Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer zai-bsc-2012-12

Bachelorarbeit

im Studiengang „Angewandte Informatik“

Arduino-gesteuerte Laser-Abtastung realer Objekte zur Generierung dreidimensionaler Gitter

Andreas Wilhelm

am Institut für

Numerische und Angewandte Mathematik

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

2. Mai 2012

Georg-August-Universität Göttingen
Zentrum für Angewandte Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel. +49 (5 51) 39-1 72 010
Fax +49 (5 51) 39-1 46 93
E-Mail office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

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

Göttingen, den 2. Mai 2012

Bachelorarbeit

Arduino-gesteuerte Laser-Abtastung realer Objekte zur Generierung dreidimensionaler Gitter

Andreas Wilhelm

2. Mai 2012

Betreut durch Dr. Jochen Schulz & Prof. Dr. Gert Lube
Lehrstuhl für Numerische und Angewandte Mathematik
Georg-August-Universität Göttingen

Inhaltsverzeichnis

Quellcodeverzeichnis

Abbildungsverzeichnis

1 Einleitung

Als Anfang der sechziger Jahre des 20. Jahrhunderts Pierre Bézier bei Renault und Paul de Casteljau bei Citroën die Bezierkurven entwickelten, war es noch undenkbar, dass die Darstellung, Modellierung und Rekonstruktion dreidimensionaler Objekte in den kommenden Jahren so an Bedeutung gewinnen sollte.

Nicht nur im Rahmen der Konstruktion und Fertigung von Industriegütern spielt Computer Aided Design (CAD) zunehmend eine große Rolle, sondern auch im privaten Bereich. Spielekonsolen wie die Xbox erkennen den Spieler und integrieren ihn direkt in das Spiel. Museen bieten ihren Besuchern aus aller Welt die Möglichkeit, virtuelle Rundgänge ganz bequem vom privaten PC zuhause zu nutzen. Es gibt sogar die ersten erschwinglichen 3-D-Drucker, wie den MakerBot, die es jedem Tüftler ermöglichen, einfach und schnell hochwertige Prototypen zu fertigen.

Aufgrund dieser ständig anwachsenden Anzahl von Anwendungsgebieten steigt auch die Nachfrage nach einfachen Methoden zur Modellierung dreidimensionaler Objekte bzw. Szenen. Man ist in der Lage, beispielsweise eine Person für ein Computerspiel oder ein Flugzeug zur Simulation thermodynamischer Flüsse am PC zu konstruieren, jedoch ist das sehr zeitaufwendig. Einfacher wäre es, ein bereits existierendes, reales Objekt als Basis einer solchen Konstruktion zu verwenden. Daher steigt die Nachfrage nach geeigneten Algorithmen zur dreidimensionalen Rekonstruktion.

In den vergangenen Jahren wurde deshalb verstärkt an Algorithmen und Verfahren zur Imitation des menschlichen Sehvermögens geforscht. Leider beschränken sich die Erfolge in diesem Bereich großteils auf aktive Verfahren, die auf eine Bestrahlung des Objekts bzw. der Szene setzen und so äußerst anfällig bezüglich Umwelteinflüssen sind.

Der enorme Hardwareaufwand, den solche aktiven Verfahren erfordern, schließt außerdem einen Einsatz im privaten Sektor größtenteils aus. Nur wenige Ausnahmen wie die Xbox Kinect haben es bisher in unsere heimischen vier Wände geschafft.

Das Ziel dieser Arbeit ist daher der Entwurf eines alternativen Verfahrens auf Basis einer für den privaten Nutzer erschwinglichen Hardware.

1.1 Alternative Verfahren

Die verschiedenen Verfahren zur Konstruktion einer dreidimensionalen Szene lassen sich in aktive und passive Verfahren unterteilen.

Die aktiven Verfahren setzen bei der Rekonstruktion der dreidimensionalen Daten auf eine Bestrahlung der Szene mit elektromagnetischen Wellen wie Licht-, Radar- oder Mikrowellen. Die am Objekt reflektierten Wellen werden mithilfe eines Sensors erfasst und dann ausgewertet. Die geometrische Interpretation der so erhaltenen Daten kann nun anhand der Signallaufzeit oder der Veränderung eines projizierten Musters in Abhängigkeit von der Position des Senders und Empfängers mit Hilfe trigonometrischer Rekonstruktionsalgorithmen ermittelt werden. Als Beispiel wäre hier die XBox Kinect zu nennen, die eine Infrarot-Punktwolke in die Szene projiziert und diese mit einer Infrarot-Kamera erfasst. Die Krümmung des Musters an Objekten lässt nun Rückschlüsse auf die Tiefeninformationen der Szene zu.

Passive Verfahren hingegen setzen auf die Rekonstruktion von Tiefeninformationen auf Basis verschiedener Bildinformationen, wie Fluchtpunkten, stereoskopische Disparität, Texturen, Bildschärfe und Schattenwurf. Abgesehen davon, dass nur mit Hilfe der Stereoskopie, für die zwei Bilder benötigt werden, brauchbare Tiefeninformationen gewonnen werden können, ist die Komplexität und mittlere Laufzeit der Algorithmen passiver Verfahren im Vergleich zu den Algorithmen aktiver Verfahren deutlich höher. Ein Beispiel für den

Einsatz passiver Verfahren sind 3-D-Aufnahmen, die auf Basis zweier Bilder verschiedener, in einem festen Abstand montierter Kameras die Stereoskopie zur Rekonstruktion der Szenengeometrie nutzen.

Ein direkter Vergleich von aktiven und passiven Verfahren ist schwer, da aktive Verfahren aufgrund des enormen Hardwareaufwandes zwar teurer, passive Verfahren jedoch in der Regel komplexer in Bezug auf die eingesetzten Algorithmen sind. Außerdem sind viele aktive Verfahren anfälliger in Hinsicht auf Umgebungsfaktoren. Eine ungünstige Bestrahlung der Szene durch Umwelteinflüsse, wie Tageslicht, können die gewonnenen Daten leicht unbrauchbar machen. Zudem erfordern aktive Verfahren den Einsatz spezieller Hardware, wodurch sich die Rekonstruktion dreidimensionaler Daten auf aktuelle Aufnahmen beschränkt.

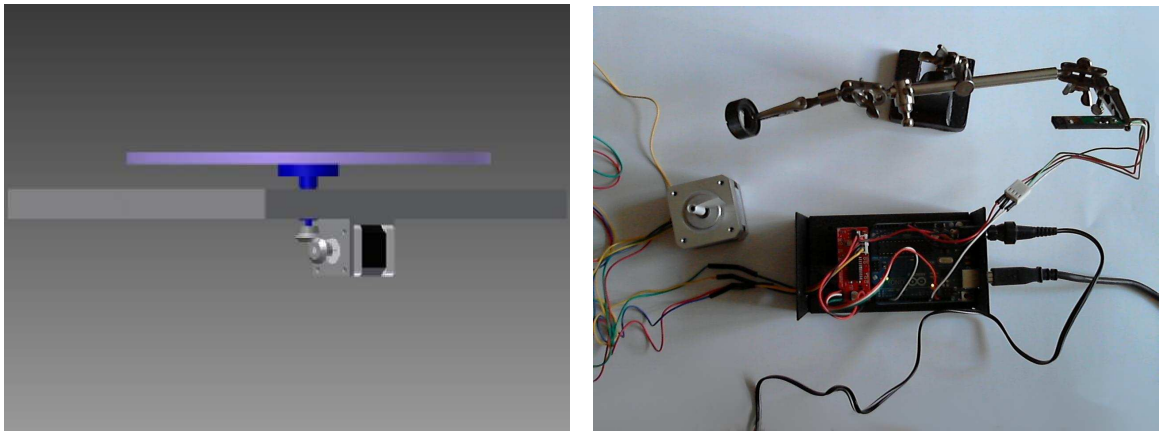
Passive Verfahren hingegen lassen sich auch auf jegliche digitalisierte Videoaufnahme anwenden, wodurch es beispielsweise möglich ist, Aufnahmen aus der Anfangszeit des Fernsehens in 3-D-Filme umzuwandeln.

Unabhängig von der Art des eingesetzten Verfahrens wird als Ergebniss eine Punktwolke im $\mathbb{R}^3$, die zur Darstellung im Computer verwendet werden kann, erzeugt.

1.2 Generierung von Punktwolken durch Laserscans

Das im Rahmen dieser Arbeit ausgearbeitete aktive Verfahren zur dreidimensionalen Rekonstruktion setzt auf eine Beleuchtung des Objekts mittels eines mit einer plankonvexen Zylinderlinse aufgefächerten Laserstrahls. Die Krümmung der so projizierten Linie am Objekt bildet die Grundlage der Auswertung der mit einer leicht versetzten Kamera aufgenommenen Bilder. Da auf diese Weise nur eine zweidimensionale Punktwolke gewonnen werden kann, wird das zu erfassende Objekt unter dem Laserstrahl schrittweise rotiert.

Auf diese Weise erhält man eine Bildfolge von Projektionen auf die Grundebene. Jedes dieser Bilder enthält nun die Tiefeninformationen innerhalb einer Ebene. Durch Kombi-



(a) Drehteller, CAD-Ansicht (b) Arduino

Abbildung 1.1: Hardware-Konstruktion

nation dieser Tiefeninformationen kann die Punktwolke eines Rotationskörpers errechnet werden, der nun alle Tiefeninformationen enthält. Durch geeignete Vernetzung der gewonnenen Punkte kann ein dreidimensionales Objekt erzeugt werden, das als Grundlage weiterer Berechnungen dienen kann.

Die Rotation des Objekts übernimmt ein von einem Arduino [BC05] gesteuerter Drehteller (siehe Abbildung 1.1). Neben der Ansteuerung des dafür benötigten Schrittmotors kümmert sich der Arduino auch um die Schaltung des Lasermoduls. Da jedoch der Arduino in seiner Leistung sehr beschränkt ist, übernimmt ein Computer die Kontrolle der Webcam und koordiniert seine Arbeit über die serielle Schnittstelle mit dem Arduino.

1.3 Aufbau und Ziel der Arbeit

Ziel dieser Arbeit ist die Erläuterung der zur Rekonstruktion von Tiefeninformationen benötigten mathematischen Grundlagen und der Entwurf sowie die Konstruktion eines kostengünstigen 3-D-Scanners und der dazugehörigen Software.

Zu diesem Zweck werden in Kapitel 2 die Grundlagen der affinen und projektiven Geometrie behandelt und die für den entworfenen Algorithmus benötigten Definitionen geliefert. Der Algorithmus selbst wird dann in Kapitel 3 vorgestellt. Seine Implementierung und die dazu eingesetzten Bibliotheken und Plattformen werden im folgenden Kapitel 4 detailliert besprochen. Abschließend werden in den Kapiteln 5 und 6 die Ergebnisse der Arbeit vorgestellt und mit denen anderer Verfahren, wie dem aus der Vergleichsarbeit [LLRS05], verglichen.

2 Mathematische Grundlagen

In diesem Kapitel werden die Grundlagen der affinen und projektiven Geometrie behandelt, die ab Abschnitt 2.2 zur mathematischen Modellierung der Problemstellung herangezogen werden. In den Abschnitten 2.6 bis 2.8 wird dann die Basis für den in Kapitel 3 vorgestellten Algorithmus geliefert.

2.1 Der affine Raum

Die Rekonstruktion einer dreidimensionalen Szene anhand eines zweidimensionalen Bildes erfordert eine mathematische Abstraktion.

Definition 2.1.1 (Affiner Raum) *Sei A eine Menge geometrischer Punkte, V ein Vektorraum und $\Sigma : A \times A \to V$ eine Abbildung. Dann nennt man (A, V) einen affinen Raum. Es gilt:*

- $\forall P, Q, R \in A : \vec{PQ} + \vec{QR} = \vec{PR}$
- $\forall P \in A, \vec{v} \in V : \exists! Q \in A | \vec{v} = \vec{PQ}$

Falls der A zugrunde liegende Vektorraum V bekannt ist, wird auch A alleine als affiner Raum bezeichnet.

Aus dieser Definition lassen sich intuitiv die folgenden Punkte ableiten:

1. Sei $O \in A$ ein beliebiger fester Punkt als Ursprung gewählt. Die Abbildung Σ, die jedem Punkt $P \in A$ den sogenannten Ortsvektor $\vec{OP} \in V$ zuordnet, ist dann eine eindeutige, wenn auch von O abhängige, Abbildung zwischen dem affinen Raum und seinem Vektorraum.

2. Die Abbildung $V \times V \to V$ mit $(v, w) \mapsto w - v$ ordnet zwei Punkten ihren Verbindungsvektor zu und ermöglicht somit die Betrachtung von V als affiner Punktraum.

Definition 2.1.2 (Affine Basis) *Sei A ein affiner Raum über dem Vektorraum V. Eine Menge von Punkten $P_0, \ldots, P_n \in A$ heißt affine Basis von A, falls die Vektoren $P_0\vec{P}_1, \ldots, P_0\vec{P}_n$ eine Basis von V bilden.*

Auf Basis dieser Definition kann nun ein Koordinatensystem wie folgt definiert werden:

Definition 2.1.3 (Koordinatensystem) *Eine Menge $(P_0, \ldots, P_n)$ mit der affinen Basis $(P_0, \ldots, P_n)$ von A heißt affines Koordinatensystem mit Ursprung P_0.*

In der weiteren Arbeit werden folgende Notationen verwendet:

Ortsvektor: Für jeden Punkt $P \in A$ und Ursprung O ist $v = \vec{OP}$ Ortsvektor von P.

Koordinate: Komponenten $(x_1, \ldots, x_n)$ von $v \in V$ heißen Koordinaten bzgl. einer Basis $(e_1, \ldots, e_n)$, gdw.

$$\vec{OP} = v = x_1 e_1 + \cdots + x_n e_n$$

2.2 Kamerabezugssystem

Affine Räume ermöglichen uns eine mathematische Abstraktion unserer Problemstellung. Dazu betrachtet man den über dem Vektorraum $V = \mathbb{R}^3$ definierten affinen Raum A mit dem Ursprung $O_V = (0, 0, 0)^T$ von V.

Die Kamera des Aufbaus ist das Projektionszentrum, der Ursprung O_V des von A erzeugten Koordinatensystems. Eine Bildebene Π repäsentiert das zweidimensionale Bild, das zur Rekonstruktion verwendet werden soll. Des Weiteren ist Π parallel zu der durch die X- und Y-Achse aufgespannten Ebene ausgerichet. Die Brennweite f der Kamera entspricht gerade dem Abstand dieser beiden Ebenen zueinander. Das Zentrum der Bildebene O_Π ist gerade der Schnittpunkt von Π mit der z-Achse, die zugleich die optische Achse der Bildebene darstellt (siehe Abbildung 2.1).

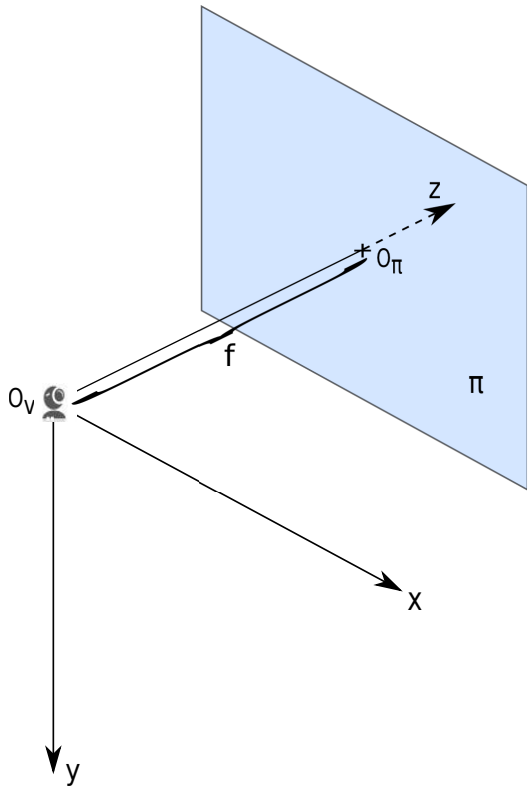


Abbildung 2.1: Kamerabezugssystem

2.3 Perspektivische Projektion

Betrachtet das menschliche Auge eine Szene, so wirken Objekte, die weiter entfernt liegen, kleiner als näher liegende Objekte gleicher Größe. Dieses Phänomen bezeichnet der Mathematiker als perspektivische Projektion oder auch Zentralprojektion. Die mit Hilfe einer Kamera aufgenommenen Bilder zeigen die erfasste Szene ebenfalls in perspektivischer Projektion. Daher ist es wichtig, dieses Gebiet genauer zu betrachten und unsere Definition eines Kamerabezugssystems zu verfeinern.

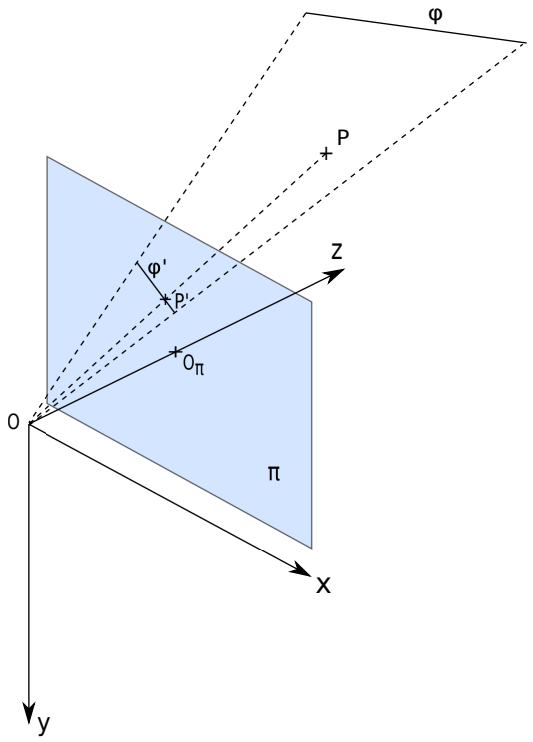


Abbildung 2.2: Perspektivische Projektion

Definition 2.3.1 (Projektionsebene) *Sei A der über dem $\mathbb{R}^3$ affine Raum mit dem Ursprung $O_V = (0,0,0)^T$. Sei Π die von den zwei orthonormalen Richtungsvektoren $\vec{u}$ und $\vec{v}$ aufgespannte und parallel zu der durch die x- und y-Achse aufgespannten Ebene ausgerichtete Ebene. Der Schnittpunkt S der durch den Ursprung O_V und einen beliebigen Punkt $P \in A$ mit $P \neq O_V$ eindeutig definierten Sichtgerade δ mit der Ebene Π lässt sich darstellen als:*

$$\vec{S} = O_\Pi + s\vec{u} + t\vec{v} \tag{2.1}$$

s und t sind dann gerade die Koordinaten des zweidimensionalen Punktes der Projektionsbene Π mit Mittelpunkt O_Π.

Anschaulich bedeutet das, dass man einem Punkt P des zugrunde liegenden dreidimensionalen Raumes einen Punkt P' der Bildebene zuordnen kann (siehe Abbildung 2.2).

Im speziellen Fall des Kamerabezugssystems mit dem Ursprung $O_V = (0,0,0)^T$ und der Brennweite f kann die Berechnung des Bildpunktes weiter vereinfacht werden.

Zu diesem Zweck muss man sich folgende Dinge klar machen:

- Der Mittelpunkt der Bildebene ist $O_\Pi = (0,0,f)^T$.
- Die beiden die xy-Ebene aufspannenden Vektoren $(1,0,0)^T$ und $(0,1,0)^T$ um die Brennweite f in z-Richtung verschoben entsprechen gerade den die Bildebene aufspannenden Vektoren u und v.

Unter diesen Voraussetzungen kann ein beliebiger Punkt $P = (X,Y,Z)^T$ auf einen Punkt $P' = (x,y,f)^T$ der Bildebene abgebildet werden, wobei gilt:

$$\begin{aligned} x &= f\frac{X}{Z} \\ y &= f\frac{Y}{Z} \end{aligned}$$

2.4 Homogene Koordinaten

Definition 2.4.1 (Homogene Koordinaten) *Sei $P = (x_1, x_2, \ldots, x_n)$ ein Punkt des $\mathbb{R}^n$. Unter homogenen Koordinaten des Punktes P versteht man das $n+1$-Tupel*

$$P_H = (hx_1, hx_2, \ldots, hx_n, h), h \in \mathbb{R}, h \neq 0 \quad .$$

Der gewählte Faktor h heißt Skalierungsfaktor.

Auf Basis dieser Definition lässt sich nun eine Abbildung $\psi : A^n \to P(A^{n+1})$ vom affinen Raum in den projektiven Raum mit $(x,y,z)^T \mapsto (x,y,z,w)^T$ definieren. Umgekehrt kann natürlich auch eine Abbildung $\psi : P(A^{n+1}) \to A^n$ vom projektiven Raum in den affinen Raum mit $(x,y,z,w)^T \mapsto (\frac{x}{w}, \frac{y}{w}, \frac{z}{w})^T$ definiert werden.

Eine affine Transformation eines Punktes kann durch eine Matrix-Vektor-Multiplikation beschrieben werden. Zu diesem Zweck wird ein Punkt um eine vierte Koordinate mit konstantem Wert 1 erweitert. Der aus dieser Erweiterung entstehende Punkt $P = (x, y, z, 1)^T$ kann durch eine Matrixmultiplikation $P' = M * P$ verschoben, skaliert oder rotiert werden.

Definition 2.4.2 (Translation) *Eine Verschiebung bzw. Translation um α, β und γ entlang der x-, y- und z-Achse eines in homogenen Koordinaten vorliegenden Punktes $P = (x, y, z, 1)^T$ kann durch Multiplikation mit einer Matrix*

$$T = \begin{pmatrix} 1 & 0 & 0 & \alpha \\ 0 & 1 & 0 & \beta \\ 0 & 0 & 1 & \gamma \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

beschrieben werden.

Definition 2.4.3 (Skalierung) *Eine Skalierung um einen Faktor α, β und γ entlang der x-, y- und z-Achse eines in homogenen Koordinaten vorliegenden Punkts $P = (x, y, z, 1)^T$ kann durch Multiplikation mit einer Matrix*

$$T = \begin{pmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

beschrieben werden.

Definition 2.4.4 (Rotation) *Eine Rotation um einen Winkel α eines in homogenen Koordinaten vorliegenden Punkts $P = (x, y, z, 1)^T$ kann durch Multiplikation mit einer Matrix beschrieben werden. Rotiert wird dabei um eine der drei Achsen des Koordinatensystems. Je nach Achse muss*

eine andere Matrix M_x, M_y oder M_z zur Rotation um die x-, y- oder z-Achse gewählt werden.

$$M_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$M_y = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$M_z = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.5 Konvolution

Die Faltung oder Konvolution zweier Funktionen f und g ist eine mathematische Operation, die eine dritte Funktion liefert, die im Regelfall eine veränderte Version einer der beiden Eingangsfunktionen ist.

Definition 2.5.1 (Konvolution) *Seien $f, g : \mathbb{R}^n \to \mathbb{C}$ zwei Funktionen. Die Faltung $(f * g)$ ist dann definiert durch das Integral $(f * g)(x) := \int_{\mathbb{R}^n} f(\theta) g(x - \theta)\, \mathrm{d}\theta$.*

Anschaulich betrachtet handelt es sich bei der eindimensionalen Faltung zweier Funktionen um eine Gewichtung der Funktionswerte der einen Funktion durch die Funktionswerte der anderen. Verwendet werden kann dies beispielsweise zur Gewichtung einzelner Werte eines Datenstroms, wie es oft Anwendung in der Signal- und Bildverarbeitung geschieht. Die Beschränktheit des Definitionsbereichs in solchen Anwendungsgebieten gibt Anlass zu der folgenden Definition:

Definition 2.5.2 (Fortsetzung des Definitionsbereichs) *Seien f und g zwei Funktionen über einem beschränkten Definitionsbereich $\mathbb{D}$. Die Faltung $(f * g)$ erfordert eine Erweiterung des Definitionsbereichs. Zu diesem Zweck setzt man die Funktion entweder außerhalb des Definitionsbereich mit der Nullfunktion oder periodisch fort.*

Diese Definition ermöglicht nun auch eine Faltung reellwertiger, beschränkter Funktionen durch eine Abbildung auf Null oder eine periodische Fortsetzung außerhalb des Definitionsbereichs. Da jedoch beispielsweise im Bereich der Bildverarbeitung keine reellwertigen, sondern diskrete Funktionen vorliegen, muss diese Definition erweitert werden.

Definition 2.5.3 (Diskrete Faltung) *Seien $f, g : \mathbb{D} \to \mathbb{C}$ zwei diskrete Funktionen mit Definitionsbereich $\mathbb{D} \subseteq \mathbb{Z}$. Dann ist die Faltung $(f * g)$ definiert durch*

$$(f * g)(x) := \sum_{i \in \mathbb{D}} f(i)g(x - i)$$

Die eindimensionale Faltung diskreter Funktionen ermöglicht nun beispielsweise die Gewichtung einzelner Pixel eines Bildes durch die gewichtete Summation des aktuell betrachteten Pixels und seiner Nachbarn. Auf diese Weise kann ein einfacher Glättungsfilter implementiert werden. Der große Nachteil der eindimensionalen Faltung ist ihre Beschränkung auf eine Achse. Es werden nur die Nachbarn in x- oder y-Richtung einbezogen.

Definition 2.5.4 (Faltungsmatrix) *Seien I, K diskrete, zweidimensionale Funktionen. Die Faltung von I und K ist definiert durch*

$$I'(x, y) := \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} I(x + i - \frac{n}{2} + 1, y + j - \frac{m}{2} + 1)K(i, j) \quad .$$

K heißt dann Faltungsmatrix mit der Dimension $n \times m$.

Eine solche Faltungsmatrix ermöglicht nun eine Gewichtung des aktuell betrachteten Bildpunktes $I(x, y)$ in Abhängigkeit von seinen Nachbarn in x- und y-Richtung.

Ein Beispiel für einen solchen einfachen Glättungsfilter ist der folgende:

$$\begin{pmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{pmatrix}$$

Dennoch arbeiten viele dieser in der Bildverarbeitung als *Kernel* bezeichneten Filter nur in einer Richtung. Daher basieren viele Algorithmen auf zwei oder mehr Kerneln. Ein Beispiel für so einen Algorithmus ist die Nutzung des nachfolgenden Sobel-Operators[1]. Er implementiert eine einfache Kantendetektion und arbeitet mit einem Kernel S_x für die x- und einen Kernel S_y für die y-Richtung.

$$S_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

$$S_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

2.6 Kalibrierung

Der Mensch ist in der Lage, anhand eines Bildes eine räumliche Vorstellung von der betrachteten Szene zu bekommen. Das Erkennen von Objekten als Ganzes macht es einfach zu entscheiden, welches Objekt im Vordergrund und welches im Hintergrund liegt. Das Wissen um die Eigenschaften der abgebildeten Objekte erleichtert zudem die Abschätzung des Abstands zu anderen im Vorder- oder Hintergrund liegenden Objekten. Die Verarbeitung von Fluchtpunkten, Schatten und Objektüberdeckung sowie die Zuordnung von Eigenschaften zu Objekten geschehen vollkommen unbewusst.

[1] Der Sobel-Operator ist eine Faltungsmatrix, die in Algorithmen zur Kantendetektion verwendet wird.

Im Computer hingegegen ist ein Bild nichts anderes als eine Matrix von Bildpunkten mit zugehörigen Farbwerten. Natürlich sind auch hier weitere implizite Informationen enthalten, jedoch ist es sehr schwer, diese nutzbar zu machen. Algorithmen zur Kantendetektion und die Bestimmung von Fluchtpunkten ermöglichen zwar die teilweise Rekonstruktion von Tiefeninformationen, dennoch ist es kaum möglich, auf die tatsächliche Größe von Objekten oder den Abstand von im Vorder- und Hintergrund liegenden Objekten zu schließen, da Informationen über die Objekte fehlen.

Es ist daher eine Kalibrierung des Systems erforderlich, die es ermöglicht, auch diese Informationen zu extrahieren. So, wie der Mensch eine im Bild gezeigte Person, ein Fahrzeug oder ein anderes Objekt, dessen Beschaffenheit und Größe er erkennt, dazu nutzen kann, Tiefeninformationen zu verarbeiten, muss auch dem Computer ein solches Objekt, dessen Eigenschaften ihm bekannt sind, als Referenz zur Hand gegeben werden.

Es gibt verschiedene Verfahren zur Kalibrierung eines Kamerasystems, dennoch ist die Vorgehensweise immer die gleiche. Anhand bekannter Informationen über die Szene werden Punkten der zweidimensionalen Projektion ihre Ursprungskoordinaten in der dreidimensionalen Szene zugeordnet. Da die Bestimmung der für die Kalibrierung relevanten Kontextinformationen der Projektion der realen Welt zu komplex ist, wird zumeist eine stark vereinfachte Umgebung zur Kalibrierung verwendet.

Es ist dennoch zu beachten, dass es Kontextinformationen gibt, die invariant unter der Position der Kamera sind, und solche, die sich in Abhängigkeit von der Kameraposition verändern. Man spricht in diesem Zusammenhang von intrinsischen und extrinsischen Kameraparametern. Als Beispiel für intrinsische Parameter sei hier die Brennweite der Kamera, für extrinsische Parameter die Entfernung der Kamera zu einem Punkt im Raum genannt.

Definition 2.6.1 (Intrinsische Kameramatrix) *Seien $x \in \mathbb{R}^3$ und $y \in \mathbb{R}^2$ Punkte der Bildebene, wobei x und y in homogener Koordinatenform vorliegen. Sei des Weiteren C eine 3×4 Matrix, so dass*

$$y \sim Cx$$

gilt. Dann heißt C intrinsische Kameramatrix. Hierbei besagt $\sim$, dass die rechte und linke Seite dieser Gleichung bis auf einen skalaren Faktor gleich sind.

Die Abbildung eines dreidimensionalen Punktes $P = (x_1, x_2, x_3)^T$ der Bildebene auf einen Punkt $P' = (y_1, y_2)^T$ des Bildes ist gegeben durch

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \frac{f}{x_3} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad ,$$

wobei der Faktor $f > 0$ die Brennweite der verwendeten Kamera ist und $x_3 > 0$ angenommen wird. Eine Umformung in homogene Koordinatenschreibweise ergibt:

$$\begin{pmatrix} y_1 \\ y_2 \\ 1 \end{pmatrix} = \frac{f}{x_3} \begin{pmatrix} x_1 \\ x_2 \\ \frac{x_3}{f} \end{pmatrix} \sim \begin{pmatrix} x_1 \\ x_2 \\ \frac{x_3}{f} \end{pmatrix}$$

Die weitere Umformung der dreidimensionalen Koordinaten liefert

$$\begin{pmatrix} y_1 \\ y_2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix}$$

bzw. $y \sim Cx$, wobei C die intrinsische Kameramatrix ist. Da C projektiv ist, gilt außerdem

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{pmatrix} \sim \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Dies ist eine vereinfachte Form der intrinsischen Kameramatrix, da angenommen wird,

dass das Zentrum der Bildebene auf der y-Achse liegt und der Kamerasensor quadratisch ist. Um diese Parameter zu berücksichtigen, lässt sich die obige Formel für Brennweiten f_x und f_y und einen Bildmittelpunkt (c_x, c_y) schreiben als

$$\lambda \begin{pmatrix} y_1 \\ y_2 \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix} \quad .$$

Die hier verwendete Rotations-Translations-Matrix

$$[R|t] = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix}$$

wird dazu verwendet extrinsische Parameter in die Berechnungen einzubeziehen. Es kann so beispielsweise die Bewegung der Kamera um ein Objekt einbezogen werden. Außerdem können durch sie die korrekten Weltkoordinaten aus den Kamerakoordinaten bestimmt werden.

Diese Form der Darstellung ist besonders wichtig, da sie von OpenCV eingesetzt wird.

2.7 Differenzielle Kantendetektion

Die Rekonstruktion der Punkte der dreidimensionalen Szene anhand der Punkte der Bildebene erfordert eine Detektion der durch den Laserstrahl erzeugten Kante und der darauf liegenden Punkte.

Da das aktuell betrachtete Bild im Regelfall mehr als nur eine Kante enthält, schlägt die Kantendetektion mit Hilfe der üblichen Algorithmen, die den Sobel- oder Prewitt-Operator nutzen, fehl. Algorithmen, die solche Faltungsmatrizen verwenden, würden auch Objektkanten detektieren und somit die Rekonstruktion des Objekts verfälschen.

Die differenzielle Kantendetektion vergleicht zu diesem Zweck zwei in der selben Objektorientierung aufgenommene Bilder, wobei eines durch den Laser beleuchtet und das andere unbeleuchtet ist. Die Differenz dieser Bilder liefert ein neues Bild, auf dem nun die eigentliche Kantendetektion ausgeführt werden kann.

Definition 2.7.1 (Farbhelligkeit) *Sei $P_I = (x, y)^T$ ein Punkt der Bildebene und $\delta(P_I) = (r, g, b)$ der dazugehörige RGB-Farbvektor. Die Helligkeit $B(P_I)$ von P_I ist definiert durch*

$$B(P_I) := \omega_r r + \omega_g g + \omega_b b \quad ,$$

wobei $\omega_r, \omega_g, \omega_b \in (0, 1)$.

Zur Bestimmung der Helligkeit eines Pixels müssen die Gewichtungsfaktoren ω_r, ω_g und ω_b geeignet gewählt werden. In der Praxis hat sich gezeigt, dass die folgende Wahl der Faktoren besonders gute Resultate liefert:

$$\begin{aligned} \omega_r &= 0.299 \\ \omega_b &= 0.114 \\ \omega_g &= 1 - \omega_r - \omega_b = 0.587 \end{aligned}$$

Auf Basis der Helligkeit eines Bildpunktes kann nun die Bilddifferenz bestimmt werden.

Definition 2.7.2 (Differenzielle Kantendetektion) *Sei $\gamma^+_{P_I}$ die Helligkeit eines Punktes der beleuchteten und γ_{P_I} der unbeleuchteten Szene. Dann bezeichnet*

$$L(P_I) := \gamma^+_{P_I} - \gamma_{P_I}$$

die Lumen-Differenz. Der Punkt P_I ist Teil der gesuchten Kante, falls

$$\sqrt{(L(P_I) * S_x)^2 + (L(P_I) * S_y)^2} > \theta$$

und

$$\sqrt{(L(P_I') * S_x)^2 + (L(P_I') * S_y)^2} \leq \theta$$

für einen beliebigen festen Grenzwert θ, den linken Nachbarn $P_I' = (x-1, y)$ und Sobel-Operatoren S_x und S_y in x- bzw y-Richtung gilt.

Die durch die differenzielle Kantendetektion akzeptierten Punkte liegen am linken Rand der durch den Laser projizierten Kante. Sie können nun durch Multiplikation mit der intrinsischen Kameramatrix C und der Rotations-Translations-Matrix $[R|t]$ auf ihre dreidimensionalen Ursprungskoordinaten abgebildet werden.

2.8 Rekonstruktion der Tiefeninformationen

Die Rekonstruktion der dreidimensionalen Punkte der Bildebene aus den zweidimensionalen Punkten des Bildes stellt noch keine Rekonstruktion der tatsächlichen dreidimensionalen Punkte dar. Zur Bestimmung der Tiefeninformationen ist ein Schnitt der vom Ursprung aus durch den Punkt P_B der Bildebene verlaufenden Geraden mit der Laserebene notwendig. Der so berechnete Schnittpunkt ist der Urpunkt P von P_B.

Um die genaue Lage und Orientierung der Laserebene E_L bestimmen zu können, muss zuerst die Objektebene E_O, auf der das zu erfassende Objekt steht, bestimmt werden. Zu diesem Zweck ist eine weitere Kalibrierung notwendig. Hierzu dient ein auf der Objektebene liegendes Quadrat. Die als zweidimensionale Vektoren vorliegenden Eckpunkte $p = (x, y)$, müssen nun auf die Bildebene projiziert werden. Dies geschieht durch Multiplikation mit der Inversen der intrinsischen Kameramatrix C und der Brennweite f der Kamera.

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = f * C^{-1} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Es ist klar, dass die Seiten des Quadrates paarweise rechtwinklig aufeinander stehen bzw. sich die gegenüberliegenden Seiten parallel zueinander befinden. Außerdem ist die Seitenlänge l des Quadrates bekannt. Um die tatsächlichen dreidimensionalen Koordinaten der

Eckpunkte $P_{B_1} \dots P_{B_4}$ zu bestimmen, müssen sie so auf der Geraden durch sich und den Ursprung verschoben werden, dass sowohl die Seitenlängen als auch die Orientierung der Seiten zueinander dem realen Abbild entsprechen. Dieses Problem kann durch das folgende nichtlineare, mehrdimensionale Gleichungssystem beschrieben werden:

$$\begin{aligned}
(s_1 P_{B_1} - s_2 P_{B_2})^2 - l^2 + (s_1 P_{B_1} - s_2 P_{B_2} - s_4 P_{B_4} + s_3 P_{B_3}) &= 0 \\
(s_2 P_{B_2} - s_3 P_{B_3})^2 - l^2 + (s_1 P_{B_1} - s_2 P_{B_2} - s_4 P_{B_4} + s_3 P_{B_3}) &= 0 \\
(s_3 P_{B_3} - s_4 P_{B_4})^2 - l^2 + (s_4 P_{B_4} - s_1 P_{B_1} - s_3 P_{B_3} + s_2 P_{B_2}) &= 0 \\
(s_4 P_{B_4} - s_1 P_{B_1})^2 - l^2 + (s_4 P_{B_4} - s_1 P_{B_1} - s_3 P_{B_3} + s_2 P_{B_2}) &= 0
\end{aligned}$$

Dieses Gleichungssystem kann nun beispielsweise mit dem Newton-Verfahren [Lub10] gelöst werden. Dieses Fixpunktverfahren berechnet iterativ eine Approximation des Ergebnisvektors. Die Iterationsvorschrift des Newton-Verfahrens lautet

$$s_{n+1} := s_n - (J(s_n))^{-1} f(s_n) \quad ,$$

wobei

$$J(x) := \frac{\partial f_i}{\partial s_j} = \begin{pmatrix} \frac{\partial f_1}{\partial s_1} & \cdots & \frac{\partial f_n}{\partial s_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial s_1} & \cdots & \frac{\partial f_n}{\partial s_n} \end{pmatrix}$$

die Jacobi-Matrix ist. Der durch Anwendung dieses Verfahrens erhaltene Ergebnisvektor

$$s_{n+1} = \begin{pmatrix} s1 \\ s2 \\ s3 \\ s4 \end{pmatrix}$$

enthält die Faktoren zur Skalierung der Eckpunkte $P_{B_1} \dots P_{B_4}$. Die durch die Skalierung erhaltenen Raumpunkte $P_1 \dots P_4$ ermöglichen nun die Aufstellung der Ebenengleichung der Objektebene

$$E_O := ((P_1 - P_2) \times (P_3 - P_2))(x - P_2) = 0 \quad .$$

Da durch die Justierung des Lasers bekannt ist, dass die Laserebene E_L senkrecht auf der Objektebene steht und durch die Punkte P_2 und P_4 verläuft, kann nun auch ihre Ebenengleichung aufgestellt werden.

$$E_L := n_{E_O} x(P_4 - P_2)(x - P_2) = 0 \quad ,$$

wobei

$$n_{E_O} = (P_1 - P_2) \times (P_3 - P_2)$$

der Normalenvektor der Objektebene ist.

Es ist nun möglich den Schnittpunkt der Geraden durch jeden Punkt P_B der Bildebene mit der Laserebene E_L zu berechnen um den dazugehörigen Urpunkt P zu bestimmen.

Der so erhaltene Punkt P liegt nun natürlich immer in der Laserebene, was nur bedingt richtig ist. Die Rotation des Objekts unter der Kamera wurde bisher noch nicht einbezogen. Um den rotierten Punkt P_R zu erhalten muss der Punkt der Laserebene durch eine Translation so verschoben werden, dass er um eine der drei Achsen rotiert werden kann, und dann wieder zurückverschoben werden.

Als Translationsvektor wird hier der Mittelpunkt M_Q des zur Berechnung der Laserebene verwendeten Quadrats gewählt.

$$M_Q = \frac{1}{2}(P_1 + P_3)$$

Die Translation des Punktes P ist dann gegeben durch

$$P_T = P - M_Q \quad .$$

Der so erhaltene Punkt P_T soll nun um die Normale der Objektebene n_{E_O} rotiert werden.

Die folgende Rotationsmatrix R_{xyz} setzt sich aus einer Koordinatensystemtransformation T_{xyz}, der inversen Transformation T_{xyz}^{-1} und der Rotationsmatrix R_x für eine Rotation um die x-Achse zusammen.

$$R_{xyz} = T_{xyz} R_x T_{xyz}^{-1} = \begin{pmatrix} tx^2 + c & txy - sz & txz + sy \\ txy + sz & ty^2 + c & tyz - sx \\ txz - sy & txy + sx & tz^2 + c \end{pmatrix}$$

Hierbei sind $s = \sin\alpha$, $c = \cos\alpha$ und $t = 1 - \cos\alpha$ sowie x, y und z die Elemente des normierten Normalenvektors n_{E_O} der Objektebene.

Es ist zu beachten, dass die Rotationsmatrix R_{xyz} eine Rotation im Weltkoordinatensystem durchführt. Um sie auf die vorliegenden Kamerakoordinaten anwenden zu können, ist eine Umrechnung von Kamerakoordinaten in Weltkoordinaten notwendig. Diese Umrechnung kann durch eine weitere Rotations-Translations-Matrix

$$[R|t] = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix}$$

beschrieben werden, die extrinsische Parameter in die Berechnungen einbezieht. Eine Überführung von Kamera- in Weltkoordinaten ist dann gegeben durch

$$P_W = R^{-1} P + t \quad .$$

Die Berechnung des tatsächlichen Objektpunktes P_R ist nun durch

$$P_R = R(R_{xyz}(R^{-1}(P - M_Q) + t))$$

gegeben.

3 Der Algorithmus

Die vorangegangenen Definitionen erlauben nun das Aufstellen des zur Rekonstruktion benötigten Algorithmus.

Algorithmus 3.1 (3-D-Rekonstruktion)

Vorbedingung:
Gegebene Anzahl an zu rotierenden Schritten A.

Nachbedingung:
Rückgabe des erzeugten Gitters.

Initialisierung:

- *Initialisiere Rotationszähler $C_R = 0$.*
- *Initialisiere für jeden Rotationsschritt einen Stapel.*

Großschritt 1. – Kamerakalibrierung
Aufstellen der intrinsischen Kameramatrix C.

Großschritt 2. – Rotations-Translations-Matrix
Aufstellen der Rotations-Translations-Matrix $[R|t]$.

Großschritt 3. – 3-D-Punktrekonstruktion

Berechne Lumen-Differenz $L(P_I)$ für alle $P_I = (x, y)$ der Bildebene.
Bestimme Laserebene E_L.
Falls die Bedingungen

- $\sqrt{(L(P_I) * S_x)^2 + (L(P_I) * S_y)^2} > \theta$
- $\sqrt{(L(P_I') * S_x)^2 + (L(P_I') * S_y)^2} \leq \theta$

erfüllt sind, führe nachfolgende Schritte aus:

- $\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = f_x C^{-1} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$
- *Berechne Schnittpunkt S der Geraden durch $(X, Y, Z)^T$ und den Ursprung O_c mit der Laserebene.*
- *Berechne $P_R = R(R_{xyz}(R^{-1}(P - M_Q) + t))$.*
- *Lege P_R auf den aktuellen Stapel.*

Großschritt 4. – Iteration

Erhöhe den Rotationszähler C_R.
Führe

- *Rotiere um einen Schritt.*
- *Gehe zu Großschritt 2.*

aus, falls $C_R < A$.

Großschritt 5. – Gittergenerierung

Erzeuge das Gitter anhand der Stapel.

4 Implementierung

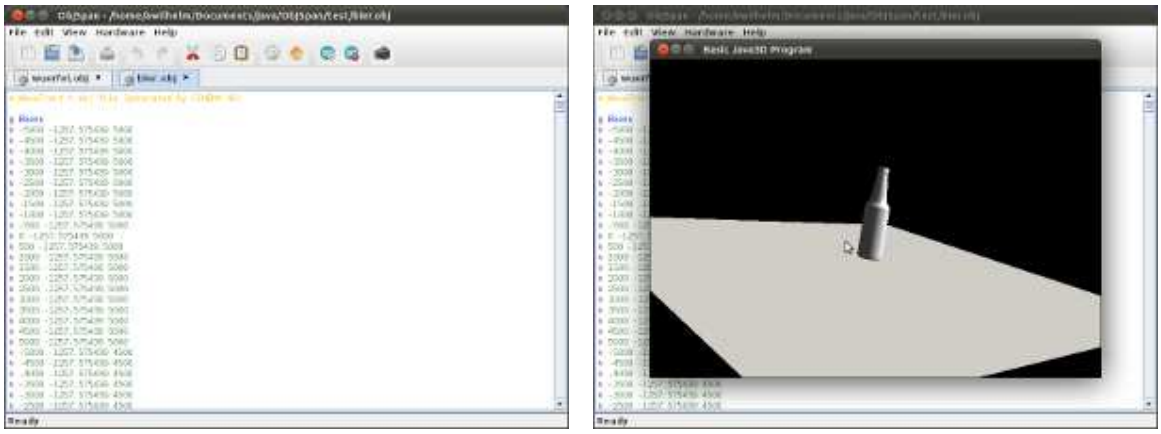
Die Implementierung der Software zur Steuerung des Scanvorgangs sowie der Berechnung der dreidimensionalen Modelle basiert auf Java. Die OpenCV-Portierung JavaCV übernimmt dabei nicht nur die Steuerung der Kamera, sondern auch die Kantendetektion und die Kalibrierung der Kamera. Die Visualisierung der resultierenden 3-D-Modelle wurde mit Java3D vorgenommen. Die Ansteuerung der übrigen Hardware übernimmt die Open-Source-Mikrocontrollerplattform Arduino, die mit Hilfe der RXTX-Bibliothek angesprochen wird.

Die entstandene Software *OpenSpan* (siehe Abbildung 4.1) ermöglicht es so, dreidimensionale Gitter auf Basis realer Objekte auf Knopfdruck zu generieren. Die erzeugten Gitter werden im Wavefront-Format (.OBJ) abgelegt und können direkt aus der Software heraus angezeigt und editiert werden. Außerdem bietet die Software die Möglichkeit, die angeschlossene Hardware (Kamera und Arduino) direkt zu testen.

4.1 Arduino und die RXTX-Bibliothek

Die zur Steuerung des Drehtischs und des Lasers verwendete Open-Source-Mikrocontrollerplattform Arduino wird aufgrund ihrer einfach zu nutzenden und flexibel einsetzbaren Hard- und Software von vielen Entwicklern zur Fertigung von Prototypen verwendet.

Die Flexibilität, das hervorragende Preis-Leistungs-Verhältnis und die einfache Handhabung qualifizierten diese Plattform auch für den Einsatz in dieser Arbeit. Eine Ansteuerung des Schrittmotors und die Schaltung des Lasers können mit nur wenigen Handgriffen und nur einigen Zeilen Quellcode umgesetzt werden.



(a) Editor für das Wavefront-Format (b) Vorschau des erzeugten Gitters

Abbildung 4.1: Die ObjSpan-Software

Das Codebeispiel 4.1 zeigt die vollständige Ansteuerung von Laser und Motor. Die Funktion *setup()* wird nur einmal beim Starten des Arduinos aufgerufen und übernimmt die Initialisierung der Ein- und Ausgabeports sowie der seriellen Verbindung. Die nachfolgende *loop()*-Funktion wird danach wiederholt ausgeführt. Sie überprüft mit Hilfe des Aufrufs *Serial.available()*, ob Daten im Puffer der seriellen Schnittstelle zur Verfügung stehen. Ist dies der Fall, wird der anliegende Befehl gelesen und von der Funktion *execute()* verarbeitet. An dieser Stelle wird die eigentliche Steuerung von Laser und Motor vorgenommen. Durch das Setzen der Steuerleitung LASER_PIN auf HIGH bzw. LOW kann der Laser aus- bzw. angeschaltet werden. Das Setzen der Steuerleitungen MOTOR_DIR_PIN und MOTOR_STEP_PIN, das in der Subroutine *rotateDeg* vorgenommen wird, legt die Drehrichtung des Motors sowie die Anzahl an Mikroschritten fest, um die gedreht werden soll. Die Subroutine kümmert sich dabei auch um die vom Motortyp abhängige Umrechnung von Grad in Mikroschritte.

```
// LED connected to pin 12.
#define LASER_PIN 12
#define MOTOR_DIR_PIN 2
#define MOTOR_STEP_PIN 3

// Commands from the java app.
#define COM_LIGHT 15
#define COM_MOTOR 16

boolean lState = true;

void setup() {
  // Setup the IO pins.
  pinMode(LASER_PIN, OUTPUT);
  pinMode(MOTOR_DIR_PIN, OUTPUT);
  pinMode(MOTOR_STEP_PIN, OUTPUT);

  // Initialize the laser pin.
  digitalWrite(LASER_PIN, HIGH);

  // Start up serial port.
  Serial.begin(9200);
}

void loop() {
  if (Serial.available()) {
    int cmd = Serial.read();
    Serial.flush();
    execute(cmd);
  }
}

void execute(int cmd) {
  switch(cmd) {
  case COM_LIGHT:
    digitalWrite(LASER_PIN, lState ? LOW : HIGH);
    lState = !lState;
    break;
  case COM_MOTOR:
```

```
40      rotateDeg(36, 1);
        break;
    }
  }

45 // Rotates a specific number of degrees (1.8 per step).
  // Negative steps used for reverse movement.
  // Speed is any number between .01 and 1.
  void rotateDeg(float deg, float speed) {
    int dir = (deg > 0) ? HIGH : LOW;
50  digitalWrite(MOTOR_DIR_PIN, dir);

    int steps = abs(deg/1.8) * 8;
    float usDelay = (1/speed) * 70;

55  for(int i = 0; i < steps; i++) {
      digitalWrite(MOTOR_STEP_PIN, HIGH);
      delayMicroseconds(usDelay);

      digitalWrite(MOTOR_STEP_PIN, LOW);
60    delayMicroseconds(usDelay);
    }
  }
```

Listing 4.1: Arduino Laser- und Motorsteuerung

Da jedoch die Leistungsfähigkeit und der zur Verfügung stehende Speicher eines solchen Mikrocontrollers stark beschränkt sind, reicht ein Arduino zur Erfassung und Verarbeitung der Bilddaten nicht aus. Es ist daher notwendig, die Steuerung der Kamera, das Speichern der Bilddaten und die Rekonstruktion der dreidimensionalen Szeneninformationen mit Hilfe eines Computers vorzunehmen (siehe Abschnitt 4.2 und 4.3). Die Kommunikation des auf dem Computer laufenden Programms mit dem Arduino zur sequenziellen Steuerung der verschiedenen Hardwarekomponenten ist daher von großer Bedeutung.

Die für den Einsatz unter Java bereitgestellte Bibliothek *librxtx-java* ermöglicht eine einfache und schnelle Kommunikation über die serielle Schnittstelle (USB).

Die Klasse *SerialCom* aus Codebeispiel 4.2 kapselt die Kommunikation über die RXTX-eigene Klasse *SerialPort*. Zudem verwaltet sie die Ein- und Ausgabeports und implementiert einen *EventListener*, der die an der seriellen Schnittstelle anliegenden Daten liest und auf der Kommandozeile ausgibt.

```
public class SerialCom implements SerialPortEventListener {
  // The serial and in/output port objects.
  private SerialPort serialPort = null;
  private InputStream input = null;
  private OutputStream output = null;

  public void connect(CommPortIdentifier com) throws Exception {
    // Open serial port.
    serialPort = (SerialPort) port.open(this.getClass().getName(), 2000);
    serialPort.setSerialPortParams(9600, SerialPort.DATABITS_8,
    SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
    Thread.sleep(2000);

    // Initialize input and output stream.
    input = serialPort.getInputStream();
    output = serialPort.getOutputStream();

    // Finally add the event listener.
    serialPort.addEventListener(this);
    serialPort.notifyOnDataAvailable(true);
  }

  public void disconnect() throws Exception {
    serialPort.removeEventListener();
    serialPort.close();
    input.close();
    output.close();
  }

  public void serialEvent(SerialPortEvent evt) throws Exception {
    if (evt.getEventType() == SerialPortEvent.DATA_AVAILABLE) {
      int available = input.available();
```

```
        byte chunk[] = new byte[available];
        input.read(chunk, 0, available);

        // Display results.
        System.out.print(new String(chunk));
      }
    }

    public void writeData(int cmd) throws Exception {
      output.write(cmd);
      output.flush();
    }
}
```

Listing 4.2: Serielle Kommunikation mit RXTX

4.2 OpenCV und JavaCV

Der hohe Abstraktionsgrad von Java erschwert die Kommunikation mit der Hardwareebene. Zur Steuerung einer Kamera ist es notwendig, auf native Systembibliotheken wie OpenCV [Bra00] zurückzugreifen.

Da Bibliotheken wie OpenCV natürlich nicht in Java, sondern in C/C++ oder anderen systemnahen Sprachen geschrieben wurden, ist es notwendig, sogenannte Wrapper für solche Bibliothek zu verwenden. Diese kapseln die Funktionalität der Systembibliothek und ermöglichen einen Zugriff auf Systemroutinen aus einer anderen Programmiersprache heraus. Ein solcher Wrapper für OpenCV ist JavaCV. Er ermöglicht es, alle Routinen zur Kamerasteuerung und Bildverarbeitung, die OpenCV seinem Anwender bereitstellt, auch in Java zu nutzen.

Die Ansteuerung der Kamera sowie die Anzeige des resultierenden Bildes gestaltet sich nun recht einfach, wie Listing 4.3 zeigt.

```
class CameraCapture {
   public static void main(String args[]) throws IOException {
      // Initialize the grabber object, ...
      OpenCVFrameGrabber grabber = new OpenCVFrameGrabber(0);

      // ... start the grabber ...
      grabber.start();

      // ... and grab an image from the camera.
      IplImage image = grabber.grab();

      // Create the window frame to display the image.
      CanvasFrame canvasFrame = new CanvasFrame("Camera Test");
      canvasFrame.setCanvasSize(image.width(), image.height());

      // Show the camera capture.
      canvasFrame.showImage(image);
   }
}
```

Listing 4.3: Kamerasteuerung mit JavaCV

Ähnlich einfach gestaltet sich auch die Kalibrierung der Kamera (Listing 4.4).

```
public class AxisChessboard {
  public void calibrate(int boardW, int boardH, int numBoards, int
      boardDt)
      throws Exception {

    // Initialize the camera, the board size and number of points
    // to be detected, the video and picture frames, grab the
    // first image and generate the grayscale.
    ...

    while (successes < numBoards) {
      // Skip every boardDt frames to allow user to move chessboard.
      if ((frame++ % boardDt) == 0) {
        // Find the chessboard corners, ...
        int found = cvFindChessboardCorners(image, boardSize, corners,
```

```
15              cornerCount, CV_CALIB_CB_FAST_CHECK);

            // ... get their Subpixel accuracy ...
            cvCvtColor(image, grayImage, CV_BGR2GRAY);
            cvFindCornerSubPix(grayImage, corners, cornerCount[0],
20              cvSize(11, 11), cvSize(-1, -1), cvTermCriteria(
                    CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1));

            // ... and draw the chess board.
            cvDrawChessboardCorners(image, boardSize, corners,
25              cornerCount[0], found);

            ...
          }

30        // Get the next image.
          image = grabber.grab();
          rawFrame.showImage(image);
        }

35      ...

        // At this point we have all of the chessboard corners we need.
        // Initialize the intrinsic matrix and calibrate the camera.
        cvCalibrateCamera2(objectPoints2, imagePoints2, pointCounts2,
40          cvGetSize(image), intrMatrix, distCoeffs, null,
            null, 0
        );

        // Save the intrinsic camera matrix and distortion coefficients.
45      cvSave("intrinsics.xml", intrMatrix);
        cvSave("distortion.xml", distCoeffs);
        grabber.stop();
      }
    }
```

Listing 4.4: JavaCV Kamerakalibrierung

Die Klasse *AxisChessboard* ist angelehnt an ein Beispiel aus *Computer Vision with the OpenCV Library* [BK08]. Zur Kalibrierung der Kamera bzw. zur Aufstellung der intrinsischen Kameramatrix wird ein beliebiges Schachfeld (siehe Abbildung 4.2) verwendet. Um das Schachfeld richtig positionieren zu können, wird das aktuelle Bild der Kamera ausgegeben. Alle paar Sekunden wird das aktuelle Bild mit Hilfe der Methode *cvFindChessboardCorners* [SYXH02] nach den inneren Ecken eines Schachfeldes durchsucht. Stimmt die Anzahl an gefundenen Ecken mit der im Aufruf der Methode *calibrate* implizit übergebenen Anzahl überein, werden die gesammelten Daten gespeichert. Anhand dieser Daten kann die Funktion *cvCalibrateCamera2* die intrinsische Kameramatrix und die Verzerrungskoeffizienten bestimmen. Die beiden resultierenden Matrizen werden in den beiden Dateien *intrinsics.xml* und *distortion.xml* abgelegt und können zu einem späteren Zeitpunkt wieder geladen werden.

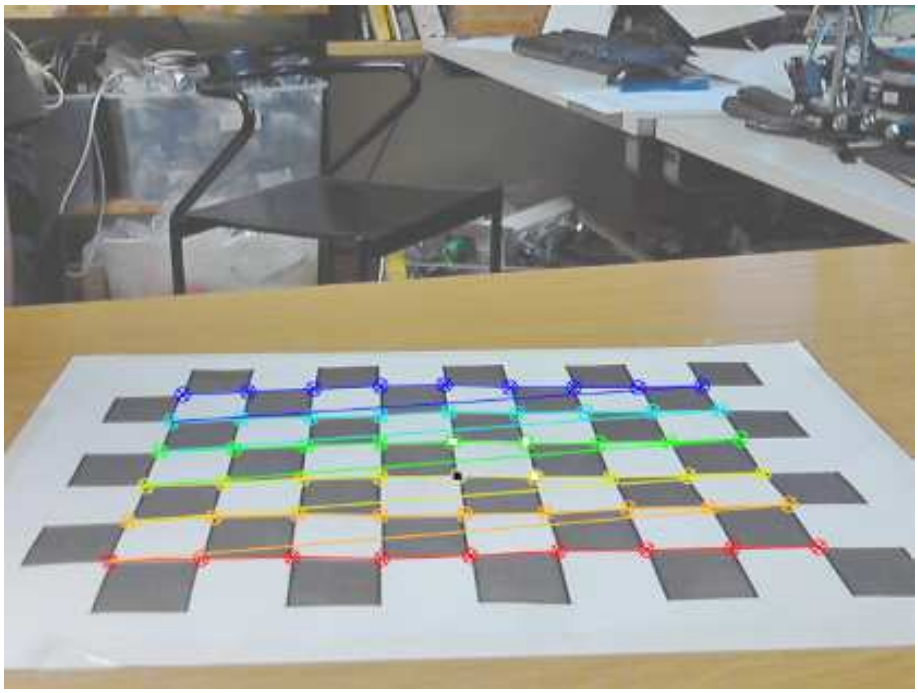
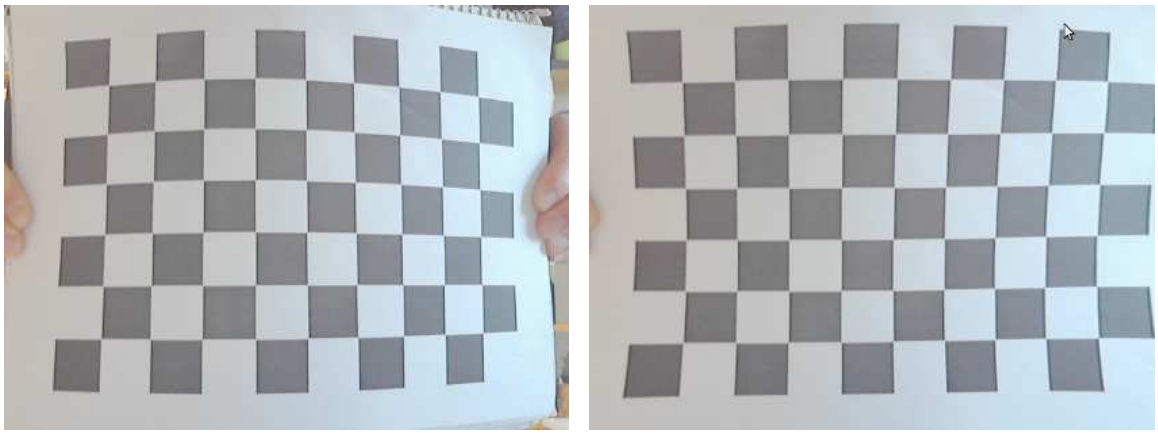


Abbildung 4.2: Detektion des Schachbretts



(a) Aufnahme (b) Korrektur

Abbildung 4.3: Korrektur verzerrter Bilder

Zur Überprüfung werden das zur Kalibrierung verwendete Bild angezeigt und ein weiteres Fenster mit dem korrigierten Bild geöffnet. Der zur Korrektur verwendete Filter wird durch die Funktion *cvInitUndistortMap* aus der intrinsischen Kameramatrix und den Verzerrungskoeffizienten erzeugt. Die Anwendung des Filters auf das Ursprungsbild übernimmt die Funktion *cvRemap*. Das Ergebnis einer solchen Korrektur ist in Abbildung 4.3 (a) zu sehen. Zum direkten Vergleich zeigt Abbildung 4.3 (b) das Ursprungsbild.

4.3 Java3D

Die Visualisierung der erzeugten Modelle basiert auf der Java3D-API von Java, welche wahlweise auf OpenGL oder Direct3D aufbaut. Der große Vorteil dieser API gegenüber beispielsweise JOGL (Java OpenGL) liegt in der Möglichkeit, Wavefront-Dateien (*.obj) direkt zu laden und anzuzeigen. Da dieses Format durch seine Einfachheit besticht, werden alle erzeugten Modelle darin gespeichert. Das direkte Laden und Anzeigen der Dateien erspart das sonst sehr aufwendige Umwandeln der für die Berechnungen verwendeten OpenCV-Vektoren in Java3D-Vektoren. Außerdem ermöglicht die API ein schnelles Umschalten zwischen der Objekt-, Gitter- und Punktwolkenansicht.

Das Beispiel aus Listing 4.5 erzeugt ein Fenster, in dem das Objekt oder die Objekte aus der *.obj-Datei angezeigt, das Koordinatensystem gedreht und an das Objekt heran- und weggezoomt werden kann.

```
public class ObjView extends JFrame {
    private BranchGroup root;

    public ObjView(Reader reader) throws IOException {
        setTitle("Basic Java3D Program");
        setSize(800, 600);
        Canvas3D canvas = new Canvas3D(SimpleUniverse.
            getPreferredConfiguration());
        getContentPane().add(canvas, BorderLayout.CENTER);
        SimpleUniverse universe = conigureUniverse(canvas, reader);
        universe.addBranchGraph(createSceneGraph());
    }

    private SimpleUniverse conigureUniverse(Canvas3D canvas, Reader reader
        ) throws IOException {
        // Setup the universe, ...
        SimpleUniverse universe = new SimpleUniverse(canvas);

        // ... add the model from file ...
        ObjectFile file = new ObjectFile(ObjectFile.RESIZE);
        root = file.load(reader).getSceneGroup();

        // ... and add some lights.
        AmbientLight ambientLight = new AmbientLight(new Color3f(Color.
            WHITE));
        ambientLight.setInfluencingBounds(new BoundingSphere());
        root.addChild(ambientLight);
        root.compile();
        return universe;
    }

    public BranchGroup createSceneGraph() {
```

```
30          BranchGroup objRoot = new BranchGroup();

            TransformGroup listenerGroup = new TransformGroup();
            listenerGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
            listenerGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
35          objRoot.addChild(listenerGroup);

            MouseRotate rotate = new MouseRotate(listenerGroup);
            rotate.setSchedulingBounds(new BoundingSphere(new Point3d(), 100));

40          MouseWheelZoom zoom = new MouseWheelZoom(listenerGroup);
            zoom.setSchedulingBounds(new BoundingSphere(new Point3d(), 100));

            listenerGroup.addChild(rotate);
            listenerGroup.addChild(zoom);
45          listenerGroup.addChild(root);

            return objRoot;
        }

50      public static void main(String args[]) throws IOException {
            new ObjView(new FileReader("wuerfel.obj")).setVisible(true);
        }
    }
```

*Listing 4.5: Anzeige von *.obj-Dateien mit Java3D*

4.4 Differenzielle Kantendetektion

Zur differenziellen Kantendetektion werden zwei Bilder von der Webcam mit Hilfe von OpenCV aufgenommen und wird in Abhängigkeit von der Helligkeit der einzelnen Pixel ihre Differenz gebildet (siehe Abschnitt 2.7). Der zu diesem Zweck verwendete Sobel-Operator wurde in die beiden Klassen *SobelXFilter* und *SobelYFilter* für die Filterung entlang der x- bzw. y-Achse aufgeteilt. Beide Filter leiten sich aus einer generischen Klasse *KernelFilter* ab, die die Konvolution eines Bildes mit dem gegebenen Kernel implementiert (siehe Abschnitt 2.5). Ein Ergebnis der Ausführung der differenziellen Kantendetektion ist in Abbildung 4.4 (b) zu sehen. Das dazugehörige Ursprungsbild ist Abbildung 4.4 (a).

(a) Objektansicht (b) Kantenansicht

Abbildung 4.4: Differenzielle Kantendetektion

5 Ergebnisse

5.1 Robustheit

Dieses Verfahren ist äußerst robust. Die differenzielle Kantendetektion arbeitet sehr zuverlässig. Ausreißer sind auch bei unterschiedlichen Lichtverhältnissen recht selten.

Die Abbildung zwischen Punkten des zweidimensionalen und dreidimensionalen Raumes stellt keine Probleme dar. Die dafür benötigten Grundlagen sind im Vegleich zu passiven Verfahren verständlich und verhältnismäßig einfach zu implementieren. Das resultierende Gitter stimmt sowohl in den relativen Abständen der Objekte zueinander als auch der Proportion der Objekte. Eine Skalierung der Szene auf die gewünschte Größe ist somit ohne Weiteres möglich.

5.2 Schwächen der Hardware

Im Verlauf dieser Arbeit haben sich einige Schwächen in Bezug auf die Hardware gezeigt. Dies betrifft zum einen die Kommunikation zwischen PC und Arduino und zum anderen die Effizienz der Kantendetektion.

Die Schwierigkeiten in der Kommunikation zwischen Arduino und Computer stellen einen Entwickler vor einige Probleme. Sie führen dazu, dass das Objekt gar nicht oder zum falschen Zeitpunkt rotiert und der Laser fehlerhaft angesteuert wird. Das wiederum verfälscht die Messergebnisse. Dieses Problem lässt sich durch die Implementierung eines geeigneten Protokolls beheben, jedoch würde dies viel Zeit in Anspruch nehmen.

Die Effizienz der Kantendetektion ihrerseits leidet unter dem starken Rauschen der Kamera, sowie der breiten Streuung des Lasers. Der Einsatz qualitativ höherwertiger Hardware würde sich hier besonders lohnen.

Ein weiteres Problem stellt die Montage von Laser und Linse dar. Die zum Zeitpunkt der Arbeit recht instabile Konstruktion erschwert eine genaue Justierung. Die Verwendung eines Schienensystems in Kombination mit geeigneten Fassungen würde sicherlich die Präzision der Apparatur und somit auch der Abtastung erhöhen.

5.3 Gewonnene Resultate

Im Rahmen der Testreihen wurden verschiedene Objekte abgetastet. Die Betrachtung der einzelnen Laserkanten zeigt, dass die Rekonstruktion der dreidimensionalen Daten zuverlässig arbeitet. Ungenauigkeiten entstanden hierbei vor allem durch die Kantendetektion, die zum Teil noch einige Außreißer lieferte.

Als besonders robust stellte sich die Detektion der Ecken des Schachbretts und die darauf basierende Bestimmung der Objekt- und Laserebene heraus. Dies zeigt auch Abbildung 4.2, in der die Positionen der erkannten Eckpunkte markiert sind. Die hohe Genauigkeit dieser Detektion der Eckpunkte auf Subpixelebene ermöglicht es, Fehler in der Rekonstruktion der Punkte nahezu auszuschließen. Störungen können nur durch Fehler in der Kantendetektion oder der Kamerakalibrierung auftreten.

Die Ergebnisse der Testreihen liegen sowohl als Punktwolken als auch als Gitterstrukturen vor. Die Abbildungen 5.1, 5.2 und 5.3 zeigen jeweils das abgetastete Objekt sowie die dreidimensionale Punktwolke.

Abbildung 5.1: Ein Läufer und das dazugehörige Gitter

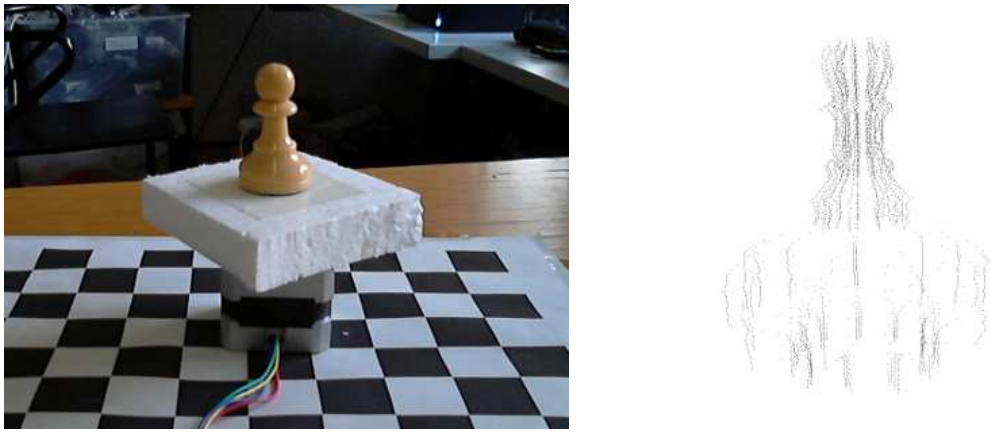


Abbildung 5.2: Ein Bauer und das dazugehörige Gitter

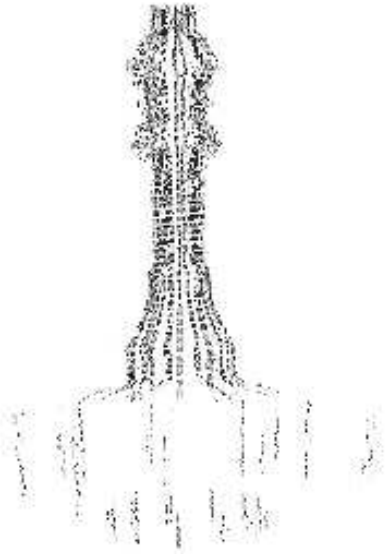


Abbildung 5.3: Eine Dame und das dazugehörige Gitter

6 Zusammenfassung und Ausblick

6.1 Vergleich der Ergebnisse

Die Tests zeigen, dass die Generierung von Punktwolken bzw. Gitterstrukturen auf Basis einer laserbasierten Objektabtastung einige Vorteile im Vergleich zu anderen Verfahren mit sich bringt.

Im Gegensatz zu passiven Verfahren ist diese Methode deutlich effektiver und liefert dabei noch bessere Ergebnisse. Der deutliche Leistungsvorteil liegt in der geringeren Komplexität der Algorithmen begründet. Auf rechenaufwendige Fluchtpunktbestimmungen und Schattendetektionen kann verzichtet werden. Zudem werden die Objekte als Ganzes und nicht nur von einer Seite erfasst. Verdeckungen die oftmals an Überhängen wie dem Griff einer Tasse entstehen, sind deutlich unwahrscheinlicher.

Im Vergleich zu anderen aktiven Verfahren, wie dem in der Vergleichsarbeit [LLRS05] vorgestellten Verfahren, zeichnet sich diese Methode durch eine vergleichbare bis höhere Qualität bei geringerem Preis aus.

Die Objektabtastung mit Hilfe von Schatten [LLRS05] ist beispielsweise äußerst empfindlich gegenüber Umwelteinflüssen wie anderen Lichtquellen. Außerdem können mit diesem Verfahren nur recht kleine Objekte abgetastet werden, da sonst der Schatten aufgrund des größeren Abstands zu breit und somit die Messergebnisse zu ungenau werden. Des Weiteren liefert die schattenbasierte Abtastung nur ein halbes Objekt, da der Schatten nur über eine Seite des Objekt bewegt wird. Dadurch wird es notwendig, das Objekt in zwei Durchgängen zu erfassen und die resultierenden Punktwolken in einem weiteren Schritt zusammenzufügen. Weitere Vergleiche mit diesem Verfahren sind leider nicht möglich,

da die dort eingesetzte „Art des Testens unter optimalen Bedingungen“ durch Simulation in einem 3-D-Bildbearbeitungsprogramm nur die „theoretische Leistungsfähigkeit des vorgestellten Verfahrens“ zeigt.

6.2 Ausblick

Trotz der einfachen bzw. kostengünstigen Umsetzung erreicht die laserbasierte Methode eine für die meisten Anwendungen ausreichend hohe Genauigkeit. Diese lässt sich jedoch im Bedarfsfall recht einfach erhöhen. Eine kürzere Belichtungszeit würde nicht nur ein dunkleres und somit einfacher zu analysierendes Bild liefern, sondern auch Ungenauigkeiten durch die Streuung des Laserlichts reduzieren. Selbstverständlich würde auch eine qualitativ höherwertige, rauschärmere Kamera mit einer höheren Auflösung das Ergebnis erheblich verbessern. Der Umstieg auf einen Infrarotlaser und eine entsprechende Kamera würde sowohl die Genauigkeit des Verfahrens erhöhen als auch eine differenzielle Kantendetektion überflüssig machen, wodurch die Dauer eines Scanvorgangs erheblich reduziert würde. Vergleichbare Vorteile würde jedoch auch ein deutlich kostengünstigerer Lichtfilter vor der Kamera mit sich bringen.

Die Dauer eines Scan-Vorgangs kann außerdem durch eine Unterteilung des Scan-Prozesses verbessert werden. Die Rekonstruktion der dreidimensionalen Koordinaten kann zu diesem Zweck parallelisiert werden. Die Auslagerung der Berechnungen in eine Cloud oder sogar auf eine Grafikkarte (beispielsweise Nvidia Cuda) würde weitere Leistungsgewinne mit sich bringen. Die Hardware arbeitet hier noch nicht an ihren Grenzen und könnte, sollte eine Parallelisierung der Software dies erfordern, durch einen schnelleren Motor und eine Kamera mit einer höheren Bildrate sogar noch weiter ausgebaut werden.

Da die aktuelle Konstruktion sowohl einen Arduino als auch einen Computer erfordert, lohnt sich sicherlich auch ein Blick in Richtung anderer Embedded-Plattformen. Aufgrund der Leistungsfähigkeit und des Preises scheint ein Umstieg auf das im November 2011 von Texas Instruments, Digi-Key und anderen Mitgliedern von `Beagleboard.org` vorgestellten

BeagleBone lohnenswert. Es ermöglicht nicht nur die direkte Ansteuerung des Motors und des Lasers, sondern auch den direkten Anschluss einer CMOS-Webcam und die direkte Verarbeitung der generierten Bilder. Auch die bereits angesprochene Parallelisierung und Auslagerung einiger Prozesse in eine Cloud lässt sich dank einer integrierten Ethernet-Schnittstelle bewerkstelligen. Besonders interessant ist es, dass trotz der vielen Vorteile das BeagleBone ähnlich einfach zu programmieren ist, wie der Arduino. Denn Dank der Ansteuerung der Ein- und Ausgabepins über Dateioperationen und der Tatsache, dass ein vollwertiges Linux auf dem kleinen Board lauffähig ist, kann unabhängig von einer Programmiersprache entwickelt werden. Eine Portierung der in Java entwickelten Oberfläche *ObjSpan* und der damit verbundenen Ansteuerung der Kamera ist somit ohne großen Aufwand möglich. Die Steuerung über Dateizugriffe macht zudem den Einsatz einer zweiten Programmiersprache und die Nutzung der RXTX-Bibliothek überflüssig, wodurch die Softwareanforderungen gesenkt und eine weitere Beschleunigung des Systems, durch Überwindung des durch die serielle Übertragung entstehenden Flaschenhalses, erreicht wird.

6.3 Fazit

Abschließend lässt sich feststellen, dass das erarbeitete Verfahren deutliche qualitative Vorteile gegenüber dem in der Vergleichsarbeit [LLRS05] vorgestellten Verfahren mit sich bringt. Zum einen wird nicht nur ein Teil des Objekts, sondern das gesamte Objekt abgetastet. Außerdem ist eine laserbasierte Abtastung unter natürlichen Begebenheiten deutlich unanfälliger gegenüber Störungen wie beispielsweise anderen Lichtquellen. Dazu kommt, dass die eingesetzte Hardware durch geeignete Erweiterungen nicht nur unempfindlicher gegenüber solchen Einflüssen gemacht werden kann, sondern auch zur Abtastung deutlich größerer Objekte fähig wird. Allerdings sollte man bei diesem Vergleich nicht außer Acht lassen, dass die Kosten der im Rahmen dieser Arbeit entwickelten Hardware deutlich höher sind.

Qualitativ ist die vorgestellte Methode mit anderen aktiven Verfahren vergleichbar, da sie kostengünstiger und auf Hardwareebene ausbaufähig ist.

Literaturverzeichnis

[BC05] M. Banzi and D. Cuartielles. The Arduino Platform. *Arduino Project Page*, 2005.

[BK08] G. Bradski and A. Kaehler. *Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.

[Bra00] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[CR01] A. Chalmers and T.-M. Rhyne. *A low cost 3D Scanner based on structured light*. Universität Sienna, 2001.

[LLRS05] P.-L. Lott, P. Lüdeking, P. Rhiem, and C. Sassenberg. *Self-made Low-cost 3D-Scanner*. Institut für Informatik Universität Münster, 2005.

[Lub10] G. Lube. *Numerische Mathematik I*. Georg-August-Universität Göttingen, 2010.

[Sch10a] R. Schaback. *Approximationsverfahren I*. Georg-August-Universität Göttingen, 2010.

[Sch10b] R. Schaback. *Handout for Computer-Aided Design, including multiaffine forms*. Georg-August-Universität Göttingen, 2010.

[SW06] R. Schaback and H. Wendland. *Kernel Techniques*. Georg-August-Universität Göttingen, 2006.

[SYXH02] W. Sun, X. Yang, S. Xiao, and W. Hu. *Robust Recognition of Checkerboard Pattern for Deformable Surface Matching in Multiple Views*. Shanghai Jiao Tong University, 2002.

A Inhalt der CD-ROM

Die der Arbeit beiliegende CD enthält die Konstruktionszeichnungen sowie die technischen Zeichnungen des Drehtellers. Des Weiteren liegen die komplette Arbeit als PDF- und LaTeX-Datei und die verwendeten Bilder bei. Selbstverständlich ist auch die komplette Applikation, die im Rahmen dieser Arbeit entwickelt wurde enthalten.

arduino Quelltext des Arduinos

construction Konstruktionszeichnungen im STEP- und Parasolid-Format, technischen Zeichnungen als PDF-Dateien und Bilder der Konstruktion

ext Externe Bibliotheken, die von der ObjSpan-Software eingesetzt werden

images Bilder der ObjSpan-Software

src Pakete der Applikation

- **arduino** Klasse zur seriellen Kommunikation mit dem Arduino
- **geometry** Geometrische Basis-Klassen
- **objloader** Lexer und Parser für Wavefront-Dateien
- **objspan** Grafische Oberfläche
- **parser** Parser und Lexer Basisklassen
- **processing** Basisklassen der Bildverarbeitung

test Wavefront-Dateien zu Testzwecken

thesis Bachelorarbeit als PDF- und LaTeX-Datei, Bilder und Quellcodes

B Technische Zeichnungen

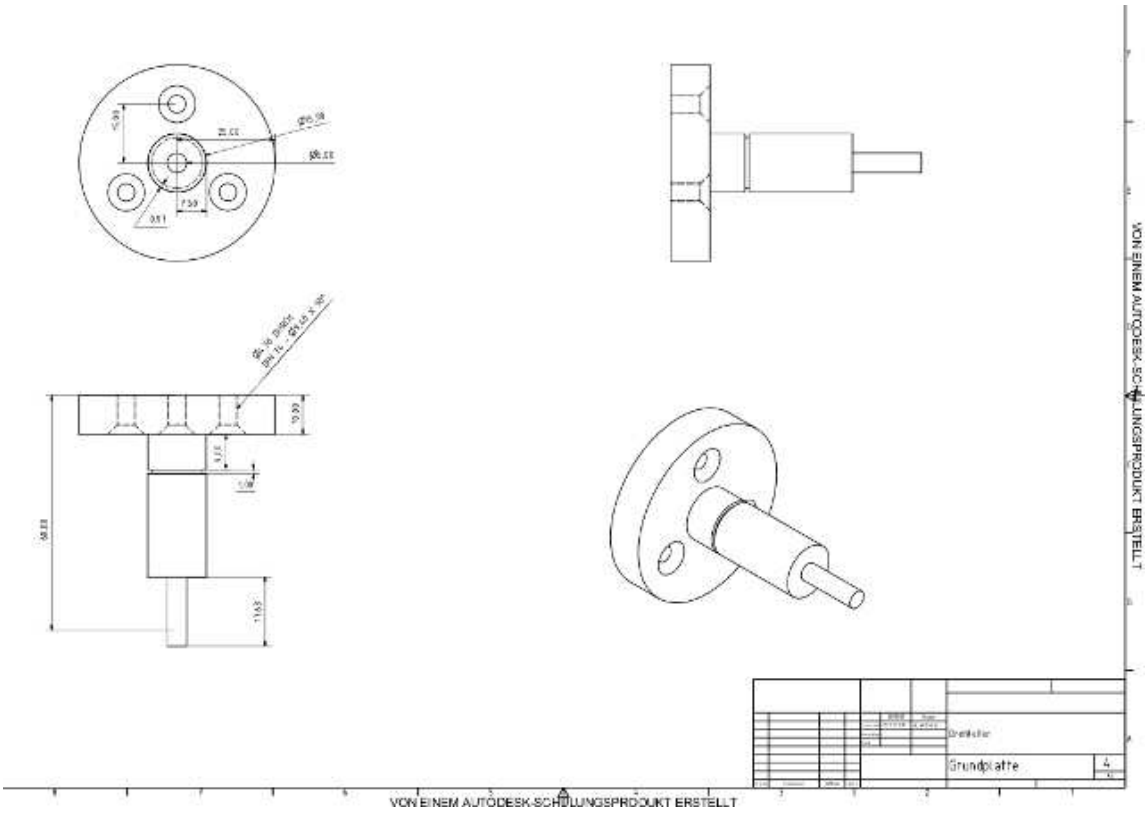


Abbildung B.1: Drehwelle

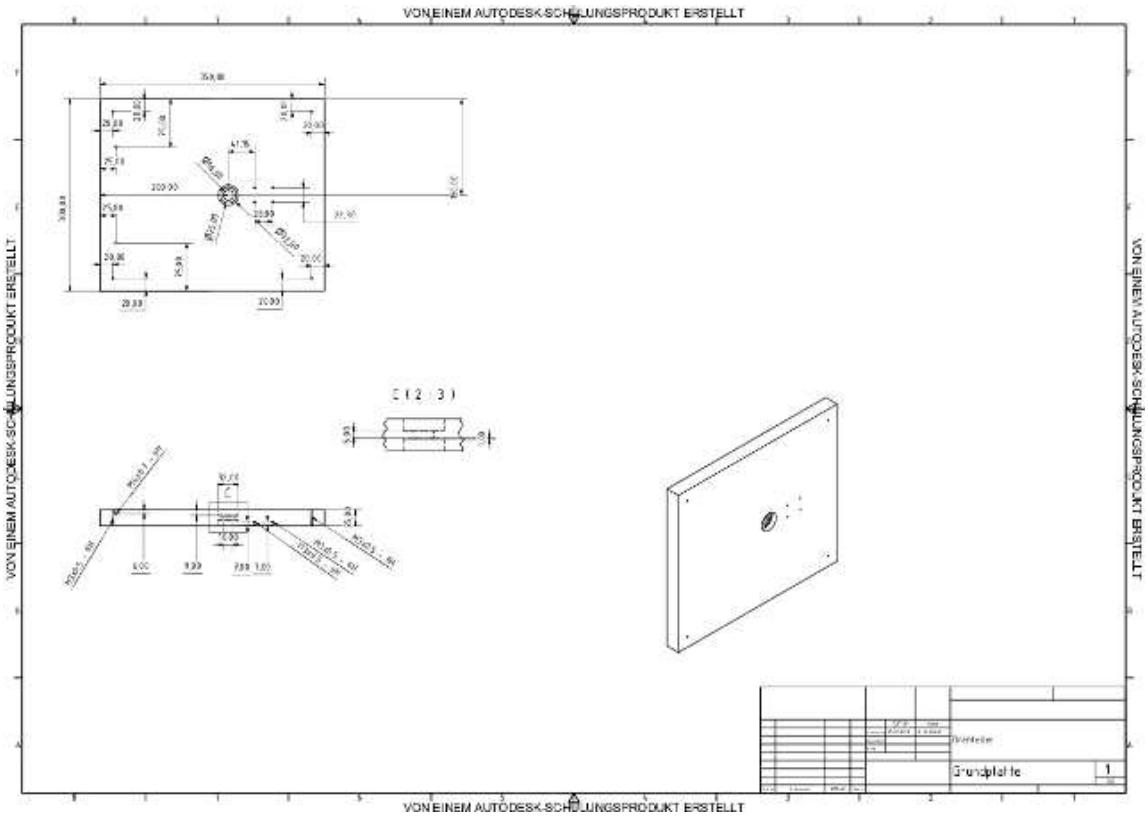


Abbildung B.2: Grundplatte

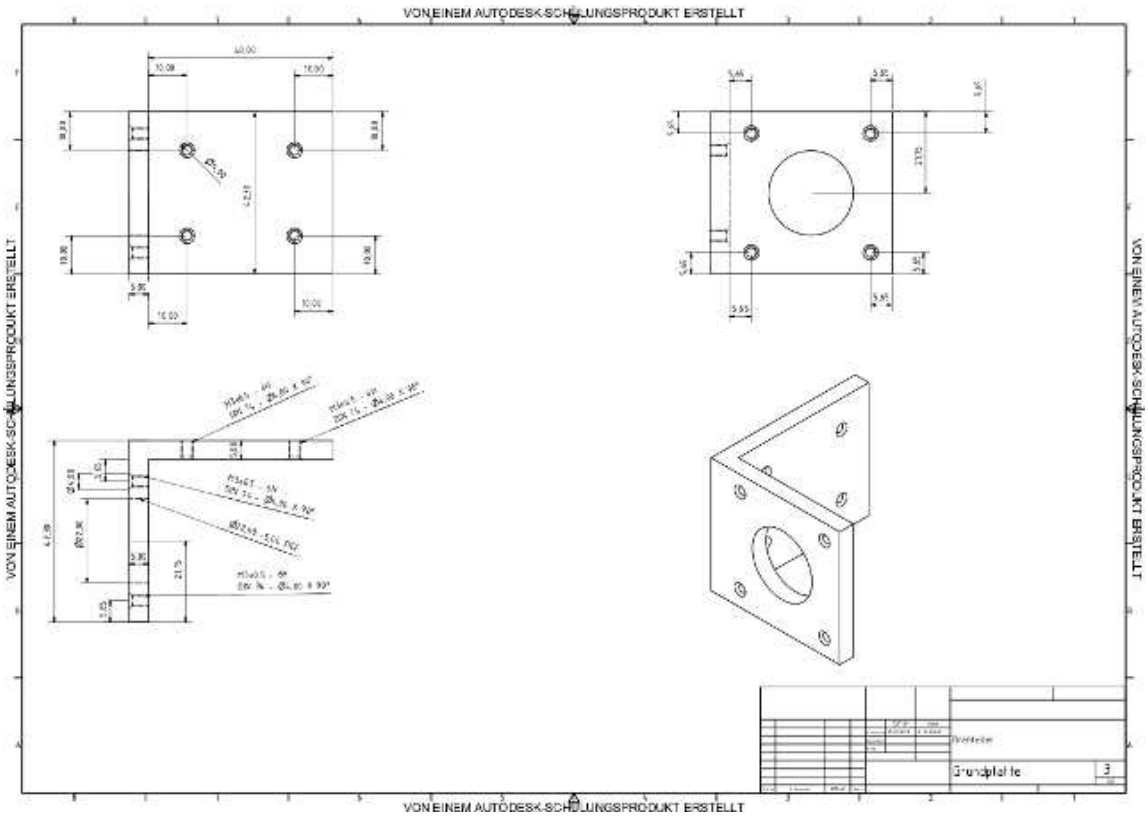


Abbildung B.3: Nema17-Winkel

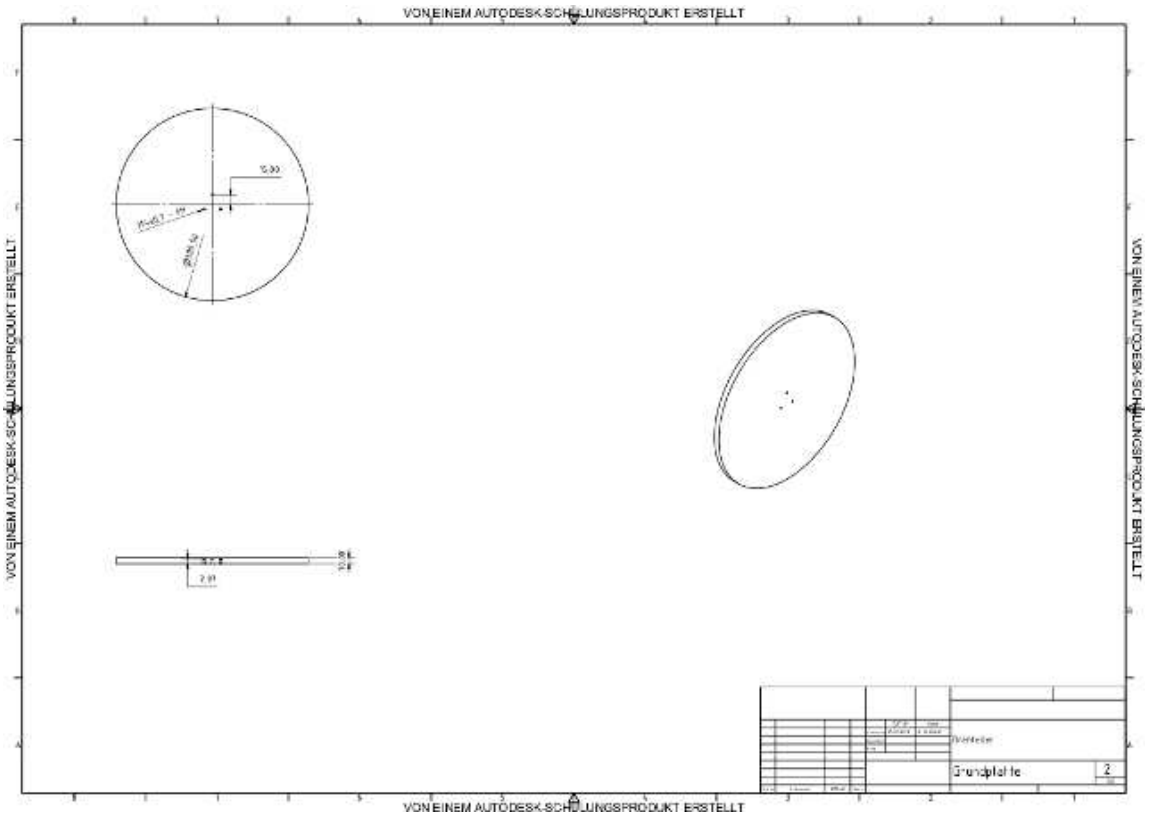


Abbildung B.4: Drehteller

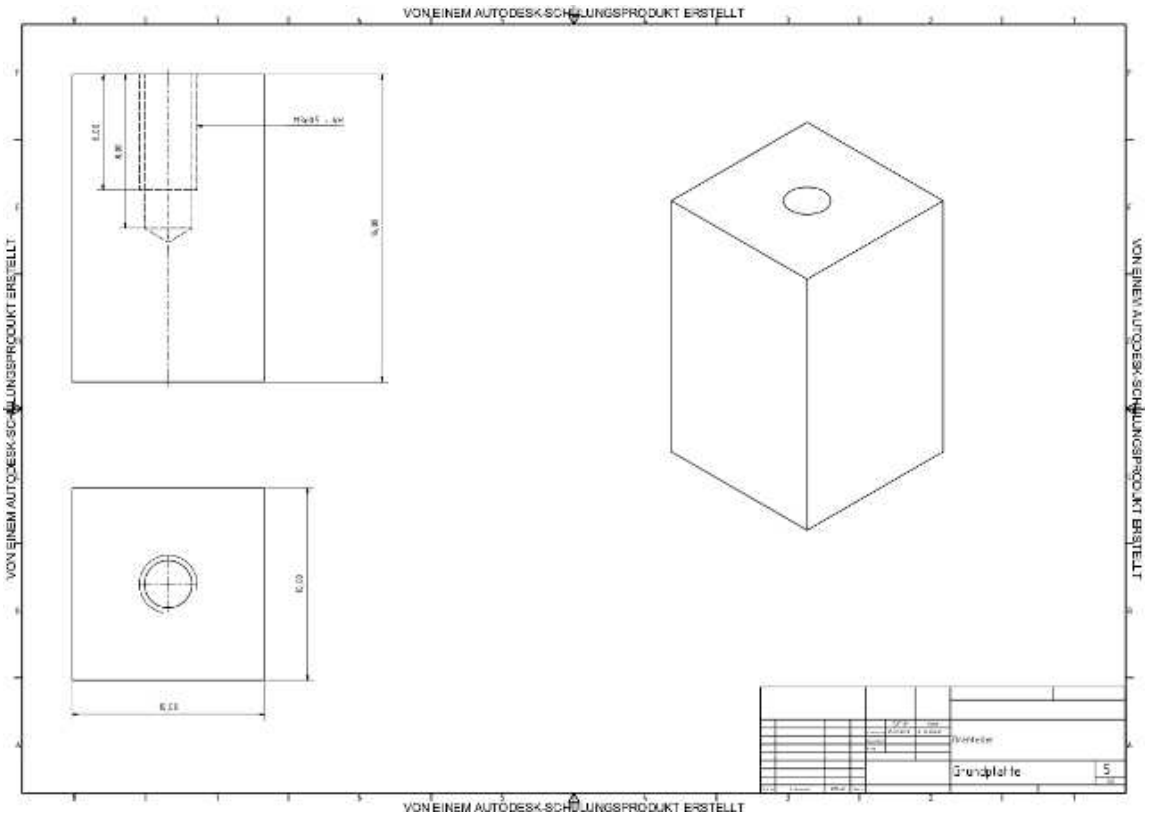


Abbildung B.5: Bein

C Quellcode

C.1 objScan.c - Arduino-Steuerung

```
// LED connected to pin 12.
#define LASER_PIN 12
#define MOTOR_DIR_PIN 2
#define MOTOR_STEP_PIN 3

// Commands from the java app.
#define COM_LIGHT 15
#define COM_MOTOR 16

boolean lState = true;

void setup() {
  // Setup the IO pins.
  pinMode(LASER_PIN, OUTPUT);
  pinMode(MOTOR_DIR_PIN, OUTPUT);
  pinMode(MOTOR_STEP_PIN, OUTPUT);

  // Initialize the laser pin.
  digitalWrite(LASER_PIN, HIGH);

  // Start up serial port.
  Serial.begin(9200);
}

void loop() {
  if (Serial.available()) {
    int cmd = Serial.read();
```

```
      Serial.flush();
      execute(cmd);
    }
  }

  void execute(int cmd) {
    switch(cmd) {
    case COM_LIGHT:
      digitalWrite(LASER_PIN, lState ? LOW : HIGH);
      lState = !lState;
      break;
    case COM_MOTOR:
      rotateDeg(36, 1);
      break;
    }
  }

  // Rotates a specific number of degrees (1.8 per step).
  // Negative steps used for reverse movement.
  // Speed is any number between .01 and 1.
  void rotateDeg(float deg, float speed) {
    int dir = (deg > 0) ? HIGH : LOW;
    digitalWrite(MOTOR_DIR_PIN, dir);

    int steps = abs(deg/1.8) * 8;
    float usDelay = (1/speed) * 70;

    for(int i = 0; i < steps; i++) {
      digitalWrite(MOTOR_STEP_PIN, HIGH);
      delayMicroseconds(usDelay);

      digitalWrite(MOTOR_STEP_PIN, LOW);
      delayMicroseconds(usDelay);
    }
  }
```

Listing C.1: objScan.c

C.2 SerialCom.java - Serielle Kommunikation

```
public class SerialCom implements SerialPortEventListener {
  // The serial and in/output port objects.
  private SerialPort serialPort = null;
  private InputStream input = null;
  private OutputStream output = null;

  public void connect(CommPortIdentifier com) throws Exception {
    // Open serial port.
    serialPort = (SerialPort) port.open(this.getClass().getName(), 2000);
    serialPort.setSerialPortParams(9600, SerialPort.DATABITS_8,
    SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
    Thread.sleep(2000);

    // Initialize input and output stream.
    input = serialPort.getInputStream();
    output = serialPort.getOutputStream();

    // Finally add the event listener.
    serialPort.addEventListener(this);
    serialPort.notifyOnDataAvailable(true);
  }

  public void disconnect() throws Exception {
    serialPort.removeEventListener();
    serialPort.close();
    input.close();
    output.close();
  }

  public void serialEvent(SerialPortEvent evt) throws Exception {
    if (evt.getEventType() == SerialPortEvent.DATA_AVAILABLE) {
      int available = input.available();
      byte chunk[] = new byte[available];
      input.read(chunk, 0, available);

      // Display results.
```

```
        System.out.print(new String(chunk));
      }
    }

    public void writeData(int cmd) throws Exception {
      output.write(cmd);
      output.flush();
    }
}
```

Listing C.2: SerialCom.java

C.3 CameraCapture.java - Kamerasteuerung

```
class CameraCapture {
   public static void main(String args[]) throws IOException {
      // Initialize the grabber object, ...
      OpenCVFrameGrabber grabber = new OpenCVFrameGrabber(0);

      // ... start the grabber ...
      grabber.start();

      // ... and grab an image from the camera.
      IplImage image = grabber.grab();

      // Create the window frame to display the image.
      CanvasFrame canvasFrame = new CanvasFrame("Camera Test");
      canvasFrame.setCanvasSize(image.width(), image.height());

      // Show the camera capture.
      canvasFrame.showImage(image);
   }
}
```

Listing C.3: CameraCapture.java

C.4 AxisChessboard.java - Kamerakalibrierung

```
public class AxisChessboard {
  public void calibrate(int boardW, int boardH, int numBoards, int
      boardDt)
      throws Exception {
    // Initialize the grabber object ...
    OpenCVFrameGrabber grabber = new OpenCVFrameGrabber(0);

    // ... and start it.
    grabber.start();

    // Get the number of points ...
    int numPoints = boardW * boardH;

    // ... and the board size.
    CvSize boardSize = cvSize(boardW, boardH);

    // Initialize the frames for the calibration, raw and undistorted
        video.
    CanvasFrame calibFrame = new CanvasFrame("Calibration");
    CanvasFrame rawFrame = new CanvasFrame("Raw Video");
    CanvasFrame undistortFrame = new CanvasFrame("Undistorted");

    // Allocate the matrices.
    CvMat imagePoints = cvCreateMat(numBoards * numPoints, 2, CV_32FC1);
    CvMat objectPoints = cvCreateMat(numBoards * numPoints, 3, CV_32FC1);
    CvMat pointCounts = cvCreateMat(numBoards, 1, CV_32SC1);
    CvMat intrMatrix = cvCreateMat(3, 3, CV_32FC1);
    CvMat distCoeffs = cvCreateMat(4, 1, CV_32FC1);

    CvPoint2D32f corners = new CvPoint2D32f(numPoints);
    int[] cornerCount = { 0 };
    int successes = 0;
    int step, frame = 0;

    // Grab the first image, ...
    IplImage image = grabber.grab();
```

```
35
      // ... show it in the raw frame ...
      rawFrame.showImage(image);

      // ... and generate its grayscale.
40    IplImage grayImage = cvCreateImage(cvGetSize(image), 8, 1);

      while (successes < numBoards) {
        // Skip every boardDt frames to allow user to move chessboard.
        if ((frame++ % boardDt) == 0) {
45        // Find the chessboard corners, ...
          int found = cvFindChessboardCorners(image, boardSize, corners,
              cornerCount, CV_CALIB_CB_FAST_CHECK);

          // ... get their Subpixel accuracy ...
50        cvCvtColor(image, grayImage, CV_BGR2GRAY);
          cvFindCornerSubPix(grayImage, corners, cornerCount[0],
              cvSize(11, 11), cvSize(-1, -1), cvTermCriteria(
                  CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1));

55        // ... and draw the chess board.
          cvDrawChessboardCorners(image, boardSize, corners,
              cornerCount[0], found);

          // If we got a good board, add it to our data.
60        if (cornerCount[0] == numPoints) {
            calibFrame.showImage(image);
            step = successes * numPoints;
            for (int i = step, j = 0; j < numPoints; i++, j++) {
              imagePoints.put(i, 0, corners.position(j).x());
65            imagePoints.put(i, 1, corners.position(j).y());
              objectPoints.put(i, 0, j / boardW);
              objectPoints.put(i, 1, j % boardW);
              objectPoints.put(i, 2, 0.0);
            }
70
            pointCounts.put(successes, 0, numPoints);
            successes++;
          } else {
```

```
          calibFrame.showImage(grayImage);
        }
      }

      // Get the next image.
      image = grabber.grab();
      rawFrame.showImage(image);
    }

    calibFrame.dispose();

    // Allocate the matrices according to the number of chessboards.
    CvMat objectPoints2 = cvCreateMat(successes * numPoints, 3, CV_32FC1)
        ;
    CvMat imagePoints2 = cvCreateMat(successes * numPoints, 2, CV_32FC1);
    CvMat pointCounts2 = cvCreateMat(successes, 1, CV_32SC1);

    // Transfer the points into the correct size matrices.
    for (int i = 0; i < successes * numPoints; ++i) {
      imagePoints2.put(i, 0, imagePoints.get(i, 0));
      imagePoints2.put(i, 1, imagePoints.get(i, 1));
      objectPoints2.put(i, 0, objectPoints.get(i, 0));
      objectPoints2.put(i, 1, objectPoints.get(i, 1));
      objectPoints2.put(i, 2, objectPoints.get(i, 2));
    }

    for (int i = 0; i < successes; ++i) {
      pointCounts2.put(i, 0, pointCounts.get(i, 0));
    }

    // Clear the memory.
    cvReleaseMat(objectPoints);
    cvReleaseMat(imagePoints);
    cvReleaseMat(pointCounts);

    // Initialize the intrinsic matrix.
    intrMatrix.put(0, 0, 1.0);
    intrMatrix.put(1, 1, 1.0);
```

```
      // Calibrate the camera.
      cvCalibrateCamera2(objectPoints2, imagePoints2, pointCounts2,
          cvGetSize(image), intrMatrix, distCoeffs, null,
          null, 0
      );

      // Save the intrinsic camera matrix and distortion coefficients.
      cvSave("intrinsics.xml", intrMatrix);
      cvSave("distortion.xml", distCoeffs);

      // Build the undistort map which we will use for all subsequent
          frames.
      IplImage mapx = cvCreateImage(cvGetSize(image), IPL_DEPTH_32F, 1);
      IplImage mapy = cvCreateImage(cvGetSize(image), IPL_DEPTH_32F, 1);
      cvInitUndistortMap(intrMatrix, distCoeffs, mapx, mapy);

      // Show the raw and the undistorted image.
      if (image != null) {
        // Show the raw image, ...
        IplImage t = cvCloneImage(image);
        rawFrame.showImage(image);

        // ... undistort the raw image ...
        cvRemap(t, image, mapx, mapy, CV_INTER_LINEAR
            | CV_WARP_FILL_OUTLIERS, cvScalarAll(0));
        cvReleaseImage(t);

        // ... and show the undistorted image.
        undistortFrame.showImage(image);
      }

      grabber.stop();
    }
  }
```

Listing C.4: AxisChessboard.java

C.5 ObjView.java - Anzeige 3-D-Objekte

```
public class ObjView extends JFrame {
    private BranchGroup root;

    public ObjView(Reader reader) throws IOException {
        setTitle("Basic Java3D Program");
        setSize(800, 600);
        Canvas3D canvas = new Canvas3D(SimpleUniverse.
            getPreferredConfiguration());
        getContentPane().add(canvas, BorderLayout.CENTER);
        SimpleUniverse universe = conigureUniverse(canvas, reader);
        universe.addBranchGraph(createSceneGraph());
    }

    private SimpleUniverse conigureUniverse(Canvas3D canvas, Reader reader
        ) throws IOException {
        // Setup the universe, ...
        SimpleUniverse universe = new SimpleUniverse(canvas);

        // ... add the model from file ...
        ObjectFile file = new ObjectFile(ObjectFile.RESIZE);
        root = file.load(reader).getSceneGroup();

        // ... and add some lights.
        AmbientLight ambientLight = new AmbientLight(new Color3f(Color.
            WHITE));
        ambientLight.setInfluencingBounds(new BoundingSphere());
        root.addChild(ambientLight);
        root.compile();
        return universe;
    }

    public BranchGroup createSceneGraph() {
        BranchGroup objRoot = new BranchGroup();

        TransformGroup listenerGroup = new TransformGroup();
        listenerGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
```

```
        listenerGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
        objRoot.addChild(listenerGroup);

        MouseRotate rotate = new MouseRotate(listenerGroup);
        rotate.setSchedulingBounds(new BoundingSphere(new Point3d(), 100));

        MouseWheelZoom zoom = new MouseWheelZoom(listenerGroup);
        zoom.setSchedulingBounds(new BoundingSphere(new Point3d(), 100));

        listenerGroup.addChild(rotate);
        listenerGroup.addChild(zoom);
        listenerGroup.addChild(root);

        return objRoot;
    }

    public static void main(String args[]) throws IOException {
        new ObjView(new FileReader("wuerfel.obj")).setVisible(true);
    }
}
```

Listing C.5: ObjView.java