UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Augmented Images - Variation von Beleuchtungsinformationen und Materialeigenschaften auf Basis von Kinect-Daten

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Niklas Gard

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	M.Sc. Gerrit Lochmann (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im März 2013

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Bachelorarbeit
Niklas Gard (Matr.-Nr. 210 100 154)

Thema: **Augmented Images – Variation von Beleuchtungsinformationen und Materialeigenschaften in Bildern auf Basis von Kinect-Daten**

Kinect von Microsoft stellte im Jahr 2010 nicht nur in der Spielindustrie eine Revolution dar. Die offene Schnittstelle ermöglichte es Entwicklern zu einem günstigen Preis, Farb- und Tiefenbilder zu erfassen, was scheinbar endlos viele innovative und weiterführende Einsatzmöglichkeiten vorstellbar machte. Aufgenommene RGB Bilder, die mit Tiefeninformation um eine Dimension erweitert sind, könnten in der automatischen Bildverarbeitung zu vielen neuen Möglichkeiten führen. So ist es auf Basis dieser Information möglich, bei einfachen Szenen und bekannter Beleuchtungssituation, die Beleuchtung aus einem Bild herauszurechnen, um die Szene anschließend mit veränderter Beleuchtungssituation oder veränderten Materialeigenschaften, zum Beispiel mit spiegelnden Oberflächen darzustellen.

Ziel der Arbeit ist es, einfache Umgebungen mit definierten Lichtverhältnissen mit Kinect aufzunehmen und auf Basis des Tiefen- und Farbbildes Beleuchtungs- und Materialeigenschaften der erfassten Szene zu variieren. Es soll gezeigt werden, ob durch Tiefeinformationen erweiterte Farbbilder einen Mehrwert bei der digitalen Photobearbeitung liefern können und inwieweit realistische Effekte auf den Kinect Bildern erzeugt werden können.

Es soll optional experimentiert werden, inwiefern eine Echtzeitfähigkeit besteht, also ob Effekte auf die Bewegung von Kinect reagieren könnten, oder, ob umgekehrt dynamische Objekte, wie zum Beispiel bewegte Lichtquellen, in die Szene eingefügt werden könnten.

Schwerpunkte dieser Arbeit sind:

1. Einarbeiten in SDKs zur Ansteuerung und zum Auslesen der Kinect Tiefenbilder
2. Einarbeitung in erforderliche GPU Programmierung und Rendering Methoden
3. Konzeption des Gesamtsystems
4. Implementierung
5. Demonstration und Bewertung der Ergebnisse
6. Dokumentation

Koblenz, den 24.09.2012

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . . . .
(Ort, Datum) (Unterschrift)

Zusammenfassung

Ein Kinect-Sensor verfügt über die Fähigkeit, gleichzeitig Farb- und Tiefenbilder aufzunehmen. In dieser Bachelorarbeit wird versucht, das Tiefenbild zu nutzen, um Beleuchtungsinformationen und Materialeigenschaften im Farbbild zu manipulieren.
Die vorgestellten Verfahren zur Beleuchtungs- und Materialmanipulation benötigen eine Lichtsimulation der Lichtverhältnisse zum Zeitpunkt der Aufnahme und übertragen dann Informationen aus einer neuen Lichtsimulation direkt zurück in das Farbbild. Da die Simulationen auf einem dreidimensionalen Modell durchgeführt werden, wird nach einem Weg gesucht, ein solches aus einem einzigen Tiefenbild zu erzeugen. Dabei wird auf Probleme der Tiefendatenerfassung des Kinect-Sensors eingegangen. Es wird ein Editor entwickelt, mit dem Beleuchtungs- und Materialmanipulationen möglich gemacht werden sollen. Zum Erzeugen einer Lichtsimulation werden einfache, echtzeitfähige Renderingverfahren und Beleuchtungsmodelle vorgestellt. Mit ihnen werden neue Beleuchtungssituationen, Schatten und Spiegelungen in das Farbbild eingefügt. Einfache Umgebungen mit definierten Lichtverhältnissen werden in Experimenten manipuliert, um Grenzen und Möglichkeiten des Sensors und der verwendeten Verfahren aufzuzeigen.

Abstract

A Kinect device has the ability to record color and depth images simultaneously. This thesis is an attempt to use the depth image to manipulate lighting information and material properties in the color image.
The presented method of lighting and material manipulation needs a light simulation of the lighting conditions at the time of recording the image. It is used to transform information from a new light simulation directly back into the color image. Since the simulations are performed on a three-dimensional model, a way is searched to generate a model out of single depth image. At the same time the text will react to the problems of the depth data acquisition of the Kinect sensor. An editor is designed to make lighting and material manipulations possible. To generate a light simulation, some simple, real-time capable rendering methods and lighting modells are proposed. They are used to insert new illumination, shadows and reflections into the scene. Simple environments with well defined lighting conditions are manipulated in experiments to show boundaries and possibilities of the device and the techniques being used.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation und Ziele

Dass das automatische Ändern von Beleuchtungssituation oder Materialbeschaffenheiten in Fotografien selbst aktuelle Bildbearbeitungsprogramme an ihre Grenzen stoßen lässt, liegt in der Natur der Fotografie. Um den Weg von Licht berechnen zu können, werden Informationen über die dem Bild zugrunde liegende Geometrie benötigt, über die das zweidimensionale Abbild der Umwelt keine direkte Auskunft gibt. Während sich eine manuelle Rekonstruktion in der Regel als sehr aufwändig erweist, sind Hardware basierte 3D-Scanner zumeist sehr teuer.
Die automatische Manipulation von Licht, Schatten oder Materialien auf Basis von Geometrie würde dennoch eine Vielzahl neuer Möglichkeiten eröffnen, sei es zur Bildkorrektur oder zur Bilderweiterung, indem zum Beispiel Spiegelungen oder Materialänderungen in ein Foto übertragen werden.

2010 veröffentlichte Microsoft mit dem **Kinect-Sensor** eine kostengünstige und handliche Hardware, die es Entwicklern dank einer offenen Schnittstelle ermöglicht, gleichzeitig Farb- und Tiefenbilder zu erfassen. In dieser Arbeit soll herausgefunden werden, ob es möglich ist, aus einem von Kinect erfassten Paar aus Farb- und Tiefenbild eine Geometrierekonstruktion zu erstellen, die als Basis zur Manipulation von Beleuchtung und Material im Farbbild dienen kann. Da bei der Datenerfassung verschiedene Probleme auftreten, etwa starke Schwankungen und Lücken im Tiefenbild, werden Techniken zur Verbesserung der Qualität der Daten vorgestellt. Weiterhin sollen Verfahren vorgestellt werden, die die Beleuchtungs- und Materialeigenschaften in Fotos manipulieren können. Da einer Beleuchtungssimulation ein 3D-Modell zugrunde liegt, wurden elementare Verfahren der Computergrafik ausgewählt, um sie durchzuführen.
Bei den Verfahren zur Beleuchtungserweiterung hängt die Qualität des Ergebnisbildes unter anderem davon ab, ob die bei der Aufnahme bestehenden Lichtverhältnisse simuliert werden können. Einfache Lichtverhältnisse sollen in dieser Arbeit manuell rekonstruiert werden.

Es wird ein Editor entwickelt, mit dem Bilder mit Kinect aufgenommen und anschließend manipuliert werden können. Es soll herausgefunden werden, ob eine Simulation in Echtzeit möglich ist. Der Editor setzt alle Verfahren, die in dieser Arbeit vorgestellt werden um und macht die Bilderweiterungen im Rahmen dieser Techniken frei konfigurierbar. Seine Entwicklung stellt Schwerpunkt und Kern der Arbeit dar.
Anhand von Beispielbildern, die mit dem Editor erzeugt wurden soll diskutiert und bewertet werden, ob die Bilderweiterung auf Basis der Kinect-Daten erfolgreich war. Falls Probleme auftreten, soll ermittelt werden, ob deren Auftreten auf qualitative Mängel der Kinect-Daten zurückzuführen sind oder ob sie durch geschicktere 3D-Rekonstruktion oder Beleuchtungssimulationen mit weiterführenden Techni-

ken behoben werden können. Anhand der ermittelten Ergebnisse wird eine Zukunftsprognose gestellt, ob und wie die Kombination aus Tiefen- und Farbkamera die Fotografie und die Bildbearbeitung beeinflussen wird.

1.2 Aufbau der Arbeit

Diese Arbeit ist unterteilt in sechs Kapitel. Im Kapitel, das auf diese Einleitung folgt, wird zunächst der Kinect-Sensor im Detail vorgestellt werden. Es soll deutlich gemacht werden, wie der Sensor ein Tiefenbild erzeugt und wo Probleme auftreten können, die beim weiteren Vorgehen eine Rolle spielen. Außerdem wird auf verwandte wissenschaftliche Arbeiten und Projekte verwiesen, bei denen Kinect ebenfalls verwendet wird.
Zur Erweiterung der Kinect-Bilder werden verschiedene bekannte Techniken aus den Bereichen Bildverarbeitung und Computergrafik verwendet, deren theoretische Grundlagen in Kapitel 3 vorgestellt werden. Kapitel 4 geht auf das Vorgehen bei der Entwicklung eines Editors zur Manipulation von Beleuchtungs- und Materialeigenschaften ein. Es gliedert sich wiederum in sechs Teile, die jeweils aufeinander aufbauen. Eingerahmt von Abschnitten, die sich mit Struktur und Aufbau der entwickelten Software beschäftigen, werden drei Schritte vorgestellt, die zur Beleuchtungs- und Materialmodifikation führen sollen. Die **Rekonstruktion** eines Kinect-Datensatzes zu einem 3D-Modell, die **Simulation** von Beleuchtung und Materialeigenschaften auf dem Modell und die **Kombination** von zwei Simulationen und dem Farbbild zu einem erweiterten Bild. In Kapitel 5 werden Ergebnisbilder gezeigt. Es wird diskutiert, ob die eingesetzten Verfahren zu einem Erfolg führten und es werden Anregungen gegeben, wie man das Ergebnis noch verbessern kann. Kapitel 6 präsentiert das Gesamtergebnis, um die oben beschriebene Zukunftsprognose zu stellen.

2 Microsoft Kinect

2.1 Der Kinect-Sensor

Kinect wurde am 4. November 2010 von Microsoft veröffentlicht. Das von Microsoft in Zusammenarbeit mit der Firma PrimeSense entwickelte Eingabegerät für die Spielekonsole Xbox 360 ermöglicht es Anwendern, Spiele ausschließlich mittels Körperbewegungen und Stimme zu steuern [CLV12] . Erstmals waren Spieler nicht mehr direkt an Eingabehardware gebunden. Die Verbreitung des Kinect-Sensors ist enorm, weltweit wurden bis Januar 2013 laut Microsoft mehr als 24 Millionen Exemplare verkauft [Meh13].
Abbildung 1 zeigt den Aufbau des Kinect-Sensors. Das Gerät vereint eine RGB-Kamera, einen Tiefensensor, ein Mikrofon-Array und einen Beschleunigungsmesser, mit dem die Orientierung des Gerätes erkannt werden kann. Der Tiefensensor setzt sich aus einem Infrarot-Projektor und einer weiteren Kamera, die mit einem Infrarot Filter ausgestattet ist, zusammen.
Die maximale Auflösung des Tiefenbildes beträgt 640*480 Pixel bei 30 Frames pro Sekunde, die maximale Auflösung des RGB-Bildes 1280*1024 Pixel bei 10 Frames pro Sekunde und 640*480 Pixel bei 30 Frames pro Sekunde. Laut Microsoft beträgt der vertikale Öffnungswinkel der Bilder 57 Grad und der horizontale Öffnungswinkel 43 Grad [Mice].
Die Verbindung von geometrischen Attributen über ein Tiefenbild und visuellen Attributen über ein Farbbild machen Kinect über den Spielebereich hinaus vielseitig einsetzbar, in Bereichen wie Computergrafik, Computer Vision, Bildverarbeitung, Mensch-Maschine-Interaktion und Robotik.
Am 16. Juni 2011 veröffentliche Microsoft ein offizielles Software Development Kit (SDK) für Windows 7, welches es ermöglicht, das Gerät zur Erstellung nicht kommerzieller Anwendungen zu verwenden. Zuvor war dies allerdings schon mit dem plattformunabhängigen SDK von PrimeSense unter dem Framework Open-NI und dem ebenfalls plattformunabhängigen Projekt OpenKinect möglich [AJL+12].
Im März 2012 erschien eine erweiterte Version von Kinect unter dem Namen *Kinect for Windows* speziell für Entwickler [CLV12]. *Kinect for Windows* und das Kinect-SDK wurden im praktischen Teil dieser Arbeit verwendet.

2.2 Erfassung von Tiefenwerten durch Structured Light

Das von der Firma PrimeSense entwickelte Verfahren zum Erstellen eines Tiefenbildes arbeitet mit einer Technik namens *Structured Light*. Der Infrarot Projektor strahlt ein vordefiniertes Muster in den Raum. Es ist für das menschliche Auge unsichtbar. Das projizierte und durch die Geometrie verzerrte Pattern wird von der Infrarotkamera, einer regulären CMOS Kamera mit Infrarotfilter, aufgenommen [AJL+12]. Im Speicher von Kinect ist ein unverzerrtes Bild des Musters in einer bekannten Distanz hinterlegt, mit dem das aufgenommene Bild verglichen wird.

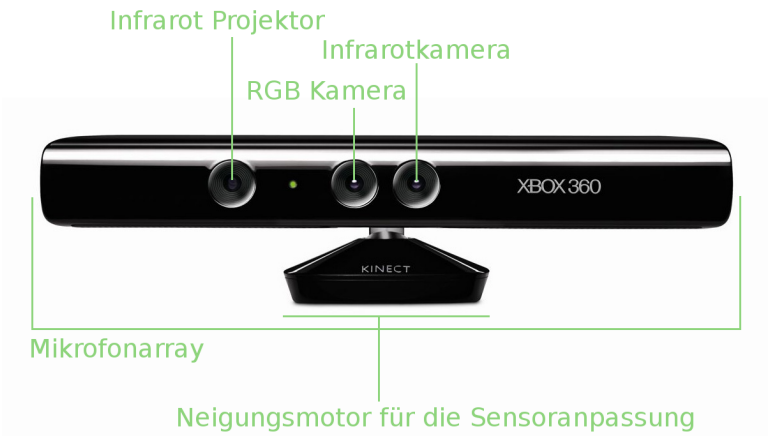

Abbildung 1: Aufbau des Kinect-Sensors[Com]

Für jeden Pixel im aufgenommenen Infrarot-Bild wird das lokale Muster in einem Fenster von 9x9 Pixeln betrachtet [CALT12]. Es wird verglichen mit dem lokalen Muster um den Pixels im gespeicherten Muster und dem Muster um die 64 benachbarten Pixeln in einem horizontalen Fenster. Man ermittelt die Position mit der größten Ähnlichkeit. Diese liefert einen Offset zwischen der für den Punkt erwarteten Position und der durch die Kamera aufgenommenen Position [CLV12].
Anhand der Tiefe des betrachteten Punktes auf einer Ebene in bekannter Distanz und dem wahrgenommenen Offset kann die karthesische Distanz des Pixels zur Kameraebene berechnet werden.
Die Berechnung kann erfolgen, da die relative Position des Projektors zur Kamera bekannt ist. Der vertikale Abstand beträgt bei Kinect 7,5 cm. Horizontal liegen Projektor und Kamera auf der gleichen Höhe [CLV12].

2.3 Genauigkeit der Kinect-Tiefenbilder

Verschiedene Faktoren beeinträchtigen die Qualität des Tiefenbildes. So ist zum Beispiel die Verwendung von Kinect im Freien nicht möglich, da der Infrarot-Anteil des Sonnenlichts das projizierte Punktemuster überdeckt [AJL+12, S. 34].
Bei reflektierenden Materialien wird das Infrarotlicht nicht zurück in die Kamera geworfen, sodass kein korrekter Tiefenwert ermittelt werden kann. Das Gleiche ist bei diffusen Materialien mit starker Lichtabsorption, zum Beispiel bei schwarzen entspiegelten Monitoren, der Fall. Diese können nicht genug Licht zurück in die Kamera werfen, um eine Ermittlung das Tiefenwertes möglich zu machen [AJL+12, S. 34].
Lücken im Tiefenbild sind außerdem neben den Übergängen von Vordergrundbereich zum Hintergrundbereich erkennbar. Diese Schatten im Tiefenbild entstehen, da der Projektor und die Kamera zur Aufnahme des Tiefenbildes etwa um 7,5 cm versetzt sind. Es wird nicht der gleiche Bereich angestrahlt, der aufgenommen

wird, was zur Folge hat, dass Bereiche im Sichtfeld der Kamera liegen, die sich hinter den vom Projektor angestrahlten Objekten befinden. Die Objekte fangen die Projektion ab. Für die aufgenommenen Bereiche, die nicht mit dem Muster angestrahlt werden, ist kein Tiefenwert zu ermitteln. Abbildung 2 verdeutlicht dieses Phänomen.

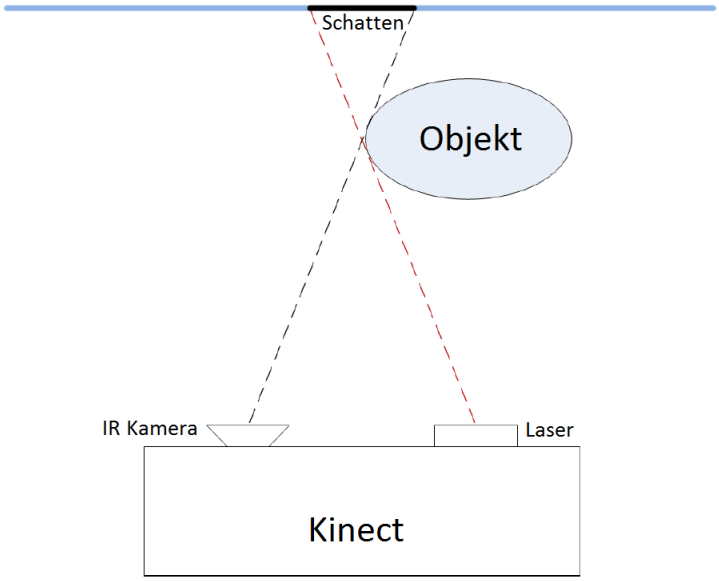

Abbildung 2: Veranschaulichung, wieso Schatten im Tiefenbild auftreten. Im Kamerafrustum liegen Bereiche, auf die kein Muster projiziert wird. Hier kann kein Tiefenwert festgestellt werden [AJL+12, S. 31]

.

Die Tiefe von Bildpunkten kann nur ermittelt werden, falls sie im von Kinect unterstützten Bereich liegt. Beim Sensor *Kinect for XBox 360* liegt dieser zwischen 0,8 und 4 m, der Sensor *Kinect for Windows* unterstützt neben diesem *Default-Modus* einen speziellen *Near* Modus um Distanzen von 0,4 bis 3 m erfassen zu können [Mica]. In [AJL+12] wurde anhand von Experimenten gezeigt, dass die ermittelten Distanzwerte vom Microsoft-SDK linearisiert werden, sodass der Bit-Wert der Tiefe eines Punktes dessen Distanz in mm entspricht. Weiterhin wurde gezeigt, dass bei stationärem Sensor ein über eine bestimmte Zeit hinweg beobachteter Punkt keine konstante Distanz besitzt. So schwankt der ermittelte Tiefenwert eines Pixels bei 1000 beobachteten Frames und einer Distanz von 2 m um etwa 40 mm. Im Tiefenbild macht sich diese Schwankung durch ein Rauschen bemerkbar.
Auch in horizontaler und vertikaler Richtung unterliegt das Tiefenbild einem Rauschen. Bei einer Distanz von 2,2 Metern liegt an Kanten eine Schwankung von insgesamt etwa 4 Pixeln in x und y Richtung vor, was etwa 15 mm entspricht. Sowohl

das Rauschen in horizontaler und vertikaler Richtung, als auch das Tiefenrauschen nehmen bei Steigerung der Distanz zu [AJL+12].

2.4 Einsatz von Kinect im Forschungsbereich

Die Einsätze von Kinect im Forschungsbereich sind vielfältig. In diesem Abschnitt werden Forschungsarbeiten vorgestellt, deren Thematik direkt oder indirekt mit der Thematik dieser Arbeit zusammenhängt.

KinectFusion In [IKH+11] stellt Microsoft das eigens entwickelte *KinectFusion* System vor, bei dem durch Bewegen des Kinect-Sensors in Echtzeit 3D-Rekonstruktionen von Innenräumen erstellt werden können. Ein Tiefenbild wird hier als diskrete Menge von 3D-Punkten in den Raum projiziert. Bei darauf folgenden Frames wird die Position der Kamera relativ vom vorherigen Frame anhand der Tiefendaten berechnet, um mit den 3D-Punkten dieses Frames die vorhandene Punktemenge zu erweitern und deren Genauigkeit zu verbessern. Laut Microsoft wird daran gearbeitet, Funktionen von *KinectFusion* in zukünftigen Versionen des Kinect-SDKs verfügbar zu machen [Whi12]. Im Zusammenhang mit *KinectFusion* findet man auch verschiedene Augmented Reality Anwendungen. Ein gescanntes Modell realer Geometrie wird zum Beispiel durch virtuelle Objekte erweitert, die dynamisch und physikalisch korrekt mit der gescannten Umgebung interagieren.

Kinitinuous Bei *KinectFusion* ist die Größe des erstellten Modells limitiert durch den verfügbaren GPU-Speicher. Das Project *Kintinuous* [WKF+12] stellt eine Erweiterung von *KinectFusion* dar bei dem die Beschränkung auf ein festes Volumen aufgehoben wird. Es können 3D-Modelle von sehr großen Regionen erstellt werden, z.B. von ganzen Gebäuden über mehrere Gebäudeetagen hinweg. Im in dieser Arbeit entwickelten Projekt, soll nur ein einzelnes Foto erweitert werden. Es wird versucht, ob auch ein einzelnes Tiefenbild mit zugehörigem Farbbild ausreicht, um Geometrie zu rekonstruieren.

GrabCut+D Durch den von Kinect und anderen Tiefenkameras gelieferten Tiefenkanal können Bildverarbeitungsprobleme, die mit RGB-Daten komplex sind, vereinfacht werden. Bei dem Projekt *GrabCut+D* [PV11] wurde mit Hilfe von Kinect-Tiefendaten das Grab Cut Segmentierungsverfahren verbessert. Es wurde ein gutes Ergebnis erzielt, da das Tiefenbild eine neue Informationsdimension liefert, die das Finden von Bildregionen erleichtert. In dieser Arbeit wird umgekehrt versucht die Qualität der Bilateralfilterung eines Tiefenbildes durch das korrespondierende Farbbild zu verbessern.

3 Theoretische Grundlagen

Um zu verstehen, wie vorgegangen wurde um Beleuchtungs- und Materialeigenschaften im Kinect-Farbbild zu verändern, wird theoretisches Grundlangenwissen aus verschiedenen Bereichen der Informatik benötigt, das in diesem Kapitel vermittelt werden soll.
Zunächst wird mit dem Bilateralfilter ein Bildverarbeitungsverfahren vorgestellt, das zur Vorverarbeitung eingesetzt wird, um die Qualität des Kinect-Tiefenbildes zu verbessern und das eingesetzt wird, bevor Neubeleuchtungen durchgeführt werden. Anschließend werden in 3.2 die Verfahren vorgestellt, die verwendet werden um Neubeleuchtungen durchzuführen. Diesen liegt eine 3D-Beleuchtungssimulation zugrunde, deren Grundlagen in 3.4 vorgestellt werden. Da die dafür verwendeten Techniken jedoch teilweise zusätzlich zum Farbbild des Kinect-Sensors noch High Dynamic Range Bilder miteinbeziehen, wird dem Kaptitel in 3.3 ein Exkurs zu diesem Thema vorangestellt.

3.1 Bilateralfilter

Das Tiefenbild des Kinect-Sensors unterliegt einem starken Rauschen. Damit die Tiefenwerte denen der tatsächlichen Geometrie näher kommen, muss das Rauschen entfernt werden. Kanten im Tiefenbild sollen jedoch gleichzeitig erhalten bleiben. In [TM98] wird mit dem Bilateralfilter ein Verfahren vorgestellt, das eine kantenerhaltende Filterung für Farb- und Grauwertbilder ermöglicht. Ein Tiefenbild kann hier genau wie ein Grauwertbild behandelt werden. Der Bildwert ist dann kein Grauwert sondern ein Tiefenwert.
Die Idee der bilateralen Filterung ist es, eine Filterung im Orts- mit einer Filterung im Werteraum zu kombinieren. Die Gewichtung eines Nachbarpixel q zum Zentrumspixel p eines Fensters F bestimmt sich damit sowohl in Abhängigkeit von ihrer räumlichen Distanz als auch in Abhängigkeit von der Distanz ihrer Bildwerte.
Es sei c eine Distanzfunktion, die räumlich nahe Pixel bevorzugt und s eine Ähnlichkeitsfunktion, die Pixel mit ähnlichem Wert bevorzugt. Der neue Wert von p bestimmt sich dann mittels Gleichung 1. Der Pixelwert $I(p)$ wird durch den Durchschnitt der ähnlichen und nahen Pixelwerte ersetzt.

$$I(p) = \frac{\sum\limits_{q \in F(p)} c(q,p) * s\Big(I(q), I(p)\Big) * I(q)}{\sum\limits_{q \in F(p)} c(q,p) * s\Big(I(q), I(p)\Big)} \tag{1}$$

Für c und $s$kann jede beliebige Distanz- bzw. Ähnlichkeitsfunktion verwendet werden. Ein Fall, der sich als einfach und effektiv erwiesen hat ist der Gauss'sche Bilateralfilter.

Beide Funktionen sind Gaussfunktionen der euklidischen Distanz ihrer Argumente. Die Distanzfunktion ist dann

$$c(q,p) = e^{-\frac{1}{2}(\frac{d(q,p)}{\sigma_d})^2} \quad q,p \in Loc \tag{2}$$

mit

$$d(q,p) = d(q-p) = \|q-p\| \tag{3}$$

σ_d gibt den Durchmesser des Gausskerns an. Die Ähnlichkeitsfunktion bestimmt sich analog. Hier gibt dann anstelle von σ_d, σ_r die Gewichtung der Bildwerte an. Man wählt σ_d und σ_r abhängig von der beabsichtigten Glättung. Abbildung 3 verdeutlicht die kantenerhaltende Wirkung des Bilateralfilters.

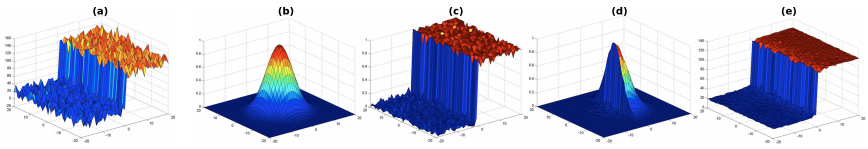

Abbildung 3: (a) Visualisierung eines mit Gauss'schem Rauschen gestörten Bildes, mit Differenz von 100 Grautönen von der rechten zur linken Seite, (b) Visualisierung der Gewichtung mittels Distanzfunktion für einen Pixel im Zentrum, auf der dunklen Seite, (c) Visualisierung der Gewichtung mittels der Ähnlichkeitsfunktion für den gleichen Pixel (d) Ergebnis der Kombination aus beiden Gewichtungsfunktionen, (e) Resultat der bilateralen Filterung. (Abbildung auf Basis von [Kro11])

3.2 Ändern von Beleuchtungs- und Materialeigenschaften

3.2.1 Rendering über Differenz- oder Quotientenbildung

Als Neubeleuchten eines Bildes bezeichnet man den Prozess, bei dem anhand von einem oder mehreren Bildern einer Szene eine beleuchtungskonforme Manipulation der Lichtquellen ermöglicht wird. Dies schließt die Manipulation der Position und Farbe bestehender Lichtquellen, sowie das Hinzufügen neuer Lichtquellen mit ein [ML04]. Änderungen von Materialien lassen sich mit ähnlichen Verfahren wie Neubeleuchtungen erzielen.
Um in einem Foto die Beleuchtung oder Materialien zu manipulieren werden

1. das Originalfoto der Szene,
2. eine 3D-Rekonstruktion der Szene und eine Kalibrierung, die es ermöglicht, 2D-Punkten im Bild einen 3D-Punkt des Modells zuzuordnen,
3. eine Rekonstruktion der Lichtverhältnisse zum Zeitpunkt der Aufnahme

benötigt. Je nach Anspruch an den Realismus müssen auch Rekonstruktionen der Reflexionseigenschaften der im Bild sichtbaren Objekte vorhanden sein.
Den Grundstein für realistische Änderungen von Beleuchtung und Materialien legte A. Fournier 1993 in [FGR92] durch die Entwicklung des *Differentiellen Renderings*. Das Verfahren wurde entwickelt um virtuelle Objekte korrekt beleuchtet in Fotos einzufügen, schließt dabei aber Änderungen von Licht und Material bereits mit ein.
Das *Differentielle Rendering* basiert auf zwei Lichtsimulationen. Die erste rekonstruiert die oben aufgezählten Punkte, um die Lichtverhältnisse im Foto nachzubilden (L_{Old}), die zweite rekonstruiert die gewünschten neuen Materialeigenschaften und Lichtverhältnisse (L_{New}).
Um einen Pixel im Bild zu verändern, verwendet man die Differenz $\triangle L = L_{Old} - L_{New}$ an den Bildkoordinaten u, v und addiert sie zum Ursprungswert eines Pixels I(u,v) um dessen neuen Wert zu erhalten [Gor08].

$$I_{New}(u, v) = I_{Old}(u, v) + \triangle L(u, v) \tag{4}$$

Ist $\triangle L(u, v)$ kleiner als null, bedeutet das, dass ein realer Pixelwert abgedunkelt wird. Ist er höher als null findet eine Aufhellung statt. Werden in der zweiten Simulation veränderte Materialien eingesetzt, werden diese durch Addieren von $\triangle L$ in das Bild übertragen.
Mit einer abgeänderten Version des Verfahrens, vorgestellt von C. Madsen und R. Laurensen in [ML04], können ähnliche Effekte erzielt werden. Sie berechnen das Verhältnis des Lichteinfalls in der neuen Simulation im Verhältnis zu dem in der alten Simulation und multiplizieren dieses mit dem ursprünglichen Pixelwert. Es ergibt sich folgende Gleichung.

$$I_{New}(u, v) = I_{Old}(u, v) * (L_{New}(u, v)/L_{Old}(u, v)) \tag{5}$$

Der folgende Beweis aus [ML04] macht deutlich, dass ein Vorgehen wie in Gleichung 5 bei Beleuchtungsänderungen plausibel ist. Es wird allerdings davon ausgegangen, dass alle im Bild sichtbaren Objekte perfekt diffus reflektieren, also dass sie Licht in alle Richtungen gleich stark reflektieren. Globale Beleuchtungseffekte werden nicht beachtet.
Unterschieden wird zunächst zwischen der Abstrahlung I und Einstrahlung L einer Oberfläche. I ist die Menge des Lichts, das die Oberfläche erreicht und L die Menge des Lichts, das sie nach außen abstrahlt. Im perfekt diffusen Fall hängen diese zusammen durch den diffusen Albedo p_d, dem Rückstrahlvermögen diffus reflektierender Oberflächen.

$$I = \frac{p_d}{\pi} * L \tag{6}$$

In einem zweidimensionalen Bild I stellt jeder Pixel eine Angabe über die auf die drei RGB-Kanäle übertragene Abstrahlung eines bestimmten Punktes im 3D-Raum dar. I_{Old} sei das unmanipulierte Bild, I_{New} das neu beleuchtete Bild.

Anhand des 3D-Modells soll jeder Punkt im Bild einem Punkt im Raum zugeordnet werden können. So kann mit den Informationen über vorhandene Lichtquellen für jeden Bildpunkt $I(u, v)$ die Einstrahlungsstärke bestimmt und in einem Bild $L(u, v)$ abgespeichert werden. Die Einstrahlungsstärke entspricht der Helligkeit des Punktes in einem weißen, perfekt diffusen 3D-Modell der Szene, gerendert mit diesen Lichtverhältnissen und dem OpenGL-Beleuchtungsmodell [ML04]. $L_{Old}(u, v)$ verwendet die Lichtverhältnisse der Originalszene, $L_{New}(u, v)$ sei berechnet mit neu bestimmten Lichtverhältnissen. Der diffuse Albedo und die Einstrahlung ändern sich für jeden Punkt einer Szene. Man kann Gleichung (3) folgendermaßen auf Bilder übertragen:

$$I(u, v) = \frac{p_d}{\pi} * L(u, v) \tag{7}$$

Ein Ändern der Lichtverhältnisse verändert die Einstrahlung an einem Bildpunkt, sein diffuser Albedo bleibt aber unverändert und kann bei der Berechnung von I_{New} beibehalten werden. Es ergibt sich:

$$\begin{aligned} I_{New}(u, v) &= \frac{p_d}{\pi} * L_{New}(u, v) \\ &= ((I_{Old}(u, v)/L_{Old}(u, v)) * L_{New}(u, v) \\ &= I_{Old}(u, v) * (L_{New}(u, v)/L_{Old}(u, v)) \end{aligned} \tag{8}$$

Die Umformung führt zu Gleichung 5 und zeigt, dass sich die Neubeleuchtung eines Pixels im perfekt diffusen Fall über das Verhältnis des neuen Lichteinfalls zum ursprünglichen Lichteinfall berechnen lässt.

3.2.2 Rekonstruktion von Licht

Um Lichtverhältnisse zum Zeitpunkt einer Fotoaufnahme zu rekonstruieren existieren verschiedene Methoden. Sind Abmessungen des Raumes und die relative Position der Lichtquellen zur Kamera, sowie deren Stärke und Größe bekannt, kann versucht werden, die herrschenden Gegebenheiten manuell zu rekonstruieren. Die Genauigkeit des Ergebnisses hängt dann von der Genauigkeit dieser Daten, sowie von den verwendeten Renderingmethoden und dem Anspruch an Echtzeitfähigkeit ab [ML04].
Sind diese Daten nicht verfügbar, kann mit Hilfe eines sogenannten *Light Probe Image* versucht werden, die Lichtverhältnisse zu rekonstruieren. Bei diesem von Paul Debevec entwickelten Verfahren wird ein High Dynamic Range Bild (siehe 3.3) von einer spiegelnden Kugel aufgenommen, die in der Mitte der Szene platziert wird. Die Light Probe wird anschließend zur Simulation einer globalen Beleuchtung verwendet [Deb98]. In 3.4.4 wird näher auf dieses Thema eingegangen. Lichtverhältnisse ausschließlich auf Basis eines einzelnen Bildes zu rekonstruieren stellt ein zumeist nicht eindeutig lösbares Problem dar, denn selbst sehr unterschiedlich konfigurierte Beleuchtungen können in einem Bild identisch aussehen [GMK03]. Um die Position von Lichtquellen zu erhalten, kann wie in [SM00]

versucht werden, im Foto vorhandene Schatten zu erkennen und diese genau mit simulierten Lichtquellen nachzubilden.
Eine Übereinstimmung der Schatten im Foto mit dem Schatten in der Simulation ist bei den genannten Verfahren besonders wichtig, wenn bestehende Lichtquellen bewegt werden sollen. Nur wenn die Schatten im Bild exakt mit den Schatten in der ersten Simulation übereinstimmen, können sie bei einer Neubeleuchtung komplett verschwinden. Ist dies nicht der Fall, bleiben Artefakte im neu beleuchteten Bild zurück [Gor08, S. 34].

3.3 High Dynamic Range Bilder

Es ist nicht möglich, High-Dynamic-Range(HDR) Bilder direkt mit einem Kinect-Sensor zu erstellen. Trotzdem soll das Prinzip kurz vorgestellt werden, da in folgenden Verfahren zusätzlich HDR-Aufnahmen herangezogen werden. Die von den meisten Digitalkameras aufnehmbare Farbtiefe für digitale Bilder beträgt je 8 Bit im Rot-, Grün, und Blaukanal, das heißt es sind 256^3 verschiedene Farben darstellbar. Man bezeichnet diese Bilder als Low Dynamic Range(LDR) Bilder. Die 8 Bit RGB-Farbkanäle reichen häufig nicht aus, um die in realen Szenen vorhandenen Helligkeitsunterschiede zu repräsentieren und geben keine Aussage über die tatsächliche Helligkeit eines Punktes. Die Pixelfarbe hängt von Kameraparametern wie Belichtungszeit und Blendenöffnung ab [RWPD06, S. 7 ff.]. Das Abbilden auf einen Wertebereich von 8 Bit führt dazu, dass Pixel bei Überbelichtung auf den hellsten und bei Unterbelichtung auf den dunkelsten Helligkeitswert abgebildet werden. Der volle Dynamikumfang der Umgebung kann nicht aufgezeichnet werden.
Als High Dynamic Range Bilder werden Bilder bezeichnet, die den vollen Dynamikbereich einer Umgebung erfassen. Eine Reihe von Bildern, die mit unterschiedlichen Belichtungszeiten aufgenommen wurden, kann zu einem HDR-Bild zu kombiniert werden. Eine kurze Belichtungszeit macht im Bild besonders helle Bereiche sichtbar, eine lange Belichtungszeit besonders dunkle Bereiche.
HDR-Bilder können zum Beispiel im *Radiance* Format (.hdr) gespeichert werden, bei dem zum Speichern eines Pixels 32 Bit benötigt werden. Drei Byte geben je eine Mantisse für den Rot-, Grün- und Blaukanal an, ein Byte dient als gemeinsamer Exponent. Mit ihm kann der der Farbwert eines Farbkanals als Fließkommazahl bestimmt werden. Der Farbwert ist direkt proportional zur tatsächlichen Leuchtdichte.[RWPD06, S. 91]
Da die Farbtiefe von regulären Monitoren beschränkt ist, können HDR-Bilder nicht auf ihnen dargestellt werden. Der Vorgang bei dem der Kontrastumfang eines HDR-Bildes wieder soweit verringern wird, dass es auf einem regulären LDR-Display darstellbar ist, wird als *Tone Mapping* bezeichnet. Es existieren viele unterschiedliche Verfahren, die ausführlich in [RWPD06] vorgestellt werden.

3.4 Rendering Grundlagen

Beiden vorgestellten Verfahren zur Neubeleuchtung von Bildern, liegen zwei Beleuchtungssimulationen auf einer geometrischen Rekonstruktion der Szene zugrunde. In diesem Kapitel werden die theoretischen Grundlagen vorgestellt, die zum Erstellen der Simulation mit OpenGL verwendet werden.

3.4.1 OpenGL-Beleuchtungsmodell und Lichtquellen

Mit dem Editor, der in dieser Arbeit erstellt wird, sollen Beleuchtungssituationen manuell konfiguriert werden können, indem Lichtquellen in einer Szene platziert werden. Die Beleuchtung die berechnet wird, basiert auf dem von OpenGL verwendeten Beleuchtungsmodell. Es gibt viele Freiheiten bei der Definition von Beleuchtungen, ist aber im Bezug auf physikalische Gegebenheiten sehr stark vereinfacht. Die Helligkeit eines bestimmten Punktes hängt von seinem Material, den definierten Lichtquellen und einem von den Lichtquellen und der Geometrie unabhängigen Umgebungslicht ab.
Für Lichtquellen und Material wird jeweils eine ambiente Farbe (a_{light} und a_{mat}), eine Farbe für diffuse Reflexion (d_{light} und d_{mat}) und eine Farbe für spiegelnde Reflexion (s_{light} und s_{mat}), angegeben. Für Materialien zusätzlich eine Emissionsfarbe e_{mat}. Die Beleuchtung eines Punktes setzt sich aus folgenden Komponenten zusammen [NFH07, S. 171 ff.]:

Materialemission Die Materialemission e_{mat} stellt den selbstleuchtenden Farbanteil eine Oberfläche dar, den ein Material unabhängig von den Lichtquellen abstrahlt.

Ambiente Beleuchtung Die ambiente Beleuchtung ist unabhängig von der Geometrie. Sie simuliert gestreutes Umgebungslicht und wird global oder pro Lichtquelle angegeben. Die ambiente Lichtfarbe berechnet sich durch

$$a_{mat} * a_{light} \tag{9}$$

Diffuse Beleuchtung Diffuse Beleuchtung geht von Oberflächen aus, die Licht unabhängig von der Blickrichtung reflektieren. Die Oberfläche sieht aus allen Richtungen betrachtet gleich aus. Die Helligkeit eines Punktes hängt vom Winkel α zwischen Lichteinfallsvektor und Normale am Punkt, auf den das Licht trifft, ab. Am stärksten ist das Licht bei senkrechtem Lichteinfall. Je flacher der Einfallswinkel, desto niedriger wird die eingestrahlte Lichtintensität. Die diffuse Lichtfarbe berechnet sich durch

$$d_{mat} * d_{light} * cos\alpha \tag{10}$$

Spiegelnde Reflexion Ein Lichtvektor wird nach dem Prinzip *Einfallswinkel gleich Ausfallswinkel*, im Bezug auf die Normale, von einer spiegelnden Oberfläche reflektiert. Die Helligkeit des Punktes ist vom Winkel zwischen der Blickrichtung und dem reflektierten Vektor abhängig. Je spitzer der Winkel, umso stärker

die Reflexion. Ein Spiegelungsexponent n gibt die Glätte des Materials an. Die reflektierte Lichtfarbe berechnet sich durch

$$s_{mat} * s_{light} * cos^n \psi \tag{11}$$

Die einzelnen Komponenten verhalten sich additiv. Materialemission und globale diffuse Beleuchtung dienen als Basisfarbe, zu der für jede Lichtquelle, die für sie berechnete ambiente, diffuse und spiegelnde Helligkeit addiert wird. Die Multiplikation zweier Farbverktoren erfolgt komponentenweise [NFH07, S. 174 ff.]. OpenGL definiert drei Modelle zur Beschreibung von Lichtquellen. Die Wahl der Lichtquelle wirkt sich darauf aus, in welchem Winkel das Licht der Lichtquelle auf die Oberfläche trifft.

Gerichtete Lichtquelle Für eine gerichtete Lichtquelle wird keine Position, sondern lediglich eine Richtung angegeben. Alle Lichtstrahlen treffen parallel auf die Oberfläche. Im OpenGL-Positionsvektor mit vier Komponenten geben die ersten drei Komponenten die Richtung des Lichtes an. Die gerichtete Lichtquelle kennzeichnet sich dadurch, dass die vierte Komponente null ist, was bedeutet, dass die Lichtquelle im Unendlichen sitzt [NFH07, S. 180].

Punktlichtquelle Eine Punktlichtquelle strahlt ihr Licht gleich stark in alle Richtungen ab. Im Positionsvektor geben die ersten drei Kompontenen die Position im 3D-Raum an. Die vierte Komponente ist ungleich null. Die Lichtstärke nimmt bei steigendem Abstand zwischen Lichtquelle und beleuchtetem Punkt ab, je nach Höhe eines konstanten, eines linearen und eines quadratischen Abschwächungsfaktors [NFH07, S. 180].

Spotlichtquelle Für ein Spotlicht werden neben der Position eine Richtung, ein Öffnungswinkel, ein Exponent und drei Abschwächungsfaktoren angegeben. Die Lichtstärke ist nicht in alle Richtungen konstant, sondern nimmt cosinusförmig ab, je weiter ein Lichtstrahl von der Hauptrichtung abweicht. Die Höhe des Exponenten gibt die Stärke dieser Abnahme an. Wie bei der Punktlichtquelle nimmt die Lichtstärke bei steigendem Abstand zwischen Lichtquelle und beleuchtetem Punkt ab [NFH07, S. 187].

3.4.2 Shadow Mapping

Zu einer veränderten Beleuchtung gehören auch veränderte Schatten. Zu deren Realisierung wird in dieser Arbeit Shadow Mapping eingesetzt, ein einfach zu implementierendes Verfahren, um in Echtzeit und unabhängig von der Komplexität der Geometrie Schatten zu berechnen. Grundlage des Verfahrens ist die Annahme, dass ein Punkt der Szene beleuchtet ist, falls nichts den direkten Weg des Lichts von der Lichtquelle zu dem Punkt blockiert. Um zu erkennen, ob dies der Fall ist, wird die Szene aus Sicht der Lichtquelle gerendert, um die Tiefenwerte der sichtbaren Geometrie in eine Textur, der Shadow Map, zu speichern. Die Textur enthält

also die Distanzen der von der Lichtquelle beleuchteten Punkte zur Lichtquelle. Diese von der Lichtquelle aus sichtbaren Punkte sind beleuchtet, die nicht sichtbaren Punkte liegen im Schatten [RLKG+09, S. 387]. In dieser Arbeit beschränkt sich der Einsatz des Shadow Mappings auf Spotlichtquellen, da für sie eine eindeutige Blickrichtung definiert ist.
Falls mehrere Lichtquellen existieren, die Schatten verursachen sollen, wird die Szene für jede Lichtquelle gerendert. Je mehr Lichtquellen existieren, desto rechenaufwändiger ist die Schattenberechnung [RLKG+09, S. 386].
In einem finalen Rendering-Durchgang aus der Kameraperspektive werden die Shadow Maps von den einzelnen Lichtquellen aus in die Szene projiziert. An einem Punkt der Szene wird jetzt der Wert, beziehungsweise die Distanz des Punktes in der projizierten Textur an dieser Stelle, mit der Distanz des Punktes zur entsprechenden Lichtquelle verglichen. Sind die Distanzen gleich, erreicht das Licht den Punkt, ohne blockiert zu werden. Ist die Distanz des Punktes größer als der Texturwert bedeutet das, dass sich etwas zwischen dem Punkt und der Lichtquelle befindet und der Punkt im Schatten liegt [RLKG+09, S. 386].
Um die Texturprojektion durchzuführen wird eine Matrix benötigt, die eine Vertex-Koordinate in eine projektive Texturkoordinate für eine bestimmte Lichtquelle überführen kann. Die Berechnung ergibt sich wie folgt.:

$$\begin{vmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{vmatrix} \times M \times V_{light} \times P_{light} \times \begin{vmatrix} x \\ y \\ z \\ w \end{vmatrix} = \begin{vmatrix} s \\ t \\ r \\ q \end{vmatrix} \tag{12}$$

M dient zur Transformation von Modell- in Weltkoordinaten, V_{light} ist die View-Matrix der Lichtquelle, P_{light} die Projektions Matrix der Lichtquelle. Die Projektionsmatrix überführt einen Vertex in den Clip-Space der Kamera mit einem Ortsbereich von [-1, 1]. Die erste Matrix im Term transformiert den Punkt aus dem Clip-Space in einen Wertebereich von [0, 1], den Wertebereich von Texturkoordinaten [RLKG+09, S. 387]. Da diese Transformation die homogenen Koordinaten verändert, müssen s, t und r durch q geteilt werden, um die Texturkoordinaten und die Tiefe zu erhalten.

Vergleicht man den Tiefenwert in der Textur und die Distanz des Punktes zur Lichtquelle, kann der Punkt entweder im Schatten liegen oder nicht. Dies führt zu harten und unnatürlichen Schattenkanten. Um weichere Schattenkanten zu erhalten, kann die Technik des *Percentage-Closer Filterings* verwendet werden [PM]. Man vergleicht hierfür die Distanz des Punktes zur Lichtquelle mit allen Texturwerten in einem Fenster um den ermittelten projizierten Texturpunkt und berechnet anschließend den Durchschnitt. Je mehr Punkte abgetastet werden, umso weicher die Schattenkanten. Je größer das abgetastete Fenster aber gewählt wird, umso mehr Vergleiche müssen gemacht werden und umso schlechter wird die Performance des Shadow Mappings.

3.4.3 Cube Mapping

Als Cube Map bezeichnet man eine Textur bestehend aus sechs quadratischen 2D-Texturen, die die sechs Seiten eines Würfels repräsentieren [RLKG+09, S. 309]. Es ist möglich, eine vollständige 360 Grad Ansicht einer Umgebung mit dem Würfel aufzuzeichnen. Jede Würfelseite zeigt ein Foto aufgenommen in positive und negative Richtung der drei Raumrichtungen (oben, unten, rechts, links, vorne, hinten). Für einen Betrachter in der Mitte des Würfels bilden sie unverzerrt die Umgebung ab. Abbildung 4 zeigt den Aufbau einer Cube Map im Vertical Cross Format, wobei P und N für positiv und negativ stehen und X, Y und Z die Achsen ausgehend von einem Koordinatenursprung in der Mitte des Würfels angeben. Cube Map-

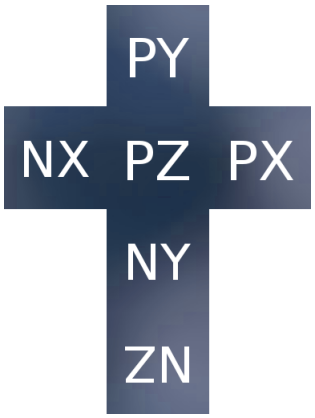

Abbildung 4: Aufbau eine Cube Map im Vertical Cross Format

ping wird in der Arbeit verwendet, um Umgebungsspiegelungen in Bilder einzufügen und um eine natürlich basierte Beleuchtung durchzuführen, wobei letzteres im nächsten Kapitel vorgestellt wird. Spiegelungen werden mit einem GLSL-Shader auf ein 3D-Modell der Szene übertragen und lassen sich anschließend wie Materialänderungen ins Foto übertragen. Um spiegelnde Objekte zu simulieren, wird angenommen, dass die Cube Map das zu rendernde Objekt umschließt. Mit einem Richtungsvektor wird ein Farbwert der Umgebung auf das Objekt übertragen. Der Winkel zwischen dem Richtungsvektor und der Normale eines Vertex muss hier der gleiche sein wie der Winkel zwischen der Normalen und der Richtung vom Punkt zum Auge des Betrachters [Wig08, S. 473]. OpenGL kann die Texturkoordinaten einer Cube Map so generieren, dass der Richtungsvektor zur Texelermittlung dienen kann [NFH07, S. 268]. Wird er als Texturkoordinate verwendet, erhält man den Texel im Schnittpunkt von Richtungsvektor und Cube Map.

3.4.4 Image Based Lighting

Lichtverhältnisse mit OpenGL zu rekonstruieren kann je nach Umgebung sehr aufwändig sein. Beim Image Based Lightning, entwickelt vom Forscher Paul Debevec, werden HDR-Cube Maps verwendet, um die Umgebungsbeleuchtung zu si-

mulieren. Grundlage des Verfahrens stellt ein so genanntes *Light Probe Image* dar, ein omnidirektionales HDR-Bild, das die einfallenden Lichtverhältnisse an einem bestimmten Punkt im Raum aufzeichnet [RLKG+09, S. 361]. Eine Light Probe kann aus einem oder mehreren Fotos von einer spiegelnden Kugel erstellt werden. Mit zwei Fotos kann bereits ein kompletter Raum erfasst werden. Abbildung 5 wurde im Rahmen dieser Arbeit eine Light Probe nach dem in [Deb] beschriebenen Verfahren erstellt. Mit Paul Debevecs Tool *HDR Shop*, kann aus einer Light

Abbildung 5: Light Probe einer Umgebung

Probe eine HDR-Cube Map erstellt werden, mit der man, wie in Kapitel 3.4.3 beschrieben, vollständig spiegelnde Objekte simulieren kann.
Bei diffus reflektierenden Objekten reflektiert ein Punkt jedoch nicht nur einen einzigen Umgebungspunkt, sondern das Licht von allen Lichtquellen, die in der Hemisphäre in Richtung seiner Normalen liegen. Dies enspricht dem gewichteten Durchschnitt aller Punkte dieser Hemisphäre. Um unnötig viele Texturzugriffe zu vermeiden, kann dieser Wert mit *HDR Shop* für jeden Punkt vorberechnet und ebenfalls als Cube Map gespeichert werden. Zur Simulation von diffus reflektierenden Oberflächen greift man nun auf diese Cube Map zu, verwendet allerdings nicht den Reflektionsvektor, sondern die Normale zum Texturzugriff [RLKG+09, S. 362].
Soll die Oberfläche nicht perfekt diffus sein, aber auch nicht perfekt spiegelnd, wird weder ein einziger Punkt noch die ganze Hemisphäre reflektiert. Auch hierfür lässt sich mit *HDR Shop* eine Cube Map vorberechnen. Ein *Phong Exponent* gibt den Glanz der Oberfäche an. Er ist für diffuse Oberflächen 1.0. Auf die spekulare Cube Map wird mit dem Richtungsvektor (Kapitel 3.4.3) zugegriffen. Theoretisch können beim Image Based Lighting beliebig viele Cube Maps verwendet werden, die unterschiedliche Reflektionsstärken simulieren. Es wird eine Gewichtung angegeben, die bestimmt, wie stark die einzelnen Texel in den finalen Farbwert eines Punktes eingehen. Abbildung 6 zeigt eine aus der Light Probe erstellte Cube Map

(links), die Cube Map nach diffuser Konvolution mit Phong Exponent von 1.0 (mitte) sowie die Cube Map nach spiegelnder Konvolution mit Phong Exponent von 50.0 (rechts).

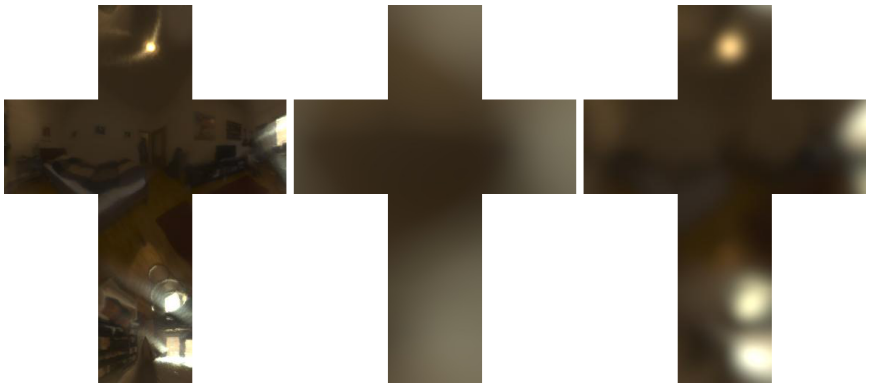

Abbildung 6: Cube Maps für Image Based Lighting: ungefiltert, nach diffuser Konvolution, nach spekularer Konvolution (v.l.n.r.).

4 Vorgehen

Zum Lösen der Aufgabenstellung wurde nach Verfahren gesucht, die Geometrie zu Hilfe nehmen, um Beleuchtung und Material zu verändern. Die zwei Verfahren aus Kapitel 3.2.1 wurden dabei gefunden. Bei beiden ist das Wissen über die Geometrie eine Voraussetzung zum Ändern von Beleuchtung und Material. In der zugehörigen Literatur wurde die Geometrie manuell rekonstruiert [ML04], [FGR92]. Es ist denkbar, dass eine Geometrierekonstruktion direkt bei Aufnahme des Fotos große Erleichterung bringen kann.

Das Tiefenbild des Kinect-Sensors liefert zu jedem Bildpunkt eine kartesische Distanz zur Kameraebene und es stellt somit keineswegs eine dreidimensionale Rekonstruktion dar. Es wird gefordert, dass zu jedem Bildpunkt im Foto ein korrespondierender Punkt in einem 3D-Modell der Szene vorhanden ist. Folglich muss im ersten Schritt ein Weg gefunden werden, das Tiefenbild des Kinect-Sensors in ein dreidimensionales Modell zu konvertieren. Kapitel 4.2 wird im Detail auf die Vorgehensweise eingehen, die eingesetzt wird, um eine 3D-Rekonstruktion mit möglichst guter Qualität zu erhalten und dabei auf zuvor erwähnte Probleme der Tiefendaten eingehen.

Liegt eine Rekonstruktion vor, wird für die Bilderweiterung gefordert, dass eine Beleuchtungs- und Materialsimulation auf diesem Modell durchgeführt werden kann. Die Grafikschnittstelle OpenGL bietet viele Möglichkeiten zur Definition von Materialien und Beleuchtungen und scheint darum gut geeignet. Damit die Simulation in Echtzeit durchgeführt werden kann, wird die OpenGL-Erweiterung GLSL verwendet, die Berechnungen mit der für 3D-Grafik optimierten Grafikhardware durchführt. Es muss eine Auswahl an Renderingtechniken gefunden werden, die geeignete Simulationen möglich machen. Grundlagen dieser Techniken wurden bereits in 3.4 geliefert. Genaueres zur Motivation und Umsetzung liefert das Kapitel *Grafiksimulation*.

Für eine Bilderweiterung werden zwei Beleuchtungssimulationen benötigt. Die erste Simulation soll die Beleuchtungs- und Materialeigenschaften, die zum Zeitpunkt der Aufnahme herrschten, rekonstruieren, die zweite die gewünschten Eigenschaften. Die Beleuchtung wird manuell rekonstruiert. Auf die manuelle Rekonstruktion von Materialien wird nicht weiter eingegangen, da in einem Foto zumeist zahlreiche unterschiedliche Materialien auftreten und dies zu aufwendig wäre.

Nachdem zwei Lichtsimulationen konfiguriert werden, besteht der letzte Schritt, um ein erweitertes Bild zu erhalten, darin, die Simulationen mit dem Foto zu kombinieren. Um dies mit OpenGL zu ermöglichen wurde eine Rendering Pipeline entwickelt, die in 4.4 vorgestellt wird.

Um die verwendete Vorgehensweise nachvollziehbar zu machen, wurde die Software **Kinect - Augmented Images** entwickelt. Die Software bildet den Rahmen dieses Kapitels. Sie soll das Vorgehen der Bilderweiterung mit einem Kinect-Sensor nicht nur nachvollziehbar, sondern auch anwendbar machen. Beleuchtungssimulationen selbst über eine Oberfläche konfigurieren zu können, was die Software ermöglichen soll, macht das ganze Verfahren zudem wesentlich intuitiver und eröffnet den nötigen Spielraum zum Experimentieren, der trotz aller Theorie für das Erhalten guter Ergebnisbilder benötigt wird.

4.1 Verwendete Technologien und Bibliotheken

Bei der entwickelten Software handelt es sich um eine Windows Presentation Foundation-Anwendung (WPF). WPF ist ein Grafik-Framework zum Erstellen von Clientanwendung für Windows und bildet einen Teil des .NET Frameworks von Microsoft. Eine WPF-Anwendung setzt sich zusammen aus *Markup* und *Code-Behind*. *Markup* bezeichnet die Darstellung der Anwendung, die mit der auf XML basierenden Programmiersprache XAML (Extensible Application Markup-Language) implementiert wurde. *Code-Behind* bezeichnet den Programmcode, der die Oberfläche verwaltet [Micd]. Hierzu wurde in dieser Arbeit die Programmiersprache C# verwendet. Weiterhin wurden die folgenden externen Bibliotheken verwendet:

Tao Framework 2.1
: Tao Framework ist eine Sammlung von Schnittstellen und Grafik Bibliotheken. Die Schnittstellen ermöglichen es, die ursprünglich für andere Programmiersprachen entwickelten Bibliotheken mit C# zu verwenden.[1]

Kinect for Windows SDK 1.6
: Das Software Development Kit liefert die Treiber, um den Kinect-Sensor unter Windows zu verwenden, sowie die APIs um Kinect-Funktionalität mit C# und XAML verwenden zu können.[2]

Es wurden die Tao Klassen *Tao.OpenGL*, *Tao.Platform.Windows* und *Tao.DevIl* verwendet. Die Erste ermöglicht die Verwendung von OpenGL und GLSL unter C#. Die gesamte 3D-Grafikprogrammierung basiert auf OpenGL. *Tao.Platform.Windows* ermöglicht das Einbinden eines OpenGL-Fensters in das XAML-Markup. *Tao.DevIl* ermöglicht die Verwendung der Programmierschnittstelle *DevIl* zum Laden und Speichern von Grafiken.

4.2 Von den Rohdaten zum 3D-Modell

Die Grundlage der durchzuführenden Bildmanipulation bildet ein Kinect-Datensatz bestehend aus Farb- und Tiefenbild. Alles was benötigt wird, um einen über USB angeschlossenen Kinect-Sensor zu erhalten, liefert das Kinect-SDK.

[1] Download unter http://sourceforge.net/projects/taoframework/

[2] Download unter http://www.microsoft.com/en-us/download/details.aspx?id=34808

Der Kinect-Sensor sendet sein Farb- und Tiefenbild als Stream, sobald ein Frame verfügbar ist wird ein Event ausgelöst, das eine Methode aufruft, in der das vorhandene Bild abgerufen werden kann.
Hier kann das Farbbild in ein Byte-Array konvertiert werden. Jeder Pixel wird als RGBA-Wert durch 4 Bytes repräsentiert. Das Tiefenbild wird in einen Array vom Typ Short konvertiert.
Es ist wichtig zu beachten, dass nur die ersten 13 Bit eines solchen Short-Wertes die Tiefeninformation enthalten und die letzten 3 Bit für einen *DepthPlayerIndex* reserviert sind, der verwendet wird, um unterschiedliche Personen vor dem Sensor zu unterscheiden [Micc]. Um einen Tiefenwert zu erhalten, muss also ein Bitshift um 3 Bit nach rechts durchgeführt werden. Die beiden Arrays bilden die Grundlage für die folgenden Operationen.

4.2.1 Angleichen von Tiefenbild und Farbbild

Bei einem Kinect-Sensor haben die Kamera zur Aufnahme des Tiefenbildes und die Kamera zur Aufnahme des Farbbildes einen Abstand von 2,5 cm [CLV12]. Dies hat zur Folge, dass die beiden Bilder aus leicht versetzten Perspektiven aufgenommen werden, ähnlich wie bei stereoskopischen Kameras. Legt man Tiefenbild und Farbbild übereinander, erkennt man diese Verschiebung. Kanten im Tiefenbild stimmen nicht mit Kanten im Farbbild überein.Die erste Zeile von Abbildung 7 verdeutlicht dies. Es ist außerdem zu erkennen, dass der Tiefensensor einen etwas kleineren Bildbereich erfasst als die RGB-Kamera.

Das Kinect-SDK liefert über die Klasse `CoordinateMapper` zwei Funktionen, um die Verschiebung für ein vollständiges Frame auszugleichen, `MapDepthFrameToColorFrame` und `MapColorFrameToDepthFrame` [Micb]. Anhand des Formats von Farb- und Tiefenbild (Auflösung und Frames pro Sekunde) und dem Tiefen-Array wird ein Array erzeugt, mit dem sich Punkte der beiden Bilder einander zuordnen lassen. Wird `MapDepthFrameToColorFrame` verwendet, enthält es für einen Punkt im Tiefenbild die 2D-Position des korrespondierenden Punktes im Farbbild. Diese Information genügt, um ein neues Farbbild zu konstruieren, das auf das Tiefenbild ausgerichtet ist. Jedem Punkt im neuen Farbbild wird der Farbwert zugewiesen, auf den der gelieferte Array an dieser Position zeigt. Bei `MapColorFrameToDepthFrame` enthält das Zuordnungsarray für einen Punkt im Farbbild die Position des korrespondierenden Punktes im Tiefenbild. Ein neues Tiefenbild kann hier auf die gleiche Art konstruiert werden. In der entwickelten Software bestimmt die Checkbox *DepthToColor*, ob das Farbbild an das Tiefenbild, oder das Tiefenbild an das Farbbild angeglichen wird.

In Abbildung 7 werden die Mapping Richtungen verglichen. In der zweiten Zeile wurde das Farbbild auf das Tiefenbild gemappt. Der vom Tiefensensor nicht erfasste Randbereich wird im neu konstruierten RGB-Bild abgeschnitten. Es sind kleine Bildartefakte, zum Beispiel am Griff des Schrankes oder an Kante von Stuhl

und Tisch zu erkennen, die durch die versetzten Kameras zu erklären sind. Einem Punkt, der im Tiefenbild aber nicht im Farbbild sichtbar ist, wird ein falscher, benachbarter Farbwert zugeordnet. In Zeile drei wurde das Tiefenbild auf das RGB-Bild übertragen. Da für den Randbereich von diesem jedoch keine Tiefenwerte vorliegen, können diese beim Mapping auch nicht ergänzt werden. Ein Vorteil des Mappings in diese Richtung ist, dass das Farbbild nicht verändert wird und somit auch keine Artefakte in diesem entstehen können. Beim Neubeleuchten kann also auf dem Originalbild gearbeitet werden. Die Genauigkeit des Mappings ist für beide Richtungen ähnlich.

Abbildung 7: Übereinanderlegen von Farbbild (rechts) und Tiefenbild (mitte) ohne Justierung (1. Zeile), mit auf Tiefenbild gemapptem Farbbild (2. Zeile) und auf Farbbild gemapptem Tiefenbild (3. Zeile). Das Tiefenbild wurde bereits als Rekonstruktion visualisiert um Kanten besser sichtbar zu machen.

4.2.2 Schließen der Lücken im Tiefenbild

In Kapitel 2.3 wurden Faktoren beschrieben, die dazu führen, dass Lücken im Tiefenbild auftreten. In der entwickelten Software wird ein iterativer Bildfilter, basierend auf [San12], verwendet, mit dem fehlende Tiefeninformation anhand der umliegenden Tiefeninformationen rekonstruiert wird. Die hierzu verwendete Methode `CreateFilteredDepthArray` erhält das ungefilterte Tiefen-Array als Argument und liefert ein gefiltertes Tiefen-Array, bei dem für jeden Bildpunkt ein

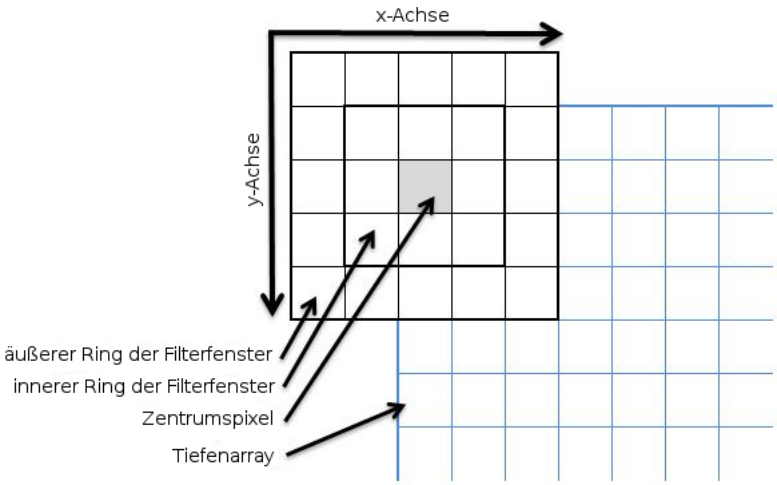

Abbildung 8: Darstellung des Filterfensters mit innerem und äußerem Ring um einen Punkt im Tiefen-Array [San12]

Tiefenwert vorhanden ist. Bereits vorhandene Tiefeninformationen bleiben unverändert.
Die Tiefendaten liegen als eindimensionales Array vor, da Breite $width$ und Höhe $height$ des Tiefenbildes bekannt sind, kann wie beim Filtern eines zweidimensionalen Bildes mit zwei verschachtelten Schleifen in x und in y Richtung über das Array gelaufen werden. Hierdurch ist die Position u, v im zweidimensionalen Tiefenbild, als auch durch $u + (v * width)$ der Index im Array bekannt. Die Methode läuft über das Array und beachtet dabei ausschließlich die Punkte, für die keine Tiefeninformation gefunden wurde. Dies kennzeichnet sich durch einen Tiefenwert kleiner oder gleich null. Für jeden Pixel werden nun 24 Nachbarpixel, aufgeteilt in zwei Ringe, betrachtet. Der erste Ring enthält die acht direkt angrenzenden Pixel, der zweite Ring enthält die Pixel, die wiederum direkt an diese acht Nachbarpixel angrenzen (siehe Abbildung 8). Es wird ermittelt, welche Tiefenwerte in dieser Nachbarschaft auftreten und wie oft sie jeweils auftreten. Tiefenwerte kleiner oder gleich null werden auch bei Betrachtung der Nachbarpixel nicht beachtet. Es existiert ein Zähler für den inneren Ring und ein Zähler für den äußeren Ring, der immer dann inkrementiert wird, wenn im entsprechenden Ring ein Pixel betrachtet wird, für den ein Tiefenwert vorliegt. Ein neuer Tiefenwert wird gesetzt, wenn mindestens für einen Pixel im inneren Ring oder für drei Pixel im äußeren Ring Tiefeninformationen gefunden wurden. Der Grenzwert für den inneren Ring wurde niedriger als der für den äußeren Ring gewählt, um direkt benachbarte Pixel zu bevorzugen. Der Wert des betrachteten Pixels wird durch den Tiefenwert ersetzt, der unter den 24 Nachbarpixeln am häufigsten vorhanden war.
Insgesamt wird das Tiefen-Array so oft durchlaufen, bis Tiefenwerte für alle Punk-

te ohne Tiefeninformation gefunden sind. Erst wenn bei einem Durchlauf kein Pixel mehr geändert wird, wird es zurückgegeben.

4.2.3 Glätten des Tiefenbildes

Zur Glättung des Tiefenbildes wird eine erweiterte Form des in Kapitel 3.1 vorgestellten Gauss'schen Bilateralfilters verwendet. Die Erweiterung besteht darin, dass bei der Filterung im Wertebereich nicht nur die Distanz der Tiefenwerte als Kriterium verwendet wird, sondern zusätzlich auch der Unterschied des Farbwertes des Zentrumspixels zum Nachbarpixel. Da bei starken Konturen in einem Farbbild oft auch Konturen der Geometrie vorliegen und Farb- und Tiefenbild so aneinander angeglichen sind, dass jedem Tiefenwert ein Farbwert zugeordnet werden kann, ist es plausibel das Farbbild als zusätzliche Informationsquelle zu verwenden. Für ein Tiefenbild I_1 und ein Farbbild I_2 erweitert sich Gleichung 1 zu Gleichung 13.

$$I_1(p) = \frac{\sum\limits_{q \in F(p)} c(q,p) * s\Big(I_1(q), I_1(p)\Big) * t\Big(I_2(q), I_2(p)\Big) * I_1(q)}{\sum\limits_{q \in F(p)} c(q,p) * s\Big(I_1(q), I(p)_1\Big) * t\Big(I_2(q), I_2(p)\Big)} \tag{13}$$

Die Funktion t ist eine Gaussfunktion der euklidischen Distanz zweier Farbwerte im CIE-Lab-Farbraum. Der CIE-Lab-Farbraum wird verwendet, da bei diesem die euklidische Distanz zweier Farben im Farbraum an ihre Distanz in der menschliche Wahrnehmung angepasst ist. Beim RGB-Farbraum wäre dies nicht der Fall. In [TM98] können mehr Informationen zum CIE-Lab Farbraum gefunden werden.

Ein Berücksichtigen des Farbwertes beim Glätten hat zur Folge, dass Geometriestrukturen mit einem Tiefenunterschied, der kleiner ist als der Rauschradius im Tiefenbild (z.B. 4 cm bei einer Distanz von 2 m) erhalten bleiben können. Über ein Sigma für t kann quasi der Detailgrad festgelegt werden, der im Tiefenbild erhalten bleiben soll. Beim Schließen der Lücken im Tiefenbild kann es zu Fehlern kommen. Falls der falsche Tiefenwert noch in einem Rahmen liegt, der von s berücksichtigt wird, gehen umliegende Punkte mit ähnlicher Farbe in die Gewichtung ein. Die Punkte, die zwar einen näheren (falsch zugeordneten) Tiefenwert haben, sich aber farblich stark unterscheiden, werden nicht berücksichtigt. Die Geometrie passt sich also an die farblich ähnliche Struktur an.
Insgesamt kann das σ der Gaussfunktion für die Tiefenwerte beim Glätten höher angesetzt werden, wenn der Farbwert berücksichtigt wird. Durch die Farbwerte ist ein zusätzliches Kriterium vorhanden, das verhindert, dass Tiefenwerte mit zu stark abweichender Farbe beim Glätten berücksichtigt werden. Im Programm kann die Zahl der Iterationen, die Größe des Kerns des Bilateralfilters und σ für alle drei Gaussfunktionen festgelegt werden. Um das Rauschen annähernd vollständig zu entfernen, muss entweder ein sehr großer Kern gewählt werden oder mehrmals über das Bild iteriert werden.

Bei einem Kinect-Tiefenbild der Größe 640*480 führen die folgenden Einstellungen zu guten Ergebnissen:

$Iterationen$	3
$Durchmesser$	21
$\sigma_{Ortsbereich}$	15
σ_{Tiefe}	8
σ_{Farbe}	3

Abbildung 9 zeigt links ein mit Phong Shading gerendertes Tiefenbild der Auflösung 640*480 ohne Glättung, in der Mitte geglättet mit den obigen Einstellungen ohne Berücksichtigung von σ_{Farbe} und rechts geglättet mit $\sigma_{Farbe} = 3$. Verbesserungen im Tiefenbild sind rechts zum Beispiel am Laptop, der Kleidung oder dem Bücherregal eindeutig zu erkennen. Der Wert für σ_{Farbe} sollte nicht kleiner als drei gewählt werden, da dies zu einer Überdetaillierung im Tiefenbild führt, was sich später störend bei Bilderweiterungen bemerkbar macht.

Abbildung 9: Gerendertes Tiefenbild ohne Glättung (links), geglättet mit Bilateralfilter (mitte), geglättet mit Bilateralfilter, der die Farbe berücksichtigt (rechts).

4.2.4 Transformation der Tiefendaten

Das Kinect-SDK liefert Tiefendaten als Array, das für jeden Bildpixel dessen Distanz zur Kameraebene in mm enthält [Mica]. Erzeugt werden soll jedoch ein perspektivisches Abbild der im zugehörigen RGB-Bild sichtbaren Geometrie.
Direktes Abbilden einer Pixelkoordinate mit zugehörigem Tiefenwert in den 3D-Raum führt wegen der unterschiedlichen Maßeinheiten und der auf Parallelprojektion basierenden Tiefendaten nicht zu einer korrekten Position. Es muss also eine Transformation gefunden werden, die einen 2D-Tiefenbildpunkt mit den Bildkoordinaten u, v und einem Tiefenwert $depth$, zu einem Punkt P (x,y,z) im 3D-Raum transformiert. Zur Berechnung werden die folgenden Parameter benötigt:

- Auflösung des Tiefenbildes
- Position und Ausrichtung der Kamera
- horizontaler und vertikaler Öffnungswinkel des Tiefensensors

Als Auflösung des Tiefenbildes wurde die maximale von Kinect unterstützte Auflösung 640*480 gewählt. Der vertikale Öffnungswinkel der Kamera wurde entsprechend der Microsoft Spezifikation auf 43 Grad festgelegt [Mice]. Für den horizontalen Öffnungswinkel haben 55,5 Grad zum besten Ergebnis geführt. Die Koordinaten werden bezüglich einer im Ursprung liegenden Kamera mit Blick in negative z-Richtung berechnet. Wichtig ist, dass die OpenGL-Kamera in dieser Position und Ausrichtung das Sichtfeld von Kinect simuliert. Das Abbild der Geometrie stimmt dann nach der Transformation in Größe und Perspektive exakt mit der Geometrie auf dem aufgenommenen RGB-Bild überein. Die hierfür verwendete Kamera ergibt sich im Programmcode durch

```
gluPerspective(43f, 640f / 480f, 100, 15000);
```

Abbildung 10 zeigt das Bildkoordinatensystem der Kinect-Tiefendaten sowie das verwendete Kamerakoordinatensystem und ihren Zusammenhang. Bei Erzeugung

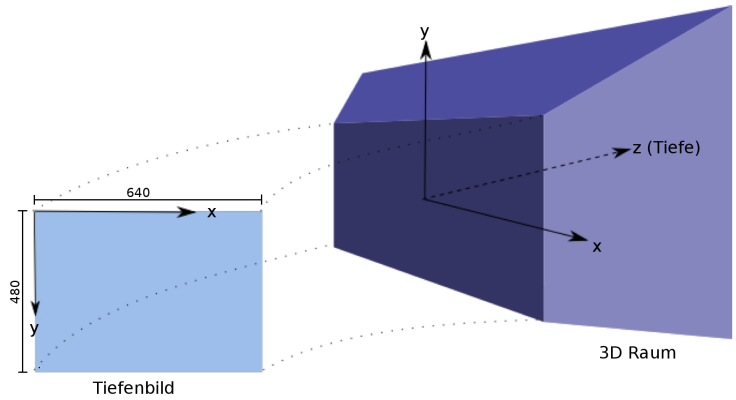

Abbildung 10: Darstellung der Transformation vom Bildbereich des Tiefenbildes in den 3D-Raum [Hö12]

der 3D-Punkte wird die Tiefe eines Punktes beibehalten, die x und y Koordinaten werden neu bestimmt. Man ermittelt zunächst mit Hilfe des bekannten Öffnungswinkels, welchen maximalen Abstand in mm ein Punkt mit einer bestimmten Distanz (diese sei der Einfachheit halber Eins) zum Bildzentrum haben darf, um noch dargestellt zu werden. Bei einem halben Öffnungswinkel β sei das Verhältnis vom maximalen sichtbaren Abstand zum Mittelpunkt zur Distanz $sin(\beta)/sin(90-\beta)$. Dieses sei in horizontale Richtung x_{Factor} und in vertikale Richtung y_{Factor}. Mit Hilfe dieser Faktoren bestimmt sich die 3D-Position eines Punktes P über

$$P(x,y,z) = \begin{cases} x = depth * x_{Factor} * \frac{u-(640/2)}{640/2} \\ y = -depth * y_{Factor} * \frac{v-(480/2)}{480/2} \\ z = -depth. \end{cases} \tag{14}$$

Durch die Subtraktion werden die Koordinaten relativ zum Bildmittelpunkt angegeben und dann durch die Division jeweils auf einen Wertebereich von -1 bis 1

übertragen. Der y- und z-Wert wird zur Übertragung ins Kamerakoordinatensystem invertiert.
Die so erzeugte Punktwolke muss nicht unbedingt die gleiche Auflösung wie das Tiefenbild haben. Im Programm gibt ein Faktor s an, jeder wievielte Punkt in horizontaler und vertikaler Richtung aufgenommen wird. Nur wenn s Eins ist, wird jeder Punkt in die Punktewolke aufgenommen. Jeder Punkt in der Punktewolke wird später zu einem Vertex im Modell. Wird s zu niedrig gewählt, sind keine Echtzeitanwendungen mehr möglich.

4.2.5 Rendern der Punktewolke

Wenn man die im vorherigen Kapitel erzeugte Punktewolke als Punkte im Raum visualisiert und mit der angegebenen perspektivischen Kamera betrachtet, erkennt man, dass alle Punkte in einem gleichmäßigen Raster angeordnet zu sein scheinen. Die Punktewolke soll als zusammenhängendes Objekt gerendert werden. Vordergrundobjekte werden also nach hinten verlängert, was allerdings von der Ursprungsposition der Kamera aus betrachtet nicht sichtbar ist. Abbildung 11 soll dies verdeutlichen. Um unsichtbare Geometrie zu ergänzen und mehrere Objekte zu rendern, wären komplizierte Algorithmen und Wissen über die Objekte nötig, was den Rahmen dieser Arbeit übersteigt. Jeder Punkt der Punktewolke wird zu einem Vertex im zu renderenden Objekt. Ein schnelles Übermitteln der Vertex-Daten an die Grafikkarte ist unter OpenGL mit einem *Vertex-Array* möglich [WND97]. Es wird ein Array angelegt, in das die Koordinaten von allen Punkte der Punktewolke in unveränderter Reihenfolge übertragen werden. Daneben wird ein Index-Array benötigt, das auf die Vertices im Vertex-Array verweist, und das später die Reihenfolge zum Zeichnen der Vertices angibt. Außerdem werden Zeiger auf je ein Array mit Texturkoordinaten und ein Array mit Eckpunktnormalen für jeden Vertex an die Grafikkarte übermittelt. Ersteres wird benötigt, damit das RGB-Bild später als Textur auf das Modell gemappt werden kann. Letzteres wird benötigt, um später eine korrekte Beleuchtung zu berechnen. Die Texturkoordinaten können bestimmt werden, da der Abstand s mit dem das Tiefen-Array abgetastet wird, sowie das Größenverhältnis des RGB-Bildes zum Tiefenbild bekannt sind. Sie bestimmen sich wie im Algorithmus 1.

```
i = 0;
for y = 0; y <= TiefenbildBreite; y+=s do
    for x = 0; x <= TiefenbildHöhe; x+=s do
        Texturkoordinaten[i, 0] = (x * VerhältnisBreite) / FarbbildBreite;
        Texturkoordinaten[i, 1] = (y * VerhältnisHöhe) / FarbbildHöhe;
        i++;
    end
end
```

Algorithm 1: Bestimmen der Bildkoordinaten

Es werden insgesamt jeweils zwei Polygone für jeden Punkt der Punktwolke, der nicht in der letzten Zeile oder Spalte liegt, erstellt. Für einen Punkt P mit Index i und einer Punktewolke mit einer Breite von w Punkten ergibt sich ergibt sich das erste Polygon aus den Punkten mit Index i, $i+w$ und $i+1$, das zweite aus Punkten mit Index $i+1$, $i+w$, $i+w+1$.
Für jedes Polygon wird in diesem Schritt eine Flächennormale erzeugt, die benötigt wird, um später Eckpunktnormalen zu berechnen. Die Flächennormale N wird für ein Polygon mit den Vertices V_0, V_1, V_2 mittels (15) bestimmt [NFH07, S. 184].

$$N = \frac{(V_0 - V_1) \times (V_2 - V_3)}{|(V_0 - V_1) \times (V_2 - V_3)|} \tag{15}$$

Die Eckpunktnormale eines Vertex wird berechnet, indem die Normalen aller an den Vertex angrenzenden Polygone zusammengerechnet und die Summe anschließend normalisiert wird [Gou71].
Pointer auf die erzeugten Arrays für Vertices, Texturkoordinaten, Eckpunktnormalen werden als `glVertexPointer`, `glTexCoordPointer` und `glNormalPointer` an die Grafikkarte übertragen. Das erzeugte Modell wird mit Quellcode 1 dargestellt. Hier bezeichnet *faces* die Anzahl der Polygone und *indices* den Index-Array.

Quellcode 1: Berechnung der Pixelfarbe

```
glDrawElements( Gl.GL_TRIANGLES, faces * 3,
      Gl.GL_UNSIGNED_INT, indices);
```

Abbildung 11 verdeutlicht das Gesamtresultat der Rekonstruktion. Man sieht rechts, wie aus einem einzigen Tiefen-Array und einem einzigen Farbbild eine geglättete, perspektivische 3D-Rekonstruktion ohne Lücken erstellt wurde.

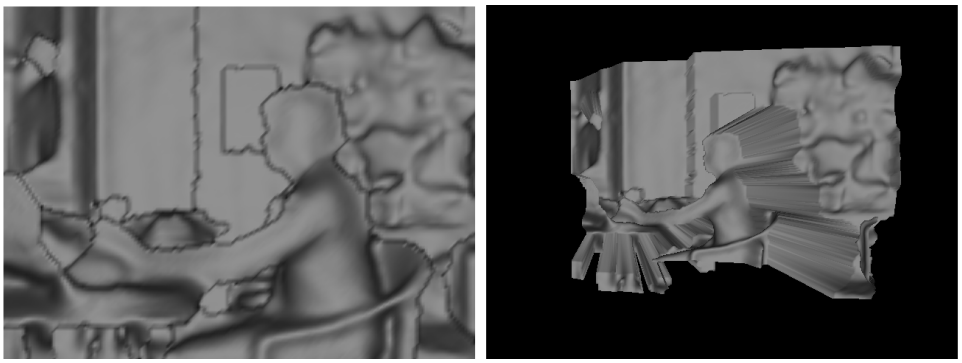

Abbildung 11: Rekonstruiertes 3D-Modell betrachtet mit Kamera im Ursprung und manipulierter Kamera. Unsichtbare Geometrie wird deutlich.

4.3 Grafiksimulationen

Auf dem im vorherigen Kapitel erzeugten 3D-Modell soll eine Licht- und Materialsimulation durchgeführt werden. Dies geschieht mittels GLSL direkt auf der Grafikhardware, was es möglich macht, einzelne Komponenten der OpenGL-Pipeline durch selbst programmierbare Shader zu ersetzen. Dabei wird die Flexibilität gegenüber der durch strenge Vorgaben begrenzten Standard OpelGL Pipeline erhöht. Die Verwendung der parallelen Rechnerarchitektur aktueller Grafikkarten soll zudem eine Interaktion in Echtzeit möglich machen.
Zusammengehörig sind jeweils ein Vertex-Shader zur Verarbeitung der Eckpunkte von Grafikprimitiven und ein Fragment-Shader zur Berechnung der Farbwerte der Pixel, aus denen sich die Geometrieprimitive zusammensetzen [RLKG$^+$09].

4.3.1 Auswahl der Verfahren

Bevor Licht und Materialsimulationen durchgeführt werden können, muss eine Auswahl getroffen werden, welche Verfahren verwendet werden sollen. Es wird auf jeden Fall ein Basisbeleuchtungsmodell benötigt, mit dem sich Lichtquellen erstellen und Beleuchtungssituationen konfigurieren lassen. Die OpenGL-Beleuchtungsberechnung bietet viele Freiheiten bei der Definition von Lichtquellen und Materialien. Mit dem Beleuchtungsmodell können keine fotorealistischen Beleuchtungen berechnet werden. Wegen der *unvollständigen* Geometrie, die in dieser Arbeit aus einem einzelnen Tiefenbild rekonstruiert wird, können das aber auch rechenaufwändigere Verfahren nicht. Der fehlende physikalische Bezug, kann sich sogar positiv auf die Neubeleuchtungen auswirken. Wie oben beschrieben wird Geometrie nach hinten verlängert. Wie in Abbildung 13 lässt sich trotzdem eine Punktlichtquelle hinter der Person auf dem Foto platzieren, da für die Beleuchtung nur die Normale und nicht die zwischen Lichtquelle und Oberfläche liegende Geometrie beachtet wird.

Alternativ wurde das realitätsnähere Image Based Lighing zur Beleuchtung des Modells ausgewählt. Auch beim Image Based Lighing hat die nach hinten verlängerte Geometrie keinen Einfluss auf das Resultat. Es ist vorstellbar, dass sich über die Verwendung einer Cube Map mit interessanten Beleuchtungssituationen ein Bild um die solche erweitern lässt. Über ein konfigurierbares Tone Mapping und HDR-Cube Maps, können Umgebung mit unterschiedlichen Helligkeiten simuliert werden. Liegt eine Reihe von Cube Maps unterschiedlicher Umgebungen vor, kann leicht zwischen diesen gewechselt werden, was dann jeweils zu vollkommen neuen Beleuchtungssituationen führen könnte.

Da bei beiden Beleuchtungsmodellen Schatten nicht automatisch erzeugt werden, wurde Shadow Mapping zur Schattenberechnung ausgewählt. Es lässt sich leicht auf mehrere Lichtquellen übertragen, was nötig ist wenn in der ersten Beleuchtungssimulation ein Schatten vorhanden ist, und in der zweiten Simulation eine

Lichtquelle hinzugefügt werden soll, die ebenfalls einen Schatten wirft. Weiterhin spricht für das Shadow Mapping seine Echtzeitfähigkeit und die Möglichkeit, weiche Schattenkanten zu simulieren.

Glänzende Oberflächen lassen sich mit Image Based Lighting oder der OpenGL-Beleuchtung simulieren. Eine interessante Erweiterung würden Umgebungsspiegelungen darstellen. Über Cube Mapping mit Zugriff auf *ungefilterte* HDR-Cube Maps lässt sich dies ohne großen Mehraufwand realisieren.

Abbildung 12: Obwohl der Vordergrund nach hinten verlängert wird, kann eine Punktlichtquellle im Hintergrund platziert werden.

4.3.2 Implementierung der Shader

Insgesamt werden sechs jeweils zusammengehörige Vertex-und Fragment-Shader eingesetzt. Bei vier von ihnen liegt der Fokus jeweils auf einem einzelnen Verfahren, zwei kombinieren verschiedene Verfahren.

PhongMultiLight *PhongMultiLight* berechnet die Beleuchtung für bis zu acht Punktlichter, Spotlichter oder gerichtete Lichter, nach dem OpenGL-Beleuchtungsmodell. Die Eigenschaften der Standard OpenGL-Lichtquellen können durch Verwendung von Shadern auf das Prinzip des *Phong Shadings* übertragen werden. Hierzu übergibt der Vertex-Shader die Normale eines Eckpunktes an den Fragment-Shader. Die Beleuchtungsberechnung wird dann im Fragment-Shader unter Verwendung der interpolierten Normale pro Pixel durchgeführt. Dies führt zu qualititiv bessern Glanzlichtern und glatteren Spotlichtkegeln, als das von OpenGL bereitgestellte *Gouraud Shading*, bei dem die Beleuchtung pro Eckpunkt berechnet und pro Pixel zwischen den Eckpunktfarben interpoliert wird [NFH07, S. 199]. Die Beschränkung auf acht Lichtquellen kommt zustande, da Verwaltung der Lichtquellen im Programm über OpenGL-Befehle geregelt wird um dann im Shader über vordefinierte Uniform-Variablen auf diese zugreifen zu können. Um zu bestimmen

welche der acht Lichtquellen für die Simulation verwendet werden sollen, wird ein Array *LightSources* als Uniform-Variable an den Fragment-Shader übertragen, das für jede Lichtquelle, an der Position ihrer Nummer, entweder 1 enthält, wenn sie verwendet wird, oder 0 wenn nicht. Dies ermöglicht ein leichtes Ein- und Ausschalten von Lichtquellen. Quellcode 2 zeigt wie anhand dieses Arrays in der *main* Methode des Fragment-Shaders zwischen den einzelnen Lichtquellen unterschieden wird.

Quellcode 2: Auswahl der aktivierten Lichtquellen

```
for (int i = 0; i < LightSources.length; i++)
{
   if (LightSources[i] == 1)
      {
         if (gl_LightSource[i].position.w == 0)
            DirectionalLight(i, n, v, amb, diff, spec);
         else if (gl_LightSource[i].spotCutoff == 180.0)
            PointLight(i, n, v, amb, diff, spec);
         else
            SpotLight(i, n, v, amb, diff, spec);
      }
}
```

Jeder der drei Lichtquellentypen ruft zur Beleuchtungsberechnung eine andere Methode auf. Zur Bestimmung des Lichtquellentyps wird geprüft, ob die vierte Positionskomponente wie bei gerichteten Lichtquellen null ist und oder der Öffnungswinkel, wie bei Punktlichtern, 180 Grad beträgt. Die Variablen *amb*, *diff* und *spec* sind Akkumulatoren, die für jede Lichtquelle um den berechneten ambienten, diffusen oder spekularen Wert erhöht werden. Die Variable *n* ist der für den Pixel interpolierte Normalenvektor, *v* ist die Vertexposition in ModelView Koordinaten. Details zur Implementierung der Methoden *DirectionalLight*, *PointLight* und *SpotLight* können in [RLKG+09, S. 274 ff.] gefunden werden. Unter Berücksichtigung der Materialeigenschaften bestimmt sich die Farbe eines Pixels wie in Quellcode 3. Farbwerte größer als 1.0 werden auf 1.0 abgebildet, um nicht darstellbare Farbwerte zu vermeiden.

Quellcode 3: Berechnung der Pixelfarbe

```
vec4 finalColor = gl_FrontLightModelProduct.sceneColor +
      amb * gl_FrontMaterial.ambient +
      diff * gl_FrontMaterial.diffuse +
      spec * gl_FrontMaterial.specular;
gl_FragData[0] = clamp(finalColor, 0.0, 1.0);
```

ShadowMapping Die *ShadowMapping*-Shader berechnen Schatten für mehrere Spotlichter mit dem Shadow Mapping Verfahren. Sie werden in einem finalen Renderdurchgang eingesetzt, um die Schatten in die Szene einzufügen. Vorher müssen jedoch Shadow Maps für alle Lichtquellen erzeugt werden. Eine Lichtquelle wird im Programm durch ein Objekt der Klasse `LightSource` repräsentiert. Die Shadow Map-Textur wird beim Erzeugen einer Lichtquelle angelegt. Die Lichtquelle

verwaltet unter anderem ihre *View*- und ihre *Projektionsmatrix*, damit jederzeit aus ihrer Sicht gerendert werden kann. Ein Spotlicht verwendet dann seinen Cutoff-Winkel als Öffnungswinkel der perspektivischen Projektionsmatrix. Die für die Schattenberechnung benötigte Texturmatrix wird ebenfalls separat für jede Lichtquelle verwaltet und über die Matrixmultiplikation aus Gleichung 12 berechnet. Nachdem aus Sicht der Lichtquelle gerendert wurde, werden die Tiefenwerte der sichtbaren Punkte über `glCopyTexSubImage2D` in die Shadow Map gespeichert. Nachdem die Shadow Maps für alle Lichtquellen erstellt wurden, werden die *ShadowMapping*-Shader aktiviert.
Im Vertex-Shader werden für alle Textur-Matritzen, die in einem Matrix-Array an den Shader übergeben werden, Schattenkoordinaten berechnet, indem sie mit den Vertexkoordinaten multipliziert werden. Sie werden an den Fragment-Shader weitergegeben.

Quellcode 4: Berechnung der Schattenkoordinaten

```
uniform mat4 TextureMatrix[7];
varying vec4 ShadowCoord[7];
//...
for (int i = 0; i< 7; i++)
   ShadowCoord[i]= TextureMatrix[i] * gl_Vertex;
```

Der Fragment-Shader erhält die Shadow Map-Texturen im Format `sampler2D-Shadow`, einem GLSL- Container speziell für 2D-Tiefentexturen. Außerdem muss ein eindimensionales Array übergeben werden, in dem markiert ist, welche Lichtquellen berücksichtigt werden, in diesem Fall alle für die Simulation aktivierten Lichtquellen, die gleichzeitig Spotlichter sind. Bei der Berechnung der Helligkeit wird dann ein Faktor *shadow* mit einbezogen, der die Helligkeitswerte reduzieren kann, falls ein Punkt im Schatten liegt.

Quellcode 5: Berechnung der Schattenstärke in der Spotlight Methode des Shadow Mapping-Shaders

```
float shadow = 1;
if (shadowCoord.w > 1)
{
   float x,y;
   for (y = -1.5 ; y <=1.5 ; y+=1.0)
      for (x = -1.5 ; x <=1.5 ; x+=1.0)
         shadow += lookup(vec2(x,y), i);
   shadow = (shadow / 16.0);
}
```

Um weiche Schattenkanten zu erhalten wird, wie in Quellcode 5, der Durchschnitt der Schattenwerte von 16 Punkten in einem 4×4 Fenster berechnet. Die `lookup` Methode erhält hierzu als Parameter einen Offset-Vektor und die Nummer *i* der aktuellen Lichtquelle (basierend auf [PM]).

Quellcode 6: Lookup-Methode

```
uniform float xPixelOffset;
uniform float yPixelOffset;
uniform sampler2DShadow ShadowMap0;
uniform sampler2DShadow ShadowMap1;
//...
float lookup ( vec2 offSet, int i)
{
if (i == 0)
   return shadow2DProj(ShadowMap0, ShadowCoord[i] +
            vec4(offSet.x * xPixelOffset * ShadowCoord[i].w,
                  offSet.y * yPixelOffset * ShadowCoord[i].w,
                  0.00005f,
                  0.0f) ).w;
else if (i == 1)
   return shadow2DProj(ShadowMap1, + vec4( ...
//...
}
```

Die Methode *shadow2DProj* führt den Schattentest mit einer Schadow Map und einem Koordinatenvektor als Argumente durch. *ShadowCoord[i]* ist die berechnete projizierte Texturkoordinate, die mit einem Offset manipuliert wird, um umliegende Texturpukte im Fenster zu testen. Um Pixelkoordinaten von *vec2 offSet* in Texturkoordinaten im Wertebereich [0,1] in x- und y-Richtung zu übersetzen, wird mit *xPixelOffset* und *yPixelOffset*, die relative Größe eines Pixels, übertragen auf diesen Wertebereich, an den Shader übermittelt. Um zu berücksichtigen, dass *shadow2DProj* automatisch eine Division durch die vierte Vektorkomponente durchführt, muss die y und z Koordinate des aufaddierten Offsets mit w multipliziert werden. Die z Koordinate wird zudem geringfügig erhöht, um Artefakte zu vermeiden.

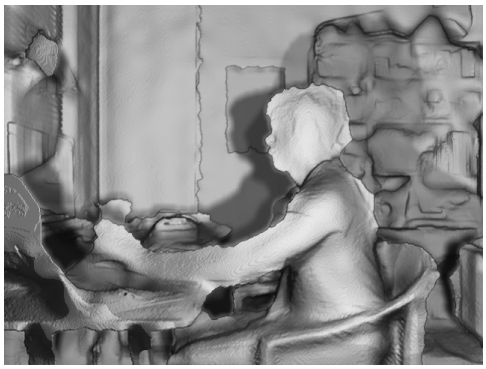

Abbildung 13: Demonstration des *ShadowMapping*-Shaders mit Schatten von drei Spotlichtern.

CubeMapping Die *CubeMapping*-Shader ermöglichen die Simulation eines Modells mit spiegelnder Oberfläche. Die Umgebung, die sich im Modell spiegeln soll,

liegt als High Dynamic Range Cube Map vor. Auf die Umgebungstextur wird über einen Richtungsvektor zugegriffen, der mit der GLSL- Funktion `reflect` automatisch berechnet werden kann [RLKG+09, S. 311]. Nach dem Texturzugriff wird ein Tone-Mapping durchgeführt, mit dem sich verschiedene Belichtungszeiten simulieren lassen [HM06]. Der Wert der Variable *exposure* kann vom Benutzer bestimmt werden. Wird der Wert klein gewählt, so ist die Umgebung sehr dunkel. Details sind in dunklen Bildbereichen zu erkennen. Wird der Wert hoch gewählt, erscheint die sich spiegelnde Umgebung sehr hell, da bei hohen Belichtungszeiten sehr viel Licht auf den Sensor treffen kann. Diese Wahlmöglichkeit gibt Flexibilität über die Helligkeit der Umgebung und somit auch über die Helligkeit des Modells. Der Wert *brightMax* stellt den höchsten Pixelwert der Cube Map dar und wird beim Laden der Textur mitbestimmt.

Quellcode 7: Texturzugriff und Tone-Mapping im Shader

```
uniform samplerCube EnvMap;
uniform float Mix;
uniform float exposure;
uniform float brightMax;
//...
vec3 reflectDir = reflect(v, normal);
vec3 envColor = vec3 (texture(EnvMap, reflectDir));
float tonefactor =
      exposure * (exposure/brightMax + 1.0) / (exposure + 1.0);
envColor *= tonefactor ;
envColor = mix (envColor, base.rgb, Mix);
```

Wird der *CubeMapping*-Shader eingesetzt, reflektiert das Modell wie ein Spiegel. Ein Punkt der Umgebung spiegelt sich in einem Punkt des Modells. Um einen Spiegelglanz zu ermöglichen, der nur leicht auf der Oberfläche liegt, existiert eine Basisfarbe, die sich durch die ambiente Materialfarbe und eine vereinfachte diffuse Beleuchtung berechnet. Über *Mix* kann bestimmt werden, wie stark der Texturwert der Umgebungstextur in den finalen Pixelwert mit eingeht. Bei einem Mix von z.B. $0,05$ ist die Spiegelung nur sehr dezent sichtbar, wohingegen die Basisfarbe bei einem Mix von 1 nicht mehr berücksichtigt wird.

ImagedBasedLighting Eine mit den von Paul Devebec vorgestellten Verfahren erstellte High Dynamic Range Cube Map erfasst das vollständige Lichtspektrum einer Umgebung. Es lassen sich wie in 3.4.4 beschrieben Cube Maps zur Simulation einer diffusen und spiegelnden Beleuchtung errechnen. Das Shaderpaar *ImagedBasedLighting* soll die Beleuchtung dieser beiden Cube Maps auf das Modell anwenden. Dies ermöglicht es, schwer zu rekonstruierende Beleuchtungssituationen direkt auf ein Modell anzuwenden, falls eine HDR-Cube Map vorliegt. Die Shader entsprechen zum Großteil den *CubeMapping*-Shadern. Der Unterschied ist, dass zwei Cube Maps übermittelt werden. Das Tone Mapping wird wie im vorherigen Abschnitt beschrieben durchgeführt, die maximale Helligkeit wird allerdings für beide Cube Maps separat übermittelt. In Kapitel 3.3 wurde ebenfalls bereits beschrieben, welche Vektoren zum Texturzugriff verwendet werden. Bei *ImagedBa-*

sedLighting geht die diffuse Materialfarbe als Basisfarbe in den finalen Farbwert mit ein. Wie stark die Materialfarbe, die diffuse und die spiegelnde Beleuchtung jeweils gewichtet werden, geben die zwei Uniform-Variablen *DiffusePercent* und *SpecularPercent* an. Wie der Farbwert berechnet wird, sieht man in Quellcode 8 (basierend auf [RLKG+09, S. 364]).

Quellcode 8: Mixen der Farben beim Inage Based Lighting

```
vec3 baseColor = (gl_FrontMaterial.diffuse).rgb;
vec3 color = mix(baseColor, diffuseColor*baseColor, DiffusePercent);
color = mix(color, specularColor + color, SpecularPercent);
gl_FragData[0] = vec4(color, 1.0);
```

LightShadowCube Der erste der beiden kombinierten Shader verbindet die *PhongMultiLight*, *ShadowMap* und *CubeMapping*, das heißt, es werden alle Lichtquellenarten unterstützt, Spotlights werfen Schatten und zusätzlich kann sich eine Umgebung im Modell spiegeln. Beim Zusammenfügen wurde die Auswahl der Lichtquellentypen wie in Quellcode 2 umgesetzt, wobei Spotlichter um den Zugriff auf Shadow Maps erweitert wurden. Der FragmentShader berechnet die Beleuchtung für einen Pixel, nimmt diesen Wert als Basisfarbe und mixt ihn wie bei den *CubeMapping*-Shadern mit einem Texturpunkt aus der Umgebungstextur.

LightShadowIBL Der zweite kombinierte Shader verbindet die Licht- und Schattenberechnung von *PhongMultiLight* und *ShadowMapping* mit der Lichtberechnung durch *ImageBasedLighting*. Dies bewirkt, dass eine auf Cube Maps basierende Beleuchtungssituation mit zusätzlichen Lichtern und Schatten erweitert und manipuliert werden kann. So können zum Beispiel Schatten, die in einer Lichtsimulation mit *Image Based Lighting* hinzugefügt werden. Die Implementation fügt lediglich Elemente der Shader zusammen, darum muss nicht näher auf den Code eingegangen werden.

4.4 Bilderweiterungen

Die vorgestellten Shader ermöglichen eine vielfältige Beleuchtungs- und Materialsimulation. Dieses Kapitel soll zeigen, wie auf diese Art erstellte Simulationen mit dem RGB-Bild zu einem neuen Bild mit veränderten Material und Beleuchtungseigenschaften kombiniert werden können.

4.4.1 Verwendete Pipeline

Bei den verwendeten Bilderweiterungsverfahren gliedert sich der Ablauf grundlegend in drei Schritte. In den ersten beiden Schritten werden Simulationen mit einer Schätzung der ursprünglichen und den neuen Beleuchtungs- und Materialeigenschaften durchgeführt, die im dritten Schritt mit dem Foto zu einem veränderten Bild kombiniert werden. Das neu beleuchtete Bild soll angezeigt werden, während

die beiden Simulationen im Hintergrund durchgeführt werden. Änderungen an einer Simulation wirken sich so direkt auf das manipulierte Bild aus.
Das 3D-Modell, aus der Ursprungsposition der Kamera betrachtet, ist durch die zuvor vorgenommenen Transformationen auf das Kamerabild ausgerichtet. Ein Pixel im zweidimensionalen Abbild des Modells korrespondiert also mit dem Pixel, der im Farbbild an der gleichen Position liegt. Die OpenGL-Erweiterung `GL_FRAME-`

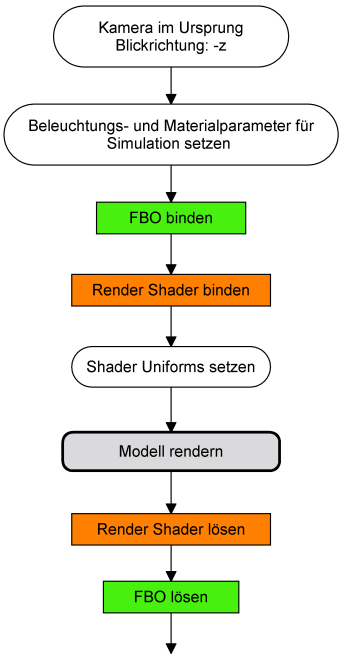

Abbildung 14: Ablauf des Offscreen Renderings

`BUFFER_EXT` ermöglicht es, den Inhalt des Color Buffers, also den sichtbaren Bildausschnitt der OpenGL-Kamera, direkt in eine Textur zu rendern. Hierzu wird ein *Framebuffer Object* (FBO) erzeugt und eine Textur wird an es gebunden. Wird das FBO vor dem Rendern aktiviert, wird in die Textur gerendert [Ahn12]. Liegen Texturen für beide Simulationen und für das Ursprungsbild vor, so kann ein neuer Pixelwert anhand der drei Texturen berechnet werden. Zur Berechnung wird ein GLSL-Shader verwendet, der im Folgenden als *Display-Shader* bezeichnet wird.
Der Shader, der zur Beleuchtungs- und Materialsimulation im 3D-Modell verwendet wird, wird im Folgenden als *Render-Shader* bezeichnet. Jeder Simulation wird ein eigener *Render-Shader* und ein eigenes FBO zugewiesen. Die Parameter, die die Beleuchtungs- und Materialsimulation definieren, sind ebenfalls für jede Simulation separat gespeichert. Abbildung 14 zeigt, welche Schritte durchgeführt werden, um eine Simulation in eine 2D-Textur zu rendern.
Nach Durchführung des Offscreen-Renderings enthält die Textur, die dem FBO zugewiesen wurde, das Resultat der Simulation unter Anwendung der für sie festgelegten Parameter. Der Gesamtablauf wird in Abbildung 15 gezeigt.

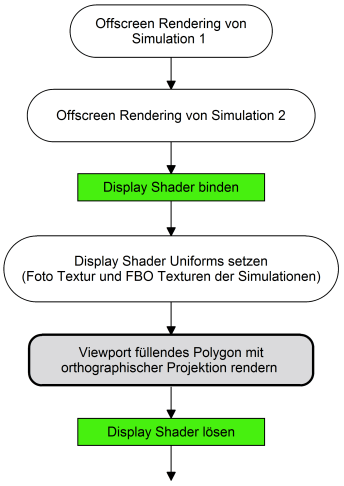

Abbildung 15: Gesamtablauf des Renderings

4.4.2 Anwendung des Display-Shaders

Wie oben beschrieben, werden dem *Display-Shader* die Foto Textur und je eine Textur für beide Simulationen übermittelt. Es wird ein Viewport füllendes Polygon gezeichnet, dessen Eckpunkten die Eckpunktkoordinaten der Texturen zugewiesen werden. Im Fragment-Shader können die Texturen dann auf Pixelbasis miteinander verrechnet werden.
Es kann zwischen zwei *Display-Shadern* gewählt werden. Der erste führt die Berechnung gemäß der Formel durch, die in 3.2.1 für das *Differentielle Rendering* angegeben wurde (9).

Quellcode 9: Fragment-Shader für Differentielles Rendering

```
uniform sampler2D Simulation1;
uniform sampler2D Simulation2;
uniform sampler2D Texture1;

void main()
{
vec4 color1 = texture2D(Simulation1, vec2(gl_TexCoord[1]));
vec4 color2 = texture2D(Simulation2, vec2(gl_TexCoord[1]));
gl_FragColor = (color2 - color1) +
               texture2D(Texture1, vec2(gl_TexCoord[0]));
}
```

Der zweite *Display-Shader* führt die Berechnung nach dem alternativen Verfahren von Madsen und Laurensen durch. Es unterscheidet sich lediglich die Berechnung des Farbwertes eines Pixels. Gemäß Formel 5 wird das Verhältnis zwischen dem Pixelwert der beiden Simulationen berechnet, das dann mit dem Farbwert in der Textur multipliziert wird.

Quellcode 10: Berechnung eines Farbwertes im alternativen Display-Shader

```
gl_FragColor = (color2 / color1) *
               texture2D(Texture1, vec2(gl_TexCoord[0]));
```

Nachdem das Rendering mit dem *Display-Shader* durchgeführt wurde, wird abhängig von der Konfiguration, das neu beleuchtete Farbbild angezeigt. Es können Feinjustierungen vorgenommen werden, um das Ergebnis zu optimieren.

4.5 Überblick über die Software

Der gesamte Bilderweiterungsvorgang wurde begleitend zu dieser Arbeit in der Software **Kinect - Augmented Images** zusammengefasst. Der Source Code der Software ist auf DVD beigelegt. Bisher wurden nur einzelne wichtige Code-Ausschnitte der Software gezeigt. Um einen Überblick über das entwickelte Programm zu geben, soll hier kurz auf Aufbau und Struktur der Software eingegangen werden.

Die Klassen `CubeMap`, `Framebuffer`, `LightSource`, `Material` und `Shader` kapseln OpenGL-Funktionalitäten, um sie später in anderen Klassen objektorientiert nutzbar zu machen. Objekte dieser Klasse verwalten dann alle für sie relevanten Objekte und Parameter selbst. Die Kapselung gibt die Flexibilität, die benötigt wird, um Simulationen mit unterschiedlichen Eigenschaften zu erzeugen, und erhöht zusätzlich die Übersichtlichkeit des Codes. Shader mit deren Erzeugung immer viel Code verbunden ist, können zum Beispiel einfach über den Aufruf eines Konstruktors und unter Angabe das Dateinamens erzeugt und kompiliert werden. Um einem solchen Shader-Objekt verschiedene Uniform-Variablen zuzuweisen, wird dann nur noch eine Code-Zeile benötigt.

Das OpenGL-Rendering wird von der Klasse `OpenGlContext` durchgeführt. Hier ist eine Reihe von Funktionen für das Rendern, abgestimmt auf bestimmte Shader vorhanden, wobei beim Erzeugen eines Shaders angegeben wird, welche davon aufgerufen werden soll.

Die Klasse `Sensor` verwaltet die aufgenommenen Tiefen- und Farbbilder. Sie enthält Methoden die das Kinect-SDK benötigen, zum Beispiel zum Mappen von Farbbild und Tiefenbild oder zum Ändern von Sensoreinstellungen. Das Kinect-SDK stellt die Klassen `KinectSensor` und `CoordinateMapper`, von denen je ein Objekt in der übergeordneten Klasse `Sensor` verwaltet und initialisiert wird.

Um zwischen den Ansichten für Simulationen und dem für das neu beleuchteten Bild wechseln zu können, die alle unterschiedlich konfiguriert sind, müssen die Parameter für diese Ansichten separat gespeichert werden, um beim Wechseln nicht verloren zu gehen. Für die Ansichten mit OpenGL-Modell verwaltet dies die Klasse `ParseSimulation`. Die Ansicht, die das manipulierte Farbbild zeigt, wird von der Klasse `ParseCombination` verwaltet. Beide sind abgeleitet von

der abstrakten Klasse `Parse`. Ein Objekt der Klasse `ParseSimulation` verwaltet alles, was bei Simulationen manipuliert werden kann, also den verwendeten *Render-* und *Display-Shader*, deren konfigurierbare Parameter, das Material, falls nötig verschiedene Reflektions Cube Maps, das Framebufferobjekt, in das gerendert wird, und die Nummern der angeschalteten Lichtquellen. Die Lichtquellen selbst werden vom `OpenGlContext` verwaltet, da diese für verschiedene Simulationen nutzbar sein sollen.

Zur Organisation der Software existieren weiterhin eine `Settings`-Klasse, die verschiedene über die GUI konfigurierbare Parameter verwaltet und auf die von praktisch allen Objekten aus zugegriffen werden kann, und eine Klasse `DataStore`, die Kinect-Bilder in .txt Dateien speichern und aus diesen laden kann, um die Benutzung der Software ohne angeschlossenen Kinect-Sensor zu ermöglichen.

Es gibt eine Reihe von statischen Klassen. `ColorSpace`[1] zur Konvertierung eines Bildes in den CIE-Lab-Farbraum, `Converter` für diverse andere benötigte Konvertierungen und `DepthMapFilter` für Filteroperationen, die auf dem Tiefenbild durchgeführt werden.

Zentrum der WPF-Anwendung bildet `MainWindow`, unterteilt in einen XAML-Teil und einen C#-Teil. In der XAML-Datei ist die gesamte Benutzungsoberfläche der Software definiert. Interaktionen eines Benutzers lösen Events aus, die wiederum im C#-Code angenommen und verarbeitet werden[2]. Der C#-Code dieser Klasse kann außerdem als Zentrum der Software angesehen werden, da hier `Sensor`, `OpenGlContext`, die Shader, die CubeMaps und die Simulationen initialisiert und verwaltet werden. Auch das Neuzeichnen des OpenGl-Fensters und das Erzeugen der Punktewolke werden hier angestoßen.

4.6 Benutzerinteraktion

4.6.1 Die Benutzerschnittstelle

Die auf XML-basierende Sprache XAML (Extensible Application Markup Language) erwies sich als sehr geeignet zum Entwurf einer Benutzerschnittstelle mit der Bilderweiterungen konfiguriert werden können. Der Vorteil liegt darin, dass die Oberfläche gut an den Programmablauf angepasst werden kann. Die einzelnen Elemente sind gruppiert und geschachtelt, sodass es möglich ist, immer die für den momentanen Arbeitsschritt benötigten Schaltflächen anzuzeigen und andere zu verstecken. Abbildung 16 zeigt die Oberfläche des Editors im Simulationsmodus, da hier alle Einstellungsmöglichkeiten gleichzeitig dargestellt werden. Die Oberfläche ist in drei Bereiche unterteilt.

[1] übernommen von http://www.mycsharp.de/wbb2/thread.php?threadid=78941
[2] ausgelagert in partielle Klasse in GuiEventHandler.cs

In der mittleren Spalte wird, falls noch kein Bild aufgenommen wurde und ein Kinect-Sensor angeschlossen ist, ein Stream des Farbbildes angezeigt. Wurde der Sensor auf das gewünschte Motiv ausgerichtet, wird mit *Snap* ein Bild aufgenommen. Es wird automatisch ein Frame abgegriffen, ein 3D-Modell erzeugt und dieses anstelle des Streams eingeblendet. Es werden dann weiter unten die vier Schaltflächen *Modell*, *Simulation 1*, *Simulation 2* und *Kombination* angezeigt. Mit den Schaltflächen kann zwischen unterschiedlichen Ansichten für die einzelnen Modi gewechselt werden. Wichtig ist zu beachten, dass Änderungen in der Modell-Ansicht sich nicht auf das Kombinationsbild auswirken. Die Modell-Ansicht soll einem Nutzer die Möglichkeit geben, Konfigurationen zu testen, ohne dass er die bereits konfigurierten Simulationen verändern muss. Ist beim Start des Programms kein Kinect-Sensor angeschlossen, wird kein Bildstream eingeblendet. Mit *Snap* wird dann ein vorgespeicherter Default-Datensatz geladen, auf dem eine Beleuchtungssimulation durchgeführt werden kann.
Ein Klick auf *Texture* ermöglicht es, in einer 3D-Ansicht das Farbbild auf das 3D-Modell mappen. Dies kann nützlich sein, um zum Beispiel Schatten in der Simulation an Schatten im Farbbild anzupassen.
Clear löscht eine konfigurierte Beleuchtungssituation und gibt die Möglichkeit ein neues Bild aufzunehmen.

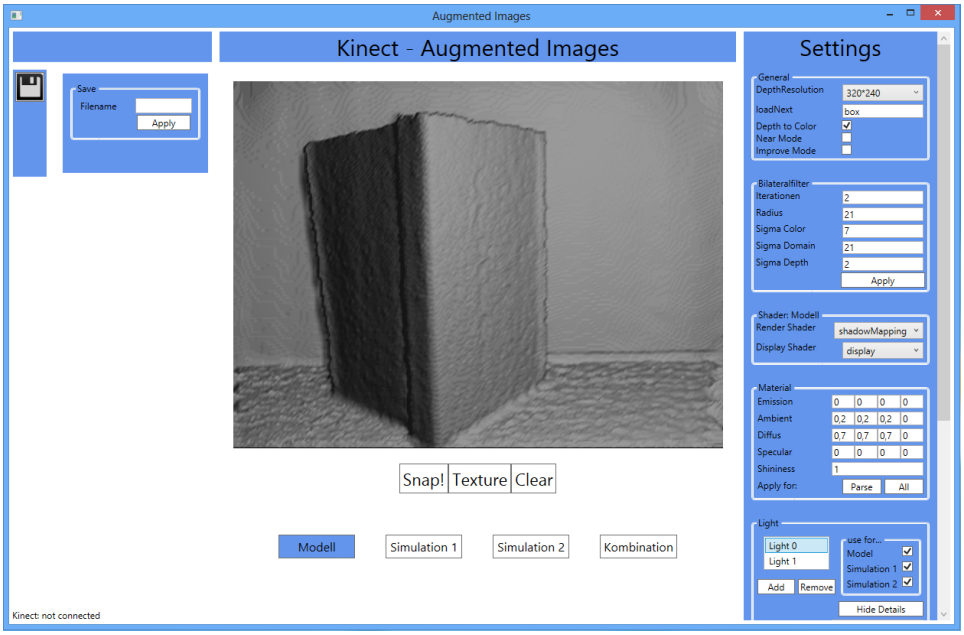

Abbildung 16: Benutzeroberfläche des Editors

Die rechte Spalte umfasst alle Einstellungsmöglichkeiten, die das 3D-Modell und den Kinect-Sensor betreffen. Unter *General* kann die Auflösung des Modells verstellt werden, es kann bestimmt werden, ob beim Erstellen des Modells das Farb-

bild an das Tiefenbild oder das Tiefenbild an das Farbbild angepasst wird und ob bei der Aufnahme des Bildes der *Near Mode* verwendet wird, um Tiefenwerte im Bereich von 40 cm bis 3 m zu erfassen oder der *Default Mode*, um einen Bereich von 80 cm bis 4 m zu erfassen. Darunter finden sich die Einstellungsmöglichkeiten für den *Bilateralfilter*. Geglättet wird immer ausgehend vom ungefilterten Tiefenbild. In der *Shader*-Box kann zwischen den sechs *Render-Shadern* gewählt werden. Gibt es zusätzliche Konfigurationsmöglichkeiten, die den Shader betreffen, werden diese unter dem Dropdown-Menü angezeigt. Außerdem kann der *Display-Shader* gewechselt werden, was sich auf das Resultat auswirkt, falls eine Neubeleuchtung betrachtet wird. Unter *Material* können die OpenGL Materialeigenschaften angepasst werden. *Parse* übernimmt sie für die sichtbare Simulation, *All* für alle Simulationen. Farbwerte werden in einem Bereich von 0 bis 1 eingetragen.
Zuletzt finden sich die Einstellungen für Lichtquellen. Da hier viele Parameter einstellbar sind, werden die Beleuchtungseinstellung gesondert in Abbildung 17 gezeigt. Entscheidend ist, dass Lichtquellen unabhängig von der momentan sichtbaren Simulation verwaltet werden. Über die drei Checkboxen neben der Liste können die Lichtquellen für die einzelnen Simulationen ein- und ausgeschaltet werden. Dies ist auch möglich, wenn das neu beleuchtete Bild betrachtet wird. So können Änderungen direkt auf dem resultierenden Farbbild betrachtet werden. Unter der Liste können Lichtquellen hinzugefügt oder entfernt werden. Es sind immer die Einstellungen für die momentan in der Liste ausgewählte Lichtquelle sichtbar.

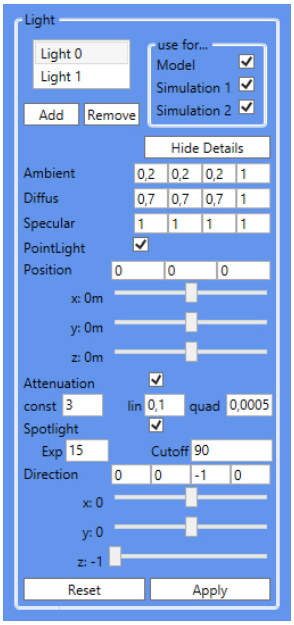

Abbildung 17: Einstellungsmöglichkeiten für Lichtquellen

In der linken Spalte findet sich ein Button um ein manipuliertes Bild zu speichern. Es wird dann im Ordner *Images* im Programmordner abgelegt. In der Menüliste ist zusätzlicher Platz reserviert. Es könnten bei Programmerweiterungen beispielsweise Pinsel oder Maskierungswerkzeuge untergebracht werden, um Änderungen direkt auf dem Tiefenbild vorzunehmen.

4.6.2 Echtzeitfähigkeit

In Kapitel 4.4.1 wurde die Pipeline des differentiellen Renderings vorgestellt und es zeigte sich, dass viele Schritte auf einmal durchgeführt werden müssen, um ein manipuliertes Bild angezeigt zu bekommen. Ein 3D-Modell, bei dem jeder Bildpunkt des Tiefenbildes zu einem Vertex konvertiert wird, hätte über 300000 Vertices. Trotz der relativ schnellen Shader ist es sehr aufwändig, für so viele Vertices eine Beleuchtung zu berechnen, vor allem wenn zusätzlich Schatten berücksichtigt werden sollen. Trotzdem ist eine hohe Auflösung in vielen Fällen wichtig für die Qualität des Ergebnisbildes, da die Geometriekanten hier am saubersten zu erkennen sind und auch kleine Geometriedetails in die Beleuchtung eingehen.
Die Lösung dieses Problems in der Software ist die Möglichkeit, die Auflösung des angezeigten Tiefenbildes immer verstellen zu können, ohne dass die konfigurierte Beleuchtungssituation oder die Glättung des Tiefenbildes verloren geht.
Dies funktioniert so, dass bei der Aufnahme eines Bildes immer die höchste Auflösung für Farb- und Tiefenbild verwendet wird und dieses im Hintergrund gespeichert bleibt.
Die Glättung wird auf dem Bild mit höchster Auflösung ausgeführt. Für das 3D-Modell werden gemäß dem Vorgehen in 4.2.4 nur bestimmte Punkte des Tiefenbildes berücksichtigt. Ein Anwender kann also, während er die Beleuchtung konfiguriert, die Auflösung wählen, bei der Änderungen auf seinem Computer in Echtzeit möglich sind. Wenn er mit dem Resultat zufrieden ist, kann er die höchste Auflösung wählen, um sein Ergebnisbild zu betrachten. Wird wegen vielen Lichtquellen und Shadow Mapping die Berechnung aufwändiger, kann er jederzeit den Detailgrad des Modells reduzieren.

Weiterhin wurde die Größe der Leinwand zum Steigern der Performance auf eine Auflösung von 640 * 480 Pixel beschränkt. Wird der Button zum Speichern betätigt, wird im Hintergrund automatisch die Leinwandgröße auf 1280*960 erhöht. Die Größe der FBOs der Simulationen wird an diese Größe angepasst und die Farbtextur mit voller Auflösung wird verwendet. Es wird anschließend in einen seperaten Framebuffer gerendert, dessen Textur dann aber nicht angezeigt, sondern in ein Byte-Array konvertiert wird. Dieses Byte-Array kann dann mit der *DevIl* Library als jpg-Datei abgespeichert werden. Anschließend werden die Auflösungen der Framebuffer und die der Farbtextur zurückgesetzt. Das Speichern wurde also mit der größtmöglichen Bildqualität durchgeführt, ohne der Performance zu schaden.

5 Ergebnisse

5.1 Beleuchtungsmanipulation

Anhand verschiedener Szenarios sollen Grenzen und Möglichkeiten der Beleuchtungsmanipulation unter Verwendung der vorgestellten Verfahren demonstriert werden. Die gezeigten Szenen weisen eine einfache Geometrie und eine leicht zu rekonstruierende Beleuchtungssituation auf. Die einzige eingesetzte Lichtquelle ist eine Spotlichtquelle, die direkt über dem Kinect-Sensor angebracht ist, sodass ihr ungefährer Lichteinfall in die Szene mit der entwickelten Software rekonstruiert werden kann. Um möglichst zutreffende Rekonstruktionen zu erhalten, wird bei der Auswahl der Bildmotive darauf geachtet, dass die sichtbaren Materialien vom Kinect-Sensor erfasst werden können. Insgesamt werden die folgenden vier Manipulationen vorgenommen.

1. Ersetzen der Spotlichtquelle durch eine schwächere Punktlichtquelle zum Abdunkeln der Szene
2. Hinzufügen einer gerichteten Lichtquelle, um ein Foto, das mit schlechten Lichtverhältnissen aufgenommen wurde aufzuhellen
3. Einfügen eines künstlichen Schattens mit einer Spotlichtquelle
4. Einfügen einer realen Beleuchtungssituation über Image Based Lighting

Bei der Auswahl der Szenarios wird darauf geachtet, alle relevanten, vorgestellten Renderingverfahren anzuwenden. Parallel dazu sollen die beiden Berechnungswege für Neubeleuchtungen verglichen werden. Das *Differentielle Rendering* wird nachfolgend als *Verfahren 1* und das Verfahren von Madsen und Laurensen als *Verfahren 2* bezeichnet.

Szenario 1 Im ersten Szenario sieht man eine Raumecke mit ausreichender Beleuchtung aufgenommen. Eine helle Spotlichtquelle ist direkt auf das Zentrum des Bildes ausgerichtet. Die sichtbare Geometrie ist geschlossen. Die vorhandene Beleuchtung soll durch eine unsichtbare Punktlichtquelle ersetzt werden. Abbildung 18 zeigt in der oberen Zeile das Ausgangsbild, eine für das Bild rekonstruierte Beleuchtungssituation und die gewünschte Beleuchtungssiutation mit einer Punktlichtquelle im hinteren Teil des Bildes in der Ecke zwischen Regal und Wand. Die im Bild sichtbare Geometrie ist sehr einfach und geradlinig. Für die Beleuchtungssimulationen wurde ein niedrig aufgelöstes Modell mit lediglich 30*40 Polygonen verwendet. Die niedrige Auflösung führt zu einem sehr weich berechneten Glanzpunkt mittels Phong Shading. Ihre Genauigkeit ist ausreichend um die sichtbare Geometrie zu erfassen. An den Ergebnisbildern in der zweiten Zeile der Abbildung ist erkennbar, dass *Verfahren 2* ein gleichmäßiger ausgeleuchtetes Ergebnisbild liefert, während *Verfahren 1* vor allem im Bereich der Tür ein leichtes Rauschen abbildet, das im 3D-Modell trotz Glättung vorhanden ist. Abgesehen von dem Rauschen, wirkt das Ergebnis mit beiden Verfahren glaubhaft.

Abbildung 18: Beleuchtungsszenario 1: Ersetzen einer Lichtquelle. Oben links: Ausgangsbild. Oben mitte: Beleuchtungssimulation der Lichtverhältnisse. Oben rechts: gewünschte Beleuchtungssimulation . Unten links: Ergebnis Verfahren 1. Unten rechts: Ergebnis Verfahren 2.

Szenario 2 Im zweiten Szenario wird die gleiche Geometrie betrachtet. Die Beleuchtung wurde aber so stark abgeschwächt, dass das Ausgangsfoto sehr dunkel ist. Schlechte Lichtverhältnisse aufzubessern könnte in der Praxis sinnvoll sein. Zur Simulation der schwachen Beleuchtung wurde die Helligkeit der Spotlichtquelle in der ersten Simulation durch die Abschwächungsfaktoren so stark reduziert, dass die Helligkeit in der Simulation in etwa der Helligkeit auf dem Foto entspricht. In der zweiten Simulation wurde die erste Lichtquelle beibehalten, zusätzlich aber eine gerichtete Lichtquelle eingefügt, die die Szene schräg von oben beleuchtet.
Man erkennt, dass die Beleuchtung bei *Verfahren 1* eher der Beleuchtungssituation in der zweiten Simulation entspricht. Dass der Boden sich grau färbt, liegt daran, dass ein weißes 3D-Modell beleuchtet wird. Wäre der Boden im Modell blau, so würde er in der ersten Simulation, bei schwachem Licht, eher schwarz und in der zweiten Simulation blau erscheinen. Das *Differentielle Rendering* würde dann einen blauen Boden in das Ergebnisbild übertragen. Bei *Verfahren 2* wird ein Problem deutlich. Ist der Helligkeitswert in der zweiten Simulation wesentlich heller als in der ersten Simulation, führt das im Ergebnisbild schnell zu Überbelichtungen. Je dunkler Simulation 1, umso heller fällt das Ergebnis aus.
Die Qualität des Tiefenbildes wird von Dunkelheit nicht beeinträchtigt, die des Farbbildes allerdings schon. Je weniger Licht vorhanden ist, umso mehr Rauschen enthält das Farbbild und umso grauer wirken die Farben. Dieses Problem kann auch durch Manipulation der Beleuchtung nicht behoben werden.

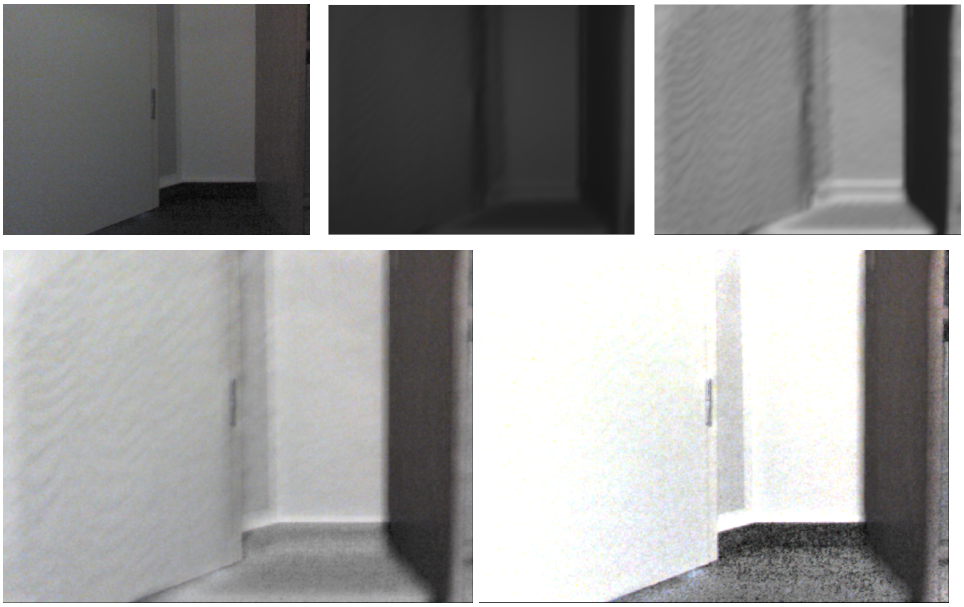

Abbildung 19: Beleuchtungsszenario 2: Einfügen einer Lichtquelle in eine dunkle Szene. Die Reihenfolge der Abbildungen entspricht der aus Abbildung 18.

Szenario 3 Im dritten Szenario wurde versucht, einen künstlichen Schatten hinzuzufügen. Die Figur im Ursprungsbild wurde frontal von einer Spotlichtquelle beleuchtet, sodass praktisch kein Schattenwurf zu erkennen ist. Da die Geometrie der Figur relativ detailliert ist, muss für das Modell eine hohe Auflösung (Polygonraster mit 320*240 Zellen) gewählt werden. Ist die Auflösung niedriger, ist die Abstufung zwischen Hintergrund und Vordergrund zu verwaschen und der Schatten schließt seitlich nicht mit dem Modell ab. Außerdem nimmt bei niedrigerer Auflösung auch der Detailgrad des Schattens immer weiter ab.
Bei der hohen Auflösung wird aber trotzdem ein Problem der Filterung, die zum Glätten des Tiefenbildes verwendet wird, deutlich. Die ebenen Flächen werden nicht auf einen einzigen Tiefenwert geglättet. Durch die kleinen Polygone sind die Abstufungen im 3D-Modell deutlich zu erkennen und übertragen sich in das Ergebnisbild.
Die Farben im mit *Verfahren 1* erzeugten Bild wirken etwas zu kräftig. Im Bild das mit *Verfahren 2* erzeugt wurde wirken sie natürlicher. Das Objekt, das den Schatten wirft, befindet sich dicht an der Wand. Hier kann auf Basis der vorhandenen Rekonstruktion ein realistischer Schatten erzeugt werden. Problematisch sind Objekte, die sich weiter von einer Wand entfernt befinden. In der Geometrie existiert eine Verbindung zwischen Objektkante und Hintergrund. Diese wird aus der Perspektive einer Lichtquelle, die das Objekt von der Seite beleuchtet, sichtbar. Sie wird auch beim Schattentest des Shadow Mapping berücksichtigt und ein Schatten der ergänzten Geometrie erscheint.
Probleme treten außerdem auf, wenn eine vorhandene Lichtquelle, die bereits für

einen Schatten sorgt, bewegt werden und sich dies auf ihren Schatten auswirken soll. Es kann zwar versucht werden den vorhandenen Schatten in der ersten Simulation zu rekonstruieren. Damit der Schatten im Ergebnisbild komplett verschwindet, muss er in der Praxis aber absolut pixelgenau rekonstruiert werden. Die Helligkeit des Schattens in der Simulation muss zudem exakt mit der Helligkeit des Schattens im Foto übereinstimmen. Mit einer nicht hundertprozentig genauen Geometrie- und Beleuchtungssimulation ist dies nicht, oder nur sehr schwierig, umzusetzen.

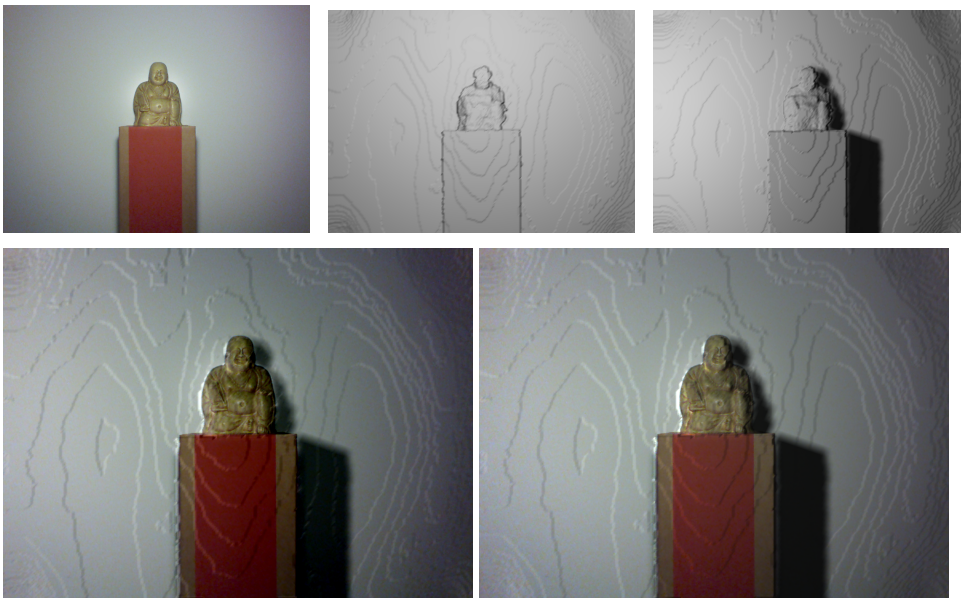

Abbildung 20: Beleuchtungsszenario 3: Einfügen eines Schattens. Die Reihenfolge der Abbildungen entspricht der aus Abbildung 18.

Szenario 4 Im vierten Szenario soll demonstriert werden, welche Effekte erreicht werden können wenn, mittels *Image Based Lighting*, Abbilder von realen Beleuchtungssituationen zum Neubeleuchten einer Szene eingesetzt werden. Verwendet wird das gleiche Ursprungsbild wie in Szenario 1. Auch die erste Beleuchtungssimulation ist damit die gleiche. Die Genauigkeit des Modells wurde verdoppelt, sodass die Geometriekanten im Modell etwas klarer zu erkennen sind. Da von diffusen Materialien ausgegangen wird und die Oberflächeneigenschaften in diesem Beispiel nicht geändert werden sollen, wird nur die mittlere Cube Map aus Abbildung 6 zum Beleuchten verwendet. Zum Tone Mapping wurde ein *Exposure* Wert von 13 (siehe 4.3.2) verwendet und dadurch eine helle Umgebung simuliert. Charakteristisch für die Beleuchtung in der Umgebung, die die Cube Map repräsentiert, war die künstliche Beleuchtung der Deckenlampe und das weiße Licht, das durch das Fenster in das Zimmer drang. Die Cube Map umschließt die Rekonstruktion so, dass das Licht des Fensters etwa auf die Türangel trifft. Wieder scheint das Ergebnisbild für *Verfahren 2* die Beleuchtung besser in das Bild zu übertragen. Dass beim textitdifferentiellen Rendering die Farbe des Bodens in der Simulation

zum Beispiel fast eins zu eins in das Ergebnisbild übernommen wird, liegt daran, dass die Materialfarben im Modell nicht korrekt rekonstruiert wurden.
Abbildung 22 zeigt die Auswirkung einer Reduzierung von *Exposure*. Nur der hellste Anteil des Lichts wird noch in die Szene übertragen. Sie erscheint dadurch insgesamt dunkler.
Image Based Lighting in der ersten Simulation zur Rekonstruktion der Beleuchtung zu verwenden ist problematisch, da es schwierig ist Cube Map und Modell für eine realistische Simulation genau aufeinander auszurichten. Weiterhin benötigt das Erstellen einer High Dynamic Range Cube Map sehr viel Zeit.

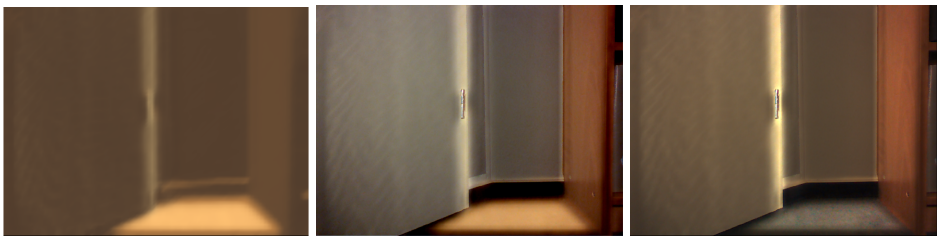

Abbildung 21: Beleuchtungsszenario 4: Anwendung von Image Based Lighting auf einer diffus reflektierenden Oberfläche. Die Reihenfolge der Abbildungen entspricht der aus Abbildung 18

Abbildung 22: Simulation der Beleuchtung mit Verfahren 2 und reduziertem *Exposure* Wert

5.2 Materialmanipulation

Im folgenden Kapitel wird demonstriert, wie mit den vorgestellten Techniken die Wirkung der im Bild sichtbaren Materialien verändert werden kann. Der Schwerpunkt liegt dabei auf dem Einfügen von Spiegelungseffekten. Hierzu werden die Parameter, die die im Bild sichtbare Oberfläche beschreiben, in der zweiten Simulation manipuliert. Dies kann in der entwickelten Software auf drei Arten erfolgen:

1. Verändern der OpenGL Materialeigenschaften

2. Einfügen einer spiegelnden Schicht, die über einen Reflektionsvektor auf eine High Dynamic Range Cube Map zugreift

3. Beim Berechnen des Farbwertes beim Image Based Lighting über einen Reflektionsvektor auf eine zweite spiegelnde Cube Map zugreifen, um so spiegelnde Materialien zu simulieren.

Einschränkung ist, dass beim momentanen Stand der Software bei einer aufgenommenen Geometrie nicht zwischen verschiedenen Materialien unterschieden wird. Das Material wird global für die gesamte Szene verwendet. Für die Beispielbilder wird das *Differentielle Rendering* eingesetzt, da bei diesem Änderungen von Materialeigenschaften eingeschlossen sind, während im Paper von Madsen und Laurensen nur Beleuchtungsänderungen gezeigt werden. In den Versuchssituationen unterschieden sich die Ergebnisse für beide Verfahren nur geringfügig.

Abbildung 23: Erweiterung eines Bildes um eine spiegelnde Komponente

Um einen Glanzeffekt in ein Bild einzufügen kann in der zweiten Simulation eine Farbe für die spiegelnde Materialkomponente eingestellt werden. Mit dem *Shininess*-Parameter kann die Glätte der Oberfläche bestimmt werden. Beim Festlegen des Helligkeitswertes für einen Punkt wird die spiegelnde Komponente auf die diffuse und ambinente Kompontene addiert. Sie ist beim differentiellen Rendering folglich das Ergebnis der Subtraktion und wird auf das Ursprungsbild addiert. Es ist also möglich, ein Bild um genau die Glanzeffekte zu erweitern, die beim Rendering berechnet werden.
Abbildung 24 zeigt verschiedene Spiegelungseffekte, die über den Zugriff auf unterschiedliche Cube Maps in ein Bild eingefügt wurden. Die Abbildungen sollen dokumentieren, wie sehr die Bildwirkung durch die unterschiedlichen Cube Maps verändert werden kann. Die Abbildungen oben links und unten rechts wurden mit High Dynamic Range Cube Maps erstellt. Das gespiegelte Licht wirkt sehr natürlich. Die Auflösung des Modells ist oben rechts am niedrigsten und wird unten

links verdoppelt und unten rechts vervierfacht. Erkennbar ist, dass dies die Genauigkeit an den Kanten erhöht, aber dass sich wie bei Szenario 3 im vorherigen Kapitel mit steigendem Detailgrad immer stärkere Muster in ebenen Flächen bilden. Die Oberfläche wirkt härter.
Wird eine spiegelnde Komponente mit Image Based Lighting hinzugefügt, stellt das Ergebnis eine Erweiterung von Abbildung 21 um zusätzliche Glanzpunkte dar. Die Stärke des Glanzes kann über einen Faktor frei reguliert werden.
Das Ändern der Materialfarbe scheint wenig sinnvoll, wenn es nur global für eine Szene durchgeführt werden kann. Es wäre hier interessanter im 3D-Modell einzelne Objekte unterscheiden zu können, die seperat konfiguriert werden können. Dennoch wäre eine Farbänderung mit *Differentiellem Rendering* möglich.

Abbildung 24: Erweiterung eines Bildes durch Spiegelungen mit verschiedenen Cube-Maps

5.3 Bewertung der Rekonstruktion

Welche Probleme bei der Rekonstruktion eines Tiefenbildes auftraten und wie versucht wurde sie zu lösen, wurde in Kapitel 4.3 gezeigt. Die Beispiele aus den beiden vorherigen Kapiteln machten deutlich, dass es der entwickelte Lösungsweg eingeschränkt möglich machte, sowohl Beleuchtungs- als auch Materialmanipulationen durchzuführen. Es ist gelungen, das Tiefenbild an das Farbbild anzugleichen, sodass Geometriekanten in beiden übereinander liegen und Manipulationen auf dem Originalbild durchgeführt werden können.
Es ist gelungen Lücken anhand der umliegenden Tiefenwerte zu schließen und ein geschlossenes Tiefenbild zu erhalten. Für kleine Lücken liefert das gewählte

Verfahren plausible Tiefenwerte. Es sollte bei der Aufnahme des Bildes darauf geachtet werden, dass nicht zu große Bildbereiche undefiniert bleiben. Hierbei kann das Tool *Kinect Explorer* von Microsoft helfen. Es ist gelungen, das Tiefenbild zu glätten, um das Rauschen des Tiefenbildes einzudämmen. Durch Nutzung des Bilateralfilters kann die Filterung so konfiguriert werden, dass nicht über Geometriekanten hinweg geglättet wird. Selbst starkes Rauschen kann durch die Verwendung von großen Filterkernen mit wenigen Iterationen unterdrückt werden. Beim Glätten kann der Farbwert hinzugezogen werden, um auch kleine Bilddetails zu erhalten. Abbildung 25 zeigt, wie wichtig die Glättung für ein Ergebnisbild ist.

Abbildung 25: Ergebnis einer leichten Beleuchtungsmanipulation auf Basis eines ungeglätteteten Tiefenbildes (links) und eines geglätteten Tiefenbildes (rechts).

Es ist außerdem gelungen die Punkte des Tiefenbildes in den 3D-Raum zu projizieren, sodass die Beleuchtung für ein perspektivisches Abbild der auf dem Farbbild sichtbaren Geometrie berechnet werden kann.
Anhand der 3D-Punktewolke ist es gelungen ein 3D-Modell mit korrekten Eckpunktnormalen zu rendern, was es möglich macht *Phong Shading* auf dem Modell anzuwenden.

Die folgenden Verbesserungen könnten in Zukunft vorgenommen werden, um die Qualität der Rekonstruktion, im Hinblick auf Bilderweiterungen zu optimieren. Für die Ergebnisbilder der Manipulationen war es störend, dass der Detailgrad des Modells global bestimmt wurde. Sinnvoller wären viele kleine Polygone an Objektkanten und wenige große Polygone in homogenen Regionen. Durch die großen Polygone würden die Muster, die in glatten Regionen entstanden sind, unterdrückt. Gleichzeitig könnten klar definierte Kanten erhalten bleiben. Abbildung 18 zeigte, dass große Polygone außerdem zu besonders weichen Beleuchtungshighlights führen. Weiterhin gab es Ungenauigkeiten an Objektkanten, die darauf zurückzuführen sind, dass Lücken im Tiefenbild nicht entsprechend der Geometrie geschlossen werden. Es wurden hierzu bereits Experimente durchgeführt, die über die bisher gezeigte Rekonstruktion hinausgehen und das Ergebnis verbessern können. Das Resultat ist noch nicht ausgereift genug sind, um in Kapitel 4 besprochen

zu werden, wird aber trotzdem als Ausblick vorgestellt. Grundlage der Experimente ist die Annahme, dass die Schatten im Tiefenbild hinter Objekten auftreten (siehe Abbildung 2). Im Tiefenbild werden der maximale und der minimale Tiefenwert bestimmt, um es in drei gleichgroße Schichten aufzuteilen. Von hinten nach vorne werden die Lücken anschließend nur innerhalb einer Schicht geschlossen. Abbildung 26 soll zeigen welche Auswirkung eine solche kleine Änderung auf das Ergebnis haben kann. Das Vordergrundobjekt ist so aufgenommen, dass es sich so nah wie möglich an der Kamera, aber gleichzeitig auch weit entfernt von der Wand, im Hintergrund befindet. Letzteres führt zu einem starken Schatten im Tiefenbild (links), ersteres zu Problemen beim Mapping von Tiefenbild und Farbbild. Beim Schließen der Lücken kommt es zu starken Ungenauigkeiten, die zum Beispiel sichtbar werden, wenn ein künstlicher Schatten eingefügt wird (mitte). Rechts sieht man, dass die Änderung am Algorithmus das Ergebnis stark verbessert hat. Teile der Lücken werden fälschlicherweise mit dem Vordergrund anstatt mit dem Hintergrund verbunden.
Zuletzt könnte versucht werden, dass 3D-Modell bereits beim Rendern in verschiedene Objekte zu unterteilen. Hier könnte, wie bei [PV11], eine Bildsegmentierung die Tiefenbild und Farbbild berücksichtigt herangezogen werden. Im Spezialfall, dass ein Bild manipuliert werden soll, auf dem Menschen zu sehen sind, kann über die integrierte Personenerkennung des Kinect-Sensors eine automatische Segmentierung vorgenommen werden. Es wäre auch denkbar, dass über ein Matching mit einem einfachen 3D-Modell so auch die unsichtbare Geometrie des Menschen geschätzt werden kann.
Insgesamt ist es gelungen aus einem einzigen Tiefen- und Farbbild ein 3D-Modell zu erzeugen, das die Basis für Bilderweiterungen bilden kann, bei dem aber trotzdem noch Optimierungsbedarf besteht um den Realismus der Ergebnisse zu erhöhen. Besonders gute Ergebnisse liefern Szenen mit einfacher Geometrie.

Abbildung 26: Experiment zur Verbesserung der Rekonstruktion

5.4 Bewertung der eingesetzten Verfahren

Szenario 1 in Kapitel 5.1 zeigte, dass es mit den verwendeten Rendering- und Bilderweiterungsverfahren möglich ist glaubhafte Beleuchtungsänderungen durchzuführen. Die verwendeten Neubeleuchtungsverfahren manipulierten das Ausgangsbild auf Pixelbasis. Es werden also keine Pixel neu gesetzt, sondern es wird lediglich der Farbwert der einzelnen Pixel manipuliert. Dies führt dazu, dass gute Ergebnisse erzielt werden können, ohne dass die Belechtungssituation des Fotos perfekt rekonstruiert wurde. Das Verfahren von Madsen und Laurensen ist insgesamt besser geeignet, um die Beleuchtung auf Basis der vorliegenden Beleuchtungs- und Geometrierekonstruktion zu verändern. Die durchgeführten Beleuchtungsänderungen wirkten homogener. Eine Materialsimulation war hierbei nicht nötig. Würde die Materialrekonstruktion vorliegen, könnten die Farbfehler die beim *Differentiellen Rendering* auftreten reduziert werden. Um einen Mehraufwand zu vermeiden muss ein Weg gesucht werden die Materialeigenschaften der Objekte auf dem Foto automatisch zu rekonstruieren. Ein Ansatz hierzu findet sich zum Beispiel in [FGR92].
Die Verwendung des OpenGL Beleuchtungsmodells macht zwar keine physikalisch korrekte Beleuchtung möglich, war aber trotzdem ausreichend, um einfache Beleuchtungsmanipulationen vorzunehmen. Für mehr Realismus kann versucht werden, die Lichtverteilung der simulierten Lichtquellen über eine so genannte *Lichtstärke-Verteilungskurve* an reale Lichtquellen anzunähern [Gor08, S. 10]. Der Einsatz rechenaufwändiger, globaler Beleuchtungssimulationen würde zudem Beleuchtungsphänomene wie indirektes Licht berücksichtigen. Für ein fotorealistisches Ergebnis wäre trotzdem ein genaues Abbild der Geometrie des Raumes notwendig.
Die Verwendung von Shadow Mapping eignet sich um Schatten in eine Szene einzufügen. Für einfache Schatten konnte das in dieser Arbeit gezeigt werden. Einschränkungen an die Schatten waren eher durch die rekonstruierte Geometrie gegeben, als durch das verwendete Verfahren. Würden mehr Informationen über die Geometrie vorliegen, könnten auch Schatten eingefügt werden, die die komplette Silhouette eines Objektes abbilden. Rekonstruktionstechniken wie in [IKH+11] und [WKF+12] könnten dies liefern. Der Raum muss allerdings durch ein Bewegen von Kinect gescannt werden. Die Rekonstruktion mit einem Klick geht verloren. Der Einsatz von *Image Based Lighting* stellte einen Versuch dar eine globale Beleuchtung zu simulieren. Eine über das Verfahren bestimmte Beleuchtung konnte in die Szene übertragen werden. Nachteil ist der hohe Zeitaufwand, der aufgebracht werden muss, um die High Dynamic Range Cube Map zu erstellen. Auch das Berechnen der Cube Maps für diffuse und spiegelnde Beleuchtung mit *HDR Shop* ist sehr zeitaufwändig. Vorteil ist, dass sich eine so aufgezeichnete Beleuchtungssituation in beliebige Szenen übertragen lässt, also nicht nur für eine einzige Neubeleuchtung erstellt wird. Im Internet finden sich zahlreiche HDR-Cube Maps mit interessanten Beleuchtungssituationen.
Das Einfügen von Spiegelungen mittels Cube Maps erlaubt es Reflexionen einzu-

fügen. Die Reflexionen beziehen sich aber nur dann direkt auf die sichtbare Umgebung, wenn für sie eine Cube Map vorliegt. Werden Cube Maps von anderen Umgebungen verwendet, können zwar optisch interessante Ergebnisse erzielt werden, die eingefügte Reflexion weist aber keinen Bezug zur Realität auf. In dieser Arbeit wurden nur globale Materialänderungen vorgenommen. Reizvoller wäre es Objekte unabhängig voneinander zu manipulieren.
Die Idee eine Software zu entwickeln, die es möglich macht alle Schritte einer Bilderweiterung, von der Aufnahme des Bildes mit Kinect bis zum finalen Ergebnisbild umzusetzen, ist sinnvoll um den durchgeführten Vorgang im Ganzen zu begreifen. Dass Konfigurationsänderungen sofort sichtbar werden, ermöglicht es, ein Ergebnisbild zu optimieren. Die freie Wahl des Detailgrades des Modells macht Beleuchtungsänderungen in Echtzeit möglich. Die Software stellt ein experimentelles Werkzeug dar, um die in dieser Arbeit entwickelten Techniken nachvollziehbar zu machen. In diesem und dem vorherigen Kapitel wurden bereits viele Anregungen gegeben wie sie noch erweitert werden kann. Um weiterführende Techniken umzusetzen, könnte sie die Grundlage für weitere Arbeiten bilden. Die Arbeit mit Cube Maps, Framebufferobjects und Shadern wird durch bereitgestellte Klassen erheblich vereinfacht. Das erweiterbare Interface umfasst die Basisfunktionen zur Beleuchtungs- und Materialmanipulation. Andere wichtige Grundlagen, wie das Laden von Kinect-Datensätzen und das Speichern eines Ergebnisbildes sind ebenfalls vorhanden.

6 Fazit und Ausblick

In dieser Arbeit wurde gezeigt, dass die Möglichkeit mit einem Kinect-Sensor gleichzeitig Farb- und Tiefendaten aufzuzeichnen, neue Wege in der Bildbearbeitung und Bilderweiterung eröffnen kann. Trotz seines günstigen Preises reichte die Qualität des Tiefenbildes aus, um mit Zuhilfenahme der ungefähren Beleuchtungssituation zum Zeitpunkt der Aufnahme, die Basis für einfache Beleuchtung- und Materialmodifikationen im Farbbild zu bilden. Die in dieser Arbeit vorgestellten Ergebnisse zeigen, in welche Richtung die Entwicklung gehen könnte. Gleichzeitig lässt sich aber auch erkennen, dass die Techniken noch weiterentwickelt werden müssen, um in vielfältigen Situationen realistische Bilder zu liefern. Anregungen hierzu wurden in dieser Arbeit gegeben.
Spannend ist vor allem, wenn die Bilderweiterung dank moderner 3D-Hardware in Echtzeit direkt auf dem Ergebnisbild durchgeführt wird. Die 3D-Simulationen bleiben komplett im Hintergrund und dienen lediglich zur Bestimmung der Helligkeit. Die Möglichkeit, Lichtquellen ohne großen Aufwand mit einem Regler direkt in einem zuvor aufgenommenen Bild bewegen zu können, stellt eine interessante und unbekannte Erfahrung dar. Grenzen der Fotografie werden scheinbar ausgehebelt.
Die Möglichkeiten der Beleuchtungs- und Materialmanipulation werden dadurch beschränkt, dass mit einem einzelnen Bild nur die auf diesem Bild sichtbare Geometrie erfasst werden kann, das Licht aber auch von der restlichen Geometrie beeinflusst wird. Projekte wie *Kinect Fusion* oder *Kinectinous* zeigen, dass Kinect dazu in der Lage ist, ganze Räume zu scannen, was die Möglichkeit der künstlichen Beleuchtungsberechnung noch verbessert.
Beachtet werden muss, dass eine Hardware verwendet wurde, die auf einen Einsatz im Spielebereich optimiert ist. Der Fokus liegt eher darauf, intuitive Gestensteuerung möglich zu machen, als millimetergenaue Geometrieabbilder erstellen zu können. Es wäre denkbar, dass in Zukunft Geräte, die mit ähnlichen Techniken arbeiten, speziell für den Einsatz zur augmentierten Bildsynthese hergestellt werden. Interessant wäre die Möglichkeit solche Sensoren in Digitalkameras zu integrieren, sodass zu jedem aufgenommenen Farbbild auch ein Tiefenbild vorliegt. Moderne Bildverarbeitungsprogramme könnten eine 3D-Rekonstruktion komplett im Hintergrund durchführen und dem Anwender dadurch die Möglichkeit geben, Lichtquellen und Materialeigenschaften direkt in seinem Bild zu manipulieren. Dies könnte die Arbeit mit Bildbearbeitungsprogrammen revolutionieren.
Wenn es möglich wäre einen Sensor herzustellen, der bei der Aufnahme eines Fotos gleichzeitig auch die Beleuchtungssituation erfasst, könnten in Kombination mit dem Tiefenbild die Materialeigenschaften rekonstruiert werden und die Freiheit an möglichen Bildmanipulationen würde sich noch vervielfachen. Da ein solcher Sensor noch nicht existiert, können in zukünftigen Arbeiten die vorgestellten Techniken zur Rekonstruktion und Simulation weiterentwickelt werden und ich denke, dass das in dieser Arbeit Gezeigte nur einen sehr kleinen Teil von dem darstellt, was diese oder ähnliche Hardware in Zukunft möglich machen kann.

Literatur

[Ahn12] S. H. Ahn. OpenGL Frame Buffer Object (FBO), 2012. http://www.songho.ca/opengl/gl_fbo.html, Abruf: 21.03.2013.

[AJL+12] M.R. Andersen, T. Jensen, P. Lisouski, A.K. Mortensen, M.K. Hansen, T. Gregersen, and P. Ahrendt. Kinect Depth Sensor Evaluation for Computer Vision Applications. Technical report, Department of Engineering, Aarhus University, 2012.

[CALT12] J. C .K. Chow, K. D. Ang, D. D. Lichti, and W. F. Teskey. Performance analysis of a low-cost triangulation-based 3d camera: Microsoft kinect system. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XXXIX-B5:175–180, 2012.

[CLV12] L. Cruz, D. Lucio, and L. Velho. Kinect and rgbd images: Challenges and applications. *2012 25th SIBGRAPI Conference on Graphics, Patterns and Images Tutorials*, pages 36–49, 2012.

[Com] Computerwoche. Kinect-Sensor. http://images.computerwoche.de/images/computerwoche/bdb/1814920/890.jpg, Abruf: 21.03.2013.

[Deb] P. Debevec. Creating a Light Probe. http://ict.debevec.org/~debevec/HDRShop/tutorial/tutorial5.html, Abruf: 21.03.2013.

[Deb98] P. Debevec. Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 189–198, New York, NY, USA, 1998. ACM.

[FGR92] A. Fournier, A. S. Gunawan, and C. Romanzin. Common illumination between real and computer generated scenes. Technical report, Vancouver, BC, Canada, Canada, 1992.

[GMK03] T. Grosch, S. Müller, and W. Kresse. Goniometric light reconstruction for augmented reality image synthesis, 2003.

[Gor08] T. Gorsch. *Augmentierte Bildsynthese (Dissertation)*. Der Andere Verlag, 2008.

[Gou71] H Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, C-20(6):623–629, June 1971.

[HM06] F. Houlmann and S. Metz. High Dynamic Range Rendering in OpenGL. Université de technologie Belfort-Montbéliard, 2006. http://transporter-game.googlecode.com/files/HDRRenderingInOpenGL.pdf, Abruf: 21.03.2013.

[Hö12] V. Högman. Building a 3D map from RGB-D sensors. Master's thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2012.

[IKH+11] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and et al. Kinectfusion : Real-time 3d reconstruction and interaction using a moving depth camera. *interactions*, page 559–568, 2011.

[Kro11] J. Krommweh. Bilaterale Filter zur Entstörung von Bildern. Universität Duisburg-Essen, 2011. http://www.uni-due.de/mathematik/krommweh/talk_Gemen_Krommweh.pdf, Abruf: 21.03.2013.

[Meh13] Y. Mehdi. Xbox execs talk momentum and the future of tv. Website, 2013. Online unter: http://www.microsoft.com/en-us/news/features/2013/feb13/02-11Xbox.aspx, abgerufen am 21.03.2013.

[Mica] Microsoft Developer Network. Coordinate spaces. http://msdn.microsoft.com/en-us/library/hh973078.aspx, Abruf: 21.03.2013.

[Micb] Microsoft Developer Network. CoordinateMapper. http://msdn.microsoft.com/en-us/library/microsoft.kinect.coordinatemapper.aspx, Abruf: 21.03.2013.

[Micc] Microsoft Developer Network. Depth Stream. http://msdn.microsoft.com/en-us/library/jj131028.aspx, Abruf: 21.03.2013.

[Micd] Microsoft Developer Network. Einführung in WPF. http://msdn.microsoft.com/de-de/library/vstudio/aa970268.aspx, Abruf: 21.03.2013.

[Mice] Microsoft Developer Network. Kinect for windows sensor components and specifications. http://msdn.microsoft.com/en-us/library/jj131033.aspx, Abruf: 21.03.2013.

[ML04] C. B. Madsen and R. Laursen. Image relighting: Getting the sun to set in an image taken at noon. In *13th Danish Conference on Pattern Recognition and Image Analysis*, pages 13–20, 2004.

[NFH07] A. Nischwitz, M. Fischer, and P. Haberäcker. *Computergrafik und Bildverarbeitung*. Vieweg & Sohn Verlag, 2nd edition, 2007.

[PM] F. Pellacini and Bunnell M. GPU Gems. `http://http.developer.nvidia.com/GPUGems/gpugems_ch11.html`, Abruf: 21.03.2013.

[PV11] F. Prada and L. Velho. Grabcut+d - VISGRAF PROJECT. Website, 2011. `http://www.impa.br/~faprada/courses/procImagenes`, Abruf: 21.03.2013.

[RLKG+09] R. J. Rost, B. Licea-Kane, D. Ginsburg, J. M. Kessenich, B. Lichtenbelt, H. Malan, and M. Weiblen. *OpenGL Shading Language*. Addison-Wesley Professional, 3rd edition, 2009.

[RWPD06] E. Reinhard, G. Ward, S. Pattanaik, and P. Debevec. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. The Morgan Kaufmann Series in Computer Graphics. Elsevier Science, 2006.

[San12] K. Sanford. Smoothing Kinect Depth Frames in Real-Time, 2012. `http://www.codeproject.com/Articles/317974/KinectDepthSmoothing`, Abruf: 21.03.2013.

[SM00] Gibson S. and A. Murta. Interactive rendering with real-world illumination. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000, Brno, Czech Republic, June 26-28, 2000*, pages 365–376. Springer, 2000.

[TM98] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proceedings of the Sixth International Conference on Computer Vision*, ICCV '98, pages 839–, Washington, DC, USA, 1998. IEEE Computer Society.

[Whi12] C. White. Kinect fusion coming to kinect for windows, 2012. `http://blogs.msdn.com/b/kinectforwindows/archive/2012/11/05/kinect-fusion-coming-to-kinect-for-windows.aspx`, Abruf: 21.03.2013.

[Wig08] S. Wigard. *XNA Game Studio Express: Spieleprogrammierung für PC und Xbox*. Redline GMBH, Heidelberg, 2nd edition, 2008.

[WKF+12] J.B. Whelan, T.and McDonald, M. Kaess, M.F. Fallon, H. Johannsson, and J.J. Leonard. Kintinuous: Spatially extended KinectFusion. In *RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, Sydney, Australia, Jul 2012.

[WND97] M. Woo, J. Neidler, and T. Davis. OpenGL Programming Guide. The Official Guide to Learning OpenGL, Version 1.1, 1997. http://www.glprogramming.com/red/chapter02.html#name6, Abruf: 21.03.2013.