgeführt werden kann. Bei der Entwicklung der Determinante, stellt man fest, dass sich ein Polynom n-ten Grades ergibt. $P(\lambda)$ heißt auch das *charakteristische Polynom* der Matrix. Über den Fundamentalsatz der Algebra lässt sich argumentieren, dass n Nullstellen existieren. Der Fundamentalsatz der Algebra besagt, dass ein Polynom vom Grad n insgesamt n Nullstellen besitzt, wobei Nullstellen mehrfach auftreten können (*Vielfachheit* von Nullstellen) [Dör88]. Nimmt λ den Wert einer dieser Nullstellen ein, so ist es ein Eigenwert. Im Fall $A \in \mathbb{R}^{3\times3}$ kann die Berechnung durch Ausnutzung besonderer Eigenschaften sehr effizient durchgeführt werden [Ebe06].

Die Berechnung der Eigenvektoren bei bekannten Eigenwerten reduziert sich auf die Lösung eines linearen Gleichungssystems mit n Unbekannten und n Gleichungen. Um den jeweiligen Eigenvektor x_i zu berechnen, genügt es, den Eigenwert λ_i in Gleichung 6.8 einzusetzen und das entstandene Gleichungssystem zu lösen. Da $x = 0$ per Definition ausgeschlossen wurde, liefert der Vektor x eine nicht-triviale Lösung des LGS und ist damit ein gesuchter Eigenvektor zum Eigenwert λ.

6.4. Gradientenabstieg

Der *Gradientenabstieg* bezeichnet ein numerisches Verfahren zur Optimierung eines Minimierungsproblems, das äquivalent zu $Ax = b$ ist. Dabei bewegt man sich von einem initialen Startwert x_0 und tastet sich in jedem Iterationsschritt näher an das Minimum heran. Um die richtige Richtung herauszufinden wird die geometrische Eigenschaft des Gradienten ausgenutzt, da dieser in die Richtung des steilsten Anstieges zeigt. D.h. es muss von einem Startpunkt ausgehend, in der Richtung des negativen Gradienten (Richtung des steilsten Abstieges) gesucht werden. Daher wird das Verfahren auch als *Verfahren des steilsten Abstiegs (steepest descent)* bezeichnet. Es wird zunächst eine Richtung $d = -\nabla f(x_0)$ bestimmt und f auf der Geraden $g = x_0 + t \cdot d$ minimiert. In dem gefundenen Minimum x_1 wird dann erneut der negative Gradient berechnet, bis das Verfahren nach endlich vielen Schritten ein lokales Minimum gefunden hat, bzw. gegen ein solches konvergiert. Um nicht zu viele Iterationen des Algorithmus durchführen zu müssen, wird oft ein Schwellwert definiert. Sinkt die Verbesserung des Wertes unter diesen Schwellwert, ist keine numerische Verbesserung mehr feststellbar und das Verfahren wird abgebrochen. Der Nachteil des einfachen Gradientenabstieges liegt darin, dass es im Allgemeinen nur langsam konvergiert, da der Weg vom Startpunkt zum Optimum einem Zick-Zack-Kurs in ggf. vielen kleinen Schritten ähnelt.
Ein besseres Verfahren zur Bestimmung des Optimums ist der *konjugierte Gradientenabstieg*. Dabei wird im jeweils nächsten Iterationsschritt nicht die Richtung $-\nabla f(x_0)$ gewählt, sondern es wird eine Richtung generiert, die von dem aktuellen Gradienten und den vorhergegangenen Richtungen abhängig ist. Zunächst wird ein Startvektor g_0 gewählt, dann werden in jedem Iterationsschritt zwei neue Richtungen g_{i+1} und h_{i+1} berechnet [PVTF02].

$$g_{i+1} = g_i - \lambda_i A \cdot h_i \tag{6.14}$$

$$h_{i+1} = g_{i+1} + \gamma_i h_i \tag{6.15}$$

Wobei die Richtungen linear unabhängig voneinander (also orthogonal) sind. Die Skalare sind durch

$$\lambda_i = \frac{g_i \cdot h_i}{h_i \cdot A \cdot h_i} \tag{6.16}$$

$$\gamma_i = \frac{g_{i+1} \cdot g_{i+1}}{g_i \cdot g_i} \tag{6.17}$$

gegeben. Diese Form des CG-Verfahrens wird *Fletcher-Reeves* Algorithmus genannt. Ein weiteres Beispiel eines solchen Verfahrens ist der *Polak-Ribiere* Algorithmus, bei dem lediglich die Berechnung des γ_i modifiziert wird [PVTF02]:

$$\gamma_i = \frac{(g_{i+1} - g_i) \cdot g_{i+1}}{g_i \cdot g_i} \tag{6.18}$$

Der Vorteil des CG-Verfahrens liegt darin, dass es garantiert nach höchstens n Schritten konvergiert, wobei n die Dimension der gegebenen Matrix beschreibt. Ein spezieller Fall des konjugierten Gradientenabstieges ist der bikonjugierte Gradientenabstieg. Das Ziel liegt in der Lösung eines linearen Gleichungssystems, welches nicht notwendiger Weise positiv definit oder symmetrisch sein muss [PVTF02].

6.5. Raycasting

Raycasting ist ein Verfahren, um 3D-Szenen mit Hilfe von Strahlen (*rays*) zu visualisieren und stellt eine Alternative zum polygonbasierten Ansatz dar. Wie Abb. 6.1 zeigt, werden Strahlen von der Beobachterposition ausgehend, durch jedes Pixel der Bildebene in die Szene emittiert. Durch Schnittpunktbestimmung kann entschieden werden, ob für das jeweilige Pixel der Bildebene ein Objekt der Szene sichtbar ist, oder nicht.

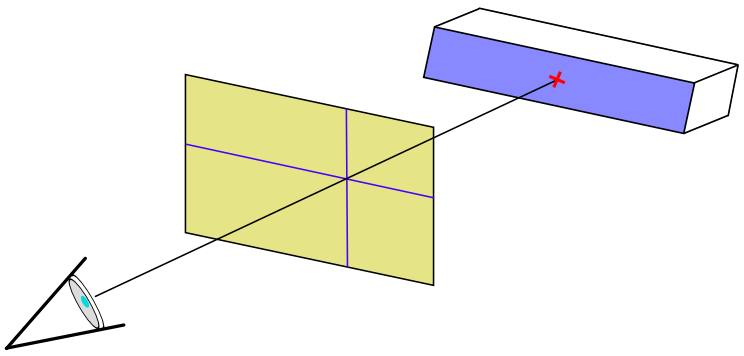


Abbildung 6.1.: Skizze des Raytracing Verfahrens.

Trifft ein Strahl ein oder mehrere Objekte der Szene, so muss ggf. zunächst der nächst gelegene Schnittpunkt berechnet werden. Das Objekt zu welchem der berechnete Schnittpunkt gehört, ist für die aktuelle Betrachterposition sichtbar und trägt mindestens an dem aktuell bearbeiteten Pixel der Bildebene bei. Den Materialeigenschaften des Pixels entsprechend kann das Pixel gefärbt werden. Trifft der Strahl kein Objekt, so wird das Pixel auf die Hintergrundfarbe der Szene gesetzt. Das Raycasting-Verfahren stellt eine simplifizierte Form des Raytracings und ein effizientes Renderverfahren dar. Der Unterschied zum aufwendigeren Raytracing liegt darin, dass keine Strahlen an einer Oberfläche der Szene reflektiert und weiterverfolgt werden. Raycasting kann theoretisch auf jede Form von Oberflächen angewendet werden, allerdings steigt je nach Komplexität der Oberfläche auch der Berechnungsaufwand. Daher wird das Verfahren meist nur auf einfache Geometrien angewendet.
In umgekehrter Richtung kann Raycasting angewendet werden, indem für jeden Voxel des Objekts berechnet wird, auf welchen Pixel der Bildebene er projiziert wird. Wichtig ist dabei die Berücksichtigung von Tiefeninformationen und Transparenz, da sonst verdeckte Objekte dargestellt werden. Raycasting bietet den Vorteil, dass auch prozedurale Geometrien bzw. Volumina dargestellt werden können, welche die polygonbasierte Pipeline nicht direkt anzeigen kann.

Kapitel 7.

Oberflächenapproximation

Die vorliegende Arbeit benutzt die im vorangegangenen Kapitel vorgestellten Verfahren zur Approximation von Oberflächen, welche durch die aufgenommenen Raumpunkte der PMD-Kamera beschrieben werden. Die Auflösung der PMD-Kamera - und damit auch die Anzahl der Raumpunkte - ist recht gering. Dies hat zur Folge, dass nur relativ wenige Punkte zur Verfügung stehen, um die Oberfläche zu approximieren. Werden nur die Punkte dargestellt, so sind die Lücken zwischen den Punkten so groß, dass der visuelle Eindruck darunter stark leidet. Eine einfache Interpolation zwischen den Raumpunkten ist nicht geeignet, um das Problem zu lösen und eine gute Approximation der Oberfläche zu erhalten. Das Ergebnis wäre zum einen sehr ungenau und zum anderen alles andere als glatt. Angestrebt wird eine glatte Oberfläche, die die tatsächliche Oberfläche so gut wie möglich approximiert. Dabei muss beachtet werden, dass Datensätze der PMD-Kamera einem Rauschen unterliegen, welches sich in Ungenauigkeiten der Punktpositionen, sowie in sogenannten *flying pixels* niederschlägt. Diesen Effekten muss entgegengewirkt werden, da die Approximation ansonsten negativ beeinflusst wird. Die softwareseitige Erzeugung weiterer Punkte wird in einem späteren Kapitel (Punktbasiertes Raycasting) diskutiert.
Die vorliegende Arbeit verwendet WLS-Oberflächen, welche eine approximierende oder interpolierende Oberfläche darstellt. Die Oberfläche S ist definiert durch eine Punktmenge, die genau diejenigen Punkte enthält, welche durch einen Projektionsoperator Ψ auf sich selbst abgebildet werden.

$$S := \{x \in B : \Psi(x) = x\} \tag{7.1}$$

Dabei ist B eine Punktmenge, die genau diejenigen Punkte enthält, welche sich in der Nachbarschaft des Punktes x befinden [ABCO+01]. Also muss zunächst ein Projektionsoperator gefunden und anschließend jeder Raumpunkt, welcher auf den PMD-Daten beruht, mit Hilfe dieses Operators projiziert werden. Ist der Ergebnispunkt mit dem Eingabepunkt identisch, gehört er zur Definitionsmenge der WLS-Oberfläche. Die Bestimmung von Ψ kann in zwei Schritte aufgeteilt werden, welche nachfolgend beschrieben

werden.
Zunächst wird eine lokale Referenzebene für jeden einzelnen PMD-Punkt gesucht, um die Oberfläche lokal über eine Funktion approximieren zu können. Diese Ebene muss möglichst nahe sowohl an dem aktuell betrachteten Punkt, wie auch an den umgebenden Nachbarpunkten liegen. Um diese Ebene zu bestimmen, wird (statt eines MLS-Ansatzes, wie bei Levin [Lev03]) ein WLS-Ansatz benutzt. In einem zweiten Schritt wird ein bivariates Polynom (von zwei Variablen abhängig) gesucht. Ist das Polynom gefunden, wird der jeweilige Raumpunkt auf dieses lokale Polynom projiziert. Da das Verfahren auf lokalen Polynomapproximationen beruht, kann die Genauigkeit der Oberflächenapproximation lokal definiert werden. Des Weiteren ist es möglich, niederfrequentes Rauschen leicht handzuhaben, da die lokalen Approximationen über Least Squares Ansätze berechnet werden [TGN+01].

7.1. Berechnung lokaler Ebenen

Für jeden Punkt wird eine lokale Ebene gesucht, die sich möglichst gut dem aktuellen Punkt, sowie den Nachbarpunkten anpasst. Dazu wird ein Punkt q und eine Normale n gesucht, welche die Ebene definieren. Zur Berechnung mit Hilfe des WLS-Verfahrens muss eine entsprechende Fehlerfunktion minimiert werden (Gleichung 7.2).

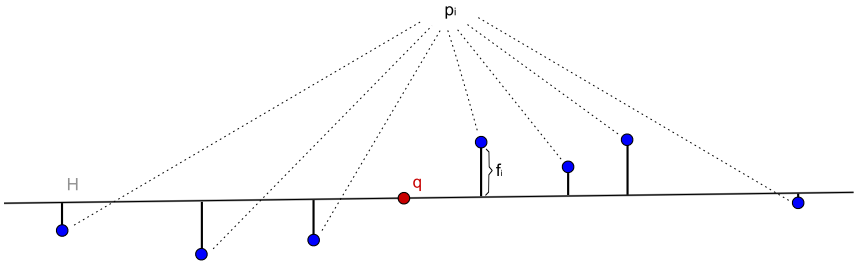


Abbildung 7.1.: Aufstellen einer lokalen Ebene durch den Punkt q.

Im Anschluss folgt die Bestimmung eines orthonormalen Koordinatensystems für die Ebene, dessen Ursprung in p liegt (siehe Abb. 7.1). Um die Ebene vollständig zu definieren, wird ein Punkt der Ebene, sowie die Normale gesucht. Dazu wird die Nachbarschaft des entsprechenden Punktes betrachtet. Die Energiefunktion

$$\min_{n,t} \sum_{p_i \in \mathbf{N}(\mathbf{r})} \langle p_i - (r + tn), n \rangle^2 \, \omega(\|p_i - p_w\|) \tag{7.2}$$

ist ein Maß für die Qualität der Position der lokalen Ebene im Raum. Wobei $\langle \cdot, \cdot \rangle$ für das innere Produkt (Skalardrodukt) und ω für eine Gewichtungsfunktion steht. p_i ist

der jeweilige Nachbarpunkt und p_w ein Gewichtungspunkt. Erreicht die Funktion ihr Minimum, so wurden der Vektor n, sowie der Skalarwert t so gewählt, dass die Entfernungen der Nachbarpunkte zur Ebene möglichst gering sind. D.h. die Ebene schmiegt sich bestmöglich an den Punkt und dessen Umgebung an. Bei dem WLS-Verfahren ist der Gewichtungspunkt ein fixer Referenzwert, für den Tejada und Ertl et al. [TGN+01] den aktuellen Punkt r vorschlagen. Da die Gewichte $\omega(\|p_i - p_w\|)$ entsprechend $\|p_i - p_w\|$ abnehmen, werden die Punkte umso stärker gewichtet, je näher sie am aktuellen Punkt liegen. Als Gewichtungsfunktion kann z.B. eine Gaußfunktion verwendet werden:

$$\omega(d) = e^{-\frac{d^2}{h^2}} \tag{7.3}$$

Um triviale Ergebnisse bei der Minimierung zu vermeiden ($n = 0$) wird weiterhin die Nebenbedingung $\|n\| = 1$ benötigt. Zur Vereinfachung machen Tejada et al. [TGN+01] bei der Berechnung der Ebene die Annahme, dass $t = 0$ und damit $r+tn = r$ ist, wodurch die Referenzebene immer durch diesen Punkt verläuft. Da die lokale Ebene sehr nahe an dem zugehörigen Punkt r liegt, ist diese Vereinfachung möglich. Die vorliegende Arbeit verwendet stattdessen:

$$q = r + tn = \frac{1}{m} \cdot \sum_i p_i = \overline{p} \tag{7.4}$$

Wobei m für die Anzahl der Nachbarpunkte steht, so dass $\overline{p}$ den Mittelwert aller Nachbarpunkte p_i ergibt. Intuitiv ergibt die Mittelwertbildung Sinn, da die Ebene gesucht ist, die den aktuellen Punkt sowie dessen Nachbarpunkte so gut wie möglich approximiert. Ein Punkt, der über die Nachbarpunkte gemittelt wird, ist damit der optimale Punkt q für die Ebene.
Das Lösen des Least-Squares Problems lässt sich durch Verwenden der Methode der Lagrange-Multiplikatoren auf ein Eigenwertproblem zurückführen [PGK02]. Die Kovarianzmatrix C des aktuellen Punktes r entspricht $A^T A$ mit

$$A = \begin{bmatrix} (p_{0_x} - \overline{p}_x) & (p_{0_y} - \overline{p}_y) & (p_{0_z} - \overline{p}_z) \\ (p_{1_x} - \overline{p}_x) & (p_{1_y} - \overline{p}_y) & (p_{1_z} - \overline{p}_z) \\ \vdots & \vdots & \vdots \\ (p_{n_x} - \overline{p}_x) & (p_{n_y} - \overline{p}_y) & (p_{n_z} - \overline{p}_z) \end{bmatrix}. \tag{7.5}$$

Der Übersichtlichkeit halber wurden die Gewichte $\sqrt{\omega_i} = \sqrt{\omega(\|p_i - p_w\|)}$ bei jedem Element der Matrix weggelassen. Die Eigenvektoren v_i der Matrix stehen laut Definition orthogonal aufeinander und entsprechen den Hauptachsen der Punktwolke, welche durch die Nachbarpunkte gebildet wird. Die Eigenwerte stellen die Varianz der einzelnen Nachbarpunkte entlang der Richtung des zugehörigen Eigenvektors dar [Han02]. Danach ist

der Eigenvektor zum kleinsten Eigenwert der Richtungsvektor in dessen Richtung die Varianz am kleinsten ist und damit der gesuchte Normalenvektor.
Die Ebene, die durch den Mittelwertpunkt verläuft und den Eigenvektor zu dem kleinsten Eigenwert (v_0) als Normale besitzt, minimiert die Summe der quadrierten Abweichungen. D.h. dass die Ebene, die durch die Eigenvektoren v_1 und v_2 die optimale Approximation der lokalen Ebene aufspannt. Für gewöhnlich ist der kleinste Eigenwert und damit auch der zugehörige Eigenvektor eindeutig. Sollte der kleinste Eigenwert mehrfach auftreten (*Vielfachheit* von Nullstellen), so ist die Normale n nicht wohldefiniert.
Zur Berechnung des Eigenwertproblems existieren verschiedene iterative numerische Verfahren, allerdings kann hier der Sonderfall ausgenutzt werden, dass es sich bei der Kovarianzmatrix um eine 3×3 Matrix handelt [Ebe06].

$$0 = -det(A - \lambda I) = -det \begin{pmatrix} a_{00} - \lambda & a_{01} & a_{02} \\ a_{01} & a_{11} - \lambda & a_{12} \\ a_{02} & a_{12} & a_{22} - \lambda \end{pmatrix} = \lambda^3 - c_2\lambda^2 + c_1\lambda - c_0 \quad (7.6)$$

Da das charakteristische Polynom vom Grad 3 ist, können bei der Nullstellenberechnung Vorteile ausgenutzt werden, so dass ein nicht iterativer Ansatz möglich ist. Die Koeffizienten des Polynoms berechnen sich durch

$$c_0 = a_{00}a_{11}a_{22} + 2a_{01}a_{02}a_{12} - a_{00}a_{12}^2 - a_{11}a_{02}^2 - a_{22}a_{01}^2 \quad (7.7)$$
$$c_1 = a_{00}a_{11} - a_{01}^2 + a_{00}a_{22} - a_{02}^2 + a_{11}a - 22 - a_{12}^2 \quad (7.8)$$
$$c_2 = a_{00} + a_{11} + a_{22} \quad (7.9)$$

Nach einigen Berechnungen können die Eigenwerte λ_0 bis λ_2 direkt berechnet werden [Ebe06]. Da die 3×3 Kovarianzmatrix C symmetrisch und positiv definit ist, sind alle Eigenwerte λ_i reellwertig.
Es bleibt zu überlegen, wie der Skalierungsfaktor h (der Gewichtungsfunktion ω) sinnvoll zu wählen ist. Wie in [Koc07] beschrieben, hängt die Wahl von h von der entsprechenden Szene ab. Grundsätzlich gibt es zwei Möglichkeiten: Ein konstanter (globaler) oder ein adaptiver (lokaler) Wert. Die Wahl eines konstanten Wertes hat den Nachteil, dass er für eine Szene heuristisch ermittelt werden muss. Ein weiterer Nachteil bei einem konstanten Skalierungsfaktor besteht bei nicht uniform verteilten Abtastpunkten [Koc07]. Denn bei Bereichen in denen eine hohe Dichte von Vertices vorliegt (z.B. bei Laserscanner-Daten), werden auch viele Punkte berücksichtigt, wodurch der Berechnungsaufwand steigt. In Bereichen mit wenigen abgetasteten Punkten dagegen (z.B. PMD-Daten), fließen eventuell zu wenige Punkte in die Berechnung mit ein, wodurch die Berechnung instabil wird und im Endeffekt eine falsche Oberflächenapproximation liefert. Wird ein konstanter

Wert für h gewählt, so hängt dieser von der Dichte der Abtastung ab.
Sowohl im Falle einer unregelmäßig abgetasteten Szene, als auch bei Echtzeitanwendungen einer unbekannten Szene, ist die Wahl eines adaptiven Parameters h der bessere Weg. In [PGK02] wird für jeden Punkt ein spezifisches h berechnet, indem zunächst die *k-Nachbarschaft* berechnet wird. Dann kann der Parameter durch $h = r/3$ berechnet werden, wobei r der Radius der *bounding sphere* der Nachbarpunkte ist. Mit dem Faktor h ist es möglich, den Grad der Glättung der Oberfläche zu bestimmen, denn je kleiner der Parameter gewählt wird, desto lokaler wird die Approximation.

7.2. Polynombestimmung

Im nächsten Schritt wird ein lokales, bivariates Polynom $g(x, y)$ bestimmt, welches die Oberfläche möglichst gut approximiert (*Polynomfitting*). Die Approximation stützt sich auf die Position des aktuellen Punktes r sowie dessen Nachbarpunkte p_i.

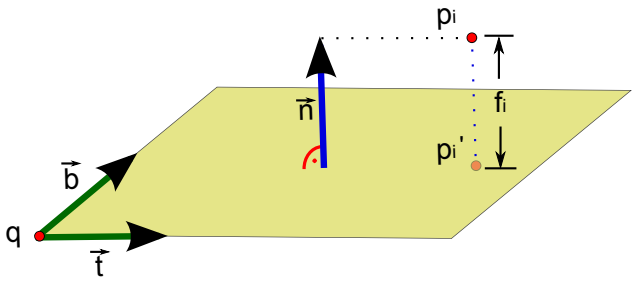


Abbildung 7.2.: Projektion auf die Referenzebene.

Um eine möglichst gute Annäherung an die tatsächliche Oberfläche zu erhalten, muss erneut eine Fehlerfunktion minimiert werden.

$$\sum_{p_i \in N(r)} (g(x_i, y_i) - f_i)^2 \omega(\|p_i - q\|) \tag{7.10}$$

Sei p_i' die Projektion des Punktes p_i auf die (im vorherigen Schritt berechnete) lokale Referenzebene. (x_i, y_i) sind die Koordinaten von p_i' bzgl. des lokalen 2D-Koordinatensystems, dessen Ursprung sich in q befindet. $f_i = \|p_i' - p_i\|$ ist eine Funktion, welche die Höhe des i-ten Nachbarpunktes über der lokalen Referenzebene angibt (siehe Abb. 7.2). p_i' berechnet sich durch

$$p_i' = \begin{pmatrix} \langle b, p_i - q \rangle \\ \langle t, p_i - q \rangle \\ \langle n, p_i - q \rangle \end{pmatrix}. \tag{7.11}$$

D.h. hier findet eine Viewing-Transformation statt, die den Ursprung des Koordinatensystems (aufgespannt aus den Vektoren b, t und n) in q transliert und p_i auf die drei Hauptachsen projiziert. Die Gewichtungsfunktion ω ist dieselbe, die bei der Approximation der lokalen Ebene benutzt wurde (z.B. Gauß-Funktion) und das Polynom $g(x, y)$ ist definiert als

$$g(x, y) = b(x, y) \cdot c \tag{7.12}$$

Wobei der Basisvektor b(x, y) geschickt gewählt und der Vektor c, der die zugehörigen Koeffizienten beinhaltet, berechnet werden muss. Die Minimierung der Fehlerfunktion liefert als Ergebnis den Vektor der Koeffizienten c, womit das Polynom vollständig definiert ist. Dazu wird ein Gleichungssystem für alle projizierten Nachbarschaftspunkte aufgestellt, welches die Form $Ax = b$ hat. Dabei ist x der Vektor, der die gesuchten Koeffizienten beinhaltet.
Eine Möglichkeit zur Lösung des Problems bietet das Cholesky-Verfahren (Kapitel 6.2). Angewendet auf die Matrix $A^T A$ liefert dieses zunächst zwei Dreiecksmatrizen, mit denen der Koeffizientenvektor effizient berechnet werden kann. Zur Erinnerung: Die Bedingungen für die Anwendbarkeit des Verfahrens sind Symmetrie und positive Definitheit der Matrix. Beide dieser Bedingungen sind offensichtlich erfüllt, so lange die Diagonalelemente der Matrix $\neq 0$ sind. Ein weiteres Verfahren zur Minimierung, das von Tejada und Ertl et al. [TGN+01] vorgeschlagen wird, ist der konjugierte Gradientenabstieg (Kapitel 6.4). Die vorliegende Arbeit verwendet im Allgemeinen das Cholesky-Verfahren und greift nur im Falle des Scheiterns des Verfahrens auf den konjugierten Gradientenabstieg zurück.

7.3. Projektion

Das Polynom, welches die Oberfläche lokal approximiert, ist vollständig bestimmt. Um den Punkt der Oberfläche zu berechnen, wird der Punkt q der lokalen Ebene H auf das Polynom projiziert. Wie in Abb. 7.3 skizziert, wird er entlang der bereits berechneten Normale verschoben. Der Projektionsoperator $\Psi(r)$ wird daher folgendermaßen definiert:

$$\Psi(r) := q + g(0, 0)n \tag{7.13}$$

Die Länge des Verschiebungsvektor wird durch den Skalierungsfaktor $g(0, 0)$ berechnet. D.h. das lokale Polynom wird im Ursprung, also an der Stelle $(0, 0)$, ausgewertet. $g(0, 0)$ entspricht damit einem Höhenwert bzw. der Entfernung der Oberfläche im Punkt q. Bis auf den konstanten Term beinhalten alle anderen Terme des Polynoms mindestens

einmal den Faktor x bzw. y. D.h. wird für $x = 0$ und $y = 0$ gewählt, so fallen alle diese Terme weg und es bleibt nur der konstante Wert übrig.

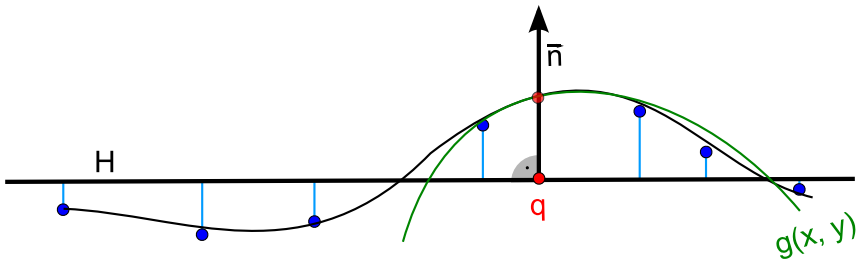


Abbildung 7.3.: Polynombestimmung und Projektion auf das lokale Polynom.

Die Wahl des Polynoms ist durch das Verfahren nicht eingeschränkt, so dass theoretisch der gesamte Polynomraum zur Verfügung steht. In der Praxis wird die Polynomwahl einerseits durch die Anzahl der Einzelterme beschränkt, da sowohl der Berechnungs-, als auch der Speicheraufwand mit jedem zusätzlichen Term steigt. Andererseits muss berücksichtigt werden, dass z.B. lineare Terme nicht gut geeignet sind, um eine Kugeloberfläche zu beschreiben. Die vorliegende Arbeit verwendet das Polynom

$$g(x, y) = Ax^2 + By^2 + Cxy + D. \tag{7.14}$$

Die quadratischen Terme (Ax^2 und By^2) eignen sich, um eine gekrümmte Oberfläche zu approximieren, während der Term Cxy eine gekippte Ebene in xy-Richtung und D eine konstante Verschiebung darstellt. Durch Kombination und Gewichtung der Einzelterme lassen sich vielfältige Oberflächen (lokal) beschreiben.

Kapitel 8.

Punktbasiertes Raycasting

Das in Kapitel 7 vorgestellte Verfahren ist geeignet, um eine Oberfläche aus einer gegebenen Punktwolke zu rekonstruieren. Um das visuelle Ergebnis zu verbessern, ist eine dichtere Punktwolke erforderlich. Zum Einfügen zusätzlicher Punkte in die Punktwolke wird oft eine Vorverarbeitung und eine Datenstruktur verwendet. Um beides zu vermeiden und eine interaktive Rekonstruktion zu gewährleisten, wird ein punkbasiertes iteratives Raycasting-Verfahren vorgeschlagen. Neben der Erhöhung der Auflösung, wird die Darstellung der Oberfläche der Beobachterposition angepasst. Der Grundgedanke der dem Verfahren zugrunde liegt, besteht darin, die gesuchten Oberflächenpunkte zu bestimmen, indem die Schnittpunkte des Strahls, durch eine Projektion der Punkte von dem Strahl auf die Oberfläche, iterativ konvergieren. In jedem Iterationsschritt wird eine Schnittberechnung des Strahls mit einem lokalen Polynom durchgeführt. Da das Polynom die Oberfläche lokal approximiert, liegt der Schnittpunkt von Strahl und Polynom nahe an dem Schnittpunkt von Strahl und Oberfläche [AA03]. Durch eine iterative Anwendung wird die Genauigkeit der lokalen Oberflächenapproximationen und damit die Oberflächenpunkte schrittweise verbessert.
In diesem Kapitel wird ein Algorithmus vorgestellt, der auf der Arbeit von Tejada und Ertl et al. [TGN+01] basiert. Ein Teil des Algorithmus ist ähnlich dem bereits diskutierten Verfahren aus Kapitel 7. Der Hauptunterschied besteht in der Verwendung des Raycasting Verfahrens, sowie einer iterativen Herangehensweise, die durch Eigenschaften des Algorithmus, bzw. dessen Implementierung bedingt ist. Bevor mit der iterativen Berechnung begonnen wird, ist ein Initialschritt notwendig. Im Folgenden werden die einzelnen Schritte des Algorithmus erläutert, wobei auf Implementierungsdetails in Kapitel 9 eingegangen wird.
Das Ziel des initialen Schrittes besteht darin, für jeden PMD-Punkt eine erste Approximation des lokalen Polynoms zu finden. Dieses kann im weiteren Verlauf des Algorithmus dazu verwendet werden, um weitere Punkte der Oberfläche zu berechnen. Um initiale Werte für die lokale Ebene und das Polynom zu bestimmen, wird sehr ähnlich vorgegangen wie bei dem Algorithmus aus Kapitel 7. D.h. es wird zunächst die Nachbarschaft für jeden Punkt p_i bestimmt, die Kovarianzmatrix zur Normalenbestimmung aufgestellt und eine Eigenwertanalyse durchgeführt. Ein Unterschied zum Algorithmus von Tejada

und Ertl et al. [TGN+01] besteht in Bestimmung der Nachbarschaftsinformationen, da Tejada und Ertl et al. auf einem bereits bekannten Datensatz arbeiten und die Nachbarschaften vorberechnen und speichern können. Die vorliegende Arbeit arbeitet auf Videostreams, so dass die Nachbarschaften effizient zur Laufzeit bestimmt werden müssen.
Die naive Suche der Nachbarpunkte kann über den euklidischen Abstand des aktuellen Punktes und jedes anderen Abtastpunktes bestimmt werden. Bei 19.200 Punkten wären demnach $19.200 \cdot 19.199$ Abstandsprüfungen notwendig. Allerdings lässt sich ein Großteil dieser Berechnungen einsparen, da keine allgemeine Punktwolke gegeben ist, sondern die Nachbarschaft auf der Bildebene ausgenutzt werden kann.
Der Normalenvektor ist dann der Eigenvektor zum kleinsten Eigenwert der berechneten Kovarianzmatrix $C = A^T A$ (siehe Kapitel 7.1). Durch die Normale und einen Punkt $q = r + tn$ ist die lokale Ebene vollständig definiert, wobei hier $q = \overline{p}$ gewählt wird (siehe Kapitel 7.1). Im nächsten Schritt soll das lokale Polynom gefunden werden, welches das LS-Problem so gut wie möglich löst. Eine Möglichkeit zur Lösung des Problems bietet das in dieser Arbeit verwendete WLS-Verfahren. Die Projektion des aktuellen Punktes auf sein lokales Polynom entfällt, da nur eine erste Approximation der lokalen Polynome gesucht wird. Das Ergebnis dieses initialen Renderdurchlaufes ist bereits eine Approximation der lokalen Polynome, welche zur ersten Schnittpunktbestimmung der Strahlen dienen.
Im iterativen Teil des Algorithmus wird das Raycasting-Verfahren umgesetzt, um weitere Oberflächenpunkte zu bestimmen. Dabei wird eine iterative Herangehensweise verwendet, bei der die berechneten Oberflächenpunkte in jedem Schritt weiter verfeinert werden. Zunächst wird ein möglicher Schnittpunkt des jeweiligen Strahls mit der Oberfläche gesucht. Dazu wird der nächst gelegene Schnittpunkt des Strahls mit den lokalen Polynomen berechnet, deren erste Approximationen bereits im initialen Schritt berechnet wurden. Die implizite Nachbarschaft wird durch Splatrendering bestimmt. Dazu werden am Viewport ausgerichtete Scheiben gerendert. Durch jedes der entstandenen Fragmente wird ein Strahl geschossen, dessen Schnittpunkt mit dem jeweiligen lokalen Polynom berechnet wird. Um den korrekten Schnittpunkt zu berechnen, muss der Strahl zunächst in das lokale Koordinatensystem transformiert werden, welches durch die Vektoren bestimmt ist, die bereits mit Hilfe der Eigenwertanalyse berechnet wurden (Normale, Binormale und Tangente). Mit der Gleichsetzung von Strahl und Polynom wird der Schnittpunkt über das Lösen des entstandenen Gleichungssystems berechnet. Statt der ursprünglichen Anzahl von $160 \cdot 120$ Punkte liegt jetzt eine deutlich höhere (von der Fenstergröße abhängige) Anzahl von Punkten vor. Die Punkte sind nur erste Approximationen der Oberflächenpunkte, aus denen im nachfolgenden Teil des Algorithmus die Oberflächenpunkte durch eine Projektion auf das PSS berechnet werden.
Zuerst wird durch jeden der berechneten Schnittpunkte eine Ebene gefittet, wobei die lokale Polynom- und Oberflächenberechnung analog zu Kapitel 7 ist. Allerdings muss die Nachbarschaftssuche entsprechend angepasst werden, da die 2D-Informationen (wie im Initialschritt) nicht ausgenutzt werden können. Eine naive Suche wäre leicht umzuset-

zen, indem die euklidische Distanz von jedem Punkt der Punktwolke zu jedem anderen Punkt berechnet wird. Dagegen spricht allerdings das Argument der Geschwindigkeit, da diese Herangehensweise entsprechend ineffizient ist. Tejada und Ertl et al. [TGN+01] schlagen eine effiziente Möglichkeit vor, welche in gewisser Weise aus der entgegen gesetzten Richtung an die Suche heran geht, wie der bereits diskutierte Algorithmus. Es werden nicht für jeden einzelnen Punkt alle weiteren Punkte auf Nachbarschaft geprüft, sondern es wird geprüft, zu welchen Punkten der aktuelle Punkt in Nachbarschaftsbeziehung steht. Wurden die Nachbarpunkte bestimmt, kann die Kovarianzmatrix aufgebaut und die Normalen über eine erneute Eigenwertanalyse berechnet werden. Die Anzahl der Nachbarpunkte wird gespeichert, so dass Punkte, die weniger Nachbarn haben als ein festgelegter Schwellwert, als Ausreißer markiert werden können. Bei diesen Punkten sind keine weiteren Berechnungen mehr erforderlich, wodurch zum einen das visuelle Ergebnis und zum anderen die Performanz verbessert wird, da weniger Instruktionen ausgeführt werden müssen. Um die Normale mit Hilfe der Kovarianzmatrix zu bestimmen, wird der Eigenvektor zum kleinsten Eigenwert berechnet. Da es sich aber als geschickt herausstellen wird, auch die Binormale und Tangente zu berechnen, werden hier alle drei Eigenvektoren berechnet und jeweils in eine Float-Textur geschrieben.
Da die lokalen Referenzebenen bekannt sind, kann mit dem Polynomfitting begonnen werden. Um die Vorteile der GPU im Umgang mit (bis zu 4×4) Matrizen nutzen zu können, wird das Polynom $Ax^2 + By^2 + Cxy + D$ (vgl. Gleichung 7.14) gewählt, dessen Koeffizienten c_0 bis c_3 durch Aufstellen und Lösen eines Gleichungssystems berechnet werden. Analog zu Kapitel 7 wird die Matrix A^TA, sowie der Vektor A^Tb aufgebaut. Mit Hilfe des Cholesky-Verfahrens (Kapitel 6.2) wird aus A^TA und A^Tb ein Vektor bestimmt, der die Koeffizienten beinhaltet. Da das lokale Polynom nun vollständig definiert ist, kann der bereits berechnete Schnittpunkt r mit Gleichung 7.13 auf das Polynom projiziert werden, wobei der Punkt q dem r entspricht. Da die Projektion approximativ senkrecht zu der Oberfläche steht, kann der Abstand zwischen der Projektion r' und dem Schnittpunkt r als Schätzung für den Abstand von r zu der Oberfläche genutzt werden [AA03]. Ist der Abstand zwischen r' und r kleiner als ein vordefinierter Schwellwert, dann ist der Schnittpunkt des Strahls mit der Oberfläche bereits gefunden: r. Damit ist das Abbruchkriterium erreicht und es sind keine weiteren Iterationen notwendig. Ist der Abstand größer, wird der Schnittpunkt zwischen Strahl und dem lokalen Polynom berechnet und als Eingabe für den nächsten Iterationsschritt verwendet.
Die nächste Iteration startet bei der Schnittpunktberechnung, die weitestgehend unverändert bleibt. Je nach Ergebnis des letzten Iterationsschritt gibt es drei Möglichkeiten zur Weiterverarbeitung:

- Iterationsende: Die Abbruchbedingung wurde bereits erfüllt, somit werden keine weiteren Berechnungen mehr durchgeführt.
- Gültiger Schnittpunkt: Das Abbruchkriterium wurde bisher noch nicht erfüllt und der bisher berechnete Schnittpunkt ist ein gültiger Punkt innerhalb der Kugel

(mit Mittelpunkt im aktuellen PMD-Punkt). In diesem Fall werden ebenfalls keine weiteren Berechnungen durchgeführt und der berechnete Schnittpunkt wird in den folgenden Schritten weiterverwendet.

- Ungültiger Schnittpunkt: Das Abbruchkriterium wurde bisher noch nicht erfüllt und der berechnete Schnittpunkt ist ungültig, d.h. es wurde kein Schnittpunkt gefunden oder der Schnittpunkt liegt außerhalb der Kugel. In diesem Fall muss ein neuer Schnittpunkt berechnet werden. Zur Erinnerung: Zur Schnittpunktberechnung eines Strahls mit einem lokalen Polynom werden zunächst Scheiben gerendert und dann Strahlen durch jedes Fragment geschossen. Dann wird der Schnittpunkt zwischen Strahl und dem Polynom berechnet, welches zum Mittelpunkt der Scheibe gehört. Wurde die Scheibe falsch gewählt, muss dieser Schritt erneut durchgeführt werden, mit dem Unterschied, dass nur Scheiben in Betracht kommen können, die hinter der Scheibe des letzten Iterationsschritts liegen. [TGN+01]

Kapitel 9.

Implementierungsdetails

In diesem Kapitel werden einige ausgewählte Implementierungsdetails des Algorithmus aus Kapitel 8 näher erläutert. Die Implementierung findet Verwendung in einem bestehenden Bildverarbeitungs-Framework in Form eines *WLSViewer*-Bausteines. Als Programmiersprache wurden C++ und GLSL verwendet. Teilweise werden bereits existierende, effiziente Implementierungen von Algorithmen z.B. aus [PVTF02] verwendet.
Als Eingabedaten sind zur Durchführung des Algorithmus im Prinzip nur eine Textur notwendig, welche die Daten der einzelnen PMD-Punkte enthält, sowie die Kalibrierungsdaten der PMD-Kamera. Die Kalibrierungsdaten beinhalten u.a. Fokuslänge, Position des Mittelpunktes und die Größe eines PMD-Pixels. Zusätzliche Eingabedaten werden verwendet, um die Punkte des Resultats am Ende der Verarbeitung entsprechend einzufärben. Zur Farbgebung des Resultats besteht neben der Möglichkeit eines aufwendigen Splatting-Verfahrens, bei dem die gewichteten Werte der Nachbarpunkte zu einem Farbwert aufaddiert werden, auch die Möglichkeit, die aufgenommenen Intensitätswerte der PMD-Kamera odereine zusätzlichen RGB-Kamera zu verwenden. Da die Intensitätswerte der PMD-Kamera (wie auch die Tiefeninformationen) eine niedrige Auflösung besitzen, anhand derer kein gutes Ergebnis möglich ist, verwendet die vorliegende Arbeit ein höher aufgelöstes Farbbild, das von einer RGB-Kamera aufgenommen wird, welche direkt auf der PMD-Kamera angebracht ist. Als weitere Eingabedaten werden die RGB-Daten in Form einer Textur, die Kalibrierungsdaten der RGB-Kamera, sowie eine Transformationsmatrix übergeben. Die Transformationsmatrix wird benötigt, um Punkte im Koordinatensystem der PMD-Kamera in das Koordinatensystem der RGB-Kamera zu transformieren. Bei der Implementierung hat sich gezeigt, dass das Ergebnis des Algorithmus durch das starke Rauschen der PMD-Daten negativ beeinflusst wird. Daher wird der Datensatz durch einen Bilateralfilter vorverarbeitet, wodurch eine Glättung erreicht wird und der Algorithmus bessere Ergebnisse liefert.
Der gesamte Algorithmus ist in mehrere Renderdurchläufe (*Multipass Rendering (MPR)*) aufgeteilt, welche iterativ durchlaufen werden. Eine Ausnahme stellt dabei der erste und der letzte Renderschritt dar, welche jeweils nur einmal ausgeführt werden. Die Trennung in mehrere Renderdurchläufe ist notwendig, da Teile des Algorithmus zu einem Zeitpunkt auf die Informationen mehrere Fragmente zugreifen müssen. Da dies nicht mög-

lich ist, werden die Ergebnisse in einem Schritt gesammelt, bevor sie in dem eigentlichen Berechnungsschritt verwendet werden. Abb. 9.1 stellt den kompletten Ablauf des Algorithmus mit den verwendeten Shader-Einheiten pro Renderdurchlauf dar. Im Folgenden wird noch einmal kurz auf diese Schritte eingegangen, wobei das besondere Augenmerk auf interessanten Problemstellungen und deren konkreten Umsetzungen liegt. Auf Umsetzungen, wie z.B. des letzten Renderdurchlaufes, der lediglich der Visualisierung der Ergebnisse dient, wird nicht eingegangen.

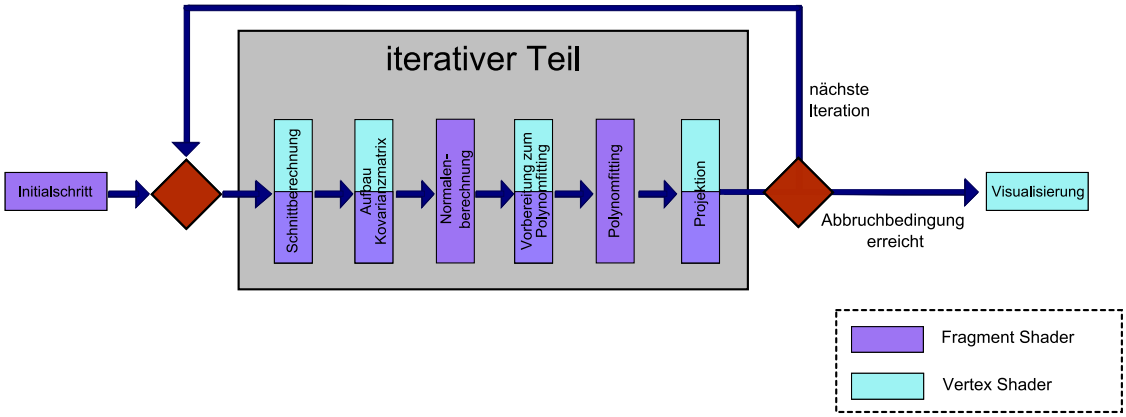


Abbildung 9.1.: Ablauf des implementierten Algorithmus. Jeder Schritt stellt einen Renderdurchlauf dar.

9.1. Initialschritt

Folgende Parameter werden an den Fragment Shader der initialen Berechnung übergeben: Der aufgenommene Datensatz in Form eines *sampler2DRect*, ein konstanter Schwellwert zur Nachbarschaftsprüfung, ein Wert für den Parameter h, der vom Benutzer über die graphische Oberfläche festgelegt werden kann, sowie Breite und Höhe der Eingabetextur. Zur Berechnung der ersten Approximation der lokalen Polynome wird ein Quad gerendert und rasterisiert, so dass pro Texel der Eingabetextur ein Fragment erzeugt wird. Die Berechnungen werden komplett im Fragment Shader durchgeführt.

9.1.1. Nachbarschaftsbestimmung

In der Arbeit von Tejada et al. [TGN+01] ist die Bestimmung der Nachbarschaft der Punkte des Datensatzes zur Laufzeit nicht notwendig, da auf bekannten Datensätzen gearbeitet wird, deren Nachbarschaftsbeziehungen in Texturen vorgespeichert werden können. Der Algorithmus der vorliegenden Arbeit verwendet Videostreams, so dass die

benötigten Informationen zur Laufzeit berechnet werden müssen. Die Daten liegen in der Eingabetextur des Bausteins vor und es kann die Eigenschaft ausgenutzt werden, dass alle Punkte in der Bildebene (mit einer Auflösung von 160 × 120) vorliegen.

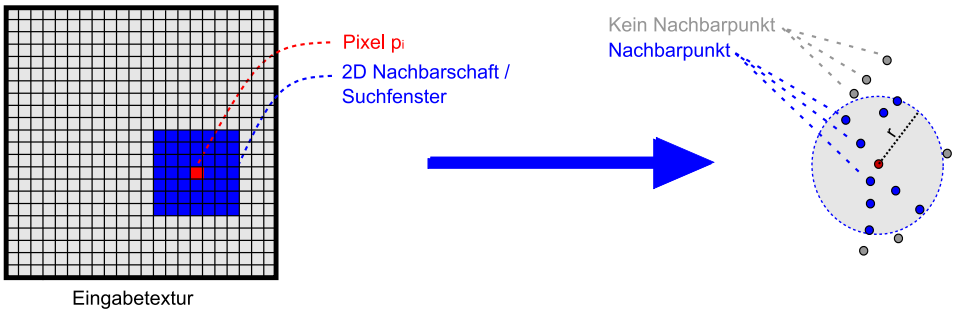


Abbildung 9.2.: Suche der Nachbarpunkte von p_i unter Ausnutzung von 2D-Informationen. Linke Seite: Definition eines Suchfensters zur Festlegung von Nachbarschaftskandidaten. Rechte Seite: Prüfung auf 3D-Nachbarschaft durch Berechnung des euklidischen Abstandes.

Notwendige Bedingung für die Nachbarschaft zweier Punkte im Raum ist die 2D Nachbarschaft und damit die Nachbarschaft der entsprechenden Texel (linke Seite von Abb. 9.2). D.h. es reicht, ein Suchfenster um jedes Texel zu definieren, in dem Nachbarn gesucht werden müssen (linke Seite von Abb. 9.2). Die aktuelle Implementierung verwendet ein Suchfenster der Größe 5, somit gibt es 120 Nachbarschaftskandidaten. Ein Nachbarschaftskandidat ist genau dann ein Nachbar, wenn sein Abstand zum betrachteten Punkt kleiner ist, als ein vordefinierter Schwellwert. Um dies zu prüfen, werden die Raumkoordinaten des aktuellen Punktes und der Kandidaten aus der Textur gelesen und der euklidische Abstand bestimmt (rechte Seite von Abb. 9.2).
Bei der Implementierung ist ein weiteres Detail zu beachten, welches in Abb. 9.3 dargestellt wird: Punkte die näher beim Betrachter liegen sind dichter beieinander, als Punkte die weiter entfernt sind. D.h. dass auch der Schwellwert zur Nachbarschaftsprüfung entsprechend der Tiefe angepasst werden muss. Dazu werden zwei verschiedene Abstandsprüfungen durchgeführt: Zunächst wird der xy-Abstand bestimmt und durch Multiplikation mit der negativen z-Komponente skaliert, um sicher zu gehen, dass Punkte, die weiter entfernt sind auch einen höheren Abstand in x- bzw. y-Richtung haben dürfen, um trotzdem Nachbarn zu sein. Als nächstes wird der Abstand in z-Richtung bestimmt und mit einem Schwellwert verglichen.

Die Nachbarpunkte werden in einem *vec3*-Array gespeichert, damit sie im nächsten Verarbeitungsschritt zur Verfügung stehen.

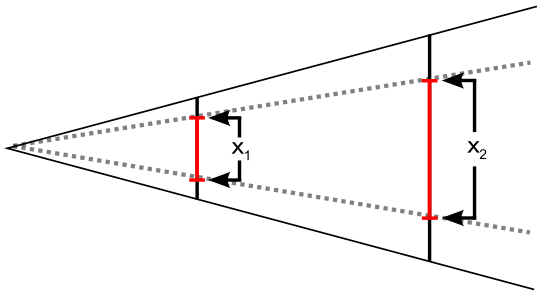


Abbildung 9.3.: Anpassung des Suchfensters für die Nachbarschaftssuche. Der Abstand x_1 auf der nahen Ebene ist geringer, als x_2 auf der weiter entfernten Ebene, obwohl die Punkte auf denselben Strahlen liegen.

9.1.2. Berechnung der lokalen Ebene

Zur Berechnung der lokalen Ebene wird die Kovarianzmatrix aufgebaut. Dazu wird zunächst der Abstand zwischen jedem Nachbarpunkt und dem Gewichtungspunkt bestimmt.

```
1    if (length(neighbors[i].xy - actual.xy) > -float(areaSize)*actual.z*
         maxDistance) continue;
2    if (length(neighbors[i].z - actual.z) > float(areaSize)*maxDistance)
         continue;
3
4    float length2D = length(weighter.xy - neighbors[i].xy);
5    float lengthZ  = abs(weighter.z - neighbors[i].z);
6
7    float weightXY = gaussian(length2D, smoothing_h*(-weighter.z));
8    float weightZ  = gaussian(lengthZ, smoothing_h);
9
10   float weight = weightXY * weightZ;
```

Listing 9.1: Berechnung des Gewichtungsfaktors durch das Produkt der Einzelgewichte der xy- bzw. der z-Richtung.

Listing 9.1 zeigt die Berechnung des Gewichtes, mit dem der Einfluss der Nachbarpunkte auf den aktuellen Punkt realisiert werden. Nach der Berechnung des Gewichtes, kann die Kovarianzmatrix aufgebaut werden (siehe Kapitel 7.2.1). Zur Erhöhung der Genauigkeit werden die Punkte vorher mit dem Faktor 100.0 skaliert, da die Einheit Meter verwendet wird und im Shader keine *double* Genauigkeit möglich ist. Beim Aufbau der Matrix wird die Symmetrie der Matrix ausgenutzt, um die Effizienz der Berechnung zu steigern. Es wird nur die obere Dreiecksmatrix berechnet, die verbliebenen Werte werden lediglich aus dem bereits berechneten Teil ausgelesen. Die Bestimmung der Normalen wird über eine Eigenwertanalyse der Kovarianzmatrix erreicht. Statt die *inverse power* Methode zu verwenden, wie Tejada et al. [TGN+01] und nur den Normalenvektor zu bestimmen,

wird in der vorliegenden Arbeit ein numerisch stabiler, nicht-iterativer Ansatz von Eberly [Ebe06] verwendet (siehe Kapitel 7.2.1).
Der Algorithmus wurde in GLSL portiert und wird komplett im Fragment Shader abgearbeitet. Die Funktionsweise ist unverändert, es wurden lediglich kleinere Änderungen vorgenommen. Anstatt der Matrix-Klassen wurden die im Shader vorhandenen, hardwareunterstützten Matrizen verwendet, analog wurden 3-elementige Arrays durch *vec3* ersetzt. Zur Berechnung der inversen Wurzel ($\frac{1}{\sqrt{x}}$) verwendet Eberly eine kurze, aber nicht triviale Methode, welche hier aufgrund der hardwareunterstützer Funktionen nicht notwendig ist. Beachtet werden muss außerdem, dass Matrizen in GLSL *column major* sind und daher Spalten- und Zeilenindizes vertauscht werden müssen.
Alle drei Eigenvektoren werden gespeichert, wobei der Vektor zum kleinsten Eigenwert der Normalen entspricht. Die anderen zwei Vektoren entsprechen der Binormalen und der Tangente der lokalen Ebene und finden im nächsten Schritt ihre Verwendung.

9.1.3. Lokales Polynomfitting

Zum Fitten des lokalen Polynoms wird zunächst die Projektion der Punkte des Datensatzes auf die berechneten lokalen Ebenen benötigt. Dazu wird jeder Punkt auf die Ebene projiziert und in das lokale Koordinatensystem transformiert. Zur Transformation werden neben der Normale auch die anderen, zusätzlich berechneten Eigenvektoren (Binormale, Tangente) verwendet (siehe Listing 9.2).

```
11    for (int i = 0; i < index; i++)
12    {
13      vec3 v = neighbors[i] - actual.xyz;
14      projNeighbors[i] = vec3(dot(binormal.xyz, v), dot(tangent.xyz, v),
            dot(normal.xyz, v));
15    }
```

Listing 9.2: Transformation in das lokale Koordinatensystem.

Über die Definition von Makros kann entschieden werden, ob das Polynomfitting über das Cholesky-Verfahren oder das *biconjugate gradient* Verfahren berechnet wird. Als Mittelweg kann das Cholesky-Verfahren benutzt werden, wann immer es möglich ist und das Gradientenverfahren in allen anderen Fällen. Zunächst wurde ein Pseudoinversenverfahren implementiert, allerdings ergab ein Vergleich mit den beiden anderen Verfahren, dass es sowohl von der Effizienz, als auch von dem Ergebnis her nicht so gut geeignet ist und wurde daher nicht weiter verfolgt.
Die Algorithmen für das *biconjugate gradient* und das Cholesky Verfahren wurden aus [PVTF02] übernommen und in GLSL umgeschrieben. Bei dem Algorithmus für die *biconjugate gradient* Methode konnten zur Effizienzsteigerung einige *if*-Abfragen entfernt werden, ohne die Funktionalität zu beeinträchtigen. Bei der Originalversion kann über

einen Parameter das Konvergenzkriterium ausgewählt werden, welches in der hier verwendeten Implementierung einfach festgelegt wurde als:

$$\left|\widetilde{A} \cdot (A \cdot x - b)\right| / \left|\widetilde{A}^{-1} \cdot b\right| < tol \tag{9.1}$$

wobei $\widetilde{A}$ der diagonale Teil der Matrix A ist [PVTF02].

Zur Weitergabe des Ergebnisses - also der Koeffizienten des Polynoms - werden FBOs verwendet. *Framebuffer Objects* (FBO) bieten eine Möglichkeit, um Ergebnisse eines Renderdurchlaufes zu speichern, damit sie in einem nächsten Schritt wieder verfügbar sind. Bei FBO handelt es sich um eine OpenGL Erweiterung (EXT_framebuffer_object), die ab OpenGL Version 1.1 verfügbar ist und ein plattformunabhängiges Offscreen-Rendering möglich macht. Zu einem FBO kann ein sogenanntes *framebuffer-attachable image* (ein 2D Pixel-Array) hinzugefügt werden, welches sowohl als Quelle, als auch als Ziel von Fragmentoperationen dienen kann. Wird es als Renderziel genutzt, so wird damit eine Art des *offscreen rendering* möglich. D.h. das Ziel in welches gerendert wird, ist nicht wie standardmäßig ein Fenster, sondern das framebuffer-attachable image.

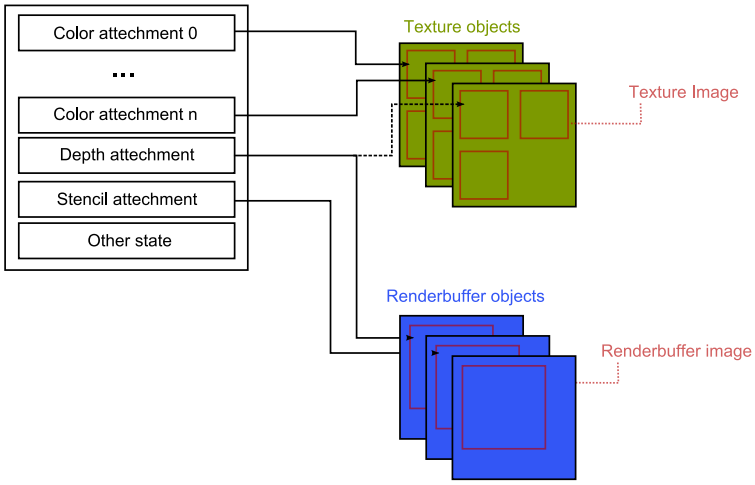


Abbildung 9.4.: Framebuffer Object Architektur. Abb. aus [Ast05]

Auch *Render To Texture* (RTT) wird unterstützt, wenn wie hier Texturen als framebuffer-attachable images benutzt werden (siehe Abb. 9.4) [AA06]. RTT erlaubt es, den Inhalt des Framebuffers direkt in eine Textur zu schreiben und wird oft für MPR verwendet. Der Vorteil liegt darin, dass der aufwendige Kopiervorgang vom Framebuffer in eine Textur eingespart wird, wenn direkt in eine Textur gerendert wird. Das in dieser Arbeit verwendete Polynom beschränkt sich auf 4 Koeffizienten ($g(x, y) = Ax^2 + By^2 + Cxy + D$).

Durch die Verwendung der vier Texturkanäle (RGBA) reicht eine einzige Float-Textur aus, um alle Koeffizienten zu speichern.

9.2. Iterativer Teil

Der iterative Teil des Algorithmus besteht insgesamt aus sechs Renderdurchläufen, in welchen jeweils Scheiben oder Quads gerendert werden. Scheiben werden immer dann verwendet, wenn eine Berechnung pro PMD-Punkt benötigt wird, wie z.B. in den Schritten *Schnittberechnung*, *Aufbau der Kovarianzmatrix* und *Vorbereitung zum Polynomfitting*. Viewport füllende Quads werden in den Schritten *Normalenberechnung*, *Polynomfitting* und *Projektion* gerendert, um pro Fragment einen Strahl zu erstellen.

9.2.1. Schnittberechnung

Im zweiten Renderdurchgang wird per Raycasting pro Bildpunkt eine Approximation eines Oberflächenpunktes für das lokale Polynom bestimmt. Die Punkte werden über Schnittpunktberechnungen im Fragment Shader zwischen dem jeweiligen Strahl und den lokalen Polynomen ermittelt. Zur Bestimmung, mit welchem Polynom der Schnittpunkt berechnet werden muss, wird das Raycasting Verfahren in umgekehrter Richtung angewendet (siehe Kapitel 6.5). Die Idee besteht darin, Kugeln um jeden PMD-Punkt zu rendern, um eine implizite Nachbarschaft zu bestimmen. Über die Fragmentkoordinaten kann auf eine Textur zugegriffen werden, die die Koeffizienten des zugehörigen Polynoms beinhaltet. Da sich Kugeln ohne weiteres nicht mit OpenGL rendern lassen, werden Scheiben verwendet, die sich an der Beobachter Position ausrichten. Diese Scheiben werden via *Point Sprites* gerendert. Dabei handelt es sich um ein Feature, welches ab der OpenGL Version 1.5 verfügbar ist und eine spezielle Form von *Billboards* darstellt. Point Sprites sind zum Beobachter ausgerichtete Quadrate, die um jeweils einen Mittelpunkt aufgespannt werden. Bei einer Änderung der Betrachterposition wird die Rotation des Quadrates um diesen Punkt durchgeführt. Mit Point Sprites lassen sich durch das Rendern eines einzelnen Punktes beliebige 2D Texturen darstellen.
Der Vorteil bei der Verwendung von Point Sprites liegt darin, dass (bei genügend großer Anzahl) das Datenaufkommen, welches über den Graphikbus an die Graphikkarte geschickt wird, stark reduziert wird. Statt den vier Eckpunkten des Quads muss nur noch der Mittelpunkt des Sprites übermittelt werden (siehe Abb. 9.5). Alles weitere wird direkt von der Graphikhardware übernommen. Da auch die Texturkoordinaten automatisch erstellt werden, wird auch hier das Datenaufkommen reduziert. Des Weiteren muss die Ausrichtung an der Betrachterposition nicht manuell vorgenommen werden. Um Point Sprites zu verwenden, wird der state *GL_POINT_SPRITE* und der Parameter

der Texturumgebung mit *glTexEnvi(GL_POINT_SPRITE, GL_COORD_REPLACE, GL_TRUE)*[1] gesetzt.

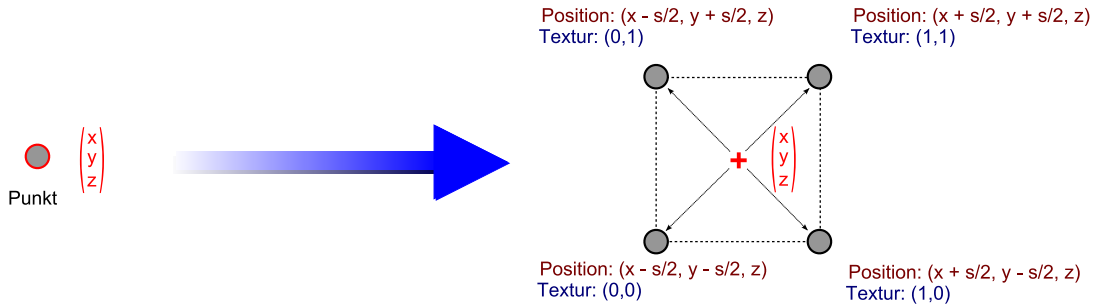


Abbildung 9.5.: Durch rendern eines Punktes werden auf der Graphikhardware 4 Punkte erzeugt, die ein Quadrat um den Mittelpunkt $(x, y, z)^T$ bilden.

Ein Problem stellt die Größe der Splats dar, die den Einflussbereich des jeweiligen Punktes angibt. Da der Benutzer sich in der Szene bewegen kann, muss die Größe in Abhängigkeit des aktuellen Frustums aktualisiert werden. In der vorliegenden Arbeit wurde die Anzahl der Pixel, die ein PMD-Pixel auf der Bildebene einnimmt, berechnet und per uniform-Variable an den Vertex Shader übergeben. Zur Berechnung wird zunächst die Höhe eines Bildschirmpixels über den Tangens des halben Öffnunswinkels (Öffnungswinkel: β in Abb. 9.6) im Bogenmaß (Faktor $\frac{\pi}{180}$) bestimmt. Die Anzahl der Pixel pro PMD-Pixel berechnet sich dann durch den Quotienten aus der PMD-Pixelhöhe und dem berechneten Wert. Zu beachten gilt außerdem die Umrechnung des PMD-Pixels in die Einheit Meter, da sich die Angaben der intrinsischen Daten auf die Einheit Millimeter beziehen (Faktor 0.001).

```
16 // OpenGL-Kontext
17 m_programPass1.setParameter1f(m_programPass1.getParameter("numb_pixel"),
       0.001 * m_pIntrinsics->sy() * height() / (0.02 * tan(M_PI * 15.0f /
       180.0f)));
18
19 //Vertex Shader
20 gl_PointSize = (numb_pixel*actual.z/(gl_ModelViewMatrix*actual).z);
```

Listing 9.3: Bestimmung der Splatgröße

In Listing 9.3 steht *sy* für die Höhe eines PMD-Pixels in mm und 0.02 für die doppelte Distanz von der Benutzerposition zur *near plane*. Der Faktor 2 erklärt sich dadurch, dass durch die Verwendung des halben Öffnungswinkels (α in Abb. 9.6) auch nur $\frac{w}{2}$ berechnet wird.

Da Scheiben statt Quadraten benötigt werden, wird auf jede Scheibe eine Alphatextur gemapped, bei der der Alphawert innerhalb des Inkreises den Wert 1.0 und außerhalb

[1] Der erste Parameter gibt die Texturumgebung an, der zweite Parameter ist der zu setzende Parameter und der letzte Parameter enthält den entsprechenden Wert.

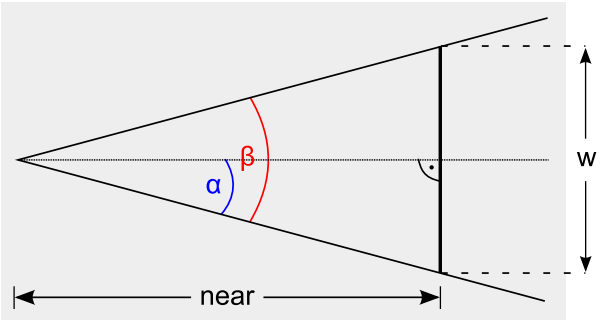


Abbildung 9.6.: Trigonometrie zur Splatgrößenberechnung.

des Kreises den Wert 0.0 hat. Im Fragment Shader wird ein Alphatest durchgeführt, der alle Fragmente verwirft, welche einen Alphawert von 0.0 haben.

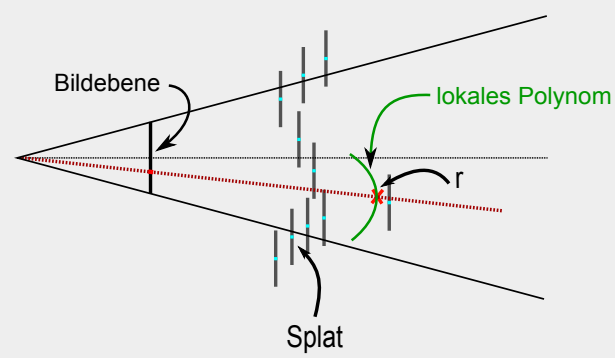


Abbildung 9.7.: Schnittpunktberechnung auf der GPU.

Durch jedes der verbliebenen Fragmente wird ein Strahl geschossen und der Schnittpunkt mit dem lokalen Polynom berechnet (siehe Abb. 9.7). Dazu muss zunächst der Strahl in das lokale Koordinatensystem transformiert werden. Die Transformation lässt sich mit der Multiplikation mit der Matrix

$$M = \begin{bmatrix} e1_x & e2_x & e3_x & v_x \\ e1_y & e2_y & e3_y & v_y \\ e1_z & e2_z & e3_z & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.2}$$

durchführen. Die Transformation von einem Koordinatensystem in ein anderes ist äquivalent mit der Veränderung der Perspektive. Für die entsprechende Rotation des Strahl sorgt die obere, linke 3×3 Teilmatrix, bestehend aus dem Normalenvektor ($e1 = (e1_x, e1_y, e1_z)^T$), der Binormalen ($e2 = (e2_x, e2_y, e2_z)^T$) und der Tangente ($e3 = (e3_x, e3_y, e3_z)^T$).

Da uns aber nicht nur die Richtung, sondern auch der Startpunkt des Vektors interessiert, muss auch eine Translation desselben stattfinden. Die Verschiebung, welche durch die letzte Spalte ($v = (v_x, v_y, v_z)^T$) erreicht wird, hat zur Folge, dass der Anfangspunkt des Strahls in dem aktuellen Punkt der Punktwolke liegt. Der Schnittpunkt kann durch Gleichsetzen des Polynoms mit der Geraden (Strahl) berechnet werden. Der Punkt, in dem Polynomgleichung und Geradengleichung übereinstimmen, muss sowohl auf dem Polynom, als auch auf der Geraden liegen und ist damit ein Schnittpunkt. Existiert keine Lösung für das Gleichungssystem, so existiert auch kein Schnittpunkt und das Fragment wird verworfen. Existiert genau eine Lösung, so muss geprüft werden, ob sich dieser innerhalb des Radius der Scheibe befindet. Ist dies nicht der Fall, so liegt der Punkt zu weit weg und das Fragment wird ebenfalls verworfen. Im Falle von zwei Lösungen für das Gleichungssystem, wird der Schnittpunkt r, welcher der Beobachterposition am nächsten gelegen ist, mittels eines Tiefentests ausgewählt und als Ergebnis in eine Textur gerendert.

9.2.2. Normalenberechnung

Durch die Implementierung einer *interaktiven* Rekonstruktion mittels eines Raycasting Ansatzes wird die Normalenberechnung algorithmisch komplizierter, wodurch eine Zweiteilung in zwei Renderpasses notwendig wird. Im ersten Renderpass wird die Kovarianzmatrix aufgebaut, die im zweiten Renderpass benutzt wird, um die Normale zu berechnen.

Aufbau der Kovarianzmatrix Im iterativen Teil des Algorithmus muss ein anderer Weg gefunden werden, um die Nachbarschaft eines Punktes zu berechnen, da der Benutzer die Möglichkeit hat, sich in der Szene zu bewegen. D.h. es ändert sich das View-Frustum und damit auch die Bildebene, was zur Folge hat, dass Punkte, die in der Eingabetextur benachbart waren, in der Eingabetextur des iterativen Teils nicht notwendiger Weise benachbart sein müssen. Die Nachbarschaftsbestimmung wird hier nach dem Vorschlag von Tejada et al. [TGN+01] durchgeführt.

Zunächst werden erneut am Viewport ausgerichtete Scheiben mit radialer Alphatextur via Point Sprites gerendert. Im Vertexshader werden die Mittelpunkte der Point Sprites an die richtigen Positionen transliert und die Größe analog zum vorangegangenen Schritt gesetzt. Der Rasterizer zerlegt diese in die einzelnen Fragmente, welche im Fragment Shader nach einem Alphatest bearbeitet werden. Für die verbliebenen Fragmente wird der Abstand zwischen dem zugehörigen Schnittpunkt des jeweiligen Fragments und dem Splatmittelpunkt bestimmt. Der Schnittpunkt wird aus der Output-Textur des vorangegangenen Renderdurchlaufes, die als *uniform*-Variable an den Shader übergeben

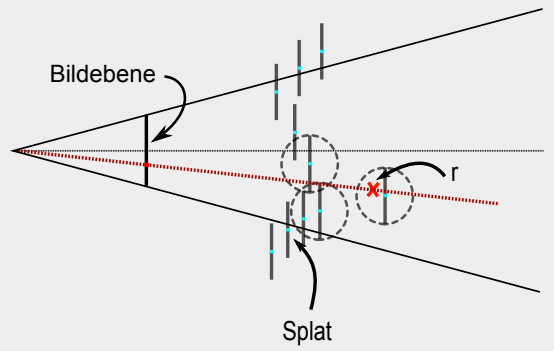


Abbildung 9.8.: Nachbarschaftsbestimmung im Raycasting Verfahren.

wird, an der Koordinate des aktuellen Fragments ausgelesen. Der Splatmittelpunkt ist der aktuell bearbeitete Vertex, der als *varying*-Variable an den Fragment Shader übergeben wird. Ist der Abstand kleiner als der gegebene Radius, so handelt es sich um einen Nachbarn (siehe Abb. 9.8), ansonsten wird das Fragment verworfen. Spätestens an dieser Stelle wird klar, warum ein einziger Renderpass nicht ausreicht, um die Normalen zu berechnen. Zur Erinnerung: Die Fragmente werden teilweise parallel verarbeitet und haben keinerlei Kenntnisse voneinander. D.h. eine Speicherung in einem Array (wie im initialen Schritt) ist hier nicht möglich.
Betrachtet man die Berechnung der Kovarianzmatrix aus Gleichung 7.5, so wird klar, dass jeder Nachbarpunkt einen Summanden zu den Einträgen der Matrix beiträgt. D.h. das Problem kann umgangen werden, indem die Einzelergebnisse für jeden Nachbarpunkt in einer Matrix akkumuliert werden, die mit 0-Werten initialisiert wurde. Im Unterschied zum initialen Schritt wird hier wie in [TGN+01] $t = 0$ und damit $q = r$ gewählt (siehe 7.1). Da das Ergebnis eine 3×3-Matrix ist, reicht eine einzige Textur nicht aus, um sie an die nächste Verarbeitungsstufe weiterzureichen. Die Lösung besteht in der Verwendung mehrere Texturen, die jeweils einen Teil der Matrix speichern. Mit *Multiple Render Targets* (MRT) ist es möglich, in einem einzigen Renderdurchlauf in mehrere Renderziele gleichzeitig zu rendern. GLSL bietet dafür das Array *gl_FragData[]*, um Daten als Output an mehrere Puffer zu schicken. Es werden 3 Float-Texturen verwendet, um die Kovarianzmatrix an die nächste Verarbeitungsstufe weiterzureichen (Abb. 9.9).

Jede Textur beinhaltet nach Verarbeitung des letzten Fragments jeweils eine Spalte der 3×3-Matrix, wobei jeder einzelne Eintrag in einem Kanal der Textur abgelegt wird. Durch Alphablending mit *glBlendFunc(GL_ONE, GL_ONE)* kann zu dem bereits vorhandenen Wert in der Textur der aktuelle Wert hinzuaddiert werden.
Gleichzeitig wird das Blending benutzt, um zu ermitteln, ob es sich bei dem aktuellen Punkt um einen Ausreißer handelt. Ein Ausreißer wird so definiert, dass die Anzahl seiner Nachbarpunkte unter einem bestimmten Schwelltwert liegt. Die Ermittlung der Anzahl der jeweiligen Nachbarpunkte wird über eine Akkumulation in der homogenen

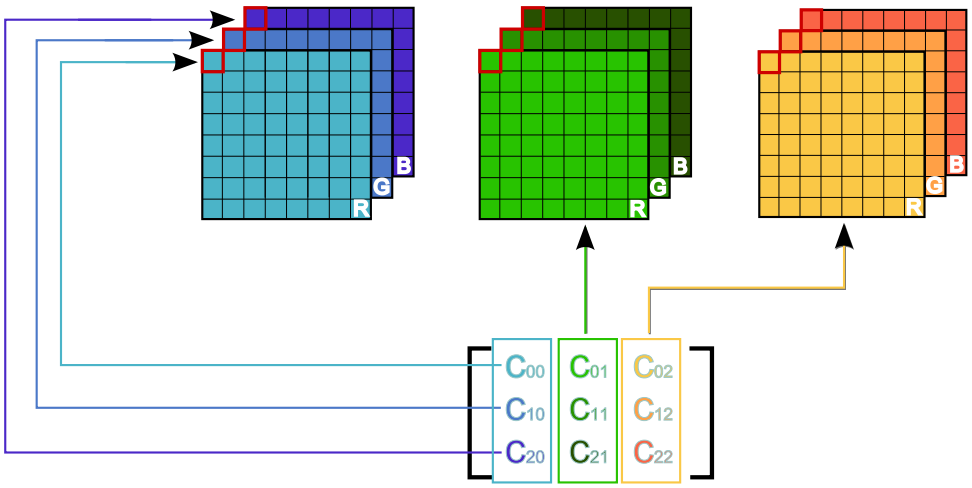


Abbildung 9.9.: Speichern der Kovarianzmatrix in 3 Float-Texturen

Koordinate erreicht. Da eine Akkumulation genau dann stattfindet, wenn eine Nachbarschaftsbeziehung vorliegt, kann der Wert 1.0 hinzuaddiert werden. Über einen Vergleich mit einem Schwellwert können die betroffenen Punkte markiert werden. In den folgenden Schritten des Algorithmus kann anhand der Markierung in der homogenen Koordinate entschieden werden, ob weitere Berechnungen durchgeführt werden müssen oder nicht. Die Implementierung lässt einen variablen Schwellwert zu, der über die GUI definiert werden kann.

Normalenberechnung Da die Kovarianzmatrix bereits berechnet wurde, können die Eigenwerte durch das Lösen des Eigenwertproblems bestimmt werden. Die Texturen, die als Renderziele im letzten Schritt benutzt wurden, dienen hier als Eingabe über 3 *uniform*-Variablen vom Typ *sampler2DRect*. Aus diesen Daten wird die 3×3 Matrix aufgebaut. Die eigentliche Berechnung verläuft analog der Berechnung im initialen Schritt.

9.2.3. Weitere Iterationen

Im Idealfall wurde bereits der gesuchte Punkt des PSS gefunden, so dass keine weiteren Iterationen notwendig sind. Ist dies nicht der Fall, wird mit dem nächsten Iterationsschritt bei der Schnittpunktberechnung gestartet. Zur Prüfung ob der berechnete Punkt bereits der gesuchte Punkt des PSS ist, wird dieser auf das lokale Polynom projiziert (siehe Gleichung 7.13) und der Abstand zwischen der Projektion und dem Originalpunkt gemessen.

```
21  vec4 localOrigin = vec4(intersect.xyz, 1.0);
22  vec4 c = texture2DRect(coeffs, gl_FragCoord.st);
23  vec4 normal = texture2DRect(normals, gl_FragCoord.st);
24
25  float marker = -1.0;
26  vec4 proj = intersect + vec4(c.w*normal.xyz, 0.0);
27
28  if(length(proj - intersect) < threshold)  marker = 2.0;   //the
        intersection with the PSS is already found
```

Listing 9.4: Test auf Abbruchbedingung

Die Berechnung (siehe Listing 9.4) wird direkt im Fragment Shader im Schritt *Projektion* durchgeführt. Ist der Abstand kleiner als ein vom Benutzer festgelegter Schwellwert, der über die GUI festgelegt werden kann, so handelt es sich bei dem berechneten Punkt bereits um einen Punkt des PSS, so dass keine weiteren Iterationen mehr notwendig sind. Der Punkt wird in der homogenen Koordinate entsprechend markiert (hier mit 2.0). Anhand dieser Markierung kann nachher entschieden werden, ob eine weitere Iteration notwendig ist, oder nicht. Ist der Abstand größer, wird ein neuer Schnittpunkt r mit dem lokalen Polynom bestimmt. Die homogene Koordinate wird auf den Wert 1.0 gesetzt, als Hinweis, dass der Schnittpunkt noch nicht der endgültige Punkt der Oberfläche ist. In beiden Fällen werden die Koordinaten des Schnittpunktes (inklusive der Markierung) in eine Float-Textur gerendert. Da die Entscheidung für jeden Punkt einzeln getroffen werden muss, kann die Schleife über die Iterationen im OpenGL-Kontext nicht direkt beendet werden. Daher wird im Fragment Shader jedes Renderpasses eine Abfrage der homogenen Koordinaten der Punkte durchgeführt. Da wenige Iterationen ausreichen sollten, um die gesuchten Schnittpunkte zu finden, ist der Overhead an Instruktionen recht gering. Es wird eine Konstante eingeführt, die vom Benutzer über die GUI festgelegt werden kann und für die maximale Anzahl von Durchläufen steht. Wird diese Anzahl erreicht, wird der Algorithmus abgebrochen.
Die nächste Iteration beginnt bei der Schnittberechnung. Sofern das Iterationsende noch nicht erreicht ist, hängt der neue Schnittpunkt von den Ergebnissen aus dem Projektionsschritt ab. Daher werden ab der zweiten Iteration die Ergebnisse des Projektionsschritt als *uniform texture2DRect* Variable an den Fragment Shader übergeben, der für die Schnittberechnung zuständig ist. Die Implementierung ermöglicht neben der Visualisierung des Endergebnisses auch die Visualisierung der Normalen, lokalen Ebenen und Polynome. Da im Falle des Abbruchs keine Berechnungen mehr durchgeführt werden und auf eine Textur nicht gleichzeitig lesend und schreibend zugegriffen werden kann, ist ein Ping Pong Verfahren notwendig. D.h. es werden zwei Texturen (in diesem Fall zwei FBOs) verwendet, von denen jeweils eine als Eingabe und eine als Ausgabe dient. Im jedem weiteren Iterationsschritt tauschen die zwei Texturen ihre Rollen, so dass die Ausgabe des letzten Schrittes immer wieder als Eingabe verwendet werden kann. Damit ist es im Falle eines Abbruchs auch möglich, das Ergebnis des letzten Iterationsschrittes des jeweiligen Shaders erneut zu schreiben, so dass die Werte nicht verloren gehen. Eine

Ausnahme stellt der Aufbau der Matrix $A^T A$ dar (5. Renderpass), da diese Zwischenergebnisse nicht mehr benötigt werden.

Kapitel 10.

Ergebnisse

Die Implementierung wurde auf einem System mit einem AMD Athlon^{TM}64 X2 Dual Core Prozessor 4200+ mit 2,21 GHz und 3,50 GB RAM, sowie einer NVIDIA GeForce 8800 GTA erreicht. Es wurden drei Ziele angestrebt: Die Erhöhung der Auflösung, die Detektion und Entfernung von Ausreißern, sowie die Glättung der Punktwolke. Im Folgenden werden die erzielten Ergebnisse des implementierten Verfahrens vorgestellt und kurz diskutiert. Zur Evaluation der Ergebnisse wurde das punktbasierte Raycasting Verfahren auf verschiedene Datensätze angewendet.
Um eine gute Oberflächenapproximation zu ermöglichen, müssen sich die lokalen, bivariaten Polynome der zu rekonstruierenden Oberfläche so gut wie möglich anpassen. Abb. 10.1 zeigt das Ergebnis des lokalen Polynomfittings in einigen Beispielpunkten, wobei zur Visualisierung des Polynoms rote Punktprimitive gewählt wurden.

Abbildung 10.1.: Durch die Generierung von zusätzlichen Punkten (im Geometry Shader) wird pro Szene jeweils ein lokales Polynom in einem Punkt dargestellt. Als Punktgröße wurde 5.0 gewählt.

Die Erhöhung der Auflösung wird in Abb. 10.2 deutlich. Auch wenn der Benutzer sich sehr nahe an die Oberfläche bewegt, sind keine einzelnen Punkte erkennbar. Je nach Fenstergröße (Höhe und Breite der Bildebene) werden entsprechend viele Punkte berechnet, so dass die Dichte der Punktwolke deutlich erhöht wird. Dadurch wird statt einer dünn besetzten Punktwolke eine geschlossene Oberfläche dargestellt.

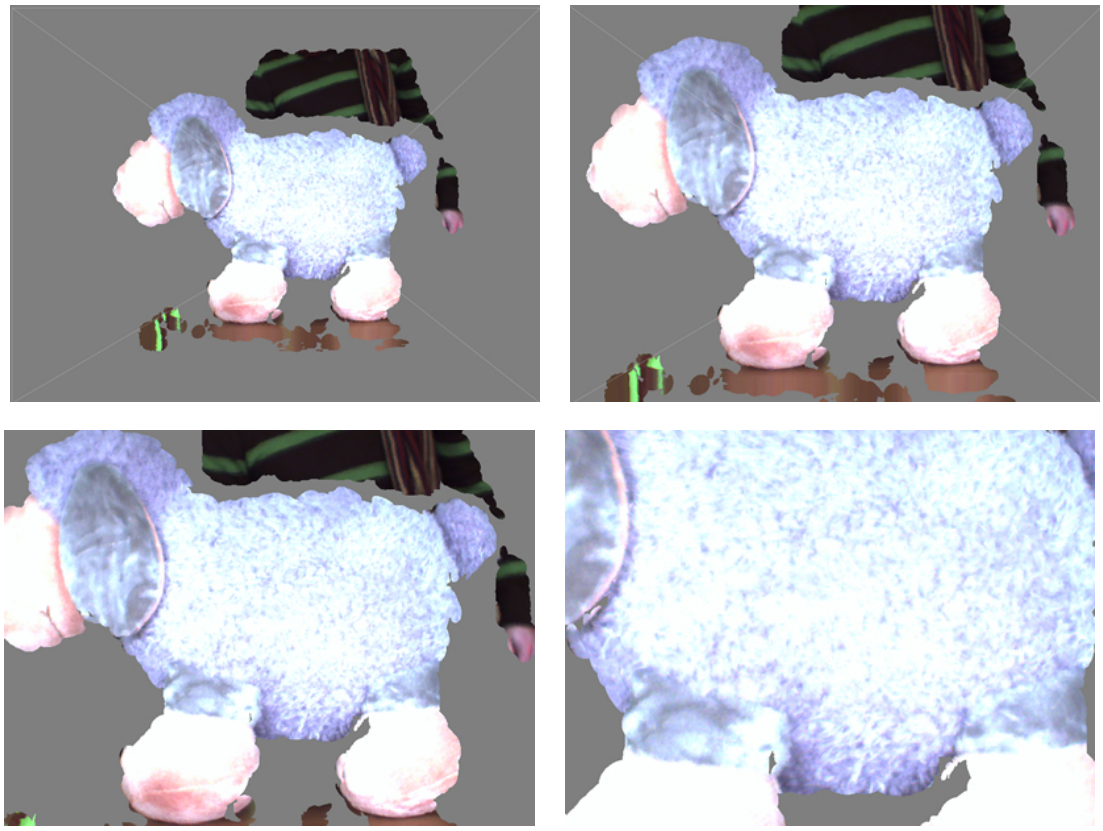
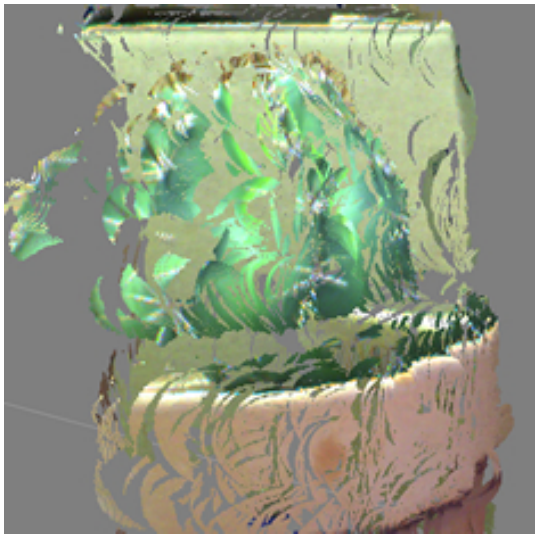
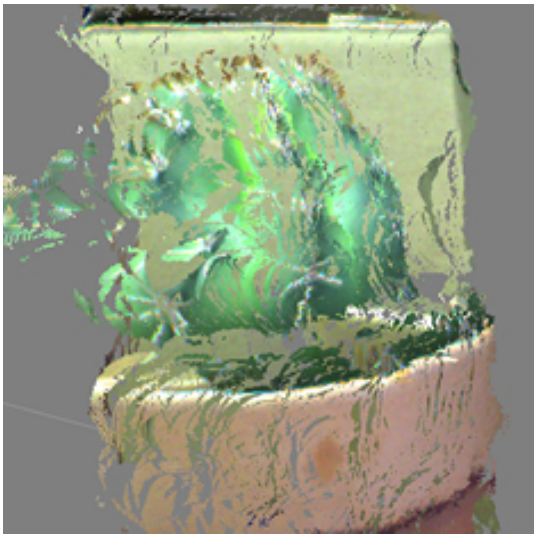


Abbildung 10.2.: Visualisierung der per Raycasting gefundenen Schnittpunkte mit dder Oberflächenapproximation.

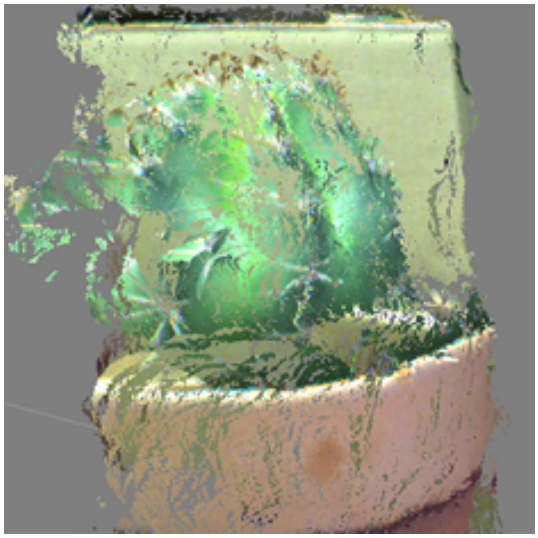
Um die Genauigkeit der Approximation zu erhöhen wurden mehrere Iterationen durchgeführt. In Abb. 10.3 und 10.4 wird die schrittweise Verfeinerung der Approximation der einzelnen Iterationsschritte auf zwei verschiedenen Datensätzen dargestellt. Zur besseren Demonstration der Unterschiede zwischen den Ergebnisbildern, wurde das Raytracing aus einer fixen Position durchgeführt, die nicht der Beobachterposition entspricht. Dadurch passt sich die Rekonstruktion der Szene nicht an die Position des Benutzers an. Wie stark sich die Verfeinerung nach den Iterationsschritten bemerkbar macht, hängt von dem verwendeten Schwellwert ab. Je größer der Schwellwert ist, desto früher wird der Iterationsprozess abgebrochen, so dass keine weitere Verfeinerung mehr vorgenommen wird.



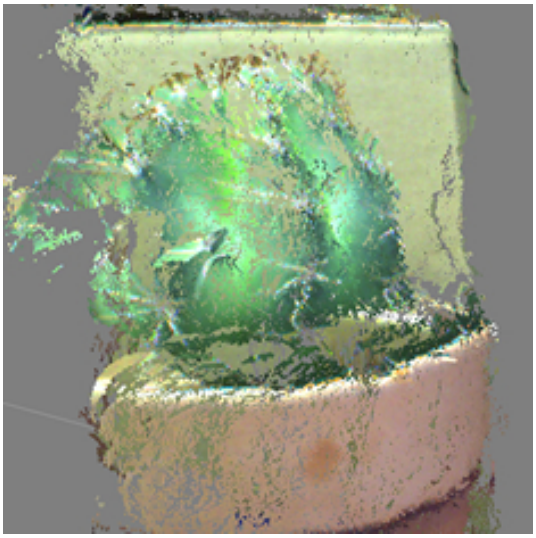
(a) Max. 1 Iterationsschritt



(b) Max. 2 Iterationsschritte

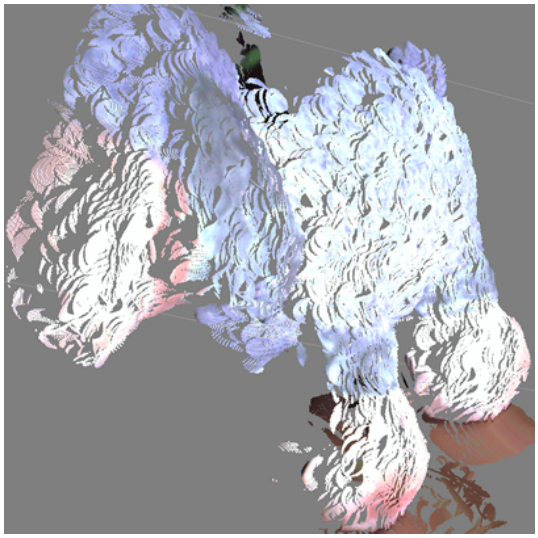


(c) Max. 4 Iterationsschritte

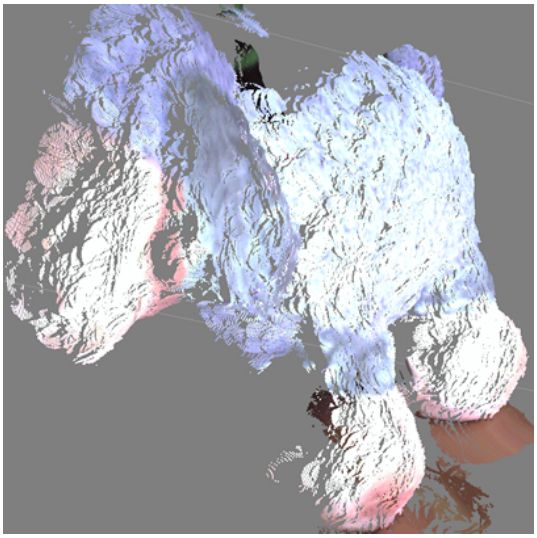


(d) Max. 6 Iterationsschritte

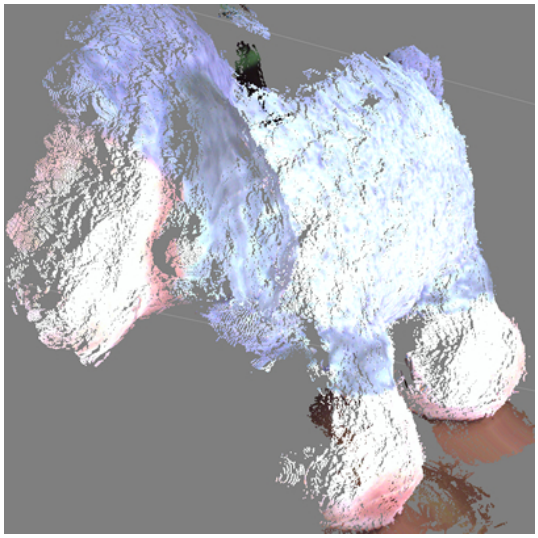
Abbildung 10.3.: Verfeinerung des Ergebnisses durch die iterative Durchführung des Algorithmus am Beispiel des Datensatzes *Kaktus* mit einem Schwellwert von 0.001.



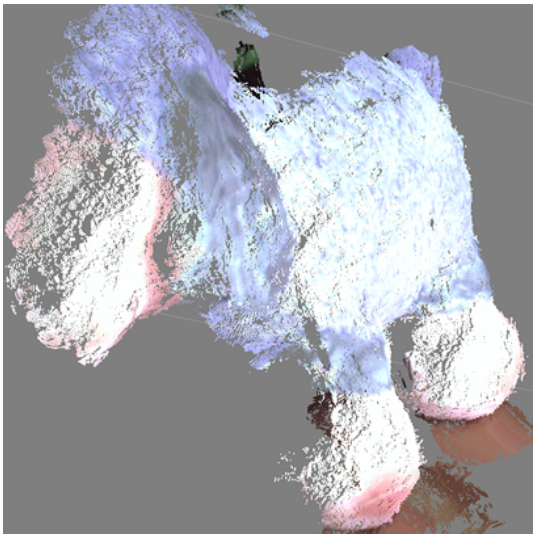
(a) Max. 1 Iterationsschritt



(b) Max. 2 Iterationsschritte



(c) Max. 4 Iterationsschritte



(d) Max. 6 Iterationsschritte

Abbildung 10.4.: Verfeinerung des Ergebnisses durch die iterative Durchführung des Algorithmus am Beispiel des Datensatzes *Jolly* mit einem Schwellwert von 0.001.

Bei den Ergebnissen der ersten Iteration sind die Splat-Scheiben noch deutlich zu erkennen, was sich nach nur wenigen Iterationen ändert. Mit jeder weiteren Iteration werden die Änderungen des Ergebnisses geringer und der Berechnungsaufwand höher (siehe Tabelle 10.2). Tabelle 10.1 zeigt für einzelne Iterationsschritte, bei wie vielen Punkten die Berechnung bereits abgeschlossen wurde (Spalte *Abbruch*) und bei wie vielen Punkten noch weitere Iterationsschritte erforderlich sind, um der Abbruchbedingung zu genügen (Spalte > *Iter.*). Die Daten beziehen sich auf zwei unterschiedliche Beobachterpositionen des Datensatzes *Kaktus*.

Iter.	Thresh.	> Iter.	Abbruch
1	0.001	104924	10089
2	0.001	97645	17368
3	0.001	91371	23642
4	0.001	86182	28831
5	0.001	82153	32860
6	0.001	78349	36664
7	0.001	75343	39670
1	0.01	42750	72263
2	0.01	27554	87459
3	0.01	20198	94815
4	0.01	16411	98602
5	0.01	13881	101132
6	0.01	11997	103016
7	0.01	10555	104458
1	0.02	9892	105115
2	0.02	4077	110928
3	0.02	1474	113537
4	0.02	1019	113994
5	0.02	649	114362
6	0.02	571	114442
7	0.02	466	114545

Iter.	Thresh.	> Iter.	Abbruch
1	0.001	226857	15977
2	0.001	211471	31363
3	0.001	200481	42345
4	0.001	190861	51968
5	0.001	183890	58944
6	0.001	177141	65690
7	0.001	171752	71082
1	0.01	103406	139428
2	0.01	69024	173810
3	0.01	51837	190997
4	0.01	43415	199419
5	0.01	36704	206130
6	0.01	31606	211228
7	0.01	27444	215390
1	0.02	21156	221678
2	0.02	10410	232424
3	0.02	3687	239147
4	0.02	2766	240068
5	0.02	1462	241372
6	0.02	1219	241615
7	0.02	699	242135

Tabelle 10.1.: Übersicht über die Anzahl der Punkte, deren Berechnung bereits abgeschlossen ist (Spalte *Abbruch*) und der Punkte, die das Abbruchkriterium noch nicht erfüllt haben (Spalte > *Iter.*). Für die linke Tabelle diente der Datensatz *Kaktus*, mit 136333 ungültigen Punkten als Basis, für die rechte Seite wurde ein Ausschnitt des Datensatzes vergrößert (8512 ungültige Punkte).

Der Grund für die hohe Anzahl an ungültigen Punkten aus Tabelle 10.1 ist der, dass die Punktwolke nicht die gesamte Bildebene bedeckt. Wie erwartet, steigt die Anzahl der

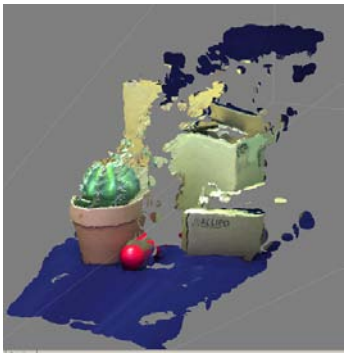
Datensatz	Iterationen	Fenstergröße	FPS
Kaktus	1	776×489	7,0
Kaktus	2	776×489	3,8
Kaktus	3	776×489	3,0
Kaktus	4	776×489	2,0
Kaktus	1	1658×934	3,0
Kaktus	2	1658×934	2,0
Jolly	1	776×489	8.0
Jolly	2	776×489	4.2
Jolly	3	776×489	3.0
Jolly	4	776×489	3.0
Jolly	1	1658×934	3.1
Jolly	2	1658×934	2.0

Tabelle 10.2.: Performanz in *frames per second* (FPS) des punktbasierten Raycastings. Die angegebenen Werte wurden bei einem Schwellwert von 0.001 über 10 Sekunden gemittelt.

Punkte, bei denen das Abbruchkriterium erfüllt ist, mit zunehmender Anzahl von Iterationen. Der Schwellwert spielt dabei eine große Rolle: Bei einem hohen Schwellwert sind nur wenige Iterationen notwendig, bis ein Großteil der Punkte das Abbruchkriterium erreicht hat. Nach Tejada und Ertl. [TGN+01] genügen zwei oder drei Iterationen aus, um alle Schnittpunkte zwischen den Strahlen und der Obefläche zu finden. Dass die evaluierten Werten des hier implementierten Verfahrens damit nicht übereinstimmen, kann daran liegen, dass in [TGN+01] ein relativ hoher Schwellwert verwendet wurde. Außerdem liegt die Vermutung nahe, dass bei PMD-Daten aufgrund der niedrigen Auflösung grundsätzlich mehr Iterationen notwendig sind, bis eine entsprechend hohe Anzahl an Punkten das Abbruchkriterium erreicht hat.

In Abb. 10.5 wird die Anzahl der Nachbarpunkte durch eine Farbcodierung visualisiert. Man kann erkennen, dass hauptsächlich die *Ausreißer* oder *flying pixels* eine geringe Anzahl von Nachbarpunkten besitzen. Durch die Festlegung eines geschickten Schwellwertes für die minimale Anzahl von Nachbarpunkten kann ein Großteil dieser Punkte frühzeitig aus der Berechnung und somit auch aus dem Endergebnis ausgeschlossen werden. Der Ausschluss aller *flying pixels* gelingt nur mit einem hohen Schwellwert, wodurch auch wesentliche Informationen der Szene verloren gehen.

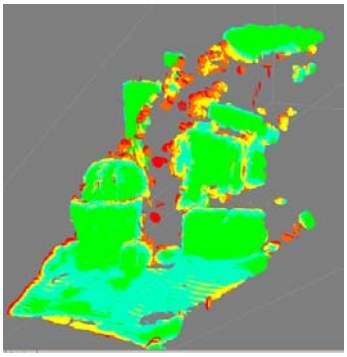
Bei nur einer Iteration kann die Anwendung als echtzeitnah bezeichnet werden. Mit steigender Anzahl von Iterationen und der Fenstergröße sinkt die Framerate stark ab. Aufgrund der hohen Berechnungszeit und der vergleichsweise geringen Verbesserung des Ergebnisses ist die Verwendung von mehreren Iterationen nur sinnvoll, so lange die Berechnungszeit keine wesentliche Rolle spielt.



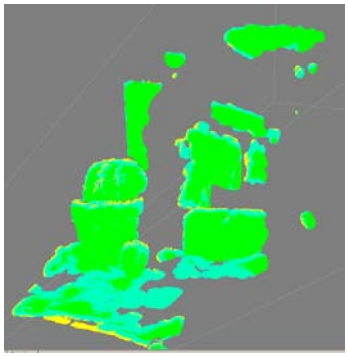
(a) Keine Grenze

(b) Mind. 10 Nachbarpunkte

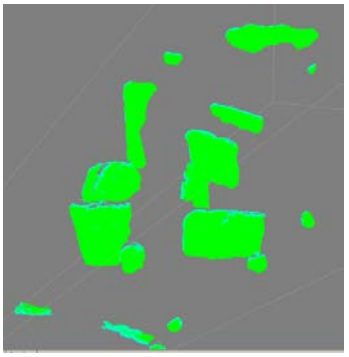
(c) Mind. 20 Nachbarpunkte



(d) Keine Grenze, farbcodiert



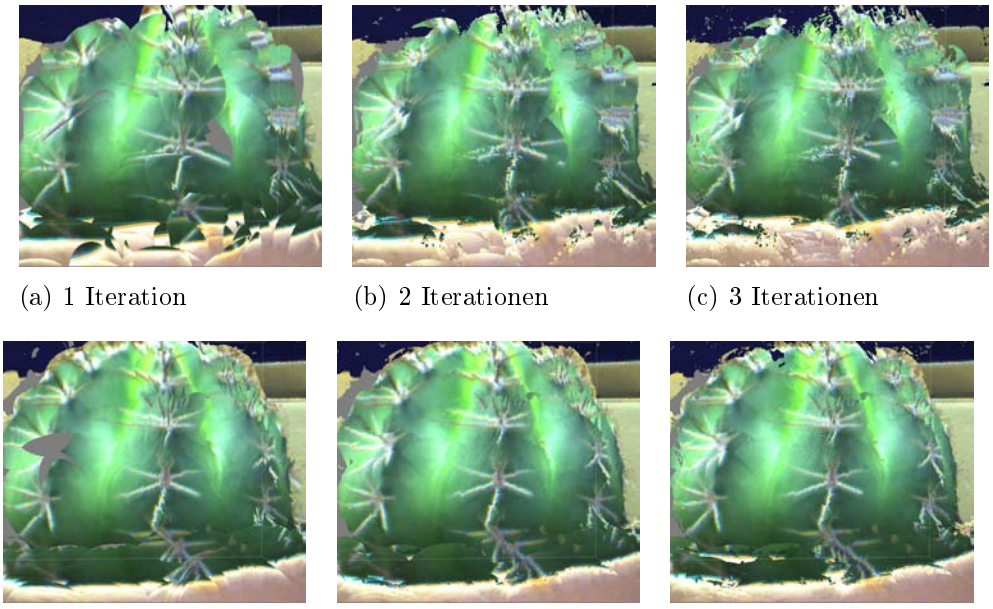
(e) Mind. 10 Nachbarpunkte, farbcodiert



(f) Mind. 20 Nachbarpunkte, farbcodiert

Abbildung 10.5.: 1. Zeile: Ergebnis mit unterschiedlichen Schwellwerten für die minimale Nachbaranzahl. 2. Zeile: Farbcodierung der Nachbaranzahl pro Punkt.

Das dritte Ziel bestand darin, durch die Anwendung des WLS-Filters eine Glättung der PMD-Daten zu erreichen. Allerdings wurde der gewünschte Glättungseffekt nicht wie erwartet erreicht. Die Erklärung liegt darin, dass die PMD-Kamera im Vergleich zu einem Laserscanner Daten liefert, deren Auflösung deutlich geringer ist und die einem stärkeren Rauschen unterliegen. Das Rauschen wirkt sich offensichtlich negativ auf das Ebenen- und das Polynomfitting aus, so dass der Algorithmus deutlich bessere Ergebnisse liefert, wenn die PMD-Daten durch einen Bilateralfilter vorgeglättet werden. Ein Vergleich zwischen Ergebnissen mit bzw. ohne die Verwendung des Bilateralfilters wird in Abb. 10.6 dargestellt.



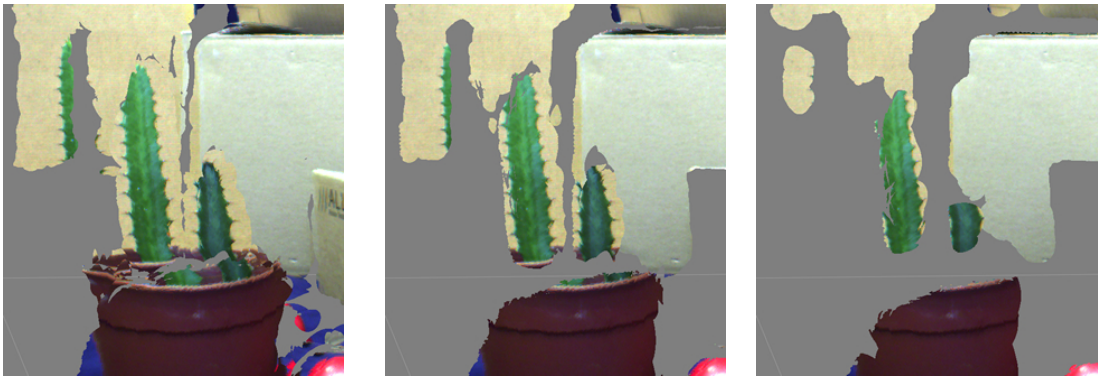
(a) 1 Iteration (b) 2 Iterationen (c) 3 Iterationen

(d) 1 Iteration, mit Bilateralfilter (e) 2 Iterationen, mit Bilateralfilter (f) 3 Iterationen, mit Bilateralfilter

Abbildung 10.6.: Obere Reihe: Ergebnis der direkten Anwendung des WLS-Filters auf die PMD-Daten. Untere Reihe: Vorverarbeitung durch einen Bilateralfilter.

Ohne die Vorverarbeitung durch den Bilateralfilter (obere Zeile von Abb. 10.6) sind die Scheiben viel deutlicher zu erkennen als bei den Ergebnissen, die durch zusätzliche Verwendung des Bilateralfilters (untere Zeile von Abb. 10.6) gewonnen wurden. Des Weiteren wirkt das Gesamtergebnis nach weiteren Iterationsschritten erheblich verrauschter, wenn keine vorherige Glättung stattfindet.
Ein Problem das bei einigen Szenen aufgetreten ist, besteht darin, dass sehr feine Details auch mit Hilfe des WLS-Filters nicht rekonstruiert werden können. In Abb. 10.7 kann man erkennen, dass statt der einzelnen Blätter der Pflanze eine ganze Fläche rekonstruiert wird. Dies lässt sich folgendermaßen erklären: Die PMD-Kamera hat Tiefenwerte für einzelne Punkte der Blätter aufgenommen. Die Punkte des Hintergrundes sind zu

weit entfernt, um als Nachbarn detektiert werden zu können. Damit werden lediglich die Punkte der anderen Blätter als Nachbarn erkannt. Da die lokale Ebene und das lokale Polynom durch die Nachbarpunkte definiert sind, wird eine geschlossene Fläche rekonstruiert. Der Fehler in diesen Punkten kann einfach behoben werden, indem der Schwellwert für die minimale Anzahl benötigter Nachbarpunkte erhöht wird. Allerdings bewirkt dies, dass ein Großteil der restlichen Punkte diesen Schwellwert unterschreitet und damit aus dem Ergebnis ausgeschlossen wird (siehe Abb. 10.5 und 10.7).



(a) Mind. 5 Nachbarpunkte (b) Mind. 15 Nachbarpunkte (c) Mind. 25 Nachbarpunkte

Abbildung 10.7.: Die Rekonstruktion von feinen Details (hier am Beispiel des Datensatzes *Kaktus1*) ist nach wie vor problematisch. Der Fehler kann durch Erhöhung des Schwellwertes für die minimale Anzahl von Nachbarpunkten verringert werden.

Kapitel 11.

Zusammenfassung und Ausblick

11.1. Zusammenfassung

Die vorliegende Arbeit hat sich mit dem Problem der interaktiven Rekonstruktion von ToF-Daten einer PMD-Kamera beschäftigt. Dabei wurden die Nachteile, die durch die Hardware gegeben sind, durch einen GPU-basierten, iterativen Algorithmus ausgeglichen. Als Grundlage dient dabei die Arbeit von Tejada und Ertl et al. [TGN+01]. Es wird in jedem PMD-Punkt zunächst eine Referenzebene und dann ein bivariates Polynom gefittet. Durch ein Raycasting-Verfahren werden durch Schnittberechnung mit den lokalen Polynomen entsprechend viele zusätzliche Punkte erzeugt. Außerdem wird die Anzahl der Nachbarpunkte für jeden berechneten Punkt festgestellt und zur Ausreißer-Detektion verwendet. Die wichtigsten Änderungen, die sich bzgl. [TGN+01] ergaben sind:

- Gewichtung:
  Die Gewichtung der Punktdaten beim Ebenen- und Polynomfitting wird angepasst, da die Verteilung der Punkte in der XY-Ebene aber nicht in z-Richtung regelmäßig ist.
- Nachbarschaftssuche:
  Der Algorithmus von Tejada und Ertl et al. [TGN+01] ist auf vorberechnete Nachbarschaftsinformationen angewiesen, daher sind zur Laufzeit keine Nachbarschaften zu berechnen. Die vorliegende Arbeit arbeitet auf Videoströmen und berechnet die Nachbarschaftsinformationen zur Laufzeit. Dabei können im Initialschritt die 2D-Informationen ausgenutzt werden, um die Suche effizienter zu gestalten.
- Ebenenfitting:
  Statt der Annahme, dass $t = 0$ und damit $q = r + tn = r$ wird im Initialschritt der Mittelwertpunkt als Gewichtungspunkt der Nachbarn $q = \overline{p}$ gewählt. Dadurch liegt der Punkt der Referenzebene optimal in der Punktwolke der Nachbarpunkte. Außerdem wird statt einer aufwendigen *inverse power* Methode ein nicht-iterativer

Ansatz aus [Ebe06] verwendet. Hierbei werden alle Eigenvektoren berechnet, da dies nachfolgende Berechnungen erleichtert.

- Polynomfitting:
  Tejada und Ertl et al. [TGN+01] verwenden ein CG-Verfahren, um das lokale Polynom zu finden, welches den aktuellen Punkt und deren Nachbarpunkte approximiert. Die vorliegende Arbeit verwendet stattdessen eine Kombination aus Cholesky-Zerlegung und der bikonjugierten Gradientenmethode.
- Minimierungsstrategie:
  Um die Performanz nicht noch weiter zu reduzieren, beschränkt sich die Implementierung auf das WLS-Verfahren (anstatt des MLS-Verfahrens), welches so gute Ergebnisse erzielt, dass durch die Verwendung von MLS keine deutliche Verbesserung zu erwarten ist.

Das Fazit, das nach der Implementierung und Auswertung der Ergebnisse gezogen werden kann ist, dass das Verfahren, welches von Tejada et al. [TGN+01] vorgestellt wurde, mit entsprechender Modifikation auch auf PMD-Daten angewendet werden kann. Ein Problem, welches sich im Verlauf der Arbeit gezeigt hat ist, dass die PMD-Daten so stark verrauscht sind, dass das Ergebnis des Verfahrens besser ist, wenn die PMD-Daten zunächst mit Hilfe eines Bilateralfilters vorverarbeitet werden. Wird das Verfahren auf eine Iteration beschränkt, ist eine echtzeitnahe Rekonstruktion möglich. Die Rekonstruktion von feinen Details wird durch den implementierten Algorithmus nicht gelöst.

11.2. Ausblick

Folgende Ideen sind bei der Ausarbeitung der vorliegenden Arbeit entstanden, welche die Ergebnisse möglicherweise verbessern könnten, aber noch nicht geprüft bzw. umgesetzt wurden:

- Polynomwahl:
  Ein Ansatzpunkt zur Verbesserung des Ergebnisses liegt in der Wahl des Polynoms. In der vorliegenden Arbeit wurde die Anzahl der Einzelterme auf 4 beschränkt, da 4er Vektoren bzw. Matrizen durch die Graphikhardware unterstützt werden und die Speicherung in eine Float-Textur leicht möglich ist. Durch die Wahl eines komplexeren Polynoms mit mehr Koeffizienten, wären z.B. zusätzliche lineare Terme möglich: $Ax^2 + By^2 + Cxy + Dx + Ey + F$. Damit können Teile der Oberfläche ggf. besser beschrieben werden, wodurch die Approximation positiv beeinflusst wird. Der Nachteil besteht allerdings darin, dass der Berechnungsaufwand steigt, da die Berechnung der Schnittpunkte mit komplexeren Polynomen deutlich aufwendiger ist. Des Weiteren muss auf Arrays anstelle von *vec3* (bzw. *vec4*) und *mat3* (bzw.

mat4) ausgewichen werden. Außerdem müssen mehr Texturen verwendet werden, um die nötigen Informationen von einem Renderpass zum Nächsten weiterzugeben.

- Kantenglättung:
  Es sind leicht Fälle zu konstruieren, in denen Kanten der Originaloberfläche herausgeglättet werden, wodurch das Ergebnis negativ beeinflusst wird (siehe Abb. 11.1). Die Kanten liegen so ungünstig innerhalb der Szene, dass die PMD-Punkte (p_i) die Ecke der Oberfläche nicht erfassen. In diesem Fall wird die tatsächliche Oberfläche (in Abb. 11.1 rot gestrichelt) durch die Referenzebene H so approximiert, dass die Kante herausgeglättet wird. Um solche Effekte zu vermeiden wäre es möglich, Informationen aus einem hoch aufgelösten Farbbild zu verwenden. Aus der Annahme, dass Pixel gleicher Farbe auch einen ähnlichen Tiefenwert besitzen, müssen die zusätzlichen Punkte, die der implementierte Algorithmus generiert, eine Auswertung der Farbwerte vornehmen anhand derer sie ihre Tiefe ggf. korrigieren. Die Verbesserung der Rekonstruktionsgenauigkeit von feinen Details, wie sie in Ab. 10.7 dargestellt werden, sind mit einer solchen Erweiterung ebenfalls vorstellbar.

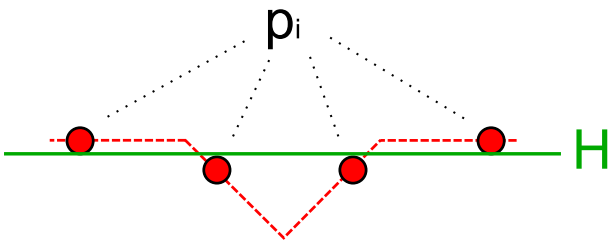


Abbildung 11.1.: Skizze einer Konfiguration von PMD-Punkten, bei der eine Kante bei der Rekonstruktion herausgeglättet wird (Draufsicht).

- Adaptives h:
  In Kapitel 7.1 wurde beschrieben, dass ein adaptiver Wert für den Skalierungsfaktor h der Gewichtungsfunktion ω sinnvoll ist. Die Implementierung verwendet aus Effizienzgründen einen konstanten Wert, der vom Benutzer festgelegt werden kann. Das Ziel einer Erweiterung könnte eine einfache Methode sein, um den Parameter h an die jeweilige Umgebung anzupassen.

- Effizienzsteigerung:
  Pro Iterationsschritt muss mehrmals ein bildschirmfüllendes Quad rasterisiert werden, so dass jeweils für *alle* Fragmente Berechnungen durchgeführt werden, unabhängig davon, ob an dieser Stelle Informationen vorliegen, oder nicht. Dadurch geht ein Teil der Berechnungszeit verloren, ohne dass ein Nutzen erreicht wird. Eine Erweiterung, die zwar keine Verbesserung der Ergebnisse mit sich bringt, aber die Performanz steigern könnte, ist die Verwendung des Stencil-Buffers. Die Idee besteht darin, den Teil der Szene zu markieren, der Informationen beinhaltet und

die Graphikhardware per Stencil-Buffer entscheiden zu lassen, wo Berechnungen durchgeführt werden sollen.

Literaturverzeichnis

[AA03] A. Adamson and M. Alexa. Ray tracing point set surfaces. In *SMI '03: Proceedings of the Shape Modeling International 2003*, page 272, Washington, DC, USA, 2003. IEEE Computer Society.

[AA06] K. Akeley and J. Allen. Ext_framebuffer_object. http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt, 2006.

[ABCO+01] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C.T. Silva. Point set surfaces. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 21–28, Washington, DC, USA, 2001. IEEE Computer Society.

[AK04] N. Amenta and Y.J. Kil. Defining point-set surfaces. *ACM Trans. Graph.*, 23(3):264–270, 2004.

[Alt02] W. Alt. *Nichtlineare Optimierung. Eine Einführung in Theorie, Verfahren und Anwendung.* Vieweg, 2002.

[Ast05] D. Astle. Next generation rendering with opengl. http://www.gamedev.net/columns/events/coverage/feature.asp?feature_id=75, 2005.

[BHMB08] M. Böhme, M. Haker, T. Martinetz, and E. Barth. Shading constraint improves accuracy of time-of-flight measurements. In *TOF-CV08*, pages 1–6, 2008.

[Can] Devkit faq. http://www.canesta.com/developers/devkit-faq.

[Can08] Canesta 101 introduction to 3d vision in cmos. http://www.canesta.com/assets/pdf/technicalpapers/Canesta101.pdf, March 2008.

[Car03] B. Carpenter. Cholesky decomposition, 2003.

[Dac02] C. Dachsbacher. *Punktbasiertes Rendering mit modernen Grafikkarten.* Diplomarbeit, Universität Erlangen-Nürnberg, Oktober 2002.

[Dör88] W. Dörfler, W. und Peschek. *Einführung in die Mathematik für Informatiker*. Hanser, 1988.

[Ebe06] D. Eberly. Eigensystems for 3x3 symmetric matrices (revisited). Technical report, www.geometrictools.com, 2006.

[FHWZ04] R. Fernando, M. Harris, M. Wloka, and C. Zeller. Eurographics 2004 tutorial abstract programming graphics hardware, 2004.

[FK03] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Han02] A. Handl. *Multivariate Verfahren*. Springer, September 2002.

[Hei99] R. Heinol, H.G. und Schwarte. Photomischdetektor erfaßt 3d-bilder. Fachzeitschrift: Elektronik, Ausgabe 12, Juni 1999.

[HF07] B. Huhle and A Fleck, S. Schilling. Integrating 3d time-of-flight camera data and high resolution images for 3dtv applications. In *3DTV CON - The True Vision*, Mai 7-9 2007.

[Hor02] E. Horber. Motion capturing. http://medien.informatik.uni-ulm.de/lehre/courses/ss02/ModellingAndRendering/07-motion-capturing.pdf, 2002.

[HSJS08] B. Huhle, T. Schairer, P. Jenke, and W. Straßer. Robust non-local denoising of colored depth data, 2008.

[Huß00] S. Hußmann. *Schnelle 3D-Objektvermessung mittels PMD/CMOS-Kombizeilensensor und Signalkompressions-Hardware*. PhD thesis, Department of Electrical Engineering and Computer Science, 2000.

[HZ04] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[KB04] L. Kobbelt and M. Botsch. A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801–814, 2004.

[Kle06] A. Klein. *Effiziente Berechnung einer dilationsminimalen Triangulierung*. Diplomarbeit, Universität Bonn, Februar 2006.

[Koc07] M. Koch. *Simplifizierung punktbasierter Geometrien Simplification of Point-based Geometry*. PhD thesis, Universität Augsburg, 2007.

[Kol] O. Kolb, A. und Loffeld. Antragspaket dynamisches 3d sehen. http://www.zess.uni-siegen.de/cms/upload/pdf/Praeambel_komplett3D.pdf, ?

[Lan00] R. Lange. *3D Time-of-Flight Distance Measurement with Custom Solid-State Image Sensors in CMOS/CCD-Technology*. PhD thesis, Department of Electrical Engineering and Computer Science, 2000.

[Lev03] D. Levin. Mesh-independent surface interpolation. *Geometric Modeling for Scientific Visualization*, 2003.

[LLK08] M. Lindner, M. Lambers, and A. Kolb. Data Fusion and Edge-Enhanced Distance Refinement for 2D RGB and 3D Range Images. *Int. J. on Intell. Systems and Techn. and App. (IJISTA), Issue on Dynamic 3D Imaging*, 2008.

[Lov05] A. Lovesey. A comparison of real time graphical shading languages. CS4983 senior technical report, Faculty of Computer Science University of New Brunswick Canada, 2005.

[LSBS99] R. Lange, P. Seitz, A. Biber, and R. Schwarte. Time-of-flight range imaging with a custom solid-state image sensor. Laser Metrology and Inspection, Proc. SPIE, Vol. 3823, 1999.

[LW85] M. Levoy and T. Whitted. The use of points as a display primitive. Technical Report 85-022, Univ. of North Carolina at Chapel Hill, Januar 1985.

[Müc07] F. E. Mücke. *Analyse GPU-basierter Feature Tracking Methoden füur den Einsatz in der Augmented Reality*. PhD thesis, Universität Augsburg, 2007.

[Müh02] K. Mühlmann. *Design und Implementierung eines Systems zur schnellen Rekonstruktion dreidimensionaler Modelle aus Stereobildern*. PhD thesis, Universität Mannheim, 2002.

[MKF+05] T. Möller, J. Kraft, J. Frey, M. Albrecht, and R. Lange. Robust 3D measurement with pmd sensors. Technical report, PMDTec, 2005.

[Nea04] A. Nealen. An as-short-as-possible introduction to the Least Squares, Weighted Least Squares and Moving Least Squares Methods for Scattered Data Approximation and Interpolation, 2004.

[Net08] Microsoft Developer Network. Pipeline stages (Direct3D 10). http://msdn.microsoft.com/en-us/library/bb205123(VS.85).aspx, 2008.

[Per07] E. Persson. ATI Radeon HD 2000 programming guide. http://ati.amd.com/developer/SDK/AMD_SDK_Samples_May2007/Documentations/ATI_Radeon_HD_2000_programming_guide.pdf, 2007.

[PGK02] M. Pauly, M. Gross, and L.P. Kobbelt. Efficient simplification of point-sampled surfaces. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 163–170, Washington, DC, USA, 2002. IEEE Computer Society.

[PKG02] M. Pauly, L. Kobbelt, and M. Gross. Multiresoultion of point-sampled geometry. *CS Technical Report*, 378, 2002.

[PM03] C. Peeper and J. L. Mitchell. Introduction to the DirectX(R) 9 high level shading language. http://www.cs.uoi.gr/~fudos/grad-exer2/hlsl-intro.pdf, 2003.

[PSSJ06] S. Patidra, B. Shiben, J. M. Singh, and Narayanan P. J. Exploiting the shader model 4.0 architecture. Technical report, International Institute of Information Technology Hyderabad, 2006.

[PVTF02] W. H. Press, W. T. Vetterling, S. A. Teukolsky, and B. P. Flannery. *Numerical Recipes in C++: the art of scientific computing*. Cambridge University Press, 2002.

[PZvBG00] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *Proc. SIGGRAPH 2000*, pages 335–342, New York, NY, USA, Juli 2000. ACM Press/Addison-Wesley Publishing Co.

[Rin07] B. Ringbeck, T. und Hagebeuker. Dreidimensionale Objekterfassung in Echtzeit - PMD Kameras erfassen pro Pixel Distanz und Helligkeit mit Videoframerate. *Allgemeine Vermessungs-Nachrichten (AVN), Heft 7/2007*, 2007.

[RL00] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000*, pages 343-352, Juli 2000.

[Ros06] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, January 2006.

[RPZ02] L. Ren, H. Pfister, and M. Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Computer Graphics Forum*, pages 461–470, 2002.

[SR408] Sr4000 data sheet. http://www.mesa-imaging.ch/pdf/SR4000_Data_Sheet_rev1.1.pdf, August 2008.

[TGN+01] E. Tejada, J. P. Gois, L.G Nonato, A. Castelo, and T. Ertl. Hardware-accelerated extraction and rendering of point set surfaces. In *Proc. Eurographics / IEEE VGTC Symposium on Visualization*, pages 21–28, 2001.

[Tho02] J. Thomas. 3D-Visualisierung mit Hilfe von Punktwolken. wwwiaim.ira.uka.de/Teaching/SeminarMedizin/Ausarbeitungen/SS2002/02_Punktwolken.pdf, 2002.

[TSE07] E. Tejada, T. Schafhitzel, and T. Ertl. Hardware-accelerated point-based rendering of surfaces and volumes. In *Proceedings of WSCG 2007 Full Papers*, pages 41–48, 2007.

[wik] 3D scanner.

[WND97] M. Woo, J. Neider, and T. Davis. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 1.1 (2nd Edition)*. Addison-Wesley Longman, July 1997.

[XSHR05] Z. Xu, H. Schwarte, B. Heinol, and T. Ringbeck. Smart pixel - photonic mixer device (PMD). New system concept of a 3d-imaging camera-on-a-chip. Technical report, PMDTec, 2005.

[YYDN07] Q. Yang, R. Yang, J. Davis, and D. Nister. Spatial-depth super resolution for range images. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 0:1–8, 2007.

[ZCa06] Technology. http://www.3dvsystems.com/technology/tech.html, 2006.

[ZPvBG01] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In *Proceedings of SIGGRAPH 2001*, pages 371-378, 2001.

Anhang A.

Spezifikation der PMD-Kamera

Name	PMD[VISION]C® 19k
Sensortyp	CMOS-Matrix Kamera mit 19200 Pixel
Detektor	$\frac{1}{2}$" Global Shutter PMD Sensor
Detektor Dimensionen	6.4 mm (H) x 4.8 mm (v)
Pixel Dimensionen	40 μ m (H) x 40 μ m (V)
Auflösung	160 (H) x 120 (V)
Optischer Füllfaktor	30 %
Empfänger Optik	C-Mount
Eindeutiger Bereich	7,5 m bei 20 MHz
z-Auflösung	$>$ 6 mm
Sichtbereich	40° bei f = 12 mm
Beleuchtungsstärke	ca. 3W optisch
Wellenlänge	870 nm
Bildrate (3D)	bis zu 15 fps (frames per second)
Digitale Schnittstellen	IEEE 802.3u, IEEE 1394
Energieversorgung	9 V ... 18 V
Gewicht	1400g

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum Unterschrift