# Dynamische Trapezkorrektur mithilfe einer Tiefenkamera auf der Basis von MaxMSP/Jitter

Diplomarbeit

vorgelegt von Thomas Orr
Matrikelnummer: 480961
Hochschule: Fachhochschule Düsseldorf
Studiengang: Ton- und Bildtechnik (Diplom)

11. Juni 2012

Erstprüfer: Prof. Dr.-Ing. Günther Witte
Zweitprüfer: Dipl.-Ing. Florian Boddin

# Abstract


## Deutsch

In den letzten Jahren wurden immer wieder neue Verfahren entwickelt, um Projektionstechnik zu automatisieren und „intelligenter“ zu machen. Innovationen wie kostengünstige, hochauflösende Tiefenkameras ermöglichen dabei neue Herangehensweisen für automatisches Projektionsmapping.

In dieser Arbeit wird ein Verfahren zur automatischen Trapezkorrektur eines Videoprojektors in Kombination mit einer *Microsoft-Kinect*-Tiefenkamera und einem Computer vorgestellt. Hierfür wurde ein theoretisches Modell entwickelt, durch welches das Ausgangsbild eines Projektors dynamisch so verzerrt werden kann, dass es bei Schrägprojektion stets als rechteckiges Bild auf der Projektionsfläche erscheint. Vorangehend wird eine Technik zur Kalibrierung der *Kinect*-Tiefenkamera mit einem beliebigen Videoprojektor präsentiert. Die theoretischen Annahmen wurden als Proof of Concept in Form einer Prototyp-Software auf Basis der grafischen Programmierumgebung *MaxMSP/Jitter* entwickelt. Dabei kamen einige Algorithmen aus der Computer-Vision-Programmbibliothek *OpenCV* zum Einsatz.

## English

In recent years many new methods have been developed in projection technology to make systems automatic and more intelligent. Recent innovations, such as inexpensive, high-resolution depth-sensing cameras, contribute to this movement and allow new approaches to machine-aided projection mapping.

The system for automatic keystone correction the author presents in this paper uses a video projector, a *Microsoft Kinect* depth-sensing camera and a computer. The author created a theoretical model for manipulating the images transmitted to the video projector in such a way that the image the viewer sees is always rectangular, regardless of the projection angle. The details are preceded by a proposal for a method of calibrating the *Kinect* camera and the projector. As a proof of concept the postulations are implemented in prototype software by the author in the *MaxMSP/Jitter* visual programming language, using algorithms from the *OpenCV* computer-vision library.

# Inhaltsverzeichnis

## Hinweise zu dieser Arbeit

Beim Lesen dieser Arbeit ist zu Beachten, dass Eigennamen, Bezeichnungen und mathematische Benennungen stets mit kursiver Schriftlage gekennzeichnet sind.

Fachbegriffe und Abkürzungen, die im Glossar auf Seite 82 näher erläutert werden, sind grau unterstrichen. Zudem werden einige Elemente der verwendeten Programmierumgebung *MaxMSP/Jitter* im Glossar näher erläutert.

Alle in dieser Arbeit enthaltenen Abbildungen wurden vom Autor selbst angefertigt, sofern nicht anders gekennzeichnet. Das Abbildungs- sowie das Tabellenverzeichnis sind auf Seite 84 und Seite 85 zu finden.

Aus Gründen der leichteren Lesbarkeit wird in dieser Arbeit auf eine geschlechtsspezifische Differenzierung, wie z. B. Nutzer/Innen, verzichtet. Entsprechende Begriffe gelten im Sinne der Gleichbehandlung für beide Geschlechter.

# Einleitung

Projektionstechnik ist seit langem in einer Vielzahl von Anwendungsbereichen vorzufinden. Vor allem im Laufe der letzten zwei bis drei Jahrzehnte hat immer günstiger werdende digitale Projektionstechnik Einzug in Seminarräume, Kinos, Theater, Museen, Messen und nicht zuletzt in die Wohnzimmer erhalten und ist auch in vielen anderen Bereichen nicht mehr wegzudenken. Trotz der raschen Entwicklung von Displaytechnologien wie LCD, Plasma, OLED oder LED-Walls und einem damit florierenden Displaymarkt bleiben Projektoren bis dato in vielen Anwendungsgebieten unersetzlich. Dies ist vor allem auf zwei Faktoren zurückzuführen: Erstens ist die Nutzung von Projektoren im Vergleich zu Displays mit ähnlicher Abbildungsgröße mit einem wesentlich geringeren Kostenaufwand verbunden. Zweitens sind Projektoren vollkommen unabhängig von der eigentlichen Bildfläche und somit nicht an die Größe des Geräts bzw. des bilderzeugenden Elements gebunden, wie dies bei Displays der Fall ist. Diese Punkte ermöglichen ein sehr hohes Maß an Flexibilität beim Gebrauch von Projektionstechnik.

Die eben angesprochene Unabhängigkeit von der Bildfläche äußert sich vor allem darin, dass mit Videoprojektoren nicht nur die orthogonale Leinwandprojektion möglich ist. Vielmehr kann theoretisch aus beliebigen Winkeln auf Flächen oder gar auf plastische Objekte projiziert werden. Vor allem im künstlerischen Rahmen, in Bereichen wie Werbung und Produkt-präsentation und in Gebieten der erweiterten und virtuellen Realität schaffen so Projektionen fundamentale Möglichkeiten. Nicht nur gewöhnliche Leinwände können so zur Informations-übertragung genutzt werden, sondern beliebige Flächen und Gegenstände können mit (projizierten) Texturen belegt werden.

Der Einsatz von Projektionstechnik erfordert jedoch im Gegensatz zum Einsatz von Displays eine individuelle Einrichtung und Anpassung an die Umwelt. Dies beinhaltet Schritte wie Farbanpassung und geometrische Korrekturen. Bei Schrägprojektion etwa muss eine Entzerrung des Bildes vorgenommen werden, sodass auf der Projektionsfläche wiederum ein unverzerrtes, der Fläche angepasstes Bild entsteht. Hierfür bieten die meisten Projektoren bereits eine Vielzahl optischer und digitaler Einstell- und Korrekturmöglichkeiten. Durch diese wird für die meisten Anforderungen eine mehr oder weniger zufrieden stellende Anpassung an Oberflächenverhältnisse, Projektionswinkel und Projektionsabstände möglich.

Neben internen Anpassungsmöglichkeiten eines Projektionsgerätes spielen Entwicklungen im Softwarebereich in diesem Zusammenhang eine entscheidende Rolle. Moderne Medienserver

bieten diverse Parameter für Farbanpassungen und erlauben geometrische Korrekturen der Ausgangsbilder, etwa zur perspektivischen Anpassung. Die Möglichkeiten reichen dabei von einfachen Funktionen zur Trapezkorrektur bis hin zu komplexen Mappings zur Projektion auf plastische Objekte.

Die manuelle Einrichtung dieser Systeme ist zeitaufwendig und erfordert professionelles Personal und spezielle Bedingungen. Zudem sind genau eingerichtete Projektionen fragil und erlauben keinerlei Veränderungen in der Anordnung von Projektionsflächen oder der Beleuchtungssituation. Einige Entwicklungen und Forschungen der letzten Jahre beschäftigen sich daher mit automatischen Systemen, die möglichst selbstständig und in kurzer Zeit eine genaue Anpassung von Projektionen an unterschiedliche Bedingungen ermöglichen. Hierzu werden Kameras oder Sensoren eingesetzt, welche Informationen über den Projektionsbereich sammeln und Korrekturen der Geometrie, Helligkeit und Farbe selbstständig durchführen. Meist beruhen diese Verfahren jedoch auf einer szenenbezogenen Kalibrierung, die bei Veränderung der Verhältnisse, etwa durch eine Umstellung des Projektors oder der Leinwand, wiederholt werden muss. Bei dynamischen Systemen hingegen werden häufig sogenannte Marker an Projektionsflächen angebracht, die in einem Kamerabild erkannt werden können. Anhand dessen kann dann eine Anpassung der projizierten Bilder vorgenommen werden. Folglich erfordern diese Methoden eine Präparierung der Projektionsflächen, was wiederum eine gewisse Inflexibilität mit sich bringt.

In dieser Arbeit wird ein Verfahren vorgestellt, welches unabhängig von Markern auf nahezu beliebigen Flächen eine automatische dynamische Trapezkorrektur ermöglicht. Mithilfe einer *Microsoft-Kinect*-Tiefenkamera wird die Lage und Orientierung des Projektors bezogen auf eine Projektionsfläche analysiert. Anhand dieser Information kann bei Schrägprojektion auf eine planare Projektionsfläche eine automatische Trapezkorrektur durchgeführt werden. Da die verwendete Tiefenkamera in Echtzeit Tiefendaten ermittelt, funktioniert das System dynamisch und erfordert lediglich eine einmalige Kalibrierung, solange die Position von Projektor und Sensor zueinander konstant bleibt. Da sich die Kalibrierung nicht auf eine konkrete Szene bezieht, ist ein ortsunabhängiger Betrieb des Systems ohne erneute Kalibrierung möglich.

# Kapitel 1: Überblick

## 1.1 Vorhandene Arbeiten

Viele Systeme in der Projektionstechnik bieten bereits Methoden zur Anpassung von Projektoren. Sowohl farbliche und geometrische Korrekturen können dabei eine Rolle spielen. Neben großen Medienserver-Systemen werden vermehrt auch kleinere Softwaresysteme angeboten, welche über weitreichende Korrekturmöglichkeiten verfügen. Sie kommen vor allem im künstlerischen oder im VJ-Bereich zum Einsatz und ermöglichen neben Farbanpassungen oft die Realisation komplexer Projektionsmappings. Programme wie *Madmapper* [Mad11], *Modul8* [Mo812] oder *GrandVJ* [gVJ12] sind in diesem Zusammenhang als Beispiele zu nennen.

Wie bereits erwähnt, erfordern die meisten Lösungen jedoch eine händische Einrichtung auf eine Projektionsfläche und müssen bei Veränderungen manuell nachkorrigiert werden. Aus diesen Gründen ist die Entwicklung automatischer Systeme erwünscht, die ohne manuelle Eingriffe ermöglichen, Projektionen an unterschiedliche Gegebenheiten anzupassen. Grundsätzlich ist bei einem solchen automatischen Verfahren eine Kalibrierung oder Analyse der Szene notwendig, um dem System Auskunft über die Umwelt zu verschaffen und die Lage des Projektors innerhalb dieser Szene zu bestimmen. Anhand dieser Kalibrierungsdaten kann dann ein projiziertes Ausgangsbild entsprechend modifiziert werden bevor es auf die Szene projiziert wird.

Für die Analyse einer Szene werden diverse Verfahren erforscht und teilweise angewendet. Yang et. al kategorisieren diese Kalibrierungs- oder Analyseverfahren im Rahmen von automatischem Projektionsmapping in passive bzw. aktive und Online bzw. Offline-systeme [Yan01]. Als aktiv werden dabei Methoden bezeichnet, bei denen der Umgebung Energie zugeführt wird. Dies könnte etwa durch eine entsprechende Illumination der Szene realisiert werden, die dann von einer Kamera aufgenommen wird und aus deren Bild im Anschluss Information über eine Szene gewonnen werden. Passive Systeme nutzen hingegen ausschließlich in der Umgebung vorhandene Energie, um die Gegebenheiten zu analysieren. Offline Methoden werden als solche bezeichnet, wenn die Kalibrierung stattfindet, während sich das System nicht im Betrieb befindet. Dagegen ermöglichen Online-Kalibrierungs-methoden den dauerhaften Betrieb und eine Korrektur zur Laufzeit, auch wenn Veränderungen

in der Szene auftreten. Tabelle 1.1 veranschaulicht die Zusammenhänge und führt einige Beispiele für die einzelnen Kategorien auf.

| | **Offline** | **Online** |
|---|---|---|
| **Passiv** | Stereoanalyse | Mechanische Anpassung |
| **Aktiv** | Unsichtbare Structured-Light-Verfahren | Laser Scan, Structured-Light-Verfahren |

*Tabelle 1.1: Kategorien der Kalibrierungs- bzw. Erfassungsverfahren nach [Yan01]*

Johnny C. Lee et. al präsentieren ein Verfahren mithilfe mehrerer Photosensoren, welche an den Ecken einer Projektionsfläche angebracht werden. Zur Kalibrierung wird ein bekanntes Muster vom Projektor ausgegeben, um so die Positionen der Sensoren bezogen auf den Projektor zu ermitteln [Lee04]. Da vom Projektor zur Kalibrierung ein definiertes Bild ausgeben werden muss, handelt es sich hierbei um ein aktives Offline-System. Mithilfe der Sensorposition bezogen auf den Projektor kann dann ein Ausgangsbild so verzerrt werden, dass es auf der Projektionsfläche im richtigen Format erscheint. Allerdings ist nach jeder Verschiebung der Projektionsfläche bzw. der Sensoren eine Neukalibrierung notwendig.

Jan Hanten erweiterte im Rahmen seiner Bachelorarbeit das Medienserver-System der Firma *Coolux* um ein dynamisches Projektionsmapping [Han08]. Dabei wurden IR-Reflektoren an den Ecken der Projektionsflächen abgebracht und die Szene aktiv mit einem IR-Scheinwerfer beleuchtet. Eine Kamera mit IR-Passfilter wird aus der Position des Projektors auf die Szene gerichtet und registriert die Eckpunkte mit den Reflektoren. In Echtzeit können mit diesem Online-System Mappings auf bewegliche Projektionsflächen realisiert werden. Ähnliche Verfahren existieren mit selbstleuchtenden IR-LEDs .

Ruigang Yang und Greg Welch stellten ein Online-Verfahren vor, bei dem keine zusätzlichen Marker oder Sensoren außer einer Kamera benötigt werden [Yan01]. Dabei wird die Projektion auf einer beliebigen Fläche von einer Kamera erfasst und durch eine Feature-Analyse Bildelemente der Projektion mit dem Original verglichen. So kann iterativ eine Anpassung der Projektion an die Projektionsfläche realisiert werden, die zunächst auf Basis einer Schätzung begonnen wird. Da das System vom projizierten Videomaterial abhängig ist, kann es zu

Problemen kommen, wenn dieses über zu wenige hochfrequente Anteile verfügt. Die Projektion einer reinen Farbfläche könnte das System demnach nicht korrigieren.

## 1.2 Ziel der Arbeit

Mit dieser Arbeit soll ein System zur automatischen dynamischen Trapezkorrektur mithilfe einer *Microsoft-Kinect*-Tiefenkamera vorgestellt werden. Dabei soll das Verfahren eine Technik zur Kalibrierung der Tiefenkamera mit einem Videoprojektor beinhalten. Hierfür soll zunächst eine theoretische Analyse des Problems erfolgen und im Anschluss eine Umsetzung auf Basis der Programmierumgebung *MaxMSP/Jitter* beschrieben werden.

Viele vorhandene Methoden zur automatischen Trapezkorrektur von Projektoren beruhen auf Markern, die an den Ecken der Projektionsfläche angebracht werden (siehe Kapitel 1.1). Dabei wird über eine Kamera die Position der Marker durch einen Erkennungsalgorithmus bestimmt. Nachfolgend wird eine Transformation des Bildes in Form einer sogenannten Homographie-Matrix ermittelt, durch welche das Ausgangsbild so verzerrt wird, dass die Marker auf der Projektionsfläche die Eckpunkte der Projektion darstellen.

Die im Rahmen dieser Arbeit entwickelte Methode soll jedoch ohne Präparierung oder Vorbereitung einer Projektionsfläche anwendbar sein. Es soll ermöglicht werden, einen Projektor, auf dem eine Tiefenkamera installiert wurde, in beliebigem Winkel auf eine planare Fläche auszurichten. Das System erkennt die Lage zur Projektionsfläche und gleicht das Bild so aus, dass für einen Betrachter ein rechteckiges, unverzerrtes Bild sichtbar wird. Da keine Marker zur Verfügung stehen, werden durch ein mathematisches Verfahren vier rechteckig angeordnete Punkte auf der Projektionsfläche bestimmt, welche den Eckpunkten der Projektion entsprechen. Mit diesen errechneten Bild-Eckpunkten kann dann ebenfalls eine Homographie-Matrix ermittelt werden, die auf das Ausgangsbild angewendet werden kann.

Die im Rahmen dieser Arbeit entwickelte *MaxMSP*-Software *C_stone* sollen zur Prüfung der theoretischen Grundlagen dienen. Der Schwerpunkt bei der Entwicklung der Software wurde auf Übersichtlichkeit und Erweiterbarkeit gelegt. Ferner wurden im Rahmen dieser Arbeit einige *MaxMSP*-Externals entwickelt, in denen Algorithmen aus der *OpenCV*-Bibliothek aus den Bereichen Kamerakalibrierung und 3D-Rekonstruktion in *MaxMSP* zugänglich gemacht wurden [OCV12]. Diese Externals können auch in vielen anderen Zusammenhängen eingesetzt werden und leisten so einen Beitrag zur Erweiterung der Programmierumgebung *MaxMSP*.

## 1.3 Mögliche Anwendungen

Wie bereits angesprochen gestaltet sich nach wie vor die Einrichtung und die Entzerrung der Projektion bei schwierigen Projektionswinkeln als problematisch und zeitaufwendig. Die bei Schrägprojektion entstehende sogenannte **Trapezverzerrung**, lässt sich bei Projektoren meist mithilfe der **Trapezkorrektur** beheben oder zumindest verringern. Bei vielen Geräten reichen jedoch die Regelbereiche nicht aus, um bei steilen Winkeln eine verzerrungsfreie Projektion zu erzielen. Vor allem, wenn horizontale **und** vertikale Verzerrung ausgeglichen werden müssen, stoßen die internen Korrekturmöglichkeiten einiger Projektoren häufig an ihre Grenzen. Meist bieten nur teure Modelle die Möglichkeiten für eine sogenannte perspektivische Korrektur. Bei dieser Funktion *(cornerpin-correction)* können die Ecken einer Projektion unabhängig voneinander verschoben werden, um eine Anpassung an die Projektionsfläche vorzunehmen. Einige Computerprogramme bieten vergleichbare Möglichkeiten.

Das hier vorgestellte Verfahren zur automatischen Trapezkorrektur soll vorerst als softwarebasierte Methode implementiert werden. Allerdings wäre auch eine Implementierung der Technik direkt im Projektor prinzipiell möglich. Die verwendete Hardware zur Tiefenerkennung ist kostengünstig. Außerdem wird nur bei der Kalibrierung des Systems die volle Auflösung des hier verwendeten *Kinect*-Tiefensensors benötigt. Im Normalbetrieb muss nur die Distanz zu wenigen Messpunkten ermittelt werden, die gemeinsam auf einer planaren Projektionsfläche liegen. Ein vollständig hardwarebasiertes System könnte demnach ab Werk kalibriert werden und dann für den Betrieb mit einem geringer aufgelösten Sensor zur Abstandserkennung ausgestattet werden.

Generell bietet sich das System vor allem für kleinere bis mittelgroße Projektionsgeräte an, die überwiegend für den mobilen Einsatz konzipiert sind. Etwa im Bildungsbereich oder bei Präsentationen sind kurze Einrichtungszeiten und flexible Positionierung eines Projektors erforderlich. Dazu kommt, dass in diesen Bereichen fast ausschließlich planare Flächen für Projektionen genutzt werden. Hier wäre eine zuverlässige automatische Anpassung der Trapezkorrektur äußerst hilfreich. Aber auch bei Festinstallationen würde eine automatische Erkennung einen hohen Zeitvorteil bringen, wenn Projektoren auf Knopfdruck an Leinwände angepasst werden könnten.

Oftmals erfordert der Gebrauch von Projektoren in Bereichen wie Theater, Oper und Performance besonders flexible Einsatzmöglichkeiten. Die Installation ist abhängig von Bühnenbild und anderen ästhetischen Faktoren, sodass selten eine optimale Platzierung möglich ist. Auch stehen in großen Veranstaltungshäusern nur geringe Einrichtungszeiten zur

Verfügung, sodass eine detaillierte manuelle Einrichtung selten durchführbar ist. Hier würde ein System zur automatischen geometrischen Anpassung Einrichtungszeiten verkürzen und die Einsatzbereiche von Projektoren erweitern.

In Bereichen der Videokunst oder in Museen, wo die Installation von Projektoren heutzutage zur Tagesordnung gehört, wäre die Möglichkeit einer schnellen Korrektur eine Erleichterung bei der Einrichtung von Ausstellungen. So wären schnelle Umpositionierungen von Video-Kunstwerken kein Problem mehr und dadurch könnte Künstlern und Kuratoren deutlich mehr Spielraum gegeben werden, ohne dass die Technik als lästiges Hindernis empfunden würde.

Sicherlich gibt es im Rahmen von Anwendungen der virtuellen und erweiterten Realität ebenfalls eine Vielzahl von Einsatzmöglichkeiten für derartige Systeme. Dabei gehören vor allem Displaytechnologien, die sich möglichst unbemerkt und dynamisch an ihre Umgebung anpassen können, zum Interessensbereich von Forschern und Wissenschaftlern. Aktuell wird vor allem im Bereich der Pico-Projektionstechnik geforscht. So könnte ein Picoprojektor mit den hier vorgestellten Korrekturmechanismen beispielsweise am Helm eines Lagerarbeiters befestigt sein und jede Fläche zur potentiellen Projektionsfläche werden lassen, auf welcher diesem wichtige Informationen über Lagerbestände angezeigt werden.

Auch die Kombination mit anderen Techniken wie etwa die automatische Farb- und Helligkeitsanpassung bei wechselnden Oberflächen eröffnet neue Möglichkeiten und erweitert den Einsatzrahmen von Projektionstechnik.

## 1.4 Übersicht über diese Arbeit

Zunächst soll in Kapitel 2 ein Überblick über die für diese Arbeit relevanten Themengebiete der projektiven Geometrie gegeben werden. Des Weiteren werden Begriffe wie Welt-koordinatensystem und Homographie-Matrix einführend erläutert. Ebenso werden hier die technischen Grundlagen der *Kinect* besprochen und erläutert, welche Daten von der *Kinect* an einen Rechner übertragen werden und wie diese dort verwendet werden können. Eine kurze Einführung in *MaxMSP* und die *OpenCV*-Library gibt zudem Auskunft über die verwendete Software. Zusätzlich wird in diesem Kapitel auf einige *MaxMSP*-Externals eingegangen, welche wichtige Funktionen der *OpenCV*-Bibliothek in *MaxMSP* verfügbar machen. Einige Externals sind im Rahmen dieser Arbeit entwickelt worden.

In Kapitel 3 wird überwiegend die Implementierung der Software *C_stone* besprochen. Dabei wird ausführlich auf die Umsetzung der theoretischen Grundlagen in der Software *MaxMSP*

eingegangen. Ein wichtiger Teil der Berechnungen wird dabei in *JavaScript* realisiert. Zudem umfasst dieses Kapitel eine Beschreibung der Benutzeroberfläche der Software *C_stone*.

Kapitel 4 beinhaltet die Analyse der Bedienbarkeit wie auch der Robustheit und geht auf Problematiken des Systems ein. Neben der Evaluation werden hier Vorschläge zur Verbesserung und Weiterentwicklung analysiert.

In Kapitel 5 findet sich eine Zusammenfassung der Ergebnisse dieser Arbeit und ein Ausblick über Möglichkeiten, Ideen und Verbesserungen, die sich durch die Entwicklung und Analyse der Software ergeben haben.

# Kapitel 2: Grundlagen

## 2.1 Projektive Geometrie – das Lochkameramodell

Projektive Geometrie behandelt unter anderem die mathematischen Grundlagen für Abbildungen der echten dreidimensionalen Welt auf eine zweidimensionale Ebene. Eine solche Abbildung, in der Literatur auch als **Projektion** bezeichnet, kann am Beispiel einer Kameraaufnahme verdeutlicht werden. Hierbei treffen Lichtstrahlen, nachdem sie durch ein Linsensystem entsprechend gebündelt wurden, auf eine zweidimensionale Ebene. Dabei handelt es sich in der Regel um einen Film oder ein elektrisches Sensorelement, welches dann die Umwandlung von Helligkeitsinformation in digitale Werte ermöglicht. Es findet also eine Transformation aus der 3D-Welt auf eine 2D-Ebene statt, bei der jedoch die Tiefeninformation, also die Information über den Abstand eines Objekts, von dem der jeweilige Lichtstrahl ausging, verloren geht.

Eine vereinfachte Betrachtung des Sachverhalts ermöglicht das sogenannte Lochkameramodell. Eine Lochkamera besteht aus einer geschlossenen Kiste, welche an einer Seite mit einem unendlich kleinen Loch ausgestattet ist, durch das Lichtstrahlen ins Innere der Kiste gelangen können (siehe Abbildung 1). Die eintretenden Lichtstrahlen treffen auf die (Bild-)Ebene gegenüber dem Loch. In diesem idealisierten Modell kann jedem Lichtstrahl, der aus der 3D-Welt ins Innere der Kamera gelangt, ein korrespondierender 2D-Punkt auf der Bildebene zugeordnet werden.

Nach dieser Abbildung ist die Tiefeninformation nicht mehr vorhanden. Es ist jedoch möglich, anhand der Bildkoordinaten im Nachhinein noch auf den ehemaligen Verlauf des Strahls außerhalb der Kamera rückzuschließen. Dieser Vorgang setzt jedoch unter anderem die Kenntnis der sogenannten intrinsischen Kameramatrix voraus, welche in Kapitel 2.2 näher besprochen wird. Diese Matrix kann somit als mathematische Beschreibung der Bezüge zwischen den 3D-Punkten und deren korrespondierenden 2D-Punkte auf einer Ebene betrachtet werden.

In Abbildung 1 wird eine Kerze im 3D-Raum über eine Lochkamera auf einer zweidimensionalen Bildfläche abgebildet. Der 3D-Punkt A, die Spitze der Flamme, wird dabei auf dem korrespondierenden 2D-Punkt B abgebildet. Wie soeben angedeutet, kann im Nachhinein nicht mehr auf die ursprüngliche Position des Punktes A geschlossen werden. Allerdings kann,

unter der Kenntnis des Abstands der Bildebene zum Loch der Lochkamera, auf den Strahl rückgeschlossen werden, auf dem sich irgendwo der Punkt A befindet.

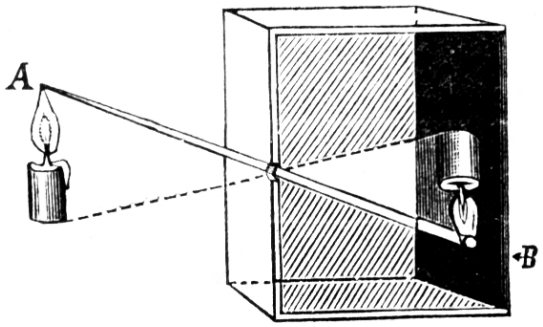


***Abbildung 1: Lochkamera***
***Quelle: pixxel-blog.de***

## 2.2 Die Kameramatrix

Eine Abbildung vom 3D-Raum auf eine 2D-Ebene, wie es bei einer Kameraaufnahme der Fall ist, kann in Form einer $3 \times 4$-Matrix dargestellt werden. In der Literatur wird folgender Zusammenhang beschrieben:

**(2.1)**
$$\tilde{m} = P\,\tilde{M}_w \quad \text{mit} \quad r = \sqrt{\frac{\left(\frac{b}{2}\right)^2}{\left(a^2+b^2+c^2\right)}} \quad \text{[SuB05]}$$

Dabei ist $\tilde{M}_w$ ein Punkt im 3D-Raum, angegeben in homogenen Koordinaten. $P$ bezeichnet die allgemeine Projektionsmatrix der perspektivischen Projektion. Sie beinhaltet dabei die intrinsische (auch: innere) Kameramatrix, eine Rotationsmatrix und einen Translationsvektor, mit denen die Lage einer Kamera im Weltkoordinatensystem beschrieben wird. Die Kameramatrix $A$ kann wie folgt dargestellt werden:

**(2.2)**
$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

$f_x$ und $f_y$ bezeichnen dabei einen horizontalen und vertikalen Skalierungsparameter. Dies entspricht der Brennweite einer Kamera in Pixeln. Durch $c_x$ und $c_y$ wird eine Verschiebung des sogenannten Kamerahauptpunkts durchgeführt. Mit der Kameramatrix kann eine Umrechnung von 3D-Koordinaten in Pixelkoordinaten des jeweiligen Bildes durchgeführt werden.

Der Prozess zur Ermittlung der intrinsischen und extrinsischen Daten einer Kamera wird als **Kamerakalibrierung** bezeichnet. Neben den intrinsischen Daten, welche im Wesentlichen die Kameramatrix beinhalten, definieren die extrinsischen Daten die Lage und Orientierung der Kamera in einem Koordinatensystem. Dabei handelt es sich meist um das übergeordnete **Weltkoordinatensystem** (siehe Kapitel 2.3). Eine Kamera, deren Kameramatrix und Position im Weltkoordinatensystem bekannt ist, wird als kalibrierte Kamera bezeichnet.

## 2.3 Das Weltkoordinatensystem

Das Weltkoordinatensystem stellt ein übergeordnetes Koordinatensystem dar, welches als Bezugskoordinatensystem für alle enthaltenen Objekte (z. B. Kameras) dient. Es handelt sich dabei immer um ein kartesisches Koordinatensystem mit orthogonalen Achsen.

Klar wird dies anhand des Beispiels einer Stereokamera, die zur Tiefenerkennung eingesetzt wird: Um eine Aussage über die Position von Objekten in der echten, dreidimensionalen Welt anhand eines Stereokamerasystems treffen zu können, muss sinnvollerweise ein übergreifendes Koordinatensystem definiert werden. Innerhalb eines solchen gemeinsamen Systems kann sowohl eine Platzierung der Kameras vorgenommen werden als auch die Positionen aller Objekte in Form von Koordinaten im gemeinsamen Koordinatensystem ausgedrückt werden. Die hierfür notwendigen Informationen sind Position, Orientierung und intrinsische Daten der Kameras (siehe Kapitel 2.2). Mit diesen wird bestimmt, wie die Weltkoordinaten mit den Bildkoordinaten der Kamera in Bezug zu setzen sind.

In einem Stereokamera-Aufbau liegt das Zentrum des Weltkoordinatensystems meist im Zentrum einer der beiden Kameras, wobei sich jeweils die z-Achse nach vorne, die x-Achse nach links und die y-Achse nach oben erstrecken. Die Kamera „blickt“ demnach entlang der z-Achse in positiver Richtung. Ist dieses System gegeben, kann die Position der zweiten Kamera relativ zu der ersten Kamera und damit relativ zum Weltkoordinatensystem beschrieben werden. Dies kann über die extrinsischen Daten der Kamera dargestellt werden. Sie beinhalten eine Rotationsmatrix und einen Translationsvektor.

Für das hier vorgestellte Verfahren zur Trapezkorrektur müssen drei Komponenten im Weltkoordinatensystem positioniert werden, um miteinander in Bezug gesetzt werden zu können: Die RGB-Kamera der *Kinect*, IR-Kamera der *Kinect* und der Videoprojektor (siehe auch Kapitel 2.5.1). Das hier verwendete Weltkoordinatensystem hat dabei seinen Ursprung im optischen Zentrum der *Kinect*-RGB-Kamera. Die Orientierung und Position der IR-Kamera und des Projektors werden in der vorliegenden Arbeit also relativ zur Orientierung und Position der RGB-Kamera angegeben. Die Ermittlung dieser Zusammenhänge, also die Kalibrierung der Komponenten untereinander, wird im Kapitel 2.7 näher besprochen.

## 2.4 Die Homographie-Matrix

Die planare Homographie beschreibt eine Abbildungsfunktion bzw. Transformation von Punkten einer Ebene auf eine andere Ebene (siehe [HZM03], [SuB05], [ORL08]). Genauer gesagt beschreibt eine Homographie eine Abbildung eines jeden auf einer Ebene befindlichen, in homogenen Koordinaten angegebenen 2D-Punktes mit den Koordinaten $m_{1x}$ und $m_{1y}$ auf einen 2D-Punkt einer anderen Ebene mit den Koordinaten $m_{2x}$ und $m_{2y}$. Eine Homographie beschreibt so eine perspektivische Transformation und kann in Form einer $3 \times 3$-Matrix dargestellt werden.

Der Zusammenhang zwischen den korrespondierenden Abbildungen der beiden Ebenen lautet dabei nach Schreer [SuB05]:

**(2.3)**
$$\tilde{m}_{2i} = H\,\tilde{m}_{1i}$$

$\tilde{m}_{1i}$ und $\tilde{m}_{2i}$ stellen dabei korrespondierende Punkt-Paare der Ausgangs- und Zielebene in homogenen Koordinaten dar. $H$ ist die $3 \times 3$-Homographie-Matrix.

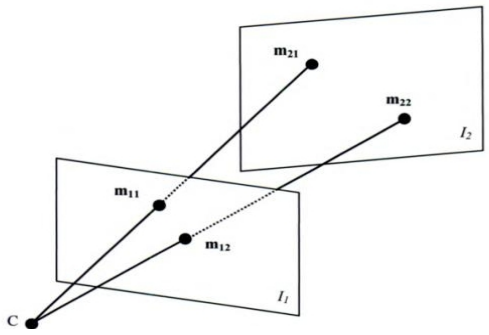


***Abbildung 2: Projektive Transformation***

***Quelle: Oliver Schreer, Stereoanalyse und Bildsynthese***

Abbildung 2 zeigt die projektive Transformation zweier Punkte von einer Ebene auf eine andere. Um die Homographie-Matrix zu ermitteln, genügen nach Schreer [SuB05] vier korrespondierende Punkte, von denen drei nicht kollinear sein dürfen.

Eine perspektivische Verzerrung eines Bildes, bei der jeder Punkt einer 2D-Ebene einen korrespondierenden Punkt auf einer anderen Ebene besitzt, kann somit durch eine Homographie beschrieben werden. Die vier Punkte, die zur Berechnung der Homographie-Matrix benötigt werden, sind im einfachsten Fall die Eckpunkte des Ursprungsbildes bzw. die gewünschten Eckpunkte des Bildes auf der Zielebene. Das Resultat einer solchen Transformation ist in Abbildung 3 dargestellt.

***Abbildung 3: Projektive Transformation durch Homographie***

Einige vorhergehende Arbeiten machen sich dies zu Nutze, indem vier Marker im Rechteck auf einer Projektionsfläche positioniert werden (siehe Kapitel 1.1). Dieses Marker-Rechteck wird von einer Kamera erfasst. Im nächsten Schritt werden diese Punkte als die vier Eckpunkte der Zielebene definiert, um die entsprechende Homographie-Matrix zu ermitteln. Die Matrix kann dann nach der Formel 2.3 auf alle weiteren Bildpunkte des Ausgangsbildes angewendet werden.

In der vorliegenden Arbeit wird ebenfalls eine Homographie-Matrix zur perspektivischen Anpassung des Ausgangsbildes angewendet. Allerdings können die Eckpunkte nicht durch Marker erfasst werden, da das System auf beliebigen Projektionsflächen funktionierten soll, ohne die Notwendigkeit, etwa Marker anbringen zu müssen. Anhand der Daten des Tiefensensors wird eine Erfassung und damit eine mathematische Beschreibung der Projektionsfläche im System möglich. Im Anschluss findet eine Berechnung der Bild-Eckpunkte statt, von denen klar ist, dass sie auf der Projektionsebene liegen müssen. Der Vorgang wird in Kapitel 3.4.2 detailliert beschrieben.

## 2.5 Der Microsoft-Kinect-Sensor

### 2.5.1 Funktionsweise und Aufbau

Der *Microsoft-Kinect*-Sensor wurde ursprünglich als Interface für die Spielkonsole *Xbox* der Firma *Microsoft* entwickelt und vertrieben. Im Gehäuse der *Kinect* sind unter anderem eine RGB-Kamera, eine IR-Kamera und ein IR-Laserprojektor untergebracht (siehe Abbildung 4). Das Gerät kann über USB mit einem Computer verbunden werden. Es überträgt neben dem Bild der RGB-Kamera eine sogenannte Tiefenmaske. Dabei handelt es sich ebenfalls um ein Videobild, welches anstatt einer Farb- oder Helligkeitsinformation eine Tiefe pro Pixel darstellt. Die übertragenen Werte repräsentieren also den Abstand oder die Entfernung eines Objekts am jeweiligen Bildpunkt der Tiefenmaske.

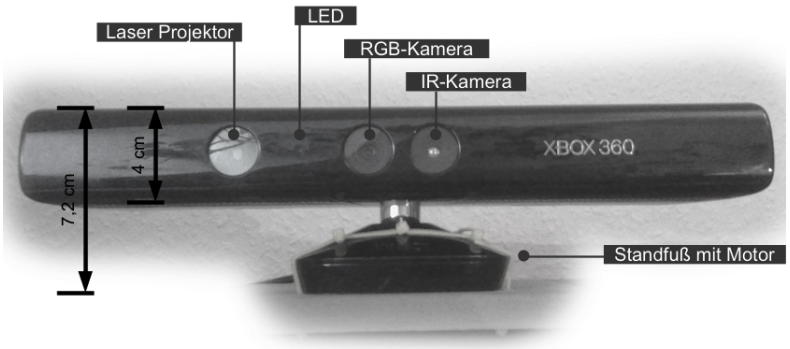


***Abbildung 4: Microsoft-Kinect-Sensor mit beschrifteten Komponenten***

Realisiert wird die Tiefenerkennung durch eine besondere Form eines sogenannten *Structured-Light*-Verfahrens [MtK12]. Hierbei wird von dem unsichtbaren IR-Laser ein bekanntes Punktemuster auf vor dem Sensor befindliche Objekte projiziert. Diese reflektieren das IR-Licht, welches von der ebenfalls im Gehäuse integrierten IR-Kamera erfasst wird. Die Videobilder des Punktemusters werden dann an einen internen Videoprozessor weitergegeben, der das aufgenommene Bild mit einem Referenzbild vergleicht. Dabei sucht ein Algorithmus nach Übereinstimmungen bzw. Unterschieden zwischen dem aufgenommenen Bild und dem Referenzbild. Werden Teile des projizierten Musters erkannt, so wird in einem nächsten Schritt die horizontale Verschiebung dieses Musterabschnitts bezogen auf das Referenzbild bestimmt. Anhand dieser Verschiebung und unter Kenntnis des Basisabstandes zwischen Projektor und Kamera können für einzelne Bildpunkte die Abstände errechnet werden. Ein solches Verfahren,

bei dem über Verschiebungen zwischen zwei Bildpunkten bei bekannter Basis auf eine Entfernung geschlossen wird, wird als **optische Triangulation** bezeichnet.

Die kodierten Tiefendaten werden dann in Form einer Tiefenmaske mit einer Wiederholrate von 30 Hz an den Computer übertragen. Mit der *Microsoft-Kinect* ist somit erstmals ein preisgünstiges Gerät auf dem Markt, mit dem Tiefenerkennung in Echtzeit und in hoher Auflösung ermöglicht wird (siehe auch [MtK12], [Orr11]). Mit einem Kaufpreis von ca. 100 € für die *Xbox*-Version der *Kinect* und bis zu 250 € für die offizielle PC-Version haben auch Privatleute und kleine Unternehmen die Möglichkeit mit dieser Technologie zu arbeiten.

### 2.5.2 Umrechnung der Tiefendaten in Weltkoordinaten

Der Datenstrom für die Tiefenmaske der *Kinect* (siehe Kapitel 2.5.1) beinhaltet die Tiefendaten als RAW-Daten, deren Verwendung jedoch einer weiteren Verarbeitung bedarf. Diese Verarbeitung oder Umrechnung geschieht entweder auf niedriger Ebene und wird beispielsweise vom Treiber übernommen oder muss in der jeweiligen Software vorgenommen werden. Bei den RAW-Werten handelt es sich um Datenwörter mit einer Wortlänge von 11 Bit, was theoretisch 2048 Abstufungen für die Entfernung ermöglicht. Tatsächlich wird aber nur ein kleiner Bereich dieser Werte genutzt. Zudem handelt es sich um eine nicht-lineare Verteilung der Werte, was auf das Verfahren der optischen Triangulation zurückzuführen ist (siehe auch Abbildung 30 auf Seite 69). Mit steigender Entfernung im Verhältnis zur Triangulationsbasis werden die Unterschiede der Verschiebung auf dem IR-Sensorchip bei gleichbleibender Differenz zweier Abstände zunehmend geringer. Das bedeutet, dass die Genauigkeit der Tiefenerkennung, bedingt durch die Triangulationsbasis und die Auflösung des IR-Sensorchips, mit steigender Entfernung abnimmt. Die Triangulationsbasis wird bei der *Kinect* durch den Abstand zwischen Projektor und IR-Kamera definiert.

Um nun die *Kinect*-Tiefeninformation verwenden zu können, muss eine Umrechnung der RAW-Daten in Meter stattfinden. Da jeder *Kinect*-Sensor produktionsbedingt leichte bauliche Unterschiede aufweist, müsste für sehr genaue Werte eine separate Kalibrierung durchgeführt werden. Es stehen jedoch bereits mehrere Näherungsformeln zur Verfügung, die sich in vielen Fällen als ausreichend erweisen. Dieser Kalibrierungsschritt kann dadurch ausgelassen werden. In der vorliegenden Arbeit kommt die Näherung (2.4) zum Einsatz. Die Abweichung zwischen gemessenem und einem mit dieser Formel berechnetem Wert liegt bei einer Entfernung von 4 m etwa bei 1,4 cm [Orr11].

**(2.4)**
$$z[m] = \frac{1.0}{RAWdepth * \ -0.0030711016 + 3.330949516} \quad \text{[Fis12]}$$

Das Ergebnis der Berechnung $z$ ist der gemessene Abstand des jeweiligen Bildpunktes und $RAWdepth$ der Wert, welcher von der *Kinect* an diesem Bildpunkt übertragen wurde.

Anhand den von der *Kinect* übertragenen Tiefenwerte ist es möglich, die 3D-Koordinaten eines jeden Punktes der Tiefenmaske zu bestimmen. Um diese Punktewolke erzeugen zu können, müssen jedoch die intrinsischen Kameraparameter der IR-Kamera bekannt sein. Diese Daten können anhand einer Kamerakalibrierung bestimmt werden, wie sie etwa durch das Projekt *RGBDemo* vorgestellt wird [HtK12]. Laut den Untersuchungen von K. Konolige und P. Mihelich für *ROS* (*Robot Operating System*) sind die Kameras der *Kinect* im Vergleich zu anderen kostengünstigen Kamerasystemen gut verarbeitet [ROS11]. Daher sind Abweichungen der intrinsischen Kameraparameter zwischen mehreren *Kinect*-Geräten relativ gering und so kann auch hier auf allgemeine Näherungen zurückgegriffen werden. Folgende Werte werden in der vorliegenden Arbeit verwendet:

$$Brennweite: \ f_x = 594.2143, \ f_y = 591.0405$$

$$Kamerahauptpunkt: \ c_x = 339.3078, \ c_y = 242.7391$$

Somit lautet die vollständige Kameramatrix der IR-Kamera nach (2.2) wie folgt:

**(2.5)**
$$A_{IR} = \begin{bmatrix} 594.2143 & 0 & 339.3078 \\ 0 & 591.0405 & 242.7391 \\ 0 & 0 & 1 \end{bmatrix}$$

Anhand dieser Matrix kann eine Umrechnung der Werte in 3D-Weltkoordinaten erfolgen. Zusammen mit dem in (2.4) errechneten Abstandswert gilt somit für einen Punkt $\tilde{M}_w$ der Punktewolke:

**(2.6)**
$$\tilde{M}_w = A_{ir} \begin{pmatrix} u \\ v \\ z \end{pmatrix}$$

Dabei sind *u* und *v* Pixelkoordinaten der Tiefenmaske und *z* der Abstandswert in Metern (siehe Formel (2.4)).

### 2.5.3 Kalibrierung der Kinect-Kameras

Die *Kinect* beinhaltet neben dem Tiefensensor eine gewöhnliche RGB-Kamera, welche ebenfalls über eine Auflösung von 640 × 480 Bildpunkten verfügt. In vielen Fällen, so auch in der vorliegenden Arbeit, ist es erwünscht, die Bilder der RGB-Kamera und Entfernungswerte der Tiefenmaske miteinander zu kombinieren. Da es sich jedoch um zwei getrennte Kameras handelt, müssen diese aufeinander angepasst, das heißt also kalibriert werden. Die Verschiebung zwischen den Kameras scheint zwar gering, kann aber je nach Anwendung zu ungenauen Ergebnissen führen. Zudem sind Öffnungswinkel und Linseneigenschaften der Kameras Faktoren, die beachtet werden müssen. Abbildung 5 zeigt, dass die Bildausschnitte der IR-Kamera und die der RGB-Kamera nicht vollständig miteinander übereinstimmen, wenn man den Abstand des Bilderrahmens zum Rand des jeweiligen Kamerabildes betrachtet.

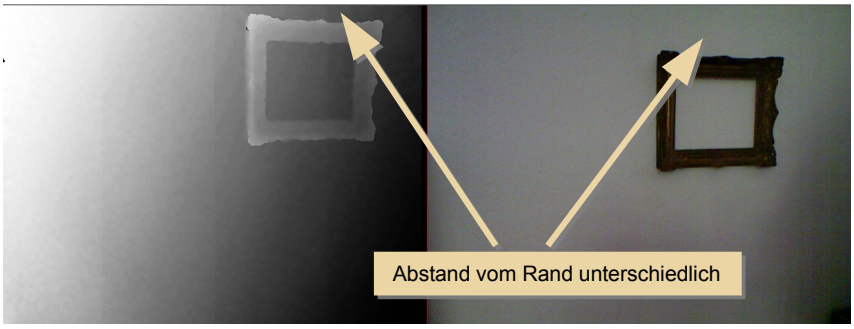


***Abbildung 5: Kinect-Tiefenmaske und RGB-Bild unkalibriert***

Im hier beschriebenen Verfahren ist eine Übereinstimmung der Tiefenmaske und der Bildpunkte der RGB-Kamera notwendig (siehe Kapitel 2.7). Um die Bildpunkte der Tiefenkamera und damit die entsprechenden Entfernungswerte auf die RGB-Kamera anpassen zu können, müssen die *Kinect*-Kameras aufeinander kalibriert werden. Dies führt zu einer deutlich erkennbaren Verzerrung des IR-Bildes und zeigt damit, wie unterschiedlich die Bildausschnitte eigentlich sind (siehe Abbildung 6). Vergleicht man nach der Anpassung den Abstand zum Rand noch einmal, so sind die Positionen des Bilderrahmens nun auf beiden Kamerabildern identisch.

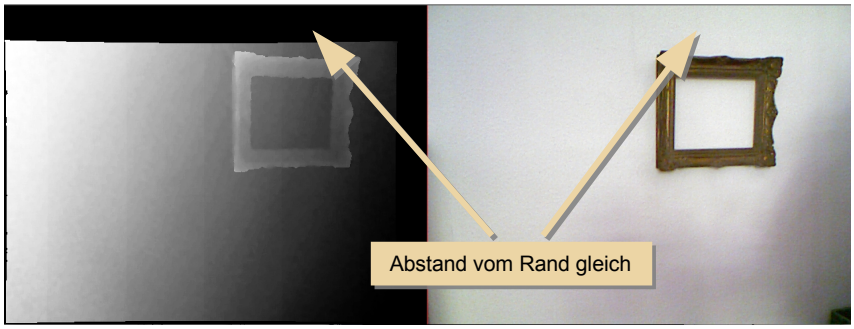


***Abbildung 6: Kinect-Tiefenmaske und RGB-Bild kalibriert***

## 2.6 MaxMSP/Jitter

Die im Rahmen dieser Arbeit entwickelte Software *C_stone* wurde mit *MaxMSP/Jitter* (auch: *MaxMSP*) entwickelt (Version 5.1.8). Dabei handelt es sich um eine leistungsfähige und flexible Programmierumgebung, welche für objektorientierte Programmierung in grafischer Form (VPL) ausgelegt ist. Die kostenpflichtige Programmierumgebung wird vom Hersteller *Cycling74* angeboten [Max11].

Im Gegensatz zu vielen textuellen Programmiersprachen ermöglicht die Entwicklung in *MaxMSP* eine sehr schnelle und intuitive Herangehensweise an Softwareentwicklung. Ein *MaxMSP*-Programm, auch **Patcher** genannt, muss nicht kompiliert werden, sondern wird ununterbrochen ausgeführt und kann zur Laufzeit beliebig verändert werden. Dabei kommunizieren Funktionsblöcke bzw. *Max*-Objekte innerhalb eines patchers über Nachrichten, sogenannte **Messages**. Diese werden zwischen Objekten meist über **patch cords** (deutsch: Verbindungskabel) geleitet, die im Patcher visuell als Linien zwischen den Objekten dargestellt werden. So können sogenannte **Outlets** einzelner Objekte mit den **Inlets** anderer Objekte verbunden werden. Ein Objekt in *MaxMSP* kann über ein Outlet Messages an andere Objekte weitergeben, respektive über seine Inlets Messages empfangen. Ein *Max*-Patcher, der zusammengehörige Objekte beinhaltet, kann als sogenannte **Abstraction** gespeichert oder als **Subpatcher** zusammengefasst werden. Innerhalb eines anderen Patchers können wiederum Abstractions als eigenständiges Objekt geladen werden. Diese Art der Enkapsulierung von Objektgruppen, die eine gemeinsame Funktion erfüllen, ist bei komplexeren *Max*-Programmen essentiell, um eine übersichtliche Programmierung zu erzielen.

Viele VPLs bieten die Möglichkeit, neben der grafischen Programmierung, Code in Schriftform in die Programme zu integrieren. Dies kann im Falle von *MaxMSP* auf mehrere Arten erfolgen.

Das *js*-Object etwa stellt ein Interface zur Einbindung von *JavaScript*-Dateien zur Verfügung. Dabei können aus *JavaScript* heraus fast alle Funktionen der *Max*-Umgebung genutzt werden, *Max*-Objekte erzeugt werden oder Messages an bereits vorhandene Objekte in einem *Max*-Patcher geschickt werden.

Für diese Arbeit ist insbesondere eine Erweiterungsmöglichkeit von *MaxMSP* interessant: Das sogenannte *Max-SDK* ist ein Software-Entwicklungs-Kit, mit dem Nutzern ermöglicht wird, über eine vorgegebene Plugin-Struktur eigene *Max*-Objekte in *C* zu entwickeln und in die Umgebung zu integrieren. Diese Objekte werden als **Externals** bezeichnet. Die Entwicklung in *C* erlaubt dabei die Einbindung anderer Bibliotheken und Funktionen, was nahezu unbegrenzte Möglichkeiten bei der Erweiterung der M*axMSP*-Plattform bietet. Das *Max-SDK* wird vom Programmhersteller kostenlos zur Verfügung gestellt.

### 2.6.1 JavaScript in Max

*MaxMSP* bietet mit dem *js*-Objekt eine unkomplizierte Möglichkeit, *JavaScript*-Dateien in die *Max*-Umgebung zu integrieren. Einige Aufgaben lassen sich in textuellen Programmiersprachen schneller oder effektiver lösen als in VPLs. Dabei wird dem Entwickler ermöglicht, über bestimmte Befehle direkt mit der *Max*-Umgebung zu kommunizieren. Auf diese Weise kann beispielsweise *Max*-Objekten explizit Nachrichten zugesendet werden oder gar neue Objekte in einem Patcher erzeugt werden.

Da es sich bei *JavaScript* um eine interpretierte Sprache handelt, die nicht kompiliert werden muss, kann eine *JavaScript*-Datei mit sofortiger Wirkung im *Max*-internen oder externen Texteditor verändert werden. Dies ermöglicht eine sehr unkomplizierte Fehlersuche und schnelle Entwicklung. Besonderheiten zur *JavaScript*-Programmierung in *MaxMSP* können der *MaxMSP*-Online-Dokumentation entnommen werden [MOD12].

### 2.6.2 OpenGL in Max

Die *Jitter*-Objektfamilie in *MaxMSP* bietet eine Reihe von Objekten, die von der *OpenGL*-API Gebrauch machen. *OpenGL* kann als Softwareschnittstelle zur Grafikhardware eines Computers verstanden werden und ermöglicht ein effektives, hardwarebasiertes 3D-Rendering [OGL04].

Alle Objekte in *MaxMSP*, die mit der *OpenGL*-API kommunizieren, sind durch das Präfix *jit.gl* in ihrer Namensgebung gekennzeichnet. *jit.gl*-Objekte erlauben den Zugriff auf eine Vielzahl

der Funktionen, die von der *OpenGL*-API bereitgestellt werden. Dies macht *MaxMSP* zu einem mächtigen Werkzeug für 3D-Grafik und hardwarebasiertes 3D-Rendering.

Um innerhalb von *MaxMSP* mit *OpenGL* kommunizieren zu können, muss ein sogenannter *OpenGL-Kontext* aufgebaut werden. Jeder *OpenGL-Kontext* hat einen individuellen Namen und beinhaltet ein *jit.gl.render*-Objekt und ein Ziel-Objekt, an welches der Renderer das soeben berechnete Frame weiterreichen kann. Meist handelt es sich dabei um eine Instanz eines *jit.window*- oder *jit.pwindow*-Objekts, mit dem der Inhalt des Render-Kontexts auf dem Bildschirm sichtbar gemacht werden kann. Zusätzlich wird ein Taktgeber benötigt, der dem Renderer den Befehl zur Berechnung eines neuen Frames erteilt. Dadurch kann die Wiederholrate für den Renderer bestimmt werden. Meist kommt hierfür das sogenannte *qmetro*-Objekt zum Einsatz.

### 2.6.3 OpenCV in Max

*OpenCV* ist eine von *Intel* entwickelte Programmbibliothek, welche leistungsfähige Algorithmen zur Bildverarbeitung und für maschinelles Sehen zur Verfügung stellt [OCV12]. In der Software *C_stone* werden insbesondere bei der Kalibrierung von Projektor und Tiefenkamera *OpenCV*-Algorithmen eingesetzt (siehe Kapitel 2.7). Hierfür wurden mehrere *OpenCV*-*C*-Funktionen in *Max*-Externals implementiert und können so in *MaxMSP* eingebunden werden. Die Entwicklung der Externals wurde durch das *cv.jit*-Projekt von Jean-Marc Pelletier ermöglicht, wodurch wichtige Grundfunktionen, wie etwa die Umwandlung von *Jitter*-Matritzen zu *OpenCV*-Matritzen verfügbar gemacht wurden [CVJ11].

Für die Implementierung der Externals kam Version 5.1.7 des *Max-SDK* zum Einsatz. Als Basis wurde *cv.jit* in der Version 1.7.2 verwendet. Die Programmierung erfolgte in *C* und ist mit der *Mac-OSX*-Entwicklungsumgebung *Xcode* 3.2.6 durchgeführt worden.

Da eine ausführliche Beschreibung aller Implementierungen den Rahmen dieser Arbeit sprengen würde, soll lediglich auf die Funktionen der entstandenen Externals und wie diese in *MaxMSP* verwendet werden können eingegangen werden. Der Quellcode des Externals *calibrateCam* ist beispielhaft im Anhang ab Seite 100 vorzufinden. Weitere Informationen über die Entwicklung von *MaxMSP*-Externals in *C* können der Dokumentation des *Max-SDK* entnommen werden [Max11].

#### 2.6.3.1 OpenCV-Max-Externals für C_stone

**calibrateCam**

Enthaltene *OpenCV*-Funktion: ***cvCalibrateCamera2()***

Das External *calibrateCam* wird zur Kamerakalibrierung eingesetzt. Die zugrundeliegende *OpenCV*-Funktion schätzt intrinsische und extrinsische Kameraparameter für eine oder mehrere Ansichten eines Kalibrierungsziels und gibt die Ergebnisse über die Outlets 1–4 an die *Max*-Umgebung zurück (siehe Tabelle 2.1). Wichtig ist, dass vor einer Kalibrierung die Auflösung der zu kalibrierenden Kamera über das *imgSize* Attribut definiert wird (*imgSize* steht für *image size*). *calibrateCam* erwartet eine Tabelle (*Jitter*-Matrix) mit den 3D-Koordinaten eines Objekts (inlet 1) und eine, welche die korrespondierenden 2D-Koordinaten der Projektion enthält (inlet 2). Als Kalibrierungsziel dient dabei meist ein planares Schachbrettmuster, dessen Eckpunkte leicht mithilfe der *OpenCV*-Funktion *findChessboardCorners()* erkannt werden können. Dabei werden die bekannten Eckpunkte des planaren Musters als **Objekt-Koordinaten** und die im Kamerabild ermittelten Eckpunkte als **Bild-Koordinaten** an *calibrateCam* übergeben.

| **Inlets** | **Typ** | **Beschreibung** |
|---|---|---|
| 1 | jit.mat: 1, float32, 3 × N | *Objekt-Koordinaten* |
| 2 | jit.mat: 1, float32, 2 × N | *Bild-Koordinaten* |
| 3 | jit.mat: 1, float32, 1 × M | Anzahl der *Objekt-* und *Bild-Koordinaten* pro Aufnahme |
| **Outlets** | **Typ** | **Beschreibung** |
| 1 | jit.mat: 1, float32, 3 × 3 | Intrinsische Kameramatrix |
| 2 | jit.mat: 1, float32, 1 × 5 | Linsen-Verzerrungs-Koeffizienten |
| 3 | jit.mat: 1, float32, 3 × M | Rotationsvektoren pro Aufnahme |
| 4 | jit.mat: 1, float32, 3 × M | Translationsvektoren pro Aufnahme |
| 5 | dump_out | Messages von Jitter-MOP (Matrix-Operator) |
| **Attribute** | **Beschreibung** | |
| imgSize | Auflösung des Kamerabildes in Pixel | |

*Tabelle 2.1: calibrateCam-External Übersicht*

In der Software *C_stone* wird das *calibrateCam*-External anders eingesetzt als bei der gewöhnlichen Kamerakalibrierung. Wo normalerweise die 2D-Koordinaten der detektierten Schachbrettecken (siehe auch Seite 23 oben) verwendet werden, werden hier 3D-Punkte, die aus der *Kinect*-Tiefenmaske gewonnen wurden, als Objekt-Koordinaten übergeben. Als Bild-Koordinaten werden die Eckpunkt-Koordinaten auf der Projektor-Bildebene eines generierten Schachbrettmusters übergeben (siehe Kapitel 2.7). Dies ist ein Vorgehen, wie es etwa bei der Kamerakalibrierung mit einem dreidimensionalen Kalibrierungsziel vorkommt.

**findChessCorners**

Enthaltene *OpenCV*-Funktionen: **cvFindChessboardCorners(), cvFindCornerSubPix(), cvDrawChessboardCorners()**

Der im External *findChessCorners* enthaltene Algorithmus durchsucht ein Schwarzweißbild nach einem Schachbrettmuster. Werden Ecken eines Schachbretts erkannt, so werden die Pixelkoordinaten dieser Ecken in Form einer *Jitter*-Matrix aus dem zweiten Outlet ausgegeben (siehe Tabelle 2.2). Zudem fügt das External dem an Inlet 1 übergebenen Schwarzweißbild eine farbliche Markierung der gefundenen Eckpunkte hinzu. Das resultierende Bild kann als optisches Feedback genutzt werden, aus dem deutlich wird, ob die Schachbrett-Ecken erkannt wurden oder nicht (siehe Abbildung 7). Die Anzahl der Eckpunkte des verwendeten Schachbrettmusters wird über das Attribut *chessBoardSize* eingestellt.

| **Inlets** | **Typ** | **Beschreibung** |
|---|---|---|
| 1 | jit.mat: 1, char, N × M | zu analysierendes Schwarzweißbild |
| **Outlets** | **Typ** | **Beschreibung** |
| 1 | jit.mat: 4, char, N × M | Schwarzweißbild mit farbiger Kennzeichnung der Ecken |
| 2 | jit.mat: 1, float32, 2 × K | Pixelkoordinaten der gefundenen Ecken |
| 3 | dump_out | Messages von Jitter-MOP (Matrix-Operator) |
| **Attribute** | **Beschreibung** | |
| chessBoardSize | Anzahl der Ecken des gesuchten Schachbrettmusters | |

*Tabelle 2.2: findChessCorners-External Übersicht*

Anmerkung zu Schachbrettmustern

Ein Schachbrettmuster wird häufig als Referenzmuster verwendet, da es für Bilderkennungsalgorithmen leicht identifizierbare Elemente aufweist. Als Ecken eines Schachbrettmusters werden die Stellen des Musters bezeichnet, an dem sich zwei schwarze Felder berühren. Diese Ecken werden von *findChessCorners* ermittelt und deren Koordinaten ausgegeben.

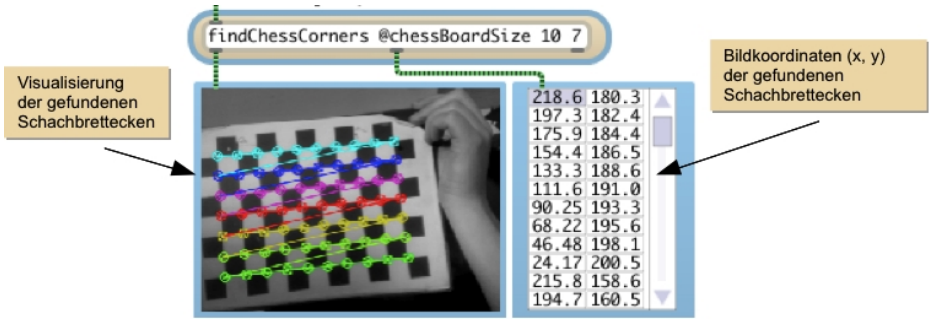


***Abbildung 7: findChessCorners-External mit Schachbrett-Visualisierung und Koordinaten***

## projectPoints

Enthaltene *OpenCV*-Funktion: **cvProjectPoints2()**

Mit *projectPoints* können die korrespondierenden 2D-Punkte einer Matrix mit 3D-Punkten auf der Kamera-Bildebene ermittelt werden. Dabei müssen an das External die Kameradaten (Inlet 2–4) übergeben werden (siehe Tabelle 2.3). Anschließend kann über das erste Inlet eine Matrix mit 3D-Punkten an das External übergeben werden. Die resultierenden 2D-Punkte werden über das erste Outlet zurückgegeben.

| **Inlets** | **Typ** | **Beschreibung** |
|---|---|---|
| 1 | jit.mat: 1, float32, 3 × N | Zu projizierende 3D-Koordinaten |
| 2 | jit.mat: 1, float32, 3 × 1 | Rotationsvektor |
| 3 | jit.mat: 1, float32, 3 × 1 | Translationsvektor |
| 4 | jit.mat: 1, float32, 3 × 3 | Kameramatrix |
| **Outlets** | **Typ** | **Beschreibung** |
| 1 | jit.mat: 1, float32, 2 × N | Korrespondierende 2D-Koordinaten auf der Bildebene |
| 2 | dump_out | Messages von Jitter-MOP (Matrix-Operator) |

*Tabelle 2.3: projectPoints-External Übersicht*

**rodrigues**

Enthaltene *OpenCV*-Funktion: **cvRodrigues2()**

Mit *rodrigues* können Rotationsvektoren der Form 3 × 1 in eine Rotationsmatrix umgewandelt werden (siehe Tabelle 2.4). Ebenso errechnet die Funktion den entsprechenden Rotationsvektor, wenn ihr eine 3 × 3-Rotationsmatrix übergeben wird. Die Erkennung, ob eine Matrix oder ein Vektor übergeben wurde, erfolgt automatisch.

| **Inlets** | **Typ** | **Beschreibung** |
|---|---|---|
| 1 | jit.mat: 1, float32, 3 × 1<br>oder<br>jit.mat: 1, float32, 3 × 3 | Rotationsvektor<br>oder<br>Rotationsmatrix |
| **Outlets** | **Typ** | **Beschreibung** |
| 1 | jit.mat: 1, float32, 3 × 3<br>oder<br>jit.mat: 1, float32, 3 × 1 | Je nach Matrixtyp des Inlet 1,<br>Rotationsmatrix oder Rotationsvektor |
| 2 | dump_out | Messages von Jitter-MOP (Matrix-Operator) |

*Tabelle 2.4: rodrigues-External Übersicht*

## 2.7 Kalibrierung Projektor – Kinect

Entscheidend für die vorliegende Arbeit ist die Annahme, dass die Projektionseigenschaften von Videoprojektoren mit denselben mathematischen Modellen wie bei einer Kamera beschrieben werden können (siehe Kapitel 2.2). Die Richtung der Lichtstrahlen, die bei der Kamera auf eine Sensorebene fallen, respektive bei einem Projektor auf einer Ebene, etwa einem LCD-Bildelement entstehen, hat demnach für die theoretische Darstellung keine Bedeutung. Dieses Überlegung wurde auch in Arbeiten wie etwa von Martynov et al. [Mar11] und Kimura et al. [Kim10] genutzt, um Verfahren zur Kalibrierung von Projektoren anzuwenden. Sie ermitteln durch eine inverse Kalibrierung die intrinsische Matrix und die extrinsischen Parameter eines Projektors. Mit diesen Daten ist es möglich, die zu Punkten im 3D-Raum korrespondierenden 2D-Punkte auf der Bildebene des Projektors zu ermitteln.

Wie bereits in den vorhergehenden Kapiteln angesprochen, handelt es sich beim hier präsentierten Aufbau um ein System mit mehren Komponenten. Projektor und *Kinect* befinden

sich physikalisch nicht am selben Ort sondern „blicken“ aus unterschiedlichen Perspektiven auf eine gemeinsame Szene. Es kann mathematisch dargestellt werden, wie ein Projektor eine von der *Kinect* aufgenommene Szene sehen würde, wenn an seiner Stelle eine Kamera mit den gleichen optischen Eigenschaften wäre. Bei der Projektor-*Kinect*-Kalibrierung müssen also die intrinsischen und extrinsischen Parameter des Projektors ermittelt werden.

Sind diese Daten bekannt, so wird es möglich, für jeden 3D-Punkt des Weltkoordinatensystems auf die Pixelkoordinaten des korrespondierenden 2D-Punkts der Bildebene des Projektors zu schließen. Anders ausgedrückt kann für jeden 3D-Punkt im Weltkoordinatensystem derjenige Pixel auf der Bildebene des Projektors ermittelt werden, durch welchen er angestrahlt würde. Das Weltkoordinatensystem hat dabei seinen Ursprung im optischen Zentrum der RGB-Kamera des *Kinect*-Sensors. Die Koordinatensysteme der IR-Kamera und des Projektors beziehen sich auf das durch die RGB-Kamera definierte Weltkoordinatensystem (siehe Kapitel 2.3).

Die Kalibrierung von Projektoren steht jedoch vor einem grundsätzlichen Problem: Ein Projektor kann, anders als eine Kamera, keinerlei Informationen aufnehmen. Bei der Kalibrierung müssen also zusätzliche Geräte (z. B. Kameras, Sensoren) eingesetzt werden, die diese Aufgabe übernehmen. Soll der Projektor zur IR-Kamera der *Kinect* kalibriert werden, kommen weitere Schwierigkeiten hinzu. Es ist nicht möglich, mit Videoprojektoren Signale im Infrarotbereich zu erzeugen. Es muss daher ein „Umweg“ über das Bild der RGB-Kamera der *Kinect* gegangen werden (siehe Abbildung 8). Da die RGB-Kamera die Signale des Projektors aufnehmen kann (etwa ein projiziertes Bild), wird prinzipiell ermöglicht, korrespondierende Punkte zwischen der Projektor-Bildebene und der durch die *Kinect* gegebenen 3D-Punkte zu bestimmen. Soll also ein Projektor auf die IR-Kamera der *Kinect* kalibriert werden, setzt dies beim hier vorgestellten Verfahren voraus, dass vorangehend die IR-Kamera und RGB-Kamera zueinander kalibriert wurden (siehe Kapitel 2.5.3).

Die Implementierung der Projektor-*Kinect*-Kalibrierung basiert auf einem Verfahren von Elliot Woods, welches in der Programmierumgebung *vvvv* umgesetzt wurde [Woo11]. Grundsätzlicher Gedanke dabei ist, den Projektor mithilfe der *cvCalibrateCamera2*-Funktion der *OpenCV*-Bibliothek auf ein 3D-Objekt zu kalibrieren [OCV12]. Die 3D-Punkte hierfür können aus der *Kinect*-Tiefenmaske errechnet werden und im Anschluss mit bestimmten Punkten auf der 2D-Bildebene des Projektors in Zusammenhang gebracht werden.

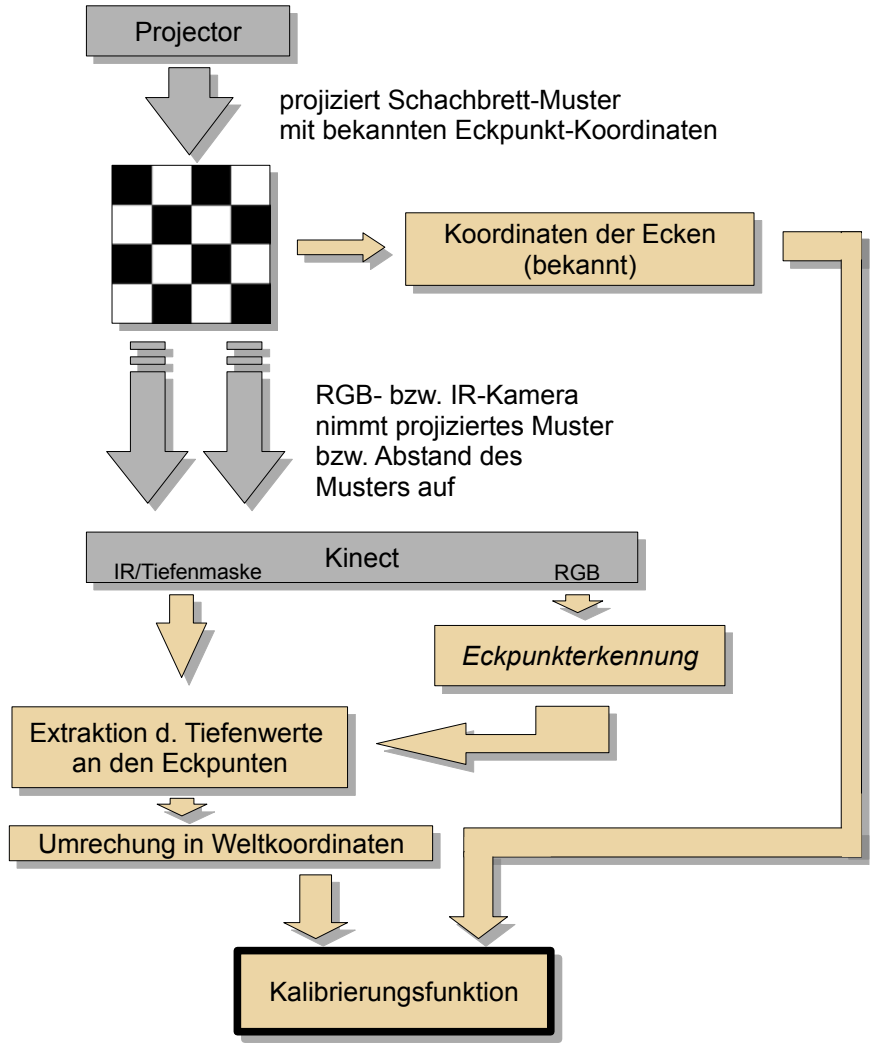


***Abbildung 8: Schematische Darstellung des Projektor-Kinect-Kalibrierungsverfahrens***

Zur Kalibrierung wird ein Schachbrettmuster vom Projektor ausgegeben (siehe Abbildung 8). Da dieses in der Software generiert wird, sind Pixelkoordinaten der Eckpunkte auf der Projektor-Bildebene bekannt. Wird nun ein *Kinect*-Sensor auf die Projektion gerichtet, so können leicht im RGB-Bild der *Kinect* die Eckpunkte des Schachbretts gefunden werden (siehe *findChessCorners* in Kapitel 2.6.3.1). Sind die Eckpunkte im RGB-Bild ermittelt, so können die entsprechenden Werte der Tiefenmaske an diesen Koordinaten extrahiert und deren Weltkoordinaten ermittelt werden. Diese Werte werden bei der Kalibrierung als *Objekt-Koordinaten* bezeichnet. Die Pixelkoordinaten auf der Projektor-Bildebene tragen hingegen die Bezeichnung *Bild-Koordinaten*. Werden nun eine ausreichende Anzahl zusammengehöriger *Objekt-* und *Bild-Koordinaten* erfasst, können Kameramatrix, Rotations- und Translationsvektor mit der *OpenCV*-Funktion *cvCalibrateCamera2()* bestimmt werden (siehe Kapitel 2.6.3.1).

Da zur Kalibrierung viele Korrelationspunkte zwischen der Projektor-Bildebene und Punkten im Weltkoordinatensystem notwendig sind, müssen mehrere Aufnahmen des Schachbretts in unterschiedlichen Auslenkungen gemacht werden. Daher sollte auf eine bewegliche (weiße) Fläche projiziert werden, deren Abstand zum Projektor und der *Kinect* zwischen einzelnen Aufnahmen verändert werden muss. Dieser Vorgang ist vergleichbar mit einem „Abscannen“ eines 3D-Raumes. Alternativ kann der Projektor gemeinsam mit der *Kinect* bewegt werden, um so unterschiedliche Aufnahmen des Schachbretts etwa auf einer Wand zu erreichen. Hierfür sollte die *Kinect* fest auf dem Projektor installiert sein. Beide Geräte dürfen in ihrer Position zueinander während und nach der Kalibrierung nicht verändert werden.

# Kapitel 3: Umsetzung

Im Folgenden wird auf die Umsetzung der in Kapitel 2 besprochenen Grundlagen besprochen. Dabei wird detailliert auf einzelne Abschnitte der *MaxMSP*-Implementierung, durch die eine Realisation der automatischen dynamischen Trapezkorrektur möglich wird, eingegangen.

## 3.1 Kurzübersicht der Hauptbereiche von C_stone

Die interne Struktur der Software *C_stone* lässt sich in fünf Hauptbereiche aufteilen, die in Abbildung 9 schematisch dargestellt sind. Eine Gesamtübersicht des Haupt-*Max*-Patchers in Form eines Screenshots kann im Anhang auf Seite 86 eingesehen werden.

***Input***

Dieser Bereich umfasst die Kommunikation mit dem *Kinect*-Sensor. Hier werden die Tiefenmaske und das RGB-Bild vom Treiber empfangen und an die Software weitergegeben. (Siehe Kapitel 3.2.)

***Calibration***

Der *Calibration*-Bereich gehört zum Programmkern von *C_Stone.* Hier finden alle nötigen Berechnungen für die Kalibrierung zwischen Projektor und *Kinect* statt. Die Objekte dieses Bereiches sind nur aktiv, wenn vom Benutzer in der GUI der *calibration mode* gestartet wurde. Nach der erfolgreichen Kalibrierung werden die ermittelten Daten an den *Runtime*-Bereich weitergegeben, wo die Berechnungen für die eigentliche Trapezkorrektur stattfinden. (Siehe Kapitel 3.3.)

***Runtime***

Der *Runtime*-Bereich gehört ebenfalls zum Programmkern von *C_stone*. Dieser Modus ist im Normalbetrieb der Software aktiv, da hier die Berechnungen zur Trapezkorrektur stattfinden. Am Ende der Prozesskette wird eine Homographie-Matrix ermittelt, die zur Verzerrung des Ausgangsbildes notwendig ist. Diese Matrix wird als Ergebnis der Berechnungen dem *Output*-Bereich übergeben. (Siehe Kapitel 3.4.)

### *Output*

Im Abschnitt *Output* findet die Verarbeitung der Videodaten statt, die vom Projektor ausgegeben werden sollen. Anhand der im *Runtime*-Abschnitt ermittelten Matrix wird hier eine Transformation des Projektor-Ausgangsbildes zur perspektivischen Korrektur durchgeführt. (Siehe Kapitel 3.5.)

### *GUI*

Als letztes Element der Software ist die grafische Benutzeroberfläche zu erwähnen. Hier sind alle wichtigen Parameter des Softwareprototyps *C_stone* zusammengefasst und benutzerfreundlich visualisiert. Zudem wird hier dem Nutzer eine Überwachung des Videoausgangssignals anhand eines kleinen Vorschaubereichs ermöglicht. (Siehe Kapitel 3.6.)

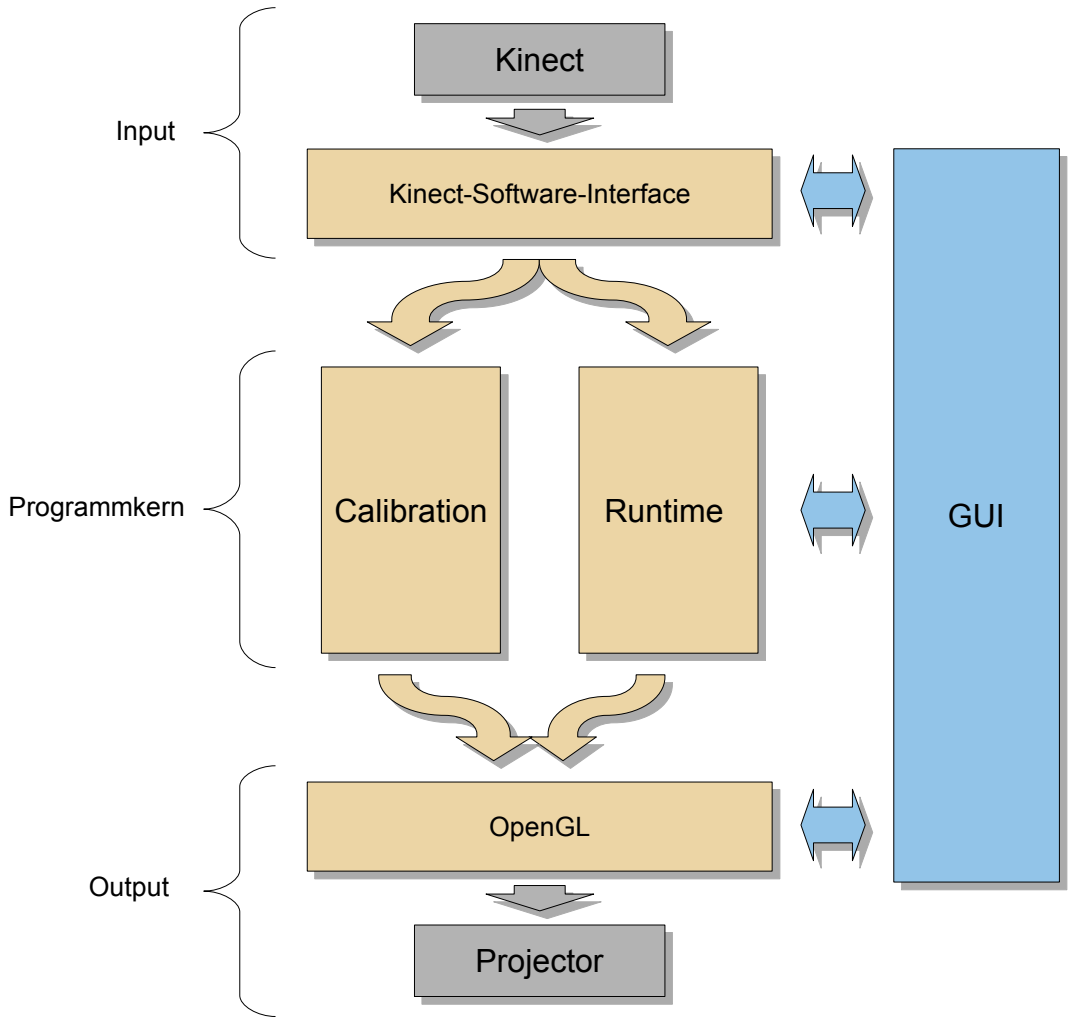


***Abbildung 9: C_stone Programmstruktur***

## 3.2 Input

Der *Input*-Bereich der Software *C_stone* umfasst die Kommunikation mit dem *Microsoft-Kinect*-Sensor (siehe Kapitel 3.2.1). Hier werden die Tiefenmaske und der RGB-Datenstrom vom Sensor empfangen und an die *MaxMSP*-Umgebung weitergegeben. Danach finden die erforderlichen Berechnungen zur Kalibrierung der *Kinect*-Kameras untereinander statt (siehe Kapitel 2.5.3). Hierfür wird die Tiefenmaske an die Abstraction *map_IR_to_RGB* weitergegeben (siehe Anhang Seite 88). Die Berechnungen der Abstraction benötigen viel Rechenleistung, was beim Betrieb auf älteren oder leistungsschwächeren Computersystemen zu Problemen führen kann. Aus diesem Grund kann die Berechnung vom Nutzer deaktiviert werden (siehe Kapitel 3.6.1). Die Genauigkeit des Systems ist in den meisten Fällen auch ohne diese Berechnungen ausreichend.

### 3.2.1 Jit.freenect.grab-External

Die Kommunikation mit dem Sensor wird durch das *jit.freenect.grab*-External für *MaxMSP* von Jean-Marc Pelletier ermöglicht [JFR12]. Hierbei handelt es sich um ein External, welches auf Basis des von der *OpenKinect*-Community entwickelten *Freenect*-Treibers mit dem *Microsoft-Kinect*-Sensor kommunizieren kann [OKI12]. Das External liefert sowohl die RAW-Tiefenmaske als auch einen Videodatenstrom der RGB-Kamera. Da bei der Nutzung von *jit.freenect.grab* keinerlei Aufbereitung der *Kinect*-Datenströme auf Treiberebene stattfindet, müssen alle notwendigen Berechnungen der Tiefendaten in der jeweiligen Software stattfinden (z. B. die Kalibrierung der *Kinect*-Kameras untereinander).

### 3.2.2 Kalibrierung der Tiefenwerte zur RGB-Kamera

Wie in Kapitel 2.5.3 angedeutet wurde, muss zunächst eine Kalibrierung der *Kinect*-Kameras vorgenommen werden. Da die RGB-Kamera mit ca. 62,7 Grad einen etwas breiteren Öffnungswinkel als die IR-Kamera (ca. 57,8 Grad) aufweist, ist im RGB-Bild ein größerer Bereich sichtbar [ROS11] (siehe Abbildungen 5 und 6 auf Seite 17). Üblicherweise findet dabei eine Anpassung des IR-Bildes auf die RGB-Kamera statt. Diese Aufgabe wird bei *C_stone* von der Abstraction *map_IR_to_RGB* durchgeführt (siehe Anhang auf Seite 88). Hier werden zunächst alle Werte der RAW-Tiefenmaske in Meter umgewandelt, gemäß Formel (2.4) auf Seite 16. Anschließend findet eine weitere Umrechnung der Tiefenmaske in Weltkoordinaten statt. Hierfür wird die Kameramatrix $A_{IR}$ der IR-Kamera benötigt, welche in Kapitel 2.5.2

vorgestellt wurde. Für nähere Informationen zur Kameramatrix und Kamerakalibrierung siehe Kapitel 2.2.

Die Lage der IR- und RGB-Kameras zueinander lässt sich nach der Kalibrierung durch die Rotationsmatrix $R$ und den Translationsvektor $t$ beschreiben. Dabei liegt der Ursprung des Weltkoordinatensystems im optischen Zentrum der RGB-Kamera. Diese Werte können durch eine individuelle Kalibrierung der Kameras des *Kinect*-Sensors ermittelt werden. Ausreichende Genauigkeit für viele Anwendungen liefern jedoch folgende allgemein anwendbare Werte für die 3 × 3-Rotationsmatrix und den 1 × 3-Translationsvektor:

**(3.1)**
$$R_{IR\text{-}RGB} = \begin{bmatrix} 0{,}99977 & 0{,}00172 & -0{,}02122 \\ -0{,}00200 & 0{,}99991 & -0{,}01289 \\ 0{,}02120 & 0{,}012933 & 0{,}99969 \end{bmatrix} \quad \text{[ROS11]}$$

**(3.2)**
$$t_{IR\text{-}RGB} = \begin{pmatrix} 0{,}02135 \\ 0{,}00250 \\ -0{,}01292 \end{pmatrix} \quad \text{[ROS11]}$$

Die Linsenverzerrungen der Kamera wurden in der vorliegenden Arbeit nicht berücksichtigt.

Um beide Kamera zueinander zu kalibrieren wird jeder Punkt der *Kinect*-Tiefenmaske anhand der Rotationsmatrix $R$ und des Translationsvektors $t$ in das Koordinatensystem der RGB-Kamera transformiert. Es gilt für einen 3D-Punkt $\tilde{M}_{kinect}$ der Tiefenmaske folgender Zusammenhang (siehe auch Kapitel 2.2):

**(3.3)**
$$\tilde{m}_{RGB} = A_{RGB}\left[R_{IR\text{-}RGB} * \tilde{M}_{kinect} + t_{IR\text{-}RGB}\right]$$

Der Punkt $\tilde{m}_{RGB}$ ist dabei der korrespondierende Punkt zum 3D-Punkt $\tilde{M}_{kinect}$ auf der Bildebene der RGB-Kamera. Er beinhaltet die Pixelkoordinaten auf der RGB-Bildebene und einen entsprechenden Tiefenwert an dieser Stelle (siehe Formel (3.4)).

**(3.4)**
$$\tilde{m}_{RGB} = \begin{pmatrix} u_{RGB} \\ v_{RGB} \\ z \end{pmatrix}$$

Für die Kameramatrix der RGB-Kamera kommen folgende Werte zum Einsatz:

$$A_{RGB} = \begin{bmatrix} 530.09194 & 0 & 328.21930 \\ 0 & 526.35860 & 268.72781 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

Die in (3.3) vorgestellte Transformation kann mithilfe des *Max*-Externals *projectPoints* realisiert werden. Da *projectPoints* als Parameter unter anderem einen Rotationsvektor erwartet, muss zunächst in einem Zwischenschritt die Rotationsmatrix $R_{IR\text{-}RGB}$ in Vektorform umgewandelt werden. Hierfür kommt das Max-External *rodrigues* zum Einsatz (siehe auch Kapitel 2.6.3.1). Abbildung 10 zeigt den Subpatcher *project_to_RGB_image_plane* mit *projectPoints* und *rodrigues*.

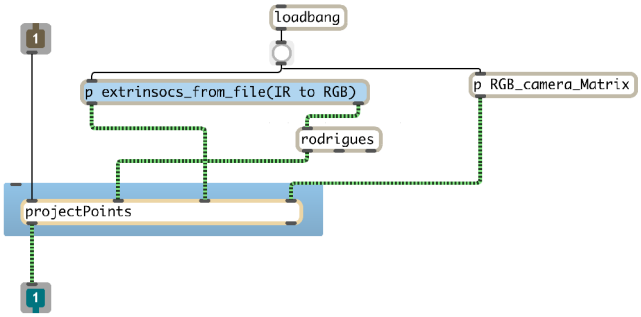


***Abbildung 10: Subpatcher project_to_RGB_image_plane***

Über Inlet 1 dieses Subpatchers wird eine Matrix mit 3D-Koordinaten der Punkte der *Kinect*-Tiefenmaske übergeben. Die resultierende Matrix mit den von *projectPoints* errechneten 2D-Punkten wird über Outlet 1 zurückgegeben.

Nach (3.4) stellen nun die ersten beiden Komponenten jedes Punktes $\tilde{m}_{RGBi}$ seine Pixelkoordinaten auf der Bildebene der RGB-Kamera dar. Die dritte Komponente repräsentiert den Abstand des Punktes zum Sensor. Somit können nun die Werte anhand der Pixelkoordinaten in einer 640 × 480-Matrix abgelegt werden. Dabei wird jeder Punkt $\tilde{m}_{RGBi}$ anhand seiner ersten beiden Komponenten als Indizes in die Matrix „einsortiert“. Die resultierende Matrix enthält die korrigierte *Kinect*-Tiefenmaske und kann für weitere Berechnungen eingesetzt werden.

## 3.3 Calibration

Bei der Kalibrierung zwischen Projektor und *Kinect* sollen die intrinsischen Daten des Projektors ermittelt werden und dessen Position und Orientierung im Weltkoordinatensystem beschrieben werden (siehe Kapitel 2.7). Teile der Umsetzung der Kalibrierung in *C_stone* basieren auf der quelloffenen *vvvv*-Software *CalibrateProjector* von Elliot Woods [vvv12] [Woo11]. Vor allem die besondere Nutzung der *OpenCV*-Funktion *calibtrateCamera2()* zur *Kinect*-Projektor-Kalibrierung beruht auf Teilen von *CalibrateProjector,* die in *MaxMSP* neu implementiert wurden (siehe auch Kapitel 2.6.3.1). Abbildung 11 zeigt einen Teil des neu entwickelten Haupt-*Max*-Patchers, welcher im Anhang auf Seite 86 zu finden ist. Darin sind alle für die Kalibrierung notwendigen *Max*-Objekte, Abstractions und Subpatcher aufgeführt.

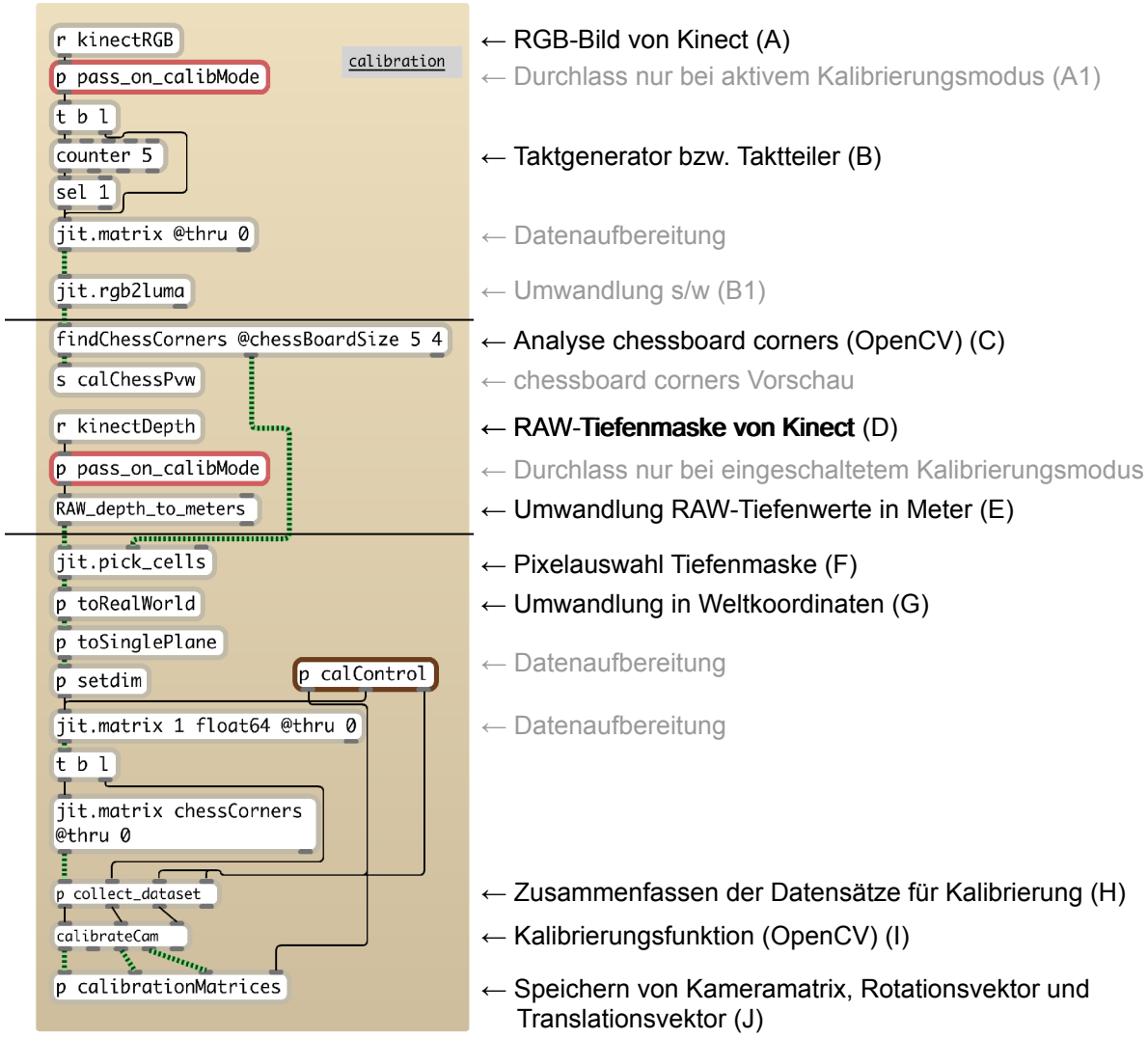


***Abbildung 11: Übersicht Calibration-Bereich***

Zu Beginn wird über das *receive*-Objekt (siehe Abschnitt A in Abbildung 11) der RGB-Datenstrom der *Kinect* empfangen. Dann findet eine Abfrage statt, ob sich die Software im Kalibrierungsmodus befindet. Ist dies der Fall, werden die Bilder vom Subpatcher *pass_on_calibMode* weitergegeben (siehe A1 in Abbildung 11). Befindet sich hingegen die Software im *Runtime*-Modus, werden hier keine Daten weitergegeben und die folgenden Objekte bleiben inaktiv.

Im nächsten Abschnitt (B) findet eine Verringerung der Framerate statt. Nur jedes fünfte Frame der RGB-Kamera wird hier weitergegeben. Dies geschieht zur „Entlastung“ der in Abschnitt C folgenden Funktion zur Erkennung des Schachbrettmusters. Dies ist notwendig, da dem External *FindChessCorners* ein *OpenCV*-Algorithmus zugrunde licgt (siehe Kapitel 2.6.3.1), welcher deutlich langsamer wird, wenn im aktuell analysierten Frame kein Schachbrettmuster zu finden ist (siehe auch Kapitel 4.6). Bliebe diese Verlangsamung aus, so könnten sich die Bilder „aufstauen“, was zu einer generellen Verlangsamung und schließlich zum Absturz des Programms führen könnte.

Erst nach einer Umwandlung des RGB-Bildes in ein Schwarzweißbild durch *jit.rgb2luma* (Abschnitt B1) werden die Frames an *FindChessCorners* weitergegeben. Dabei wird ein Kontrollbild generiert (*FindChessCorners* Outlet 1), welches für den Nutzer in der GUI sichtbar ist (siehe Kapitel 3.6.2). Somit kann während des Kalibrierens überprüft werden, ob die Ecken des projizierten Schachbrettmusters von *FindChessCorners* erkannt wurden. Aus dem zweiten Outlet des *FindChessCorners*-External werden die Pixelkoordinaten als Matrix an *jit.pick_cells* (Abschnitt F) weitergegeben.

Von *jit.pick_cells* werden nun die im RGB-Bild erkannten Eckpunkte aus der Tiefenmaske ausgewählt. Hierzu werden zunächst die RAW-Tiefenwerte des aktuellen Frames der Tiefenkamera (Abschnitt D) in Meter umgewandelt. Dies geschieht durch den Patcher *RAW_depth_to_Meters* gemäß Formel (2.4), welcher in Kapitel 2.5.2 näher erläutert wird. Abbildung 12 zeigt die Implementierung der Formel (2.4) anhand eines *jit.expr*-Objekts.

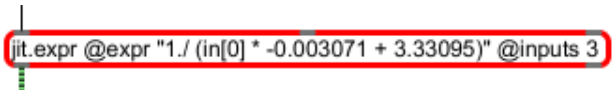


***Abbildung 12: jit.expr in RAW_depth_to_meters***

In Abschnitt F findet nun die Auswahl einzelner Punkte aus der Tiefenmaske statt. Ziel ist es, die 3D-Koordinaten der im RGB-Bild erkannten Eckpunkte zu bestimmen. *Jit.pick_cells* generiert eine 3 × N-Matrix, wobei N die Anzahl der Ecken im Schachbrett ist. In den drei Spalten der Matrix werden die Pixelkoordinaten u, v und der Tiefenwert z an den jeweiligen Pixeln abgelegt. Für die Punkte in dieser Matrix können nun im darauffolgenden Abschnitt G die Weltkoordinaten bestimmt werden (siehe Kapitel 2.5.2). Screenshots der Abstraction *jit.pick_cells* und des Subpatchers *toRealWorld* befinden sich im Anhang auf Seite 89.

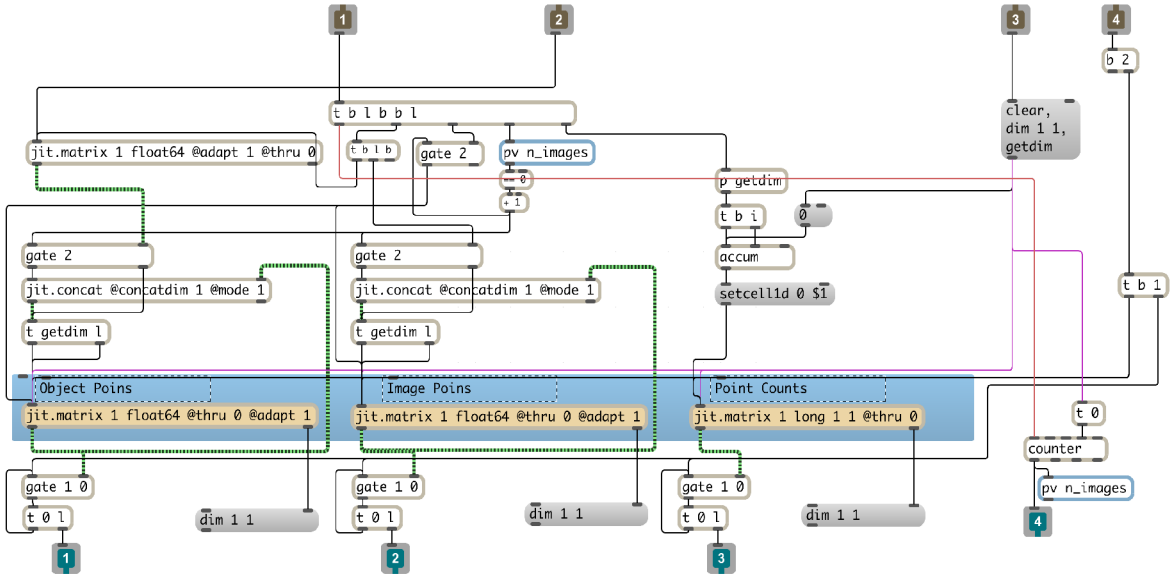


***Abbildung 13: Subpatcher collect_dataset***

Der Subpatcher *collect_dataset* (Abbildung 11, Abschnitt H) stellt ein wichtiges Element der Kalibrierung dar. Abbildung 13 zeigt die Inhalte des Subpatchers, wo sich im blau unterlegten Teil drei *jit.matrix*-Objekte befinden. In den ersten beiden dieser drei Matrizen werden die *Objekt-Koordinaten* und dazugehörigen *Bild-Koordinaten* gespeichert. Das dritte *jit.matrix*-Objekt enthält eine 1 × 1-Matrix mit der Gesamtanzahl der gesammelten Punktepaare. Diese Zahl ergibt sich aus der Anzahl der Ecken des projizierten Schachbrettmusters multipliziert mit der Anzahl der Schachbrett-Aufnahmen. Da die Aufnahmen des Schachbrettmusters nacheinander durchgeführt werden, müssen die Matrizen nach jeder Aufnahme um die neuen Punkten erweitert werden. Dies geschieht durch das Objekt *jit.concat* (siehe Abbildung 13). Der Inhalt aller drei *jit.matrix*-Objekte wird dann dem Objekt *calibrateCam* übergeben (Abbildung 12, Abschnitt I). Hierfür muss der Nutzer einen Button betätigen, nachdem ausreichend viele Aufnahmen des Schachbretts gemacht wurden (siehe Kapitel 3.6.2).

Durch *calibrateCam* werden nun die Kameramatrix, ein Rotations- und ein Translationsvektor ermittelt und im Anschluss im Subpatcher *calibrationMatrices* in weiteren *jit.matrix*-Objekten gespeichert. Abbildung 14 zeigt die im Subpatcher *calibrationMatrices* enthaltenen Objekte. Die *jit.matrix*-Objekte sind dabei mit „cameraMat“, „rvec“ und „tvec“ benannt. Durch diese Benennung kann nun in der gesamten *Max*-Umgebung von gleichnamigen *jit.matrix*-Objekten auf die Daten zugegriffen werden, ohne dass sie über patch cords verbunden sein müssen. Die Daten werden auf diese Art im Objekt *pointProjector* des *Runtime*-Abschnitts aufgerufen und können direkt verwendet werden (siehe Kapitel 3.4.3).

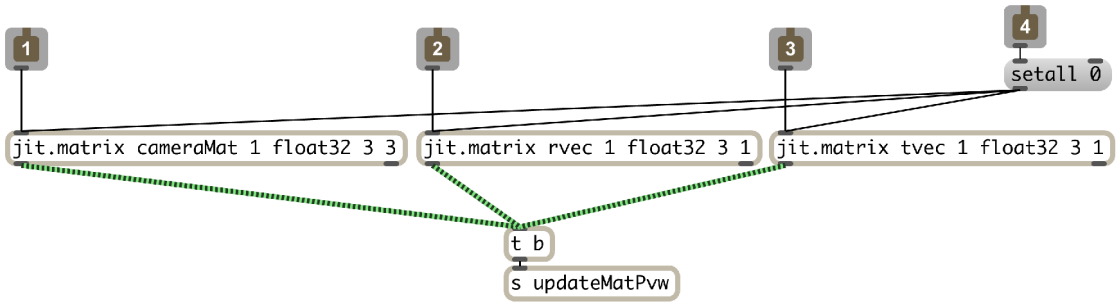


***Abbildung 14: Subpatcher calibrationMatrices mit jit.matrix-Objekten zur Speicherung der Kalibrierungsergebnisse***

## 3.4 Runtime

### 3.4.1 Übersicht

Abbildung 15 zeigt, welche Komponenten im Abschnitt *Runtime* der Software enthalten sind. Diese Prozesskette ist im Normalbetrieb aktiv. In diesem Programmabschnitt werden die Tiefenmaske der *Kinect* analysiert und daraus eine entsprechende Homographie-Matrix zur perspektivischen Transformation des Projektorbildes erzeugt.

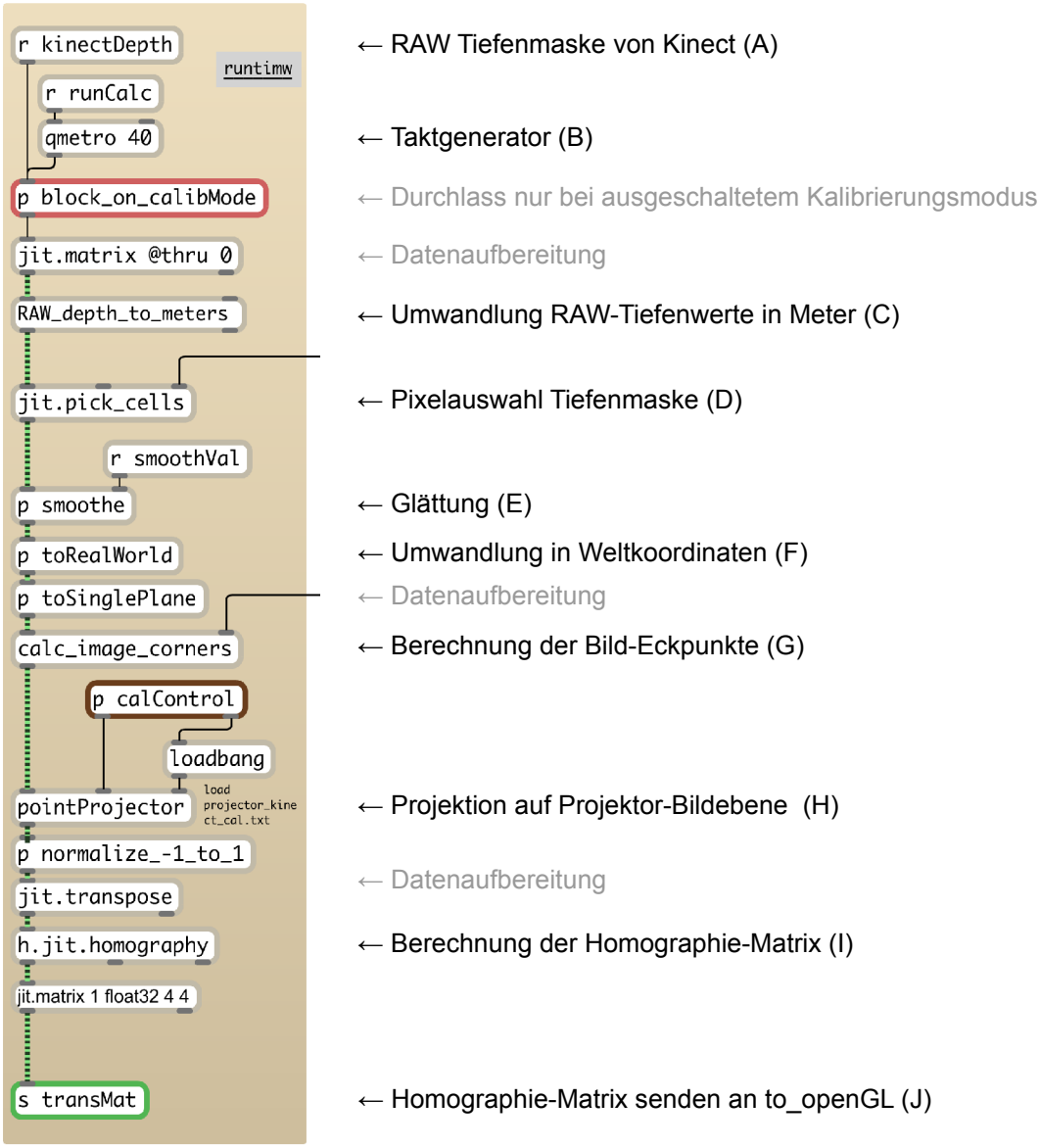


***Abbildung 15: Übersicht Runtime***

Im Abschnitt A (Abbildung 15) werden zunächst die *Kinect*-Tiefendaten über ein *receive*-Objekt empfangen. Als Taktgeber der ganzen Objektkette dient ein *qmetro*-Objekt (Abschnitt B). Mithilfe dieses Objekts sollen Geschwindigkeitsschwankungen in der Übertragung des *Kinect*-Interfaces ausgeglichen werden. Die Berechnung der Homographie-Matrix findet alle 40 ms, also 25 Mal pro Sekunde, statt, vorausgesetzt das Computersystem verfügt über die erforderliche Rechenleistung (siehe auch Kapitel 4.6).

In Abschnitt B wird eine Umrechnung der RAW-Tiefendaten in Meter durchgeführt. Hierbei kommt wiederum der Patcher *RAW_depth_to_meters* zum Einsatz (siehe auch Abbildung 12). Hiernach werden bestimmte Pixel der Tiefenmaske der Tiefenmaske ausgewählt, mit denen die Projektionsfläche *Ep* beschrieben werden soll. Dafür wird dem Objekt bzw. der Abstraction *jit.pick_cells* in Abschnitt D über das dritte Inlet eine Liste mit drei Punkten übergeben (siehe auch *jit.pick_cells* Screenshot auf Seite 89). Die Positionen dieser Punkte in der Tiefenmaske können über die GUI vom Nutzer bestimmt werden (siehe Kapitel 3.6.4). Auf Basis dieser Punkte werden im folgenden Schritt die Punkte P, $K_1$ und $K_2$ ermittelt, welche die Projektionsfläche *Ep* im 3D-Raum beschreiben.

Vor der Umwandlung der Punkte in 3D-Weltkoordinaten findet eine Glättung der Tiefenwerte statt, welche vom Subpatcher *smoothe* in Abschnitt E erledigt wird (siehe auch Anhang, Seite 90). Diese Glättung sorgt für weichere Übergänge zwischen Korrekturzuständen bei Veränderungen des Projektionswinkels. Die Intensität der Glättung kann dabei vom Nutzer eingestellt werden (siehe Kapitel 3.6.4). Auf diese Weise kann *C_stone* an unterschiedliche Anforderungen angepasst werden. Ist eine sehr schnell wechselnde Korrektur erforderlich, so sollte die Reaktionsgeschwindigkeit gering gehalten werden. Das erhöht allerdings auch die Anfälligkeit der Software auf Rauschen innerhalb der *Kinect*-Tiefenmaske.

Im nächsten Schritt (siehe Abbildung 15, Abschnitt F) werden die Weltkoordinaten der drei Punkte aus der *Kinect*-Tiefenmaske ermittelt (siehe Kapitel 2.5.2) und anschließend eine Matrix mit diesen Weltkoordinaten an die Abstraction *calc_image_corners* weitergegeben (siehe Abbildung 16). Dort findet ein weiterer wichtiger Teil der Berechnungen statt. Es werden anhand der drei übergebenen Punkte die gewünschten Bild-Eckpunkte der Projektion errechnet (G). Innerhalb der Abstraction *calc_image_corners* werden die Koordinaten der Punkte an ein *JavaScript*-Objekt (siehe „*js math.3d.js*“-Objekt in Abbildung 16) weitergeleitet, wo die Berechnung stattfindet. Auf die *JavaScript*-Implementierung des *math.3d.js*-Scripts wird im Kapitel 3.4.2 ausführlich eingegangen.

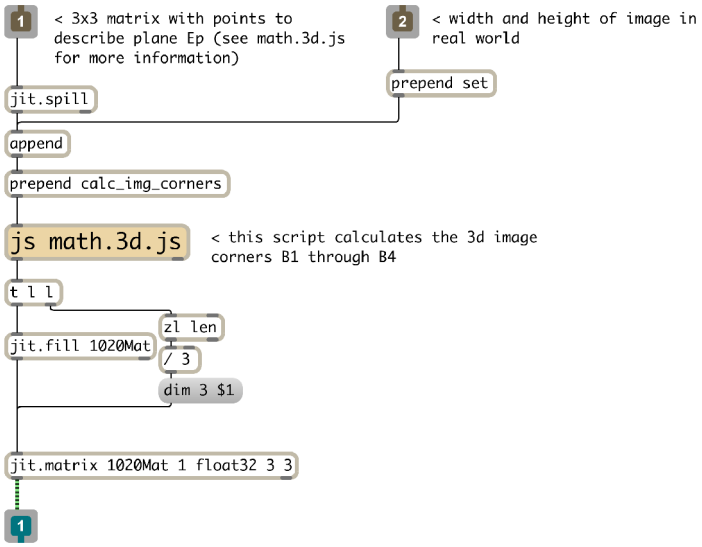


***Abbildung 16: Abstraction calc_image_corners***

Zum Schluss werden die berechneten Eckpunkte im Weltkoordinatensystem an den Subpatcher *pointProjector* weitergegeben. *pointProjector* errechnet anhand der Projektordaten die korrespondierenden 2D-Punkte auf der Projektorbildebene (detaillierte Beschreibung in Kapitel 3.4.3). Im Anschluss daran kann mittels dieser vier Bild-Eckpunkte auf der Projektor-Bildebene eine Homographie-Matrix erzeugt werden (siehe Kapitel 3.4.4). Diese Matrix ist das Ergebnis der Berechnungen des *Runtime*-Abschnitts von *C_stone*. Über das *send*-Objekt in Abschnitt J der Abbildung 15 wird die Matrix zur Verwendung an den *to_openGL*-Patcher weitergeleitet (siehe Kapitel 3.5.1). Die Matrix ist grundlegend für die perspektivische Transformation, durch welche die Trapezkorrektur in *C_stone* realisiert wird.

### 3.4.2 Berechnung der Bild-Eckpunkte der Projektion

Dieses Kapitel beschäftigt sich detailliert mit der Berechnung der Bild-Eckpunkte. Die Bild-Eckpunkte sind jene 3D-Punkte deren korrespondierende 2D-Punkte auf der Projektionsfläche für die Berechnung der Homographie-Matrix notwendig sind. Die Berechnung wird anhand eines *JavaScripts* durchgeführt, welches im Subpatcher *calc_image_corners* implementiert ist (siehe auch Abbildung 15, Abschnitt G).

Zu Beginn werden ein Punkt $P$ und zwei weitere Punkte ($K_1$ und $K_2$) der *Kinect*-Tiefenmaske ausgewählt. Wichtig ist, dass diese Punkte nicht kollinear sind, da mithilfe dieser Punkte eine

mathematische Darstellung der Projektionsfläche *Ep* erfolgen soll. $P$, $K_1$ und $K_2$ sind daher in einem Dreieck angeordnet.

Im nächsten Schritt werden die 3D-Koordinaten der Punkte $P$, $K_1$ und $K_2$ ermittelt (siehe Kapitel 2.3), um anschließend anhand dieser Punkte die Ebene *Ep* zu definieren. Für *Ep* in Koordinatenform mit $r$ und $s$ als Parameter gilt:

**(3.6)**
$$Ep: \ \vec{x} = P + r * (K_1 - P) + s * (K_2 - P)$$

Folgende Bedingungen gelten für die Bild-Eckpunkte $B_1$ bis $B_4$:

- Alle vier Punkte liegen auf der Ebene *Ep*.
- Sie bilden auf der Ebene ein Rechteck.
- Die Strecken $B_1B_2$ und $B_3B_4$ verlaufen parallel zur Grundebene *Exz*, die von den Koordinatenachsen $x$ und $z$ des Weltkoordinatensystems aufgespannt wird.

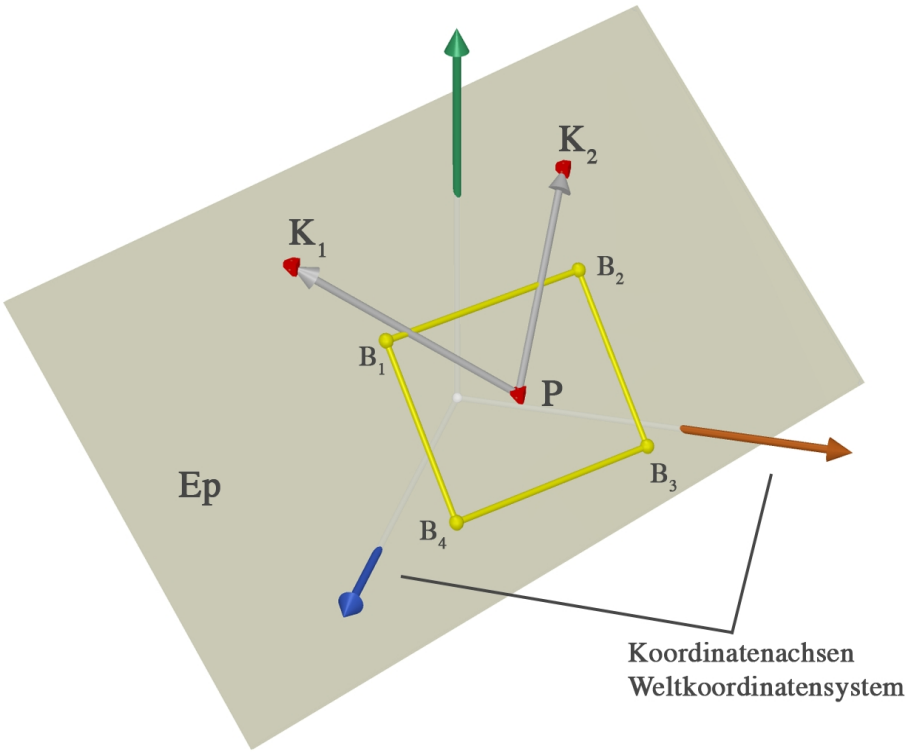


***Abbildung 17: Ep mit Punkten P, K1, K2 und B1–B4***

Um nun die Koordinaten der Bild-Eckpunkte $B_1$ bis $B_4$ bestimmen zu können, muss zunächst die Größe des Bildes in horizontaler und vertikaler Richtung bekannt sein. Diese Größen werden als $h$ und $b$ bezeichnet und beschreiben die tatsächliche Größe der Projektion im Weltkoordinatensystem. Sie können in der Software vom Nutzer eingestellt werden (siehe dazu Kapitel 3.6.4, *Image size*).

Zunächst wird eine Hilfsebene $HE_m$ definiert, die den Punkt $P$ enthält und parallel zur Grundebene Ebene *Exz* verläuft. Somit kann $HE_m$ mit folgender Gleichung beschrieben werden:

$$HE_m: \; N_{HE_m} * (\vec{x} - P) = 0 \quad \text{mit dem Normalenvektor} \quad N_{HE_m} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{3.7}$$

Diese Hilfsebene wird von der Projektionsebene *Ep* in der Geraden *g* geschnitten. Der Punkt $P$ liegt ebenfalls auf dieser Geraden und kann somit als deren Aufpunkt verwendet werden. Gleichung (3.8) definiert *g*, wobei *r* den Parameter und $a_1$, $a_2$, $a_3$ die Komponenten des Vektors darstellen.

$$g: \; \vec{x} = P + r * \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \tag{3.8}$$

Der Zusammenhang wird in Abbildung 18 verdeutlicht.

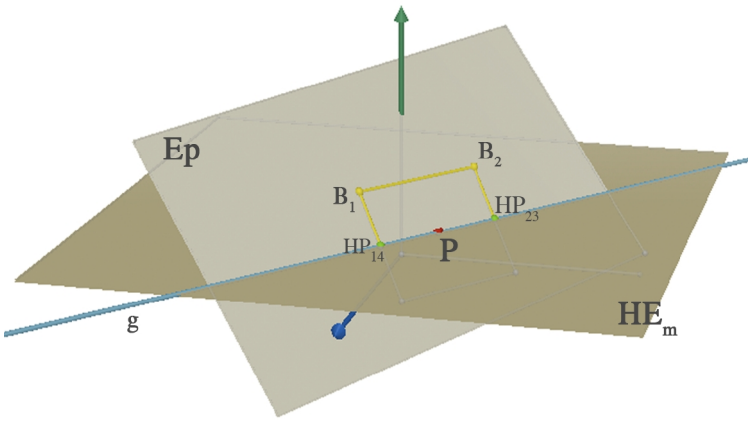


***Abbildung 18: Ep, $HE_m$ und Schnittgerade g***

Auf der Geraden *g* befinden sich die Hilfspunkte $HP_{14}$ und $HP_{23}$. Sie liegen jeweils zwischen den gesuchten Eckpunkten $B_1$ und $B_4$, $B_2$ und $B_3$, also jeweils in der Mitte der linken und rechten Bildkante. (Die Punkte $B_3$ und $B_4$ werden in Abbildung 18 von der Hilfsebene $HE_m$ verdeckt.)

Die Koordinaten der Hilfspunkte können dann über den Abstand $\frac{b}{2}$ entlang der Geraden *g* bestimmt werden. Dabei gilt:

**(3.9)**
$$|HP_{14} - P| = \frac{b}{2} \quad \text{und} \quad |HP_{23} - P| = \frac{b}{2}$$

Da beide Hilfspunkte auf der Geraden *g* liegen, erhält man nach Einsetzen der Geraden g in die Gleichung (3.9):

**(3.10)**
$$\left| \begin{pmatrix} p_1 + r * a_1 \\ p_2 + r * a_2 \\ p_3 + r * a_3 \end{pmatrix} - P \right| = \frac{b}{2}$$

**(3.11)**
$$\sqrt{(r * a_1)^2 + (r * a_2)^2 + (r * a_3)^2} = \frac{b}{2}$$

**(3.12)**
$$r = \sqrt{\left(\frac{\left(\frac{b}{2}\right)^2}{(a^2 + b^2 + c^2)}\right)}$$

Durch Umformung der Gleichung (3.10) kann nun auf (3.11) und folglich auf (3.12) geschlossen werden. Als Lösung der Gleichung (3.12) existieren zwei Ergebnisse: *r* und *-r*. Durch Einsetzen beider Lösungen für den Parameter *r* in der Geraden *g* erhalten wir die Punkte $HP_{14}$ und $HP_{23}$.

Nun werden zwei weitere Hilfsebenen definiert, indem jeweils die Vektoren zwischen den Punkten $HP_{14}$, $HP_{23}$ und dem Punkt *P* gebildet werden. Diese Vektoren stellen die

Normalenvektoren der Hilfsebenen $HE_{links}$ und $HE_{rechts}$ dar, wobei die Aufpunkte jeweils durch $HP_{14}$ und $HP_{23}$ definiert werden. So sind die Normalenvektoren gegeben mit:

**(3.13)** $$N_{HE_{links}} = HP_{14} - P \quad \text{und} \quad N_{HE_{rechts}} = HP_{23} - P$$

Damit gilt für die Darstellung der Ebenen in Normalenform:

**(3.14)** $$HE_{links}: \quad N_{HE_{links}} * \left(\vec{x} + HP_{14}\right) \quad \text{und} \quad HE_{rechts}: \quad N_{HE_{rechts}} * \left(\vec{x} + HP_{23}\right)$$

Die Schnittgeraden dieser Ebenen (3.14) mit der Projektionsebene *Ep* werden als *hgl* und *hgr* bezeichnet und mit den Punkten $HP_{14}$ bzw. $HP_{23}$ als Aufpunkt definiert. Sie enthalten die gesuchten Punkte $B_1$, $B_4$ bzw. $B_2$, $B_3$ (siehe Abbildung 19).

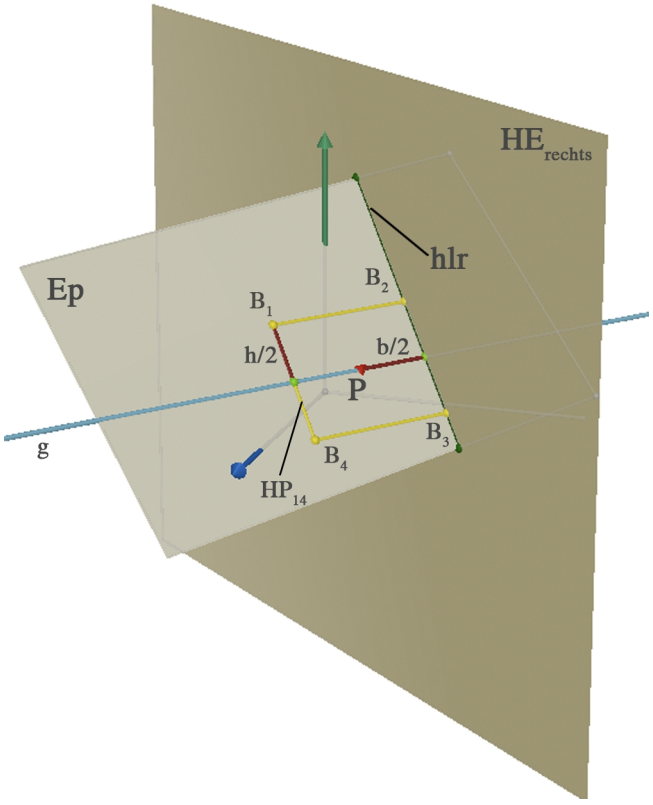


***Abbildung 19: Ep, $HE_{rechts}$ und Schnittgerade hlr***

Ähnlich wie in (3.9) kann nun über eine Abstandsberechnung auf die Punkte $B_1$ und $B_4$ geschlossen werden. Diese Punkte liegen auf der Geraden *hgl* und ihr Abstand zum Hilfspunkt $HP_{14}$ beträgt $\frac{h}{2}$. Selbiges gilt für die Punkte $B_2$ und $B_3$ auf der Geraden *hgr*.

#### 3.4.2.1 Das JavaSkript math.3d.js

Für die hier benötigten Berechnungen wurden einige Objekte und Funktionen in dem *JavaScript*-Dokument *math3d.js* implementiert, beispielsweise für den Umgang mit Vektoren oder zur Schnittpunktbestimmung.. Zunächst folgt eine Übersicht über einige Funktionsobjekte zur Darstellung einfacher Elemente der analytischen Raumgeometrie. Danach werden einige der Grundfunktionen wie etwa zur Ermittlung des Skalarproduktes oder zur Vektoraddition vorgestellt.

**Punkt**

Für die Beschreibung eines Punktes oder eines Vektors wird hier das *JavaScript*-interne Array-Objekt verwendet. Codezeile 1 zeigt die Implementierung eines Punkts (oder Vektors) *O* mit den Koordinaten (1|3|-4):

```
1                    var O = new Array(1,3,-4);
```

**Gerade**

Mit dem Objekt *Line* lassen sich Geraden im dreidimensionalen Raum definieren.

```
2                    function Line(PointP, Vec){
3                            this.originPoint = PointP;
4                            this.Vec = Vec;
5                            }
```

Die dem Objekt *Line* zugehörigen Variablen *originPoint* und *Vec* repräsentieren dabei den Aufpunkt auf der Geraden und einen Vektor. Bei der Instanziierung eines Objektes vom Typ *Line* müssen diese Werte als Parameter übergeben werden (siehe Codezeile 2).

Folgendes Beispiel demonstriert nun die Instanziierung eines *Line*-Objekts mit dem Punkt *myPoint* und dem Vektor *myVec*:

```
6                var myPoint = new Array(1, 4, -2); // define a point
7                var myVec = new Array(2, 2, 2); // define a vector
8                var aLine = new Line(myPoint, myVec); // instantiate the line
```

### Fläche

Neben dem *Line*-Objekt wird das Objekt *Plane* beschrieben, mit dem eine Ebene im dreidimensionalen Raum repräsentiert werden kann.

```
9                function Plane(PointP, PointK1, PointK2){
10
11                   this.originPoint = PointP;
12                   this.VecA = Vec_from_points(PointP, PointK1);
13                   this.VecB = Vec_from_points(PointP, PointK2);
14
15                      // cross product between vectors -> normalvector
16                   this.normVec = Cross_product(this.VecA, this.VecB);
17
18                            // d is for the plane equation: N * p = d
19                   this.d = Scalar_product(this.normVec, this.originPoint);
20                   }
```

Hierbei müssen zur Instanziierung drei Punkte ($P$, $K_1$, $K_2$) als Parameter übergeben werden, durch welche die Ebene definiert wird (siehe Codezeile 9). Durch den Aufruf der Funktion *Vec_from_Points()* (Codezeile 21) werden dann zwei Vektoren errechnet, welche die Ebene zusammen mit dem Aufpunkt $P$ aufspannen. Zusätzlich werden der Normalenvektor errechnet und die Konstante $d$, welche aus dem Skalarprodukt des Normalenvektors und des Ortsvektors des Punktes $P$ besteht. Die Konstante $d$ wird innerhalb von Funktionen zur Schnittpunktberechnung zwischen Ebenen verwendet. Beispielsweise in der Funktion *Intersect2Plane()* im Anhang auf Seite 93, Codezeile 176, wird diese benötigt.

### Vektor aus zwei Punkten

*Vec_from_points()* stellt eine einfache Funktion dar, der zwei Punkte in Form von Arrays übergeben werden. Die Funktion gibt den Vektor vom ersten zum zweiten Punkt als Array zurück.

```
21              function Vec_from_points(a,b){
22                  var result = new Array();
23                              for(var i = 0; i < a.length; i++)
24                                      result[i] = b[i] - a[i];
25                  return result;
26                  }
```

### Kreuzprodukt zweier Vektoren

Mit *cross_product()* kann das Kreuzprodukt zweier Vektoren ermittelt werden. Das Ergebnis ist ebenfalls ein Vektor.

```
27              function Cross_product(a,b){
28                          var result = new Array();
29
30                                  result[0] = a[1]*b[2] - a[2]*b[1];
31                                  result[1] = a[2]*b[0] - a[0]*b[2];
32                                  result[2] = a[0]*b[1] - a[1]*b[0];
33
34                          return result;
35                          }
```

### Skalarprodukt zweier Vektoren

Die Funktion *Scalar_product()* kann zur Ermittlung des Skalarprodukts zweier Vektoren eingesetzt werden.

```
36              function Scalar_product(a,b){
37                  var result = 0.;
38                          result = a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
39                  return result;
40                  }
```

Diese Funktionen sind einige Beispiele aus der *JavaScript*-Datei *math.3d.js*, die im Rahmen dieser Arbeit entstanden ist. Die vollständige Implementierung kann im Anhang auf Seite 92 eingesehen werden.

#### 3.4.2.2 Eckpunktberechnung in math.3d.js

Die JavaScript-Funktion *Calc_img_corners()* beinhaltet alle in Kapitel 3.4.2 besprochenen Schritte zur Berechnung der Punkte $B_1$ bis $B_4$. Als Übergabeparameter benötigt die Funktion drei Punkte ($P$, $K_1$ und $K_2$) und zwei zusätzliche Werte, welche die Breite und die Höhe des Bildes beschreiben (hier mit *h_img_size* und *v_img_size* bezeichnet). (Siehe auch Codezeile 383 auf Seite 98.) Diese Funktion wird jedes Mal aufgerufen, wenn eine neues Frame der *Kinect*-Tiefenmaske bereitsteht und die Homographie-Matrix berechnet werden soll.

Im ersten Schritt wird die Ebene *Ep* und die Hilfsebene *HE_m* parallel zur Ebene *Exz* erzeugt (Codezeile 41). Ebenso werden die Variablen $B_1$, $B_2$, $B_3$ und $B_4$ als Arrays definiert, da sie später 3D-Punkte enthalten sollen.

```
41                          var Ep = new Plane(P, K1, K2);
42
43                          var y_vec = new Array(0,1,0);
44                          var HE_m = new Plane_norm(P, y_vec);
45
46                          var B1 = new Array();
47                          var B2 = new Array();
48                          var B3 = new Array();
49                          var B4 = new Array();
```

Im Anschluss wird in Codezeile 50 mithilfe der Funktion *Intersect_2plane()* die Schnittgerade $g$ zwischen den Ebenen *Ep* und *HE_m* ermittelt. Die Implementierung dieser Funktion kann im Anhang auf Seite 93, Codezeile 176, eigesehen werden.

```
50                          var g = Intersect_2plane(Ep, HE_m);
```

Im folgenden Schritt werden die Punkte entlang der Geraden *g* im Abstand $\frac{b}{2}$ ermittelt. Hierzu wird die Funktion *distance_on_line()* aufgerufen, deren Implementierung ebenfalls im Anhang eingesehen werden kann (Seite 92, Codezeile 93). *distance_on_line()* erwartet zwei Parameter: Eine Linie *l* und einen Abstand *d*. Werden diese an die Funktion übergeben, gibt sie die Koordinaten **zweier** Punkte zurück, die um den Abstand *d* vom Aufpunkt der Geraden *l* entfernt sind. Das erklärt den nächsten Schritt (Codezeile 51), in dem der Aufpunkt der Geraden *g* als Punkt *P* definiert wird. Dies ist nötig, um die Hilfspunkte *HPl* und *HPr* mit dem Abstand *d* vom Punkt *P* zu ermitteln:

```
51                          g.originPoint = P;
```

Hiernach wird *distance_on_line()* aufgerufen und der Rückgabewert zunächst in die Variable *temp* geschrieben. Da die Funktion zwei 3D-Punkte in einem Array zurückgibt, müssen die Werte aus dem *temp*-Array entsprechend auf die Arrays *HPl* und *HPr* aufgeteilt werden (siehe Codezeilen 52-62).

```
52                          var temp = distance_on_line(g, h_img_size/2);
53
54                          var HP_l = new Array();
55                          var HP_r = new Array();
56
57                          HP_l[0] = temp[0];
58                          HP_l[1] = temp[1];
59                          HP_l[2] = temp[2];
60                          HP_r[0] = temp[3];
61                          HP_r[1] = temp[4];
62                          HP_r[2] = temp[5];
```

Ausgehend von den Hilfspunkten *HPl* und *HPr* (Codezeile 54, 55) können nun zusammen mit dem Punkt *P* die zwei Hilfsebenen $HE_{links}$ und $HE_{rechts}$ definiert werden. Hierbei ist zu beachten, dass diese Hilfsebenen mit dem Funktionsobjekt *Plane_norm()* und nicht mit *Plane()* erzeugt

werden, wie etwa in Codezeile 41. *Plane_norm()* erlaubt die Erzeugung von Ebenen mithilfe eines Punktes und einem Normalenvektor anstatt von drei Punkten auf der Ebene (Implementierung siehe Seite 93, Codezeile 164).

```
63                              var Nl = Vec_from_points(HP_l,P);
64                              var Nr = Vec_from_points(HP_r,P);
65                              var HE_links = new Plane_norm(HP_l, Nl);
66                              var HE_rechts = new Plane_norm(HP_r, Nr);
```

Um nun die Geraden *hgl* und *hgr* zu bestimmen, werden im nächsten Schritt die Hilfsebenen $HE_{\text{links}}$ und $HE_{\text{rechts}}$ mit der Projektionsebene *Ep* geschnitten (Codezeilen 67 und 68).

```
67                              var hg_l = Intersect_2plane(HE_links, Ep);
68                              var hg_r = Intersect_2plane(HE_rechts, Ep);
```

Zuletzt werden nun die Punkte $B_1$ bis $B_4$ über die jeweiligen Abstände zu den Hilfspunkten *HPl* und *HPr* bestimmt. Hierbei wird jedoch der vertikale Abstand *v_img_size*, also die Höhe des Bildes, verwendet.

```
69                              hg_l.originPoint = HP_l;
70                              hg_r.originPoint = HP_r;
71
72                              temp = distance_on_line(hg_l, v_img_size/2);
73                              B1[0] = temp[0];
74                              B1[1] = temp[1];
75                              B1[2] = temp[2];
76                              B4[0] = temp[3];
77                              B4[1] = temp[4];
78                              B4[2] = temp[5];
```

```
79
80                              temp = distance_on_line(hg_r, v_img_size/2);
81                              B2[0] = temp[0];
82                              B2[1] = temp[1];
83                              B2[2] = temp[2];
84                              B3[0] = temp[3];
85                              B3[1] = temp[4];
86                              B3[2] = temp[5];
```

Als letzte Zeile der Funktion folgt nun der Befehl, die Werte von $B_1$ bis $B_4$ über das Outlet 0 auszugeben (siehe Codezeile 87). Dabei handelt es sich um das linke Outlet des *js*-Objekts innerhalb des *Max*-Patchers. So werden die Ergebnisse aus dem *JavaScript* an *MaxMSP* als Liste übergeben und können dort weiterverarbeitet werden (siehe dazu auch Abbildung 16 auf Seite 39).

```
87                  outlet(0, B1, B2, B3, B4);
```

### 3.4.3 Projektion der Bild-Eckpunkte auf die Projektor-Bildebene

Dieses Kapitel bezieht sich auf den Subpatcher *pointProjector* in Abschnitt H der Abbildung 15. Durch die in Kapitel 3.4.2 besprochene Methode werden die 3D-Koordinaten der Bild-Eckpunkte im Weltkoordinatensystem ermittelt. Nun sollen die korrespondierenden Punkte in Pixelkoordinaten auf der Bildebene des Projektors bestimmt werden. Abbildung 20 veranschaulicht die Transformation der 3D-Eckpunkte auf die 2D-Projektor-Bildebene.

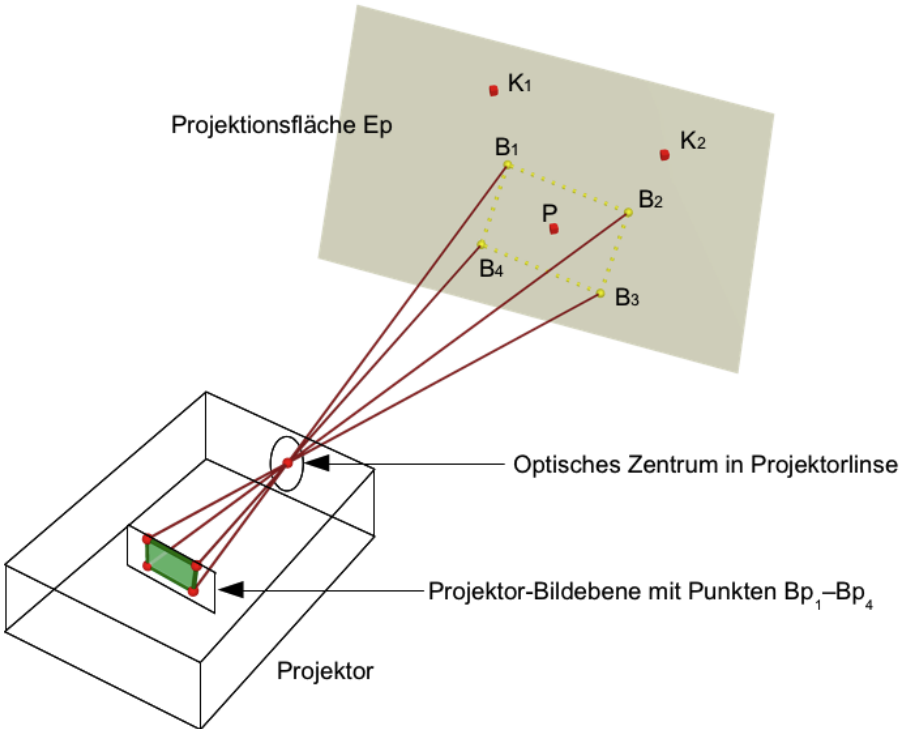


***Abbildung 20: Zusammenhang der Punkte $B_1$–$B_4$ auf Ep und deren korrespondierende Punkte $Bp_1$–$Bp_4$ auf der Projektor-Bildebene***

Die errechneten Punkte $B_1$–$B_4$ beschreiben auf der Projektionsfläche ein Rechteck. Tatsächlich ist jedoch nur aus dem Blickpunkt eines Betrachters, der genau orthogonal zu der Ebene steht und sich auf der Höhe des Punktes *P* befindet, auch wirklich ein Rechteck zu erkennen. Blickt man schräg auf die Projektionsfläche wirken die Punkte perspektivisch verzerrt. Auf die Bildebene eines Projektors bezogen bedeutet dies nun, dass bei Schrägprojektion die Eckpunkte auf der Bildebene nicht in einem Rechteck angeordnet sein können, um bei Projektion auf eine planare Projektionsfläche als rechteckig wahrgenommen zu werden.

In Kapitel 2.2 wurde die Formel (2.1) als $\tilde{m} = P\tilde{M}_w$ mit $P = A[R|t]$ eingeführt. Die Matrix *P* enthält sowohl die intrinsischen als auch die extrinsischen Daten des Projektors. Mit diesem Zusammenhang können nun die projizierten Punkte $Bp_1$-$Bp_4$ errechnet werden. Dabei handelt es sich um die korrespondierenden 2D-Punkte der 3D-Punkte $B_1$–$B_4$ auf der Projektionsebene *Ep*. Für diese Berechnung gilt somit der in (3.15) dargestellte Zusammenhang.

**(3.15)**
$$Bp_i = P\,B_i$$

Diese Transformation wird über das External *pointProjector* realisiert, in welchem das External *projectPoints* enthalten ist. Wie in Abbildung 21 zu sehen, werden dem External *projectPoints* die Kameramatrix, ein Rotationsvektor und ein Translationsvektor in Form von *Jitter*-Matritzen übergeben. Entweder werden dabei die im *calibration mode* der Software erstellten Daten verwendet (links in Abbildung 21) oder die Datei *kinect_projector_cal.txt* wird eingelesen, um gespeicherte Kalibrierungsinformationen zu verwenden.

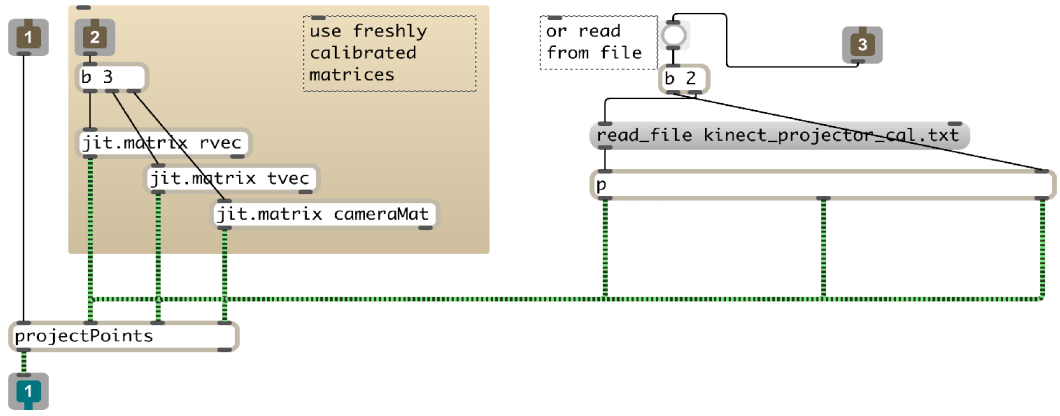


***Abbildung 21: Abstraction pointProjector mit projectPoints External***

Jedes mal wenn nun eine Matrix mit vier 3D-Punkten $B_1$-$B_4$ über das Inlet 1 der *pointProjector*-Abstraction übergeben wird, wird damit eine Matrix mit den korrespondierenden 2D-Punkten $Bp_1$–$Bp_4$ über das Outlet 1 ausgegeben (Inlet 1 und Outlet 1 sind auf Abbildung 21 zu erkennen).

### 3.4.4 Berechnung der Homographie-Matrix

Die Berechnung der Homographie-Matrix ist der letzte Schritt, der im *Runtime*-Abschnitt von *C_stone* durchgeführt wird. Die Berechnung findet unter Einsatz des Externals *h.jit.findhomography* statt (siehe Abbildung 15, Abschnitt I) [HON11].

Im vorherigen Kapitel wurde beschrieben, wie die gesuchten Eckpunkte $Bp_1$–$Bp_4$ des Zielbildes auf der Projektor-Bildebene ermittelt werden können. Nun sollen hingegen alle Bildpunkte eines beliebigen Ausgangsbildes so transformiert werden können, dass an der Wand ein rechteckiges Bild entsteht. Wie in Kapitel 2.4 beschrieben wurde, genügen vier Punkte einer Ausgangsebene und deren korrespondierende Punkte auf der Zielebene, um eine entsprechende Transformationsmatrix zu ermitteln. Ist diese bekannt, kann sie auf alle weiteren Punkte der Ausgangsebene angewendet werden, um entsprechend deren korrespondierende Punkte auf der Zielebene zu bestimmen.

*h.jit.homography* beinhaltet die *OpenCV*-Funktion *cvFindHomography()*. Das External erwartet eine 2 × 4-Matrix, in der jeweils die x- und y-Werte der Bildpunkte auf der Zielebene enthalten sind. Dabei werden diesen Zielpunkten intern die normalisierten Eckpunkte einer Ausgangsebene gegenübergestellt. Aus diesem Grund werden Werte zwischen −1 und 1 für die Beschreibung der Eckpunktpositionen auf der Zielebene benötigt und keine absoluten Pixelkoordinaten. Dabei entspricht der Punkt mit den Koordinaten (−1 | −1) der linken oberen Bildecke und der mit den Koordinaten (1 | 1) der Ecke rechts unten im Bild. Die Normalisierung der Punkte $Bp_1$–$Bp_4$, welche nach der Transformation in Kapitel 3.4.3 in absoluten Pixelkoordinaten angegeben sind, erfolgt im Subpatcher *normalize_-1_to_1* (siehe Abbildung 15 auf Seite 37 unterhalb des Abschnitts H).

Nach der Normalisierung durch *normalize_-1_to_1* erhalten wir die Matrix *dst_points* (siehe (3.16)).

**(3.16)**

$$dst\_points = \begin{bmatrix} Bp_{1x} & Bp_{2x} & Bp_{3x} & Bp_{4x} \\ Bp_{1y} & Bp_{2y} & Bp_{3y} & Bp_{4y} \end{bmatrix}$$

Diese Matrix wird an *h.jit.homography* weitergeleitet, wo die Berechnung der Homographie-Matrix stattfindet. Codezeilen 88 bis 100 sind Quellcode von *h.jit.homography* entnommen und verdeutlichen, welche Parameter an die *OpenCV*-Funktion *cvFindHomography()* übergeben werden.

```
88              src_points = cvCreateMat( 4,2 , CV_32FC1 );
89              // fill matrix
90                          cvSetReal2D(src_points, 0, 0, -1);
91                          cvSetReal2D(src_points, 0, 1, 1);
92                          cvSetReal2D(src_points, 1, 0, 1);
93                          cvSetReal2D(src_points, 1, 1, 1);
94                          cvSetReal2D(src_points, 2, 0, 1);
95                          cvSetReal2D(src_points, 2, 1, -1);
96                          cvSetReal2D(src_points, 3, 0, -1);
97                          cvSetReal2D(src_points, 3, 1, -1);
98                          // […]
99              // call cvFindHompgraphy()
100             cvFindHomography( src_points, dst_points, &outmat2, 0, 0, 0);
```

In Codezeile 88 wird eine Matrix namens *src_points* erzeugt, die in den darauffolgenden Zeilen mit Werten gefüllt wird. Über die Funktion *cvSetReal2D()* wird jeweils der Wert einer Zelle gesetzt. Nach der Ausführung der Codezeilen 90–97 enthält die Matrix *src_points* die in (3.17) dargestellten Werte.

**(3.17)**

$$src\_points = \begin{bmatrix} -1 & 1 & 1 & -1 \\ 1 & 1 & -1 & -1 \end{bmatrix}$$

In der darauffolgenden Zeile (Codezeile 100) ist der eigentliche Aufruf der Funktion *cvFindHomography()* beschrieben. Neben der Matrix *src_points* wird die Matrix *dst_points* übergeben. Sie enthält dabei die normalisierten Punkte $Bp_1$–$Bp_4$, welche dem External *h.jit.homography* in Form einer *Jitter*-Matrix als Zielpunkte überreicht werden (siehe (3.16)).

Das Ergebnis der Berechnungen von *cvFindHomography()* wird in die Matrix *outmat2* geschrieben, die als drittes Argument an die Funktion übergeben wurde (siehe Codezeile 100). Diese Matrix wird schließlich an die *Max*-Umgebung zurückgeführt. Sie enthält die gesuchte Homographie-Matrix und wird vor der Übergabe auf zwei Arten formatiert:

1. als 4 × 4-*OpenGL*-Matrix (Outlet 1)
2. als 3 × 3-*OpenCV*-Matrix (Outlet 2).

Je nach Gebrauch können diese Matrizen unterschiedlich eingesetzt werden. Im Fall von *C_stone* wird die eigentliche Transformation des Ausgangsbildes bzw. -videos in *OpenGL* realisiert. Somit wird hier die 4 × 4-*OpenGL*-Matrix, welche nun als Transformationsmatrix an den Subpatcher *to_OpenGL* weitergegeben wird, verwendet (siehe Kapitel 3.5.1).

## 3.5 Output

### 3.5.1 Die to_openGL Abstraction

Der *Output*-Bereich der Software *C_stone* wird allein durch die *to_openGL*-Abstraction dargestellt. Ein Screenshot der Abstraction ist im Anhang auf Seite 91 einzusehen. Sie erledigt wichtige Funktionen der Kommunikation mit der *OpenGL*-API. Das Ausgangsbild des Projektors wird in einem *OpenGL*-Render-Kontext mithilfe eines *jit.gl.sketch*-Objekts erzeugt. Dazu wird bei der Initialisierung im *jit.gl.sketch*-Objekt eine Ebene generiert. Das Videobild wird dieser Ebene als Textur zugewiesen. Die Verzerrung des Ausgabebildes anhand der ermittelten Homographie-Matrix wird durch eine Modifikation des *OpenGL-Modelview-Matrix-Stacks* erreicht [OGL04]. Dies geschieht über die Message „glmultmatrix 1006TRANSFORM“, die dem Objekt bei der Initialisierung des Patchers gesendet wird. Tabelle 3.1 veranschaulicht, wie mit der *to_openGL*-Abstraction innerhalb von *MaxMSP kommuniziert werden kann.*

| **Inlets** | **Typ** | **Beschreibung** |
|---|---|---|
| **1** | jit.mat: 4, char, N × M<br>(z. B. Video) | Matrix wird einer Ebene im OpenGL-Kontext als Textur zugewiesen; diese Ebene wird entsprechend der Matrix an Inlet 2 verzerrt. |
| **2** | jit.mat: 1, float32, 4 × 4 | Korrekturmatrix; definiert die perspektivische Verzerrung der in Inlet 1 empfangenen Textur. |
| **3** | jit.mat: 4, char, N × M<br>(z. B. Video) | Matrix wird einer Ebene im OpenGL-Kontext als Textur zugewiesen; ist immer bildfüllend bezogen auf den Videoausgang. |
| **4** | Message | Messages an das *jit.window*-Objekt |
| **Argumente** | 1: Name des *OpenGL*-Render-Kontext | |

*Tabelle 3.1: to_openGL-Abstraction Übersicht*

## 3.6 GUI

Mit der Entwicklung von *C_stone* sollte ein Prototyp erstellt werden, mit welchem die prinzipiellen Überlegungen zur Umsetzung der hier vorgestellten Trapezkorrektur überprüft und getestet werden können. Hierfür wurde in *MaxMSP* auch eine Benutzeroberfläche erstellt, die während der Testphasen (siehe Kapitel 4) einen schnellen Zugriff auf die wichtigen Parameter der Software bot.

Der blaue Bereich in Abbildung 9 auf Seite 29 steht für alle Elemente der Benutzerinteraktion. Die strikte Trennung von Interfaceelementen und Programmkern in *MaxMSP* ist nicht vorgesehen. Dies äußert sich vor allem darin, dass sogenannte UI-Objekte neben ihrer Funktion als Interfaceobjekt meist auch andere Funktionen erfüllen. Aus Gründen der Übersicht bietet es sich aber bei komplexen Programmen an, bei der Implementierung eine „weiche“ Trennung zwischen Interface und Programmkern zu realisieren.
Einzelne Objekte können in *MaxMSP* dem sogenannten Präsentationsmodus zugewiesen werden. Befindet sich ein Patcher in diesem Modus, so verschwinden alle Objekte, die diesem nicht explizit zugeordnet wurden. Die übrigen dem Modus zugeordneten Objekte können nun beliebig neu angeordnet und etwa in ihrer Größe verändert werden, sodass eine benutzerfreundlichere Anordnung der UI-Elemente möglich ist. Beendet man den Präsentationsmodus in *MaxMSP*, so befinden sich alle Objekte wieder am ursprünglichen Ort innerhalb des Patchers.

Bei allen Bedienelementen in *C_stone* handelt es sich um Standard-UI-Objekte der *MaxMSP*-Umgebung. Für die bessere Übersicht und ein einheitliches Aussehen wurde jedoch ein individuelles Farbprofil auf alle Elemente angewandt.

Der Haupt-Max-Patcher der Software *C_stone* ist im Anhang auf Seite 86 mit allen Objekten eingeblendet dargestellt. Die darauffolgende Abbildung zeigt denselben Patcher im Präsentationsmodus, für den die Objekte in vier Gruppen angeordnet wurden (siehe Seite 87). Dieser Patcher wird allerdings beim Programmstart nicht direkt geladen. Unter Verwendung eines sogenannten *bpatcher*-Objekts bietet *MaxMSP* eine besondere Möglichkeit, Abstractions in einem übergeordneten Patcher zu integrieren. Wie durch ein Fenster kann mittels eines *Bpatchers* in eine Abstraction „hineingeschaut“ werden. Für die Software wurde ein weiterer Patcher programmiert, welcher lediglich das *bpatcher*-Objekt und vier Buttons beinhaltet (siehe Abbildung 22). Der Haupt-Patcher der Software *C_stone* wird nun im *bpatcher* geladen und der Nutzer bestimmt über die Knöpfe, in welchen Bereich des Patchers „geschaut“ werden soll.

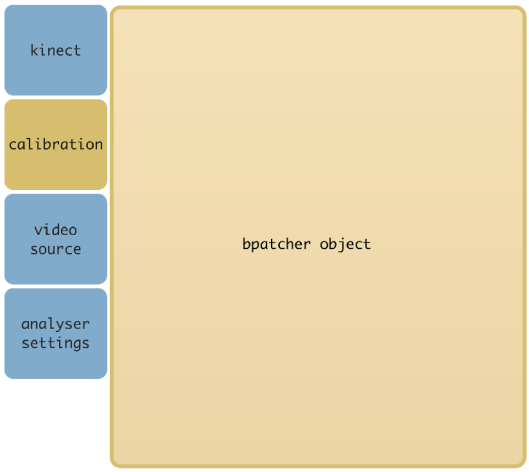


***Abbildung 22: C_stone GUI Patcher***

Über die Buttons im GUI-Patcher (links in Abbildung 22) können die einzelnen Ansichten der Oberfläche sichtbar gemacht werden. In den folgenden Abschnitten wird genauer auf die Bedienelemente und deren Funktion in den einzelnen Ansichten eingegangen werden. In Abbildung 23 sind Screenshots aller vier Ansichten in einer Übersicht dargestellt.

### 3.6.1 Kinect

In dieser Gruppe ist ein Button zur Aktivierung des *Kinect*-Sensors enthalten. Der Button muss nach dem Starten der Software betätigt werden. Wird die *Kinect* nicht über den Button aktiviert, kann keine automatische Trapezkorrektur stattfinden. Außerdem steht ein Vorschaufenster zur Betrachtung der *Kinect*-Tiefenmaske zur Verfügung. Optional kann in dieser Ansicht vom Nutzer die Kalibrierung zwischen den *Kinect*-Kameras aktiviert bzw. deaktiviert werden. (Siehe Abbildung 23 links oben).

### 3.6.2 Calibration

Im GUI-Abschnitt *Calibration* sind alle Elemente zur Durchführung der Projektor-*Kinect*-Kalibrierung enthalten. Zunächst kann in dieser Ansicht – über das Setzen eines Häkchens – der Kalibrierungsmodus aktiviert werden. Ist der Modus aktiv, wird auf dem Projektor automatisch ein Schachbrettmuster ausgegeben und im Vorschaufenster des *Calibration*-Bereichs erscheint ein Bild mit den erkannten Ecken des Schachbrettmusters (ähnlich Abbildung 7 auf Seite 23). Wurden alle Ecken erkannt, so kann mit dem *grab*-Button eine Aufnahme gespeichert werden.

Dabei wird der Zähler, dessen Stand rechts neben dem Button angezeigt wird, inkrementiert. Nachdem die Aufnahmen durchgeführt wurden, kann anhand des Buttons *calculate* die Berechnung der Kalibrierung ausgeführt werden. Die errechnete intrinsische Matrix des Projektors sowie Rotations- und Translationsvektor werden dann rechts in den Tabellen angezeigt. Wenn diese Werte nun nach Einschätzung des Nutzers in Ordnung sind, können sie mit dem Button *use calib* übernommen werden. Bei fehlerhaften Werten kann die Kalibrierung mit *reset* zurückgesetzt werden, um anschließend den Vorgang zu wiederholen.

Bei Betätigung des Buttons *load cal file* wird in der Datei *kinect_projector_cal.txt* nach abgespeicherten Kalibrierungsdaten gesucht. Die Textdatei muss sich im Verzeichnis des Patchers befinden. Sie wird automatisch geladen, wenn die Software *C_stone* gestartet wird.

(Siehe Abbildung 23 rechts oben.)

### 3.6.3 Video Source

In dieser Ansicht werden Einstellungen bezüglich der Videoquelle vorgenommen. Es können zwei parallele Quellen ausgewählt werden: Die erste erscheint korrigiert auf dem Projektor-Ausgangsbild, je nach dem, wie der Projektor auf die Wand gerichtet ist. Die zweite Quelle wird immer im Vollbild ausgegeben und ist von der Korrektur nicht betroffen. Sie kann etwa bei Tests oder Versuchen mit der Software als Referenz verwendet werden, sodass die Auswirkungen der Korrektur besser veranschaulicht und verstanden werden können. Zudem lässt sich hier die Einstellung *chessboard* wählen, die das für die Kalibrierung notwendige Schachbrettmuster ausgibt. Die Helligkeit dieses Schachbrettmusters lässt sich dabei zwischen den Werten 0 und 100 % regulieren.

Als korrigierte Ausgabe lassen sich drei unterschiedliche Quellen wählen:

- *movie*
- *desktop*
- *pattern*

Bei der Einstellung *movie* kann über den Button „*open…“* ein Video geladen werden, welches bei Betätigung des *play*-Buttons abgespielt wird. *desktop* gibt einen beliebigen Bereich des PC-Desktops auf dem Projektor wieder. Der Bereich kann über die Werte für *size* und o*ffset* eingestellt werden. Ist *pattern* angewählt, so wird ein animiertes Punktemuster zu Demo- bzw. Testzwecken ausgegeben. In dieser Ansicht kann zudem noch das Seitenverhältnis der Quelle über ein Menü ausgewählt werden. Ebenso steht eine Vorschau des Ausgangsbildes zur

Verfügung, welches aktuell auf dem Projektor ausgegeben wird. Da diese Vorschaufunktion eine hohe CPU-Last verursacht, kann sie über das Setzen eines Häkchen wahlweise aktiviert oder deaktiviert werden. (Siehe Abbildung 23 links unten.)

### 3.6.4 Analyser Settings

In der Ansicht *Analyser Settings* lassen sich weitere Einstellungen vornehmen, die sich direkt auf das Verhalten der Korrektur auswirken. Zum einen besteht die Möglichkeit die Korrekturfunktion von *C_Stone* zu unterbrechen bzw. zu pausieren. Hierfür muss *activate dynamic correction* (ADC) abgewählt werden. Dabei wird etwa ein Video zwar weiter abgespielt, aber die aktuelle Korrekturposition wird gehalten und nicht weiter verändert. Die Einstellung *update output* unterbricht hingegen sowohl die Videoausgabe als auch die Korrektur. Sie stellt demnach eine Funktion zum generellen Pausieren der Software und zur Entlastung der CPU des Computers dar.
Unterhalb dieser Einstellungsmöglichkeiten befindet sich eine Anzeige für die durchschnittliche Framerate, die aktuell mit *C_stone* erreicht wird. Diese Anzeige ist hilfreich bei Versuchen und Tests, die sich auf die Zuverlässigkeit und die Geschwindigkeit des Systems beziehen.
Unmittelbar unter der Anzeige befindet sich der *smoothe correction* Wert. Über diesen Parameter kann die Reaktionsgeschwindigkeit des Systems bei Korrekturen eingestellt werden (siehe auch Abbildung 15, Abschnitt E). Große Werte verlangsamen die Reaktionszeit, sorgen aber auch für geschmeidigere und weichere Übergänge unterschiedlicher Korrekturzustände.
Der darauffolgende Parameter *Image size* wirkt sich auf die tatsächliche Größe der Projektion aus. Mit tatsächlicher Größe ist dabei Höhe der Projektion in Metern, wie sie an der Projektionsfläche erscheint, gemeint. Sie bleibt, unabhängig vom Abstand des Projektors zur Projektionsfläche, konstant.
Der letzte Parameter *triangle size* bezieht sich auf die Größe des Dreiecks, welches durch die drei Punkte im darunterliegenden Vorschaufenster dargestellt wird. Dabei handelt es sich um eine Visualisierung der Stellen des *Kinect*-RGB-Bildes, an welchen die drei Punkte $P$, $K_1$ und $K_2$ zur mathematischen Definition der Projektionsebene *Ep* ermittelt werden (siehe auch Kapitel 3.4.2). Die Entfernung dieser drei Punkte zueinander kann über *triangle size* eingestellt werden. Ebenso kann die Position des Dreiecks direkt im Vorschaufenster durch Verschieben des grauen Kreises erreicht werden. Dabei ist zu beachten, dass sich dadurch das gesamte korrigierte Bild im Ausgangsfenster bzw. der Projektion mit verschiebt. (Siehe Abbildung 23 rechts unten.)

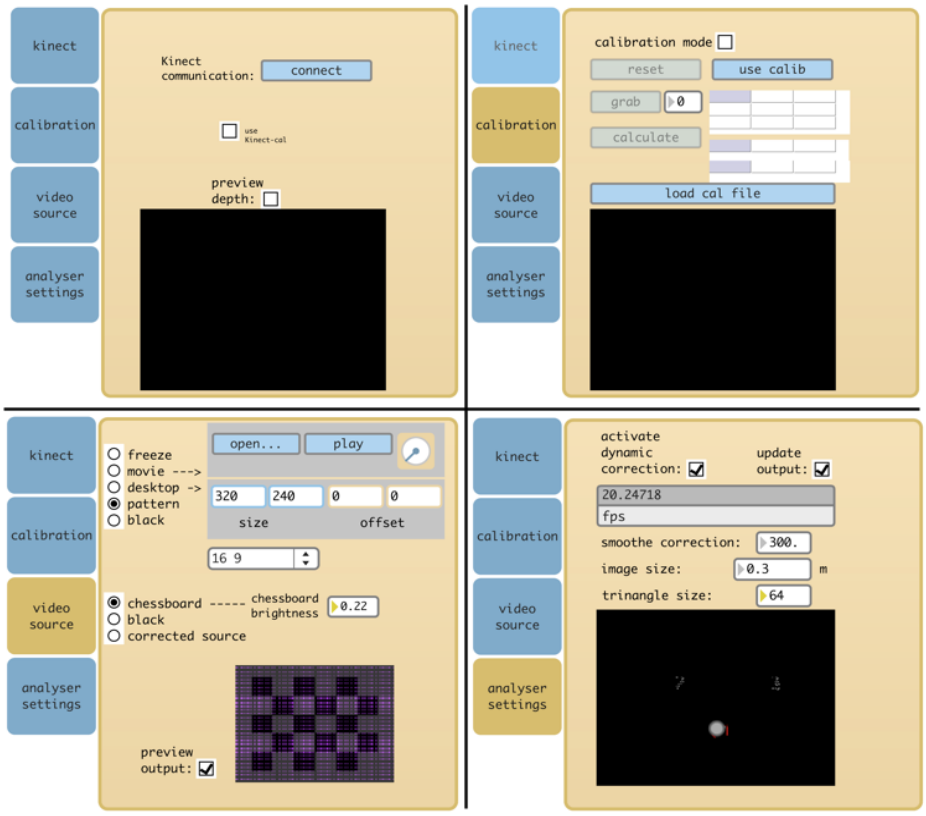


*Abbildung 23: Die vier Ansichten der C_stone-Benutzeroberfläche*

# Kapitel 4: Prüfung und Bewertung

In diesem Kapitel sollen, neben einer Beschreibung der Zusammenstellung der Komponenten, einige Eigenschaften des Systems analysiert werden und auf die Anwendbarkeit der zugrundeliegenden Software *C_stone* eingegangen werden. Dabei beleuchten Versuche die Praxistauglichkeit, deuten jedoch auch auf Schwierigkeiten hin, die etwa im Zusammenhang mit der *Kinect*-Tiefenerkennung entstehen können. Zudem wird anhand einiger Tests deutlich, wie hoch die CPU-Belastung beim Betrieb von *C_stone* für einen Computer ist, was einen Eindruck über die Leistungsfähigkeit der Software vermittelt.

## 4.1 Komponenten und Systemaufbau

### 4.1.1 Komponenten

Der technische Aufbau des Systems beinhaltet im Wesentlichen drei Komponenten:

- *Microsoft-Kinect*-Sensor

Der *Kinect*-Sensor sollte möglichst unbeweglich auf dem Projektor befestigt werden. Zum Anschluss des Sensors an den PC ist eventuell spezielles Zubehör notwendig (Netzteil mit USB-Adapter). Da die Übertragung der Daten von der *Kinect* zum PC per USB eine hohe Bandbreite erfordert, sollte sichergestellt sein, dass ein *Kinect*-Sensor als einziges Gerät an einem *highspeed-USB-Port* angeschlossen ist. Die Übertragung benötigt einen Großteil der verfügbaren Bandbreite eines Ports.

- Videoprojektor

Prinzipiell ist *C_stone* unabhängig vom Projektormodell einsetzbar. Voraussetzung ist, dass sich der Videoprojektor an einen PC anschließen lässt und die Grafikhardware die Nutzung eines sekundären Monitors unterstützt. Die Auflösung und das Seitenverhältnis des Projektors sind wie gewünscht zu wählen. Ebenso ist bei Projektoren mit Zoomobjektiv eine gewünschte Zoom-Einstellung vorzunehmen. Nach der Kalibrierung mit *C_Stone* dürfen keine dieser Parameter verändert werden. Idealerweise sollte die Brennweite des Projektors mit der

Brennweite der *Kinect*-Kameras übereinstimmen oder zumindest in einem ähnlichen Bereich liegen, um bessere Kalibrierungsergebnisse zu erzielen.

- *Apple* PC

Die aktuelle *C_Stone* Software funktioniert ausschließlich mit dem *Apple*-Betriebssystem *Mac OS X*. Grund hierfür ist, dass einige der verwendeten *Max*-Externals, welche für dieses Projekt entstanden sind, auf *Mac OS X* -Basis entwickelt und kompiliert wurden.

### 4.1.2 Aufbau und Ausrichtung

Idealerweise sollte die *Kinect* nahe der Optik des Projektors installiert werden. Dabei bietet sich in vielen Fällen die Installation der *Kinect* direkt auf dem Gehäuse des Projektors an. Die Komponenten sollten möglichst unbeweglich miteinander verbunden und nach der Kalibrierung nicht mehr verschoben werden. Abbildung 24 zeigt den hier verwendeten Aufbau.

***Abbildung 24: Kinect auf Projektor***

Die Ausrichtung der *Kinect* sollte durch Betrachtung des Bildausschnitts der RGB-Kamera überprüft werden. Dabei muss der ganze Projektionsbereich im Bild der RGB-Kamera zu sehen sein, vorausgesetzt die Brennweite des Projektors lässt dies zu. Weist der Projektor einen wesentlich geringeren Öffnungswinkel verglichen mit der *Kinect* auf, so ist die Ausrichtung so vorzunehmen, dass sich der Projektionsbereich des Projektors etwa in der Mitte des RGB-Bildes befindet (siehe Abbildung 25).

Als Versuchsgerät diente ein Videoprojektor des Typs *AstroBeamX250* des Herstellers *Anders & Kern*, ein LCD-Projektor mit einer nativen Auflösung von 1024 × 768 Bildpunkten. Dieser Projektor besitzt auch im vollen Weitwinkel einen deutlich kleineren Öffnungswinkel im

Vergleich zur *Kinect*. Dies wird in Abbildung 25 deutlich, in welcher eine Aufnahme des Sichtbereichs der *Kinect*-RGB-Kamera dargestellt ist. Der Projektor projiziert hier zur Unterstützung bei der Ausrichtung der *Kinect* eine bildfüllende, weiße Farbfläche auf eine Wand.

***Abbildung 25: Projektionsbereich des Projektors aus Sicht der Kinect-RGB-Kamera***

In den Systemeinstellungen des Betriebssystems muss zur Verwendung der Software *C_stone* der Projektor als sekundärer Monitor definiert werden. Zusätzlich muss sichergestellt werden, dass er rechts vom Hauptbildschirm positioniert wird, bevor die Software gestartet wird. Abbildung 26 zeigt die Positionierung unter *Systemeinstellungen → Monitore → Anordnen.*

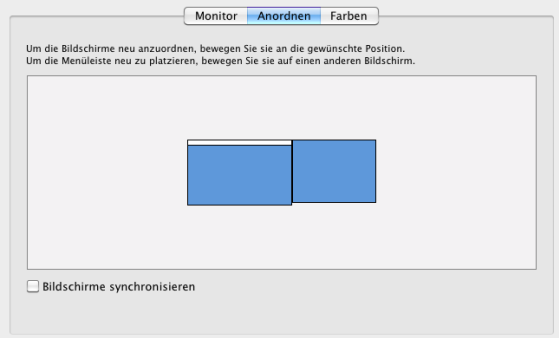


***Abbildung 26: Monitoranordnung in den Mac-OS-X-Systemeinstellungen***

## 4.2 Kalibrierung

Die Kalibrierung der *Kinect* und des Projektors ist ein wichtiger Arbeitsschritt für eine zuverlässige und korrekte Trapezkorrektur im Normalbetrieb. Im Praxistest erweist sich die Kalibrierung jedoch als relativ fehleranfällig. So ist es in vielen Fällen erforderlich, die Kalibrierung mehrmals durchzuführen, bis korrekte Werte für intrinsische und extrinsische Parameter des Projektors ermittelt werden können. Dies könnte auf mehrere Faktoren

zurückzuführen sein: Die Algorithmen der *OpenCV-calibrateCamera*-Funktion berechnen eine Schätzung der Projektordaten anhand einer Ausgleichsrechnung über die Methode der kleinsten Quadrate [ORL08]. Die Ermittlung der Näherungsfunktionen reagiert sehr sensibel auf minimale Veränderungen in den Objekt- und Bild-Koordinaten (siehe Kapitel 3.3). Hier wäre eine örtliche Vorfilterung der Tiefendaten hilfreich, um Rauschen zu verringern und konstantere Werte zu erreichen. Dabei könnten in einem bestimmten Radius um die Analysepunkte mehrere Werte zusammengefasst und eine Mittelwertbildung durchgeführt werden.

Da in der Prototyp-Software *C_stone* keinerlei Fehlererkennung für die Kalibrierung integriert ist, muss vom Nutzer entschieden werden, ob die errechneten Werte richtig sind oder evtl. Fehler enthalten. Der Praxistest zeigte, dass es an dieser Stelle hilfreich wäre, den Kalibrierungsschritt benutzerfreundlicher zu gestalten und die Fehleranfälligkeit zu verringern.

Abbildung 27 zeigt das projizierte Schachbrettmuster an einer weißen Fläche. Im Vorschaufenster der Software *C_stone* (Abbildung 27, rechts) wird dem Nutzer angezeigt, ob Schachbrettecken erkannt wurden oder nicht. Vom Nutzer sollten gute Bedingungen für die Tiefenerkennung während der Kalibrierung sichergestellt werden. Dabei muss darauf geachtet werden, dass kein Sonnenlicht in den Raum gelangt und eine Distanz von ca. 1 bis 1,5 m zur Projektionsfläche eingehalten wird.

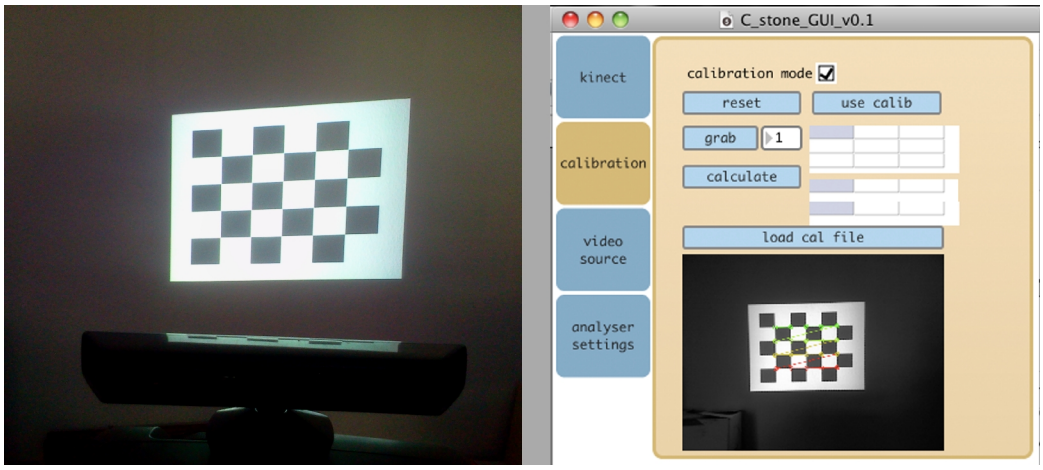


***Abbildung 27: Schachbrettprojektion im Kalibrierungsmodus und Vorschau Calibration***

## 4.3 Projektionsbereich und Bildgröße

Der **Projektionsbereich** und die **Bildgröße** sind im Kontext dieser Arbeit differenziert zu betrachten. Der Projektionsbereich ist durch die Brennweite und damit durch den Öffnungswinkel des verwendeten Projektors definiert. Die Bildgröße hingegen bestimmt, wie groß das korrigierte Ausgangsbild tatsächlich auf der Projektionsfläche erscheint. Dieser Parameter kann vom Nutzer durch Einstellungen in der Software beeinflusst werden (siehe 3.6.4). Die Bildgröße wird in Metern angegeben und ist unabhängig vom Projektionsbereich. So könnte ebenfalls eine Bildgröße eingestellt werden, die vom Projektionsbereich des Projektors nicht erreicht werden kann. In diesem Fall wäre nur ein Ausschnitt des Bildes zu sehen.

Neben der Bildgröße lässt sich vom Nutzer die Position des projizierten Bildes innerhalb des Projektionsbereichs bestimmen. Im Normalbetrieb („Runtime“, siehe Kapitel 3.4) analysiert *C_stone* die Tiefenmaske der *Kinect* an drei Punkten. Diese drei Analysepunkte definieren eine Ebene, und gemäß der Lage dieser Ebene zum Projektor wird das Ausgabebild verzerrt. Die Messpunkte sind in einem gleichseitigen Dreieck auf der Projektionsfläche angeordnet. Ihre Position kann in der GUI-Ansicht „*analiser settings*“ anhand des RGB-Bildes der *Kinect* eingesehen und verändert werden (siehe Kapitel 3.6.4). Ebenso können hier vom Nutzer die Abstände der Punkte zueinander beeinflusst werden und so etwa eine Anpassung an unterschiedlich große Projektionsflächen erreicht werden. Durch eine Verschiebung des grauen Kreises lassen sich die Punkte zudem im RGB-Bild positionieren. Dadurch wird auch die Projektion auf der Projektorbildebene verschoben, da der graue Kreis den Mittelpunkt des projizierten Bildes repräsentiert.

## 4.4 Maximale Projektionswinkel

Zur Ermittlung der minimalen und maximalen Projektionswinkel, bei denen die Trapezkorrektur von *C_stone* zufrieden stellende Ergebnisse liefert, gestaltete sich der Versuchsaufbau wie folgt: Ein Videoprojektor vom Typ *AstroBeamX250* und die *Microsoft-Kinect* wurden, wie in Abbildung 24 gezeigt, aufgebaut und im *calibration mode* kalibriert (siehe Kapitel 3.6.2). Der Projektor wurde auf dem Fußboden mithilfe eines projizierten Kreuzes genau entlang einer Markierung zunächst orthogonal zu einer beweglichen weißen Projektionsfläche ausgerichtet (siehe Abbildung 28). Die bewegliche Projektionsfläche wurde dann um ihre horizontale und anschließend um die vertikale Achse gedreht, um die maximalen Projektionswinkel zu

ermitteln. Dabei wurde sichergestellt, dass sich die jeweilige Drehachse stets in einem Abstand von 1,4 Metern zum Projektor befand.

***Abbildung 28: Versuchsaufbau Winkelmessung***

Der Versuch lieferte für die horizontale Korrektur einen maximalen Projektionswinkel von über 70° in beide Richtungen (siehe Abbildung 29). In vertikaler Richtung wurden bis zu Winkeln von etwa 60° zufriedenstellende Ergebnisse erreicht. (0° in horizontaler und vertikaler Richtung entspricht dabei der in Abbildung 28 dargestellten Aufstellung.)

***Abbildung 29: Maximale Auslenkung horizontal***

Die Ergebnisse zeigen, dass das vorgestellte Verfahren bis zu sehr großen Winkeln eine dynamische Korrektur erreichen kann. Allerdings kam es in den jeweiligen Extrema zu deutlichen Artefakten und unerwünschten Nebeneffekten, die auf mehrere Faktoren zurückzuführen sind. Vor allem ein stets vorhandenes, unerwünschtes Alternieren des Bildes zwischen Korrekturzuständen steigerte sich proportional zum Projektionswinkel, bis es bei großen Winkeln als sehr störend auffiel. Ebenso nahmen Fehler bezüglich der absoluten Größe des Bildes immer stärker zu. So wurde beim vorliegenden Versuch eine Bildgröße von 0.7 m definiert (siehe auch Kapitel 3.6.4). Bei einer horizontalen Auslenkung, wie sie in Abbildung 29 zu sehen ist, wurde eine Breite von 62 cm gemessen, was einer Abweichung von ca. 14 % zur gewünschten Größe entspricht. Da jedoch das Seitenverhältnis der Projektion bzw. des dargestellten Bildes nicht beeinträchtigt wurde, stellte sich diese Eigenschaft im Allgemeinen als weniger störend heraus.

Die Ursprünge der auftretenden Probleme lassen sich in drei Kategorien einteilen:

- *Kinect*-Tiefenerkennung und -Datenverarbeitung
- Kalibrierung
- Optische Eigenschaften des Projektors

Je nachdem in welchem Winkel die *Kinect* auf eine Fläche gerichtet ist, kann es zu einer Verschlechterung der Tiefenerkennung kommen. Diese Eigenschaft ist abhängig vom Material bzw. der Beschichtung einer Oberfläche und den damit verbundenen Reflexionseigenschaften. So kann das projizierte IR-Muster möglicherweise vom *Kinect*-internen Analysealgorithmus schlechter erkannt werden, da große Anteile des projizierten Lichtes bei flachen Einfallswinkeln nicht zur IR-Kamera zurück gelangen. Leichte Unebenheiten auf der Projektionsfläche können ebenfalls Probleme hervorrufen und die Erkennung beeinträchtigen (siehe auch Kapitel 4.7, Flächenbeschaffenheit).

In *C_stone* findet eine Umrechnung der von der *Kinect* übertragenen RAW-Tiefendaten in Meter statt (siehe Kapitel 2.5.2). Die vorgestellte Formel zur Umrechnung der Daten ist eine Näherungsformel, die gewisse Abweichungen vom eigentlichen Abstandswert aufweist (siehe Formel 2.4). Hier wäre eine separate Kalibrierung pro Sensor notwendig, um eventuellen Abweichungen entgegenzuwirken und eine höhere Genauigkeit zu erreichen.

Abweichungen und Ungenauigkeiten der Kalibrierung äußern sich ebenfalls bei extremen Projektionswinkeln. So kann es vorkommen, dass eine misslungene Kalibrierung bei horizontaler Schrägprojektion zufriedenstellende Ergebnisse ermöglicht, jedoch in vertikaler

Richtung schon bei 30°-Winkeln zu auffälligen Fehlern führt. Dies könnte durch Optimierungen im Kalibrierungsverfahren verbessert werden. Denkbar wäre eine bessere Filterung bzw. Glättung der *Kinect*-Tiefenwerte oder die automatische Erkennung geeigneter Schachbrettaufnahmen.

Das weiter oben angesprochene Problem der instabilen Bildlage, das sich durch ein Alternieren zwischen Korrekturzuständen äußert, wäre ebenfalls durch eine entsprechende Filterung der Tiefendaten zu verringern. Dabei wäre eine örtliche Mittelung um die Analysepunkte in der *Kinect*-Tiefenmaske ein entscheidender Faktor zur Lösung.

Unabhängig von den Problemen, die durch die Tiefenerkennung der *Kinect* oder durch die Datenverarbeitung entstehen, treten bei Schrägprojektion einige Artefakte auf, die auf die optischen Eigenschaften von Projektoren zurückzuführen sind. Allen voraus soll an dieser Stelle die Schärfentiefe von Projektoren angesprochen werden. Je nach Projektormodel und -optik muss hier bedacht werden, dass nur in einem bestimmen Bereich, bezogen auf die Entfernung des Projektors zur Projektion, eine scharfe Darstellung möglich ist. Zudem nimmt bei Schrägprojektion die Leuchtdichte mit der Entfernung ab, da pro Flächeneinheit weniger Bildpunkte auf die Projektionsfläche fallen. Bei gleichbleibender Bildgröße nimmt zudem die absolute Auflösung zur Darstellung des korrigierten Bildes ab, da ein Bild, bezogen auf die Lage des Projektors, auf einem kleineren Bereich dargestellt werden muss.

Bei den eben dargestellten Punkten handelt es sich jedoch um Probleme, die sich nur teilweise durch Optimierung der hier vorgestellten Software oder des zugrundeliegenden Verfahrens verbessern lassen. So wäre etwa die Implementierung eines Helligkeitsausgleichs über die Entfernungsinformation der *Kinect* realisierbar. Lediglich durch die Auswahl entsprechender Optiken und bestimmter Projektionsgeräte lassen sich Probleme der Projektors, wie beispielsweise zu geringe Tiefenschärfe, reduzieren.

## 4.5 Distanz zur Projektionsfläche

Beim Betrieb des hier vorgestellten Systems muss eine maximale bzw. minimale Distanz zur Projektionsfläche eingehalten werden. Abgesehen von optischen Einschränkungen seitens des Projektors, sind die minimalen und maximalen Projektionsabstände von der Tiefenerkennung der *Kinect* abhängig. In Abbildung 30 wird dargestellt, wie die Genauigkeit der *Kinect*-Tiefenerkennung mit steigender Entfernung abnimmt. Das Schaubild ist das Resultat einer Messreihe, bei der ein Versuchsgerät in 1 cm-Schritten von einer Wand wegbewegt wurde und

die Werte jeweils in der Mitte der RAW-Tiefenmaske gespeichert wurden. Der an der jeweiligen Position gemessene Wert wird in Abbildung 30 auf der y-Achse dargestellt und als *Bildwert* bezeichnet. Zudem wurde im Schaubild eine Ausgleichskurve (*Fit*) abgebildet, welche den Verlauf einer Näherungsfunktion an die Messpunkte darstellt.

Anhand des Schaubildes wird deutlich, dass die Genauigkeit der Tiefenerkennung rapide abnimmt. So beträgt die Quantisierungsschrittweite bei 2 Metern etwa 1,2 cm. Bei 5 Metern hingegen sind es nur noch etwa 10 cm [Orr11]. Dies beeinflusst auch die Trapezkorrektur mit *C_stone*. Die in Kapitel 4.4 angesprochene, unruhige Bildlage wird mit steigender Entfernung wahrnehmbarer. Auch hier wäre eine Filterung durch Mittelung von nahe gelegenen Punkten um die Analysepunkte der Tiefenmaske eine Maßnahme, welche die Ergebnisse verbessern würde. Der Praxistest zeigte jedoch, dass Abstände um 5 Meter kein Problem für *C_stone* darstellen. Bei dem Versuch waren ideale Lichtbedingungen vorhanden (Raum abgedunkelt, weiße Projektionsfläche).

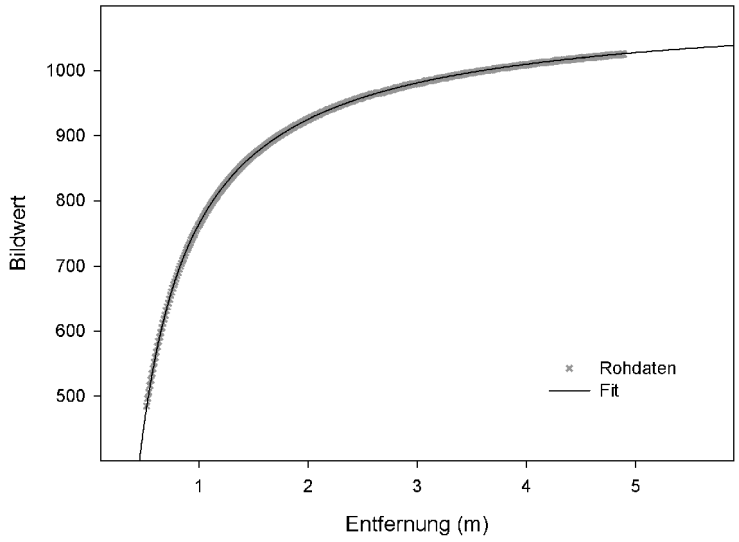


***Abbildung 30: Kinect Bildwerte bezogen auf Entfernung***

Am unteren Ende des Schaubilds in Abbildung 30 ist eine deutlichere Grenze der erfassten Rohdaten sichtbar. Die Messung von Abständen mit der *Kinect* unter circa 60 cm stellte sich als nahezu unmöglich heraus. Bei diesen Entfernungen werden in Bereichen der Tiefenmaske keine Werte mehr übertragen oder sie sind von starkem Rauschen überlagert. Die Tiefenauflösung hingegen ist hier am höchsten. So beträgt beispielsweise die Quantisierungsschrittweite bei einem Abstand von 70 cm etwa 0,16 cm [Orr11]. Beim Betrieb von *C_stone* muss an den

Analysepunkten mindestens ein Abstand von 60 bis 70 cm zur Projektionsfläche eingehalten werden.

Abgesehen von den eben besprochenen Einschränkungen des *Kinect*-Sensors ist hier nochmals auf Faktoren wie etwa der Schärfebereich des verwendeten Projektors hinzuweisen. So wird es selten möglich sein, das Bild eines Projektors bei einem Projektionsabstand von 50 cm scharf stellen zu können.

## 4.6 Anforderungen an den PC

Für diese Arbeit wurde auch eine Analyse der Leistungsanforderungen durchgeführt, welche die Software *C_stone* an einen Computer stellt. Hierfür wurden einige Versuche gemacht, bei denen in mehreren Belastungsstufen jeweils die durchschnittliche erreichte Framerate der Software betrachtet wurde. Als erstes Versuchsgerät *(PC 1)* kam ein *MacBook white* zum Einsatz. Zum Vergleich wurde ein leistungsfähigeres Modell der *MacPro*-Serie *(PC 2)* ausgewählt. Detaillierte technische Daten beider Modelle sind in Tabelle 4.1 aufgeführt.

| | **PC 1** | **PC 2** |
|---|---|---|
| **Typ** | MacBook4,1 | MacPro1,1 |
| **CPU** | Intel Core 2 Duo mit 2,1 GHz | Dual-Core Intel Xeon 2,66 GHz |
| **Hauptspeicher** | 3 GB, 800 MHz | 7 GB, 1,33 GHz |
| **Grafik** | Intel GMA X3100, 144 MB | NVIDIA GeForce 7300 GT, 256 MB |
| **Betriebssystem** | Mac OS X 10.6.8 | Mac OS X 10.6.8 |
| **Software** | MaxMSP/Jitter 5.1.8 | MaxMSP/Jitter 5.1.9 |

*Tabelle 4.1: Technische Daten der Versuchs-PCs*

**Runtime**

Die Anforderungen an den Computer variieren stark, je nachdem welche Videoquelle in der Software *C_Stone* ausgewählt wird. Allein die Decodierung eines hochauflösenden Videos, welches in einem stark komprimierten Format vorliegt, stellt bereits eine hohe Anforderung an den Rechner. Die Berechnungen, die von *C_Stone* durchgeführt werden, und die hohe

Datenrate der *Kinect* können ebenfalls zu niedrigen Frameraten des Systems führen. Im Folgenden wurden daher jeweils die durchschnittlichen Frameraten der Software unter unterschiedlichen Bedingungen analysiert und verglichen. Dabei wurde jeder Versuch dieser Reihe einmal mit eingeschalteter Korrektur und danach mit ausgeschalteter Korrektur (ADC aus) durchgeführt. Die Software wurde auf beiden Testgeräten jeweils eine Minute in vier Belastungsstufen betrieben:

1. Standbild
2. Video (640 × 480), Photo Jpeg
3. Video (1920 × 1080), H.264
4. Desktop-Übertragung vom Hauptbildschirm (640 × 480)

Die Belastungsstufen sind in Abbildung 31 farblich sortiert:

1. Standbild: „still“, Blau
2. Video (640 × 480): „lowres“, Beige
3. Video (1920 × 1080): „highres“, Gelb
4. Desktop-Übertragung: „desktop“, Orange

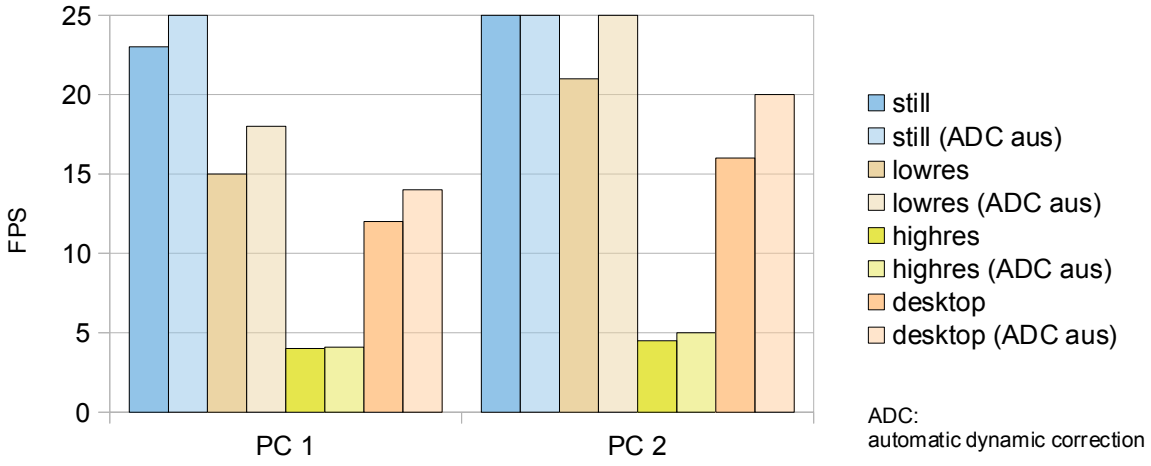


***Abbildung 31: Analyse der Framerate (FPS) im Runtime-Modus***

Die Ergebnisse zeigen, dass der Betrieb von *C_stone* relativ hohe Anforderungen an einen Computer stellt. Dies ist nicht zuletzt auf die hohe Systemauslastung aufgrund der *Kinect*-

Datenübertragung zurückzuführen. Die Schnittstelle benötigt alleine für die Übertragung der RAW-Tiefendaten eine Bandbreite von 640×480×16 Bit/Frame, was etwa 4,7 Mbit pro Frame entspricht. Zudem müssen die RGB-Daten mit 640×480×24 Bit/Frame verarbeitet werden, was etwa 7 MBit pro Frame entspricht. Insgesamt muss demnach eine Datenrate von mindestens (4,7 Mbit + 7 Mbit) × 30Hz = 350 Mbit/s von der USB-Schnittstelle und dem Computer verarbeitet werden. Dabei sind noch keinerlei Steuerdaten und sonstige im USB-Protokoll vorhandenen Daren mit einberechnet, sodass die tatsächliche Datenrate insgesamt noch etwas höher liegt.

Die Ergebnisse in Abbildung 31 zeigen auch, dass die Deaktivierung der Korrektur (ADC – *automatic dynamic correction*) zwar einen geringen Leistungsvorteil bringt, jedoch offensichtlich nicht der alleinige Grund für die CPU-Belastung durch *C_stone* ist. Dies fällt vor allem bei der Betrachtung der gelben Balken in Abbildung 31 auf. Hier erbringt die Deaktivierung der dynamischen Korrektur kaum Leistungsvorteile. Allein die Decodierung und Darstellung eines hochauflösenden *h.264*-codierten Videos in *MaxMSP* scheint eine hohe Systembelastung hervorzurufen.

Eine Umsetzung des theoretischen Modells auf Basis einer anderen Programmiersprache würde vermutlich eine etwas effektivere Programmierung zulassen. Auch eine Umlagerung weiterer Berechnungen auf die GPU wäre an einigen Stellen denkbar. So könnten beispielsweise die Berechnungen der Transformationen von 3D-Punkten auf 2D-Ebenen auf Grafikebene ausgeführt werden, um eine höhere Gesamt-Performance zu erzielen. Die hohe Datenrate der *Kinect* bleibt jedoch enthalten und stellt auch für leistungsfähigere Computer eine Herausforderung dar.

### Calibration

Im Kalibrierungsmodus sind zwei Fälle zu unterscheiden:

1. ein Schachbrettmuster wird im RGB-Bild erkannt
2. das Schachbrettmuster kann vom Algorithmus nicht erkannt werden.

Abbildung 32 stellt die Messergebnisse des Versuchs dar, bei dem die Leistung im Kalibrierungsmodus untersucht wurde. Die erste Messung wurde bei erkennbarem Schachbrettmuster durchgeführt. Hierfür wurde der Projektor auf eine weiße Wand gerichtet auf der die Schachbrettecken für die Software gut identifizierbar waren. Die zweite Messung wurde bei abgedecktem Projektor durchgeführt, sodass kein Schachbrettmuster im Bild der *Kinect* erkennbar war.

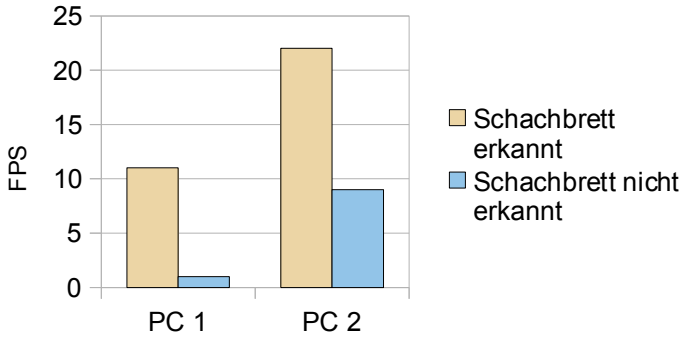


***Abbildung 32: Analyse der Framerate in Kalibrierungsmodus***

Beim zweiten Fall ist eine deutliche Verringerung der Framerate erkennbar. Dies ist auf das External *findChessCorners* zurückzuführen (siehe auch Kapitel 2.6.3.1). Der verwendete *OpenCV*-Algorithmus wird sehr langsam, wenn kein Schachbrettmuster im Bild erkannt wird. Da dieser Modus beim Betrieb von *C_stone* nur selten und für kurze Zeit während der Einrichtung verwendet wird, stellt sich hier die Verlangsamung als nicht störend heraus.

## 4.7 Einflüsse und Systemstabilität

Die Genauigkeit des Systems ist maßgeblich abhängig von der *Kinect*-Tiefenerkennung. Da es sich dabei um ein optisches Verfahren zur Tiefenerkennung handelt, bei dem Licht im IR-Bereich auf Objekte projiziert wird und von diesen wieder reflektiert wird, gibt es eine Reihe äußerer Einflüsse, welche die Funktionalität beeinträchtigen können (siehe auch Kapitel 2.5.1).

### Lichtbedingungen

Umgebungslicht mit hohem IR-Anteil kann zu Problemen bei der *Kinect*-internen Analyse des projizierten IR-Musters führen. Beispielsweise Sonnenlicht, aber auch Licht von Halogenleuchtmitteln, kann zu einer Fehleranfälligkeit bis hin zum Ausfall der *Kinect*-Tiefenerkennung führen. Der Versuch in Abbildung 33 zeigt die *Kinect* Tiefenmaske neben dem Bild der RGB-Kamera. Die *Kinect* wurde hierbei auf eine Szene gerichtet, in der ein weißes Blatt von einer 30 W-Halogenlampe im Abstand von ca. 40 cm beleuchtet wurde. In der Tiefenmaske (Abbildung 33, links) deuten schwarz gefärbte Bildbereiche auf Regionen hin, in denen keine Erfassung von Tiefenwerten möglich ist. Dort wird das *Kinect*-IR-Muster vom Infrarotanteil des Halogenlichts überlagert. Unterschiedliche Grautöne in den übrigen Bereichen der

Tiefenmaske repräsentieren hingegen eine bestimmte Entfernung bzw. Tiefe am jeweiligen Bildpunkt. Somit war an den grauen Stellen eine Erfassung des IR-Musters und damit eine Berechnung der Abstandswerte möglich, da die Intensität des Halogenlichtes nicht ausreichte, um die Tiefenerkennung zu stören.

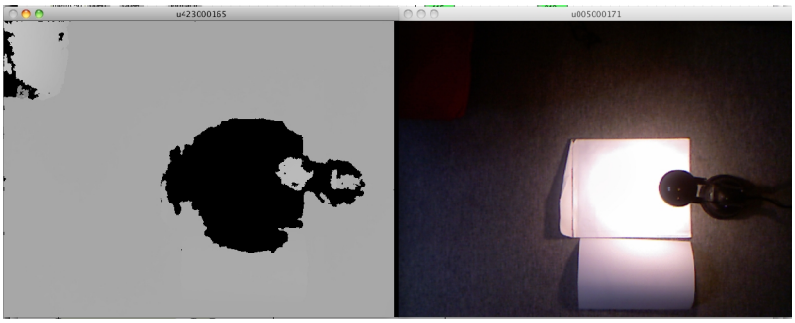

***Abbildung 33: Tiefenmaske unter Einwirkung einer Halogenlampe***

Da beim hier vorgestellten System der Einsatz der *Kinect* nur in Zusammenhang mit Videoprojektoren stattfindet und diese im Allgemeinen eher im Innenraum und bei geringem Umgebungslicht eingesetzt werden, kommen diese Probleme in der Praxis kaum zum Tragen. An dieser Stelle soll ergänzend erwähnt werden, dass sich die Projektion selbst nicht störend auf die Tiefenerkennung auswirkt. Dies ist auf den geringen IR-Anteil der in Videoprojektoren eingesetzten Entladungslampen zurückzuführen.

### Flächenbeschaffenheit

Neben Lichtbedingungen spielt die Oberflächenbeschaffenheit von Objekten für die *Kinect*-Tiefenerkennung eine entscheidende Rolle. Diffus reflektierende Materialien, wie etwa eine matt weiße Wand, ermöglichen dabei eine gute Tiefenerkennung. Das von der *Kinect* projizierte Muster kann dann optimal vom Algorithmus erkannt werden und in Abstandswerte umgesetzt werden. Dunkle oder gar spiegelnde Flächen können jedoch zu inkorrekten Messergebnissen in der Tiefenmaske führen und damit auch zu fehlerhaften Korrekturen der Software *C_Stone*.

Ein Bespiel für eine kritische Oberfläche kann ebenfalls der Abbildung 33 entnommen werden. Das Gehäuse der Halogenlampe ist aus einem schwarzen, glänzenden Kunststoff hergestellt. Betrachtet man die Tiefenmaske an der Stelle des Lampengehäuses, so sind ebenfalls schwarze Bereiche erkennbar. Hier wurden, zum einen durch die Farbe, zum anderen aber durch die

spiegelnde Eigenschaft des Materials, Lichtstrahlen unzureichend in Richtung *Kinect*-IR-Kamera zurückgeworfen.

Ebenso können sehr raue oder nicht planare Flächen zu Problemen bei der Verwendung von *C_stone* führen. Testweise wurde in diesem Zusammenhang auf Flächen bzw. Objekte wie zum Beispiel auf einen Vorhang projiziert, um die Eigenschaften von *C_stone* auf nicht planaren Flächen zu analysieren. Der Versuch zeigte, dass bei großen Entfernungen Unebenheiten in der Projektionsfläche eine geringere Rolle spielen und die Anwendung von *C_stone* weniger stark beeinflussen. Dies ist auf die geringer werdende Auflösung der Tiefenerkennung zurückzuführen (siehe auch Kapitel 4.5).

An gewissen Stellen, wie z. B. an Objektkanten oder in Übergangsbereichen zwischen unterschiedlich gut reflektierenden Materialien, kann die Entfernungsmessung problematisch sein. Oft kommt es dort zu rauschenden Werten in der Tiefenmaske. Auch dies wird in Abbildung 33 deutlich. Bei der Betrachtung des Lampengehäuses in der Tiefenmaske fällt auf, dass die Übergänge nicht scharf sind, sondern unförmige, ausgefranste Ränder die Kanten verwischen. Bei *C_stone* äußert sich dies in einer Art Zittern oder Alternieren zwischen zwei Korrekturzuständen. Dies kann durch die Einstellung „*smoothe correction*“ etwas verringert werden (siehe Kapitel 3.6.4).
Ein weiterer Lösungsansatz, um die Verwendung von *C_stone* auf rauen, unebenen Flächen zu verbessern und resistenter gegen Rauschen der Tiefenwerte zu machen, wäre die örtliche Mitteilung der Tiefenwerte. So könnte etwa bei der Projektion auf einen Vorhang der Mittelwert mehrerer Werte der Tiefenmaske errechnet werden, welche in einem bestimmten Radius um die gewünschten Analysepunkte liegen. Da bei der Verwendung von *C_stone* von nahezu planaren Projektionszielen ausgegangen wird, könnte diese Mittelung über große Bereiche in der Tiefenmaske durchgeführt werden.

**Objekte im Analysebereich**

Gegenstände oder Personen im Bereich einer der Analysepunkte der Software *C_Stone* können schnell zu falschen Ergebnissen führen. Wenn sich etwa eine Person bei einer Präsentation im Projektionsbereich aufhält, ändert sich evtl. sprunghaft das korrigierte Bild, je nachdem wo sich die Person befindet. Dies kann sich in sehr auffälligen und störenden Änderungen zwischen zwei Korrekturzuständen äußern. *C_Stone* bietet daher die Möglichkeit, die Korrektur zu pausieren. Ist das System zur Zufriedenheit des Nutzers eingerichtet, kann die Funktion „activate dynamic correction“ deaktiviert werden (siehe Kapitel 3.6.4). Die aktuelle Korrektur

wird nun gehalten und ist nicht mehr abhängig von der Projektionsfläche. Zudem findet durch die Deaktivierung eine Entlastung der CPU statt (siehe Kapitel 4.6).

Ebenso wäre an dieser Stelle denkbar, ein weiteres zeitliches Filter zu implementieren, welches solch sprunghafte Veränderungen zwischen zwei Abständen in der Kinect Tiefenmaske registriert und ignoriert. Tritt eine Person in den Projektionsbereich ein, so wird eine plötzliche Veränderung der Position eines der drei Analysepunkte registriert. Dies könnte vom System als unerwünschtes Ereignis erkannt werden und der vorherige Korrekturzustand gehalten werden. Selbstverständlich müssten solche Funktionen vom Nutzer aus- und einschaltbar sein.

### Zuverlässigkeit

Um die Stabilität des Systems zu bewerten, wurde *C_stone* sowohl im Kalibrierungsmodus als auch im Normalbetrieb auf zwei PCs über einen längeren Zeitraum betrieben. Dabei wurden zwei Versuchsreihen durchgeführt (PC 1 und PC 2, siehe Tabelle 4.1):

- Runtime-Modus mit lowres-Video (640 × 480)

Zunächst wurde im Normalbetrieb ein 640 × 480-Video auf dem Projektor ausgegeben. Nach 60 Minuten wurde jeweils der Versuch abgebrochen. Die CPU Auslastung blieb während der gesamten Versuchsdauer konstant und auch die Speicherauslastung zeigte keine bemerkenswerten Veränderungen.

- Calibration-Modus

Problematischer zeigte sich der Test im Kalibrierungsmodus. Hier fiel nach kurzer Laufzeit eine wachsende Speicherauslastung auf, was evtl. auf ein Speicherleck im External *findChessCorners* zurückzuführen ist. Nach etwa 20 Minuten beendet das Betriebssystem auf beiden Testgeräten die Software automatisch.

# Kapitel 5: Zusammenfassung und Ausblick

## 5.1 Zusammenfassung

Ziel dieser Arbeit war es, ein Verfahren zur automatischen dynamischen Trapezkorrektur zu entwickeln. Als Werkzeug standen eine *Kinect*-Tiefenkamera und die grafische Programmierumgebung *MaxMSP/Jitter* zur Verfügung. Das Verfahren sollte so angelegt sein, dass die Trapezkorrektur unabhängig von Markern und damit unabhängig von einer Präparierung der Projektionsflächen funktioniert. Dieses Ziel wurde mit der Entwicklung der Prototyp-Software *C_stone* erfolgreich umgesetzt.

In Kapitel 2 wurden sowohl elementare Themen der projektiven Geometrie, Kamera- bzw. Projektorkalibrierung als auch grundlegende technische Eigenschaften der verwendeten Komponenten besprochen. Darauf aufbauend wurden theoretische Modelle entwickelt und weiterverarbeitet, welche letztendlich sowohl eine Kalibrierung der Komponenten Projektor und *Kinect* als auch eine Berechnung der Homographie-Matrix zur Verzerrung eines Videobildes ermöglichen. Anhand der in Echtzeit berechneten Homographie-Matrix kann eine Korrektur des Ausgangsbildes vorgenommen werden, sodass bei Schrägprojektion stets ein rechteckiges Bild auf einer planaren Projektionsfläche erscheint. Ein wichtiger Schritt in diesem Zusammenhang war auch die Umsetzung der Berechnung der Bild-Eckpunkte, worauf in Kapitel 3.4.2 ausführlich eingegangen wurde. Diese Idee setzte dabei die Basis für das ortsunabhängige Verfahren zur Trapezkorrektur in Echtzeit.

Die theoretischen Annahmen und Überlegungen konnten erfolgreich in *MaxMSP/Jitter* implementiert werden, was in Kapitel 3 ausführlich dargestellt wurde. Neben der Implementierung der notwendigen Berechnungen im Programmkern von *C_stone*, wurde eine Benutzeroberfläche entwickelt, welche die wichtigsten Parameter und Funktionen für Präsentations- und Analysezwecke zugänglich macht.

Die Nutzung der Programmierumgebung *MaxMSP/Jitter* ermöglichte dabei eine übersichtliche und leicht nachvollziehbare Programmierung und machte die Software *C_stone* zur geeigneten Basis für Versuche und für weitere Implementierungen. Die Realisierung erforderte zudem die Anwendung einiger Funktionen der *Computer-Vision*-Bibliothek *OpenCV*. Da die in diesem Zusammenhang entstandenen *Max*-Externals in vielen anderen Bereichen Anwendung finden können, stellt deren Entwicklung eine Erweiterung der Programmierumgebung an sich dar. So

wird beispielsweise durch die in Kapitel 2.6.3.1 vorgestellten Externals die Kamerakalibrierung in *MaxMSP* ermöglicht, wofür zuvor keine Möglichkeiten geboten waren.

Eine ausführliche Analyse der wichtigsten Eigenschaften und Merkmale des Systems und der Software *C_stone* stellen die Leistungsfähigkeit aber auch die Einschränkungen der Methode dar. So zeigte sich, dass die Software relativ hohe Anforderungen an Computersysteme stellt und daher zum reibungslosen Betrieb einen sehr leistungsfähigen Computer benötigt. Jedoch zeigte sich auch, dass die automatische Trapezkorrektur nicht maßgeblich für die hohe Rechenanforderung ist. Vielmehr scheinen die hohe Datenrate der *Kinect* und die simultane Decodierung hochauflösender, stark komprimierter Videos das System zu belasten (siehe auch Kapitel 4.6).

## 5.2 Ausblick

Die hier erarbeitete und vorgestellte Methode erwies sich als geeignetes Verfahren zur automatischen dynamischen Trapezkorrektur auf planaren Flächen. Der Software-Prototyp *C_stone* vermittelt dabei erste Eindrücke welche Möglichkeiten durch eine dynamische Trapezkorrektur gegeben sind. Um jedoch alle Möglichkeiten einer softwarebasierten Lösung auszuschöpfen, wäre die Integration des Systems etwa in eine bestehende Medienserver- oder VJ-Software sinnvoll. Auch eine Umsetzung weiterer Programmkomponenten in einer Programmiersprache wie *C* oder *C++* wäre ein Schritt, durch welchen die Effektivität und Leistungsfähigkeit der Software gesteigert werden könnte.

Weitere Verbesserungsvorschläge beziehen sich vor allem auf die Kalibrierung zwischen Projektor und *Kinect*. Durch die Implementierung eines Vorfilters könnten Fehler, die aufgrund von verrauschten Werten der Tiefenmaske entstehen, minimiert werden. Ebenso könnten die Neugestaltung der Benutzeroberfläche und der Einbau von Fehlererkennungsalgorithmen die Handhabung der Software verbessern. Für gezielte und effektive Korrekturen der Software müsste jedoch vorher eine detaillierte Analyse eines geplanten Anwendungsbereichs durchgeführt werden. Somit wäre beispielsweise die Ausgestaltung einer Benutzeroberfläche bei der Integration von *C_stone* in eine bestehende Software hinfällig.

Der sicherlich sinnvollste Weg wäre jedoch die vollständig hardwarebasierte Umsetzung des Systems direkt im Videoprojektor. Dazu müssten die erforderlichen Komponenten wie die Tiefenkamera und eine Einheit für Video-Signalverarbeitung im Gehäuse des Projektors untergebracht werden. Dies würde dem Nutzer ein hohes Maß an Flexibilität bieten und eine

komplett automatische „Plug-and-Play“-Trapezkorrektur ermöglichen, die nicht nur im Präsentationsbereich, sondern auch auf Bühnen oder in der Medienkunst weite Einsatzmöglichkeiten finden würde. Zwar bedarf es bei einer hardwarebasierten Umsetzung an vielen Stellen weiterer Entwicklungen, die in Kapitel 2 erarbeiteten theoretischen Modelle und deren Umsetzung in Kapitel 3 bieten jedoch die erforderlichen Grundlagen für diesen Schritt.

# Literatur- und Linkverzeichnis


[Mad11] GarageCUBE: madmapper Software - Homepage
www.madmapper.com (letzter Zugriff: 14.5.12)

[Mo812] GarageCUBE: Modul8 Software - Homepage
www.garagecube.com/modul8/index.php (letzter Zugriff: 14.5.12)

[gVJ12] Arkaos: GrandVJ Software - Homepage
www.garagecube.com/modul8/index.php (letzter Zugriff: 14.5.12)

[Yan01] Ruigang Yang, Greg Welch: Automatic and Continuous Projector Display and Surface Calibration Using Every-Day Imagery (2001)

[Lee04] Johnny C. Lee, Paul H. Dietz, Dan Maynes-Aminzade, Ramesh Raskar, Scott E. Hudson: Automatic Projector Calibration with Embedded Light Sensors (2004)

[Han08] Jan Hanten: Kamerabasierte Echtzeit-Erkennung bewegter Flächen zur korrekten Texturprojektion (2008)

[OCV12] willowgarage: OpenCV - documentation, download and wiki
opencv.willowgarage.com (letzter Zugriff: 30.4.2012)

[SuB05] Oliver Schreer: Stereoanalyse und Bildsynthese (2005)

[HZM03] Richard Hartley, Andrew Zisserman:
Multiple View Geometry in Computer Vision (2003)

[ORL08] Gary Bradski, Adrian Kaehler: Learning OpenCV (2008)

[MtK12] Sean Kean, Jonathan Hall, Phoenix Perry: Meet the Kinect - An introduction to Programming Natural User Interfaces (2012)

[Orr11] Thomas Orr: Funktionsweise und Genauigkeit der Kinect-Tiefenerkennung und praktisches Anwendungsbeispiel im Theaterkontext (2011)

[Fis12] Matthew Fisher: Kinect - Interpreting Sensor Values
www.graphics.stanford.edu/~mdfisher/Kinect.html (letzter Zugriff: 30.4.2012)

[HtK12] Jeff Kramer, Nicolas Burrus, Florian Echtler, Daniel Herrera C., Matt Parker: Hacking the Kinect (2012)

[ROS11] Kurt Konolige, Patrick Mihelich: ROS.org - Kinect operation
www.ros.org/wiki/kinect_calibration/technical (letzter Zugriff: 30.4.2012)

[Max11] cycling74: Max5 API, Version 5.1.7 (2011)

[MOD12] cycling74: Max5 Online Documentation, http://cycling74.com/
docs/max5/vignettes/intro/docintro.html (letzter Zugriff: 8.5.12)

[OGL04] Richard S. Wright, Jr. and Benjamin Lipchak:
OpenGL Superbible (third edition) (2004)

[CVJ11] Jean-Marc Pelletier: cv.jit - Computer Vision for Jitter
jmpelletier.com/cvjit/ (letzter Zugriff: 30.4.2012)

[Mar11] Ivan Martynov, Joni-Kristian Kamarainen, Lasse Lensu:
Projector Calibration by “Inverse Camera Calibration” (2011)

[Kim10] Makoto Kimura, Masaaki Mochimaru, Takeo Kanade:
Projector Calibration using Arbitrary Planes and Calibrated Camera (2010)

[Woo11] Elliot Woods: Calibrating Projectors and Cameras: Practical Tools
www.kimchiandchips.com/blog/?p=725 (letzter Zugriff: 3.5.12)

[JFR12] Jean-Marc Pelletier: jit.freenect.grab
jmpelletier.com/freenect/ (letzter Zugriff: 7.5.12)

[OKI12] Open-Kinect-Community: OpenKinect-Wiki
openkinect.org/wiki/Main_Page (letzter Zugriff: 7.5.12)

[vvv12] vvvv group, joreg, Max Wolf, Sebastian Gregor, Sebastian Oschatz:
vvvv - Software Homepage, www.vvvv.org (letzter Zugriff: 21.5.12)

[HON11] Hannes Köcher: The Honspage - Software - h.jit.homography,
www.hons.at/software/ (letzter Zugriff: 4.5.12)

# Glossar

| | |
|---|---|
| API | engl. application programming interface, Programmierschnittstelle |
| External | In *C* programmiertes Plugin, welches in der *Max*-Umgebung als Objekt eingebunden werden kann |
| extrinsisch | Im Bereich Computer Vision beschreiben extrinsische Kameraparameter die Position und Orientierung einer Kamera oder eines Projektors im Weltkoordinatensystem (siehe auch Kapitel 2.2). |
| FPS | engl. frames per second, Bilder pro Sekunde |
| GUI | engl. graphical user interface, grafische Benutzeroberfläche |
| Instanziierung | Begriff aus der objektorientierten Programmierung: die Erzeugung eines Objekts (auch: einer Instanz) einer bestimmten Klasse. |
| intrinsisch | Im Bereich Computer Vision beschreiben intrinsische Kameraparameter die inneren Parameter einer Kamera. Sie beinhalten Brennweite und Kamerahauptpunkt aber auch weitere Faktoren wie die Schräge des Sensorelements (engl. skew), die Pixelgröße und Linsen-Verzerrungs-Koeffizienten (siehe auch Kapitel 2.2). |
| IR | Infrarot |
| jit.<objektname> | Präfix für Bezeichnungen von *Max*-Objekten der *Jitter*-Objektfamilie |
| Jitter | Name der *MaxMSP*-Komponente zur Manipulation von Videodaten und Matrizen. |
| Kamerahauptpunkt | Schnittpunkt der optischen Achse mit der Bildebene einer Kamera |
| Kameramatrix | 3 x 3-Matrix, welche die intrinsischen Parameter einer Kamera enthält |
| kollinear | auf einer Gerade liegend |
| Mapping | siehe Projektionsmapping |
| Marker | für einen Bildanalyse-Algorithmus deutlich identifizierbares Element |
| Outlet/Inlet | *Max*-Objekte können über Outlets und Inlets untereinander verbunden werden, um sich gegenseitig Nachrichten senden zu können. |
| patch cord | Verbindung zwischen Outlet und Inlet zweier *Max*-Objekte |
| Patcher | Programm oder Teil eines Programms in *MaxMSP,* welches meist mehrere *Max*-Objekte beinhaltet, die gemeinsam eine gewisse Funktion erfüllen. |
| Projektionsmapping | Bezeichnung für einen Vorgang, bei dem Videodaten so modifiziert werden, dass ein Bild perspektivisch korrekt auf eine Szene oder auf ein Objekt projiziert wird. |

| | |
|---|---|
| RAW | Rohdatenformat |
| SDK | engl. software development kit, Software-Entwicklungs-Kit |
| Subpatcher | Ein *Max*-Patcher innerhalb eines übergeordneten Patchers, in dem aus Gründen der Übersicht bestimmte Objekte zusammengefasst wurden. |
| Tiefenkamera | Kamera, die statt Farb- oder Helligkeitswerten Distanzen pro Pixel erfasst (siehe Kapitel 2.5, *Microsoft-Kinect*-Sensor) |
| UI | engl. user interface, Benutzeroberfläche |
| VPL | engl. visual programming language, grafische Programmiersprache |

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Anhang

## Screenshot des Haupt-Patcher der MaxMSP-Software C_stone

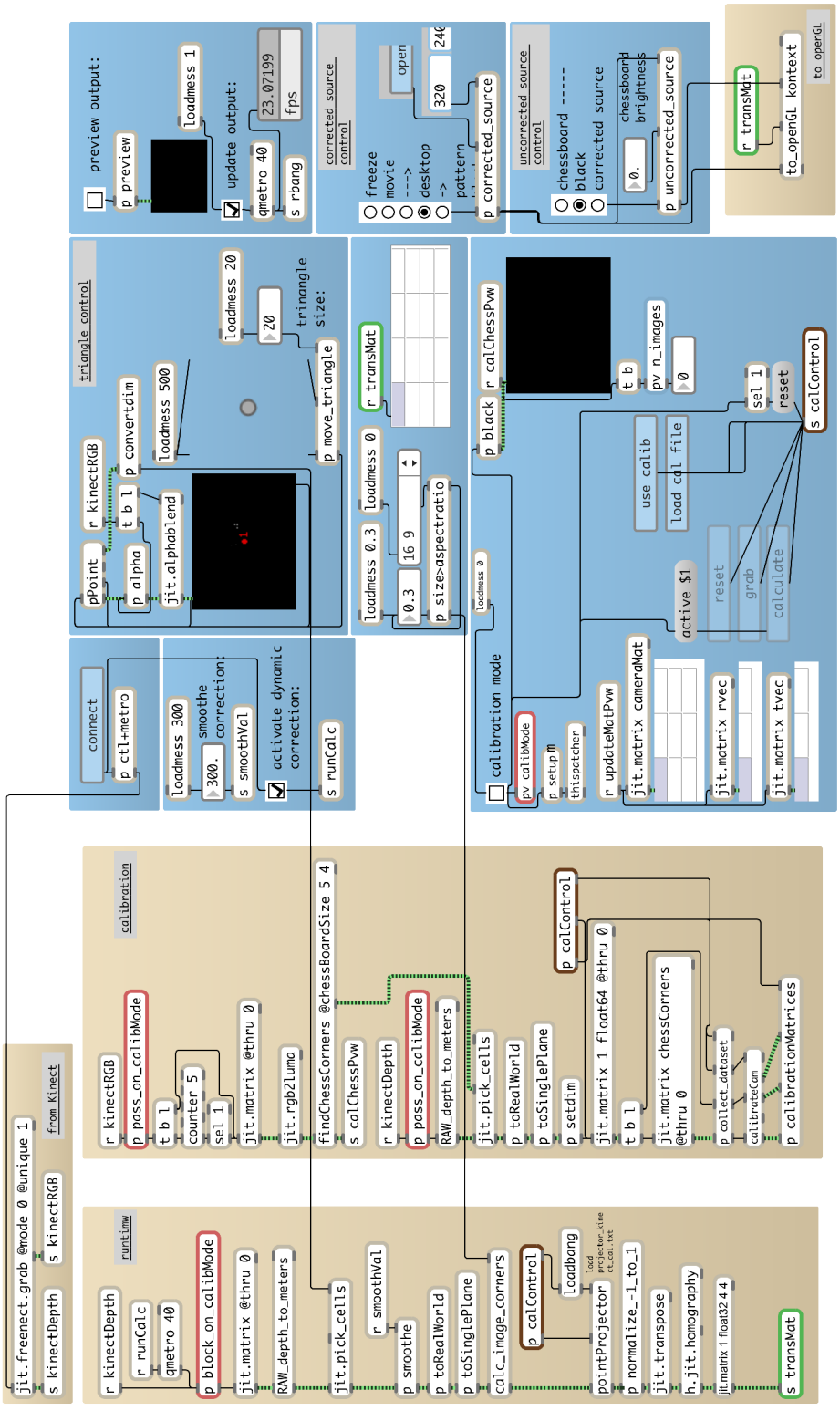


*Abbildung 34: C_stone-Haupt-Patcher in der Gesamtübersicht*

# GUI-Elemente des C_stone-Haupt-Max-Patchers

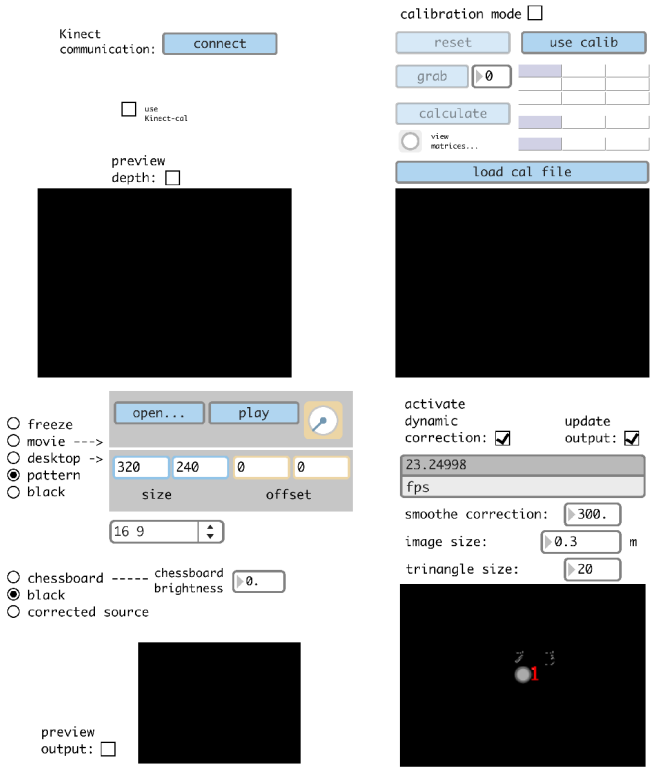


*Abbildung 35: C_stone-Haupt-Max-Patcher im Präsentationsmodus (GUI-Elemente sichtbar)*

## Screenshot der map_IR_to_RGB-Abstraction

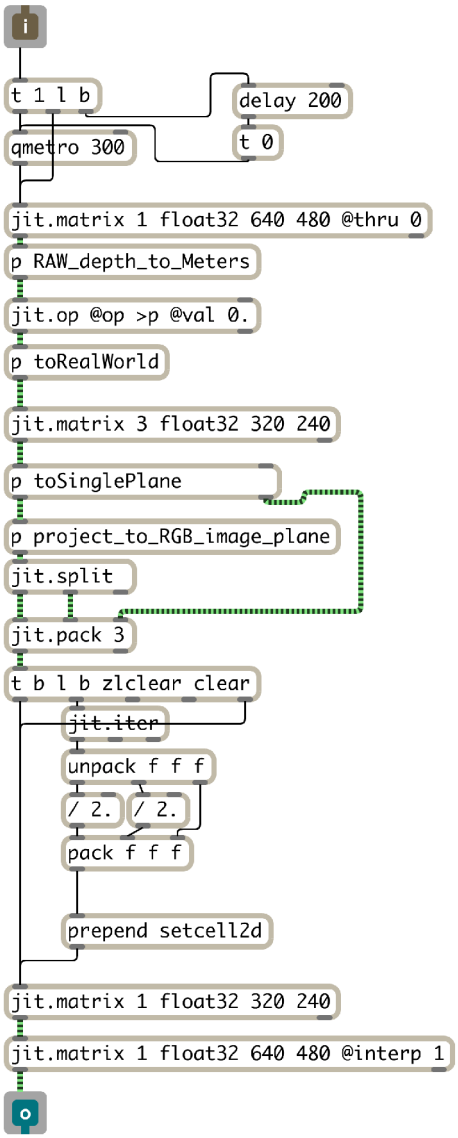


*Abbildung 36: map_IR_to_RGB-Abstraction*

## Screenshot der Abstraction jit.pick_cells

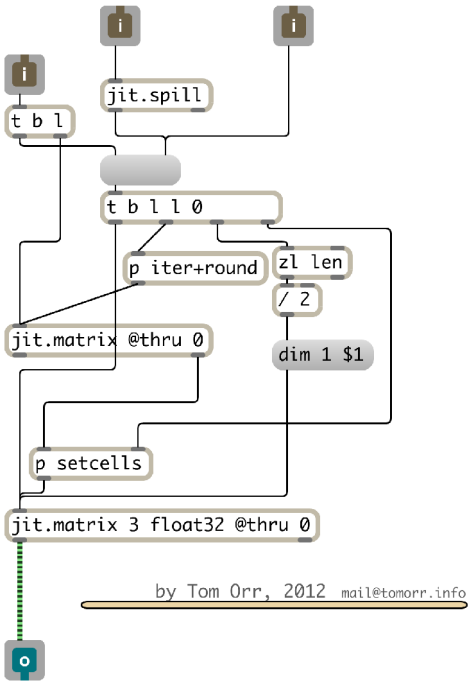


*Abbildung 37: Abstraction jit.pick_cells*

## Screenshot des Subpatchers toRealWorld

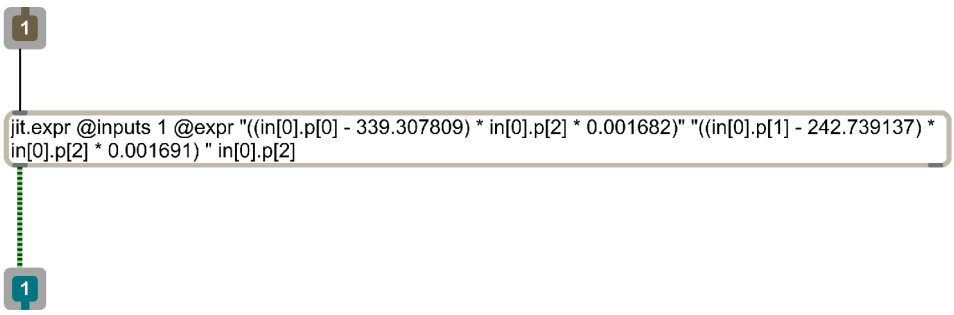


*Abbildung 38: Subpatcher toRealWorld*

## Screenshot des Subpatchers smoothe

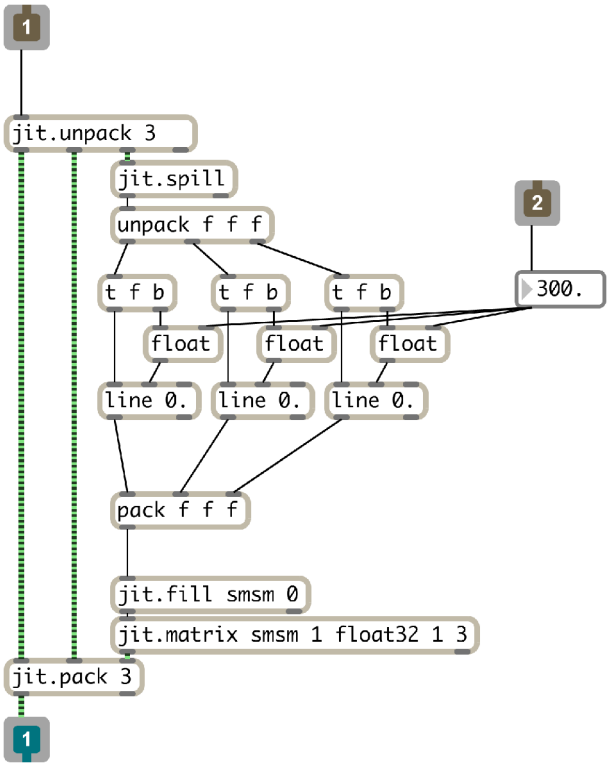


*Abbildung 39: Subpatcher smoothe*

## Screenshot der Abstraction to_openGL

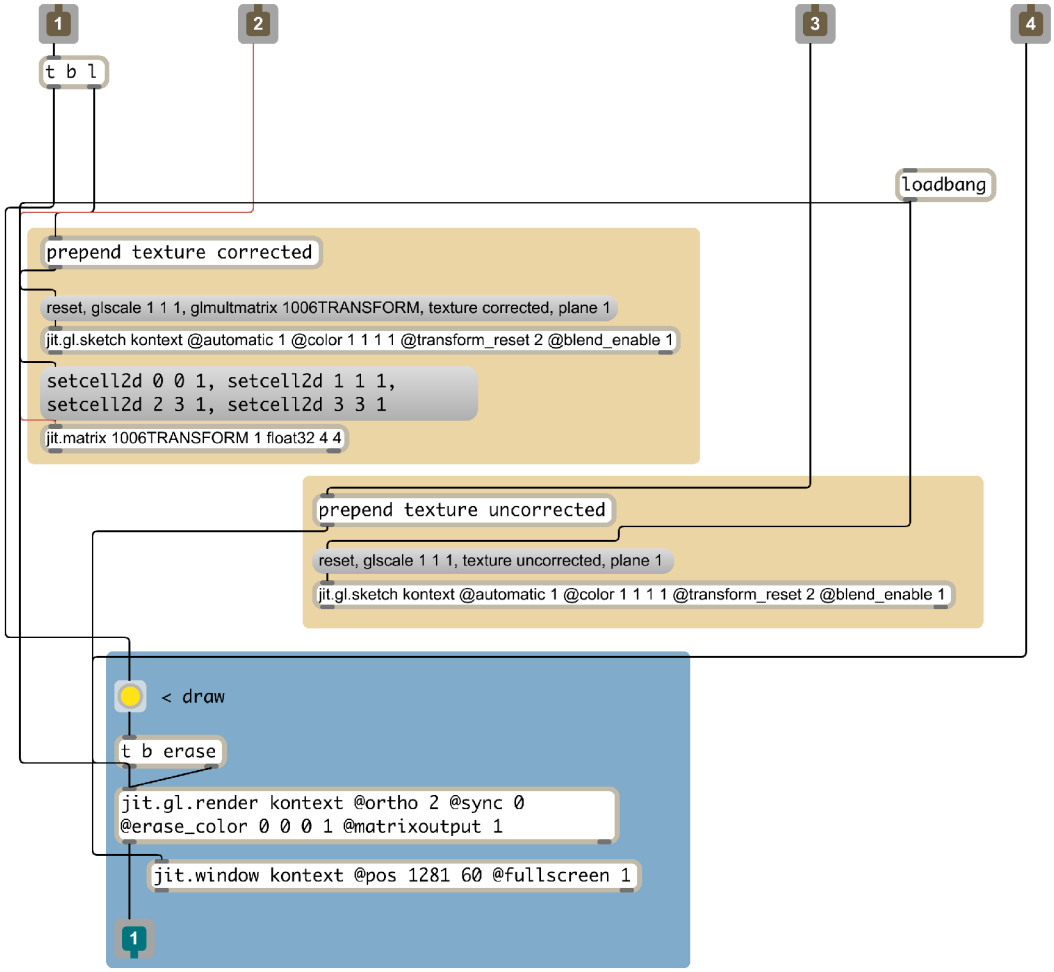


*Abbildung 40: Abstraction to_openGL*

## Quellcode math3d.js zur Berechnung der Bild-Eckpunkte

```
101                // note: points and vectors are always arrays with three elements
102
103                autowatch = 1;
104                inlets = 1;
105                outlets = 2;
106
107                //------ THE STRAIGHT LINE ------//
108                Line.local = 1;
109                function Line(PointP, Vec){
110                            this.originPoint = PointP;
111                            this.Vec = Vec;
112                      }
113
114                //------ DISTANCE ON LINE  ------//
115                // given a line and a distance this
116                // function calculates the Point(s)
117                // on the line with the given dis-
118                // tance to the origin point of the
119                // line
120
121                distance_on_line.local = 1;
122                function distance_on_line(l, d){ // l: the line, d: the distance
123                      var result = new Array();
124                      var vec_sqrt = l.Vec[0]*l.Vec[0] + l.Vec[1]*l.Vec[1] + l.Vec[2]*l.Vec[2];
125                      var r = Math.sqrt (d * d / vec_sqrt);
126
127                      // figure out the points by working out
128                      // the point on the line with parameter r
129                      // for r positive (first point)
130                      result[0] = l.originPoint[0] + r * l.Vec[0];
131                      result[1] = l.originPoint[1] + r * l.Vec[1];
132                      result[2] = l.originPoint[2] + r * l.Vec[2];
133
134                      r = r * -1;
135
136                      // for r negative (second point)
137                      result[3] = l.originPoint[0] + r * l.Vec[0];
138                      result[4] = l.originPoint[1] + r * l.Vec[1];
139                      result[5] = l.originPoint[2] + r * l.Vec[2];
140
141                      return result;
142                      }
                   // returns an array with 6 numbers, first three are first point, second three second
143
```

```
144                 //-------- THE PLANE --------//
145                 Plane.local = 1;
146                 function Plane(PointP, PointK1, PointK2){
147
148                               this.originPoint = PointP;
149                               this.pointA = PointK1;
150                               this.pointB = PointK2;
151
152                               this.VecA = Vec_from_points(this.originPoint, this.pointA);
153                               this.VecB = Vec_from_points(this.originPoint, this.pointB);
154
155                               this.normVec = Cross_product(this.VecA, this.VecB);
156
157                               // d is for the plane equation: N * p = d
158                               this.d = Scalar_product(this.normVec, this.originPoint);
159                         }
160
161                 //-------- THE PLANE WITH NORMAL --------//
162                 // make a plane from a normal vector and a point on the plane
163                 Plane_norm.local = 1;
164                 function Plane_norm(PointP, NormVec){
165
166                               this.originPoint = PointP;
167                               this.normVec = NormVec;
168
169                               // d is for the plane equation: N * p = d
170                               this.d = Scalar_product(this.normVec, this.originPoint);
171                         }
172
173                 // intersection of two planes. function returns a line.
174                 // see http://paulbourke.net/geometry/planeplane/ for more information
175                 Intersect_2plane.local = 1;
176                 function Intersect_2plane(planeA, planeB){
177
178                               var N1 = planeA.normVec;
179                               var N2 = planeB.normVec;
180                               var d1 = planeA.d;
181                               var d2 = planeB.d;
182
183                               // dot products:
184                               var N1N1 = Scalar_product(N1, N1);
185                               var N2N2 = Scalar_product(N2, N2);
186                               var N1N2 = Scalar_product(N1, N2);
187
188                               var determinant = N1N1 * N2N2 - N1N2 * N1N2;
189
190                               var c1 = (d1 * N2N2 - d2 * N1N2)/determinant;
191                               var c2 = (d2 * N1N1 - d1 * N1N2)/determinant;
192                 var lPoint = Vector_add(Scalar_multiplication(N1, c1), Scalar_multiplication(N2,c2));
```

```
193                             var lVec = Cross_product(N1,N2);
194
195                             var result = new Line(lPoint, lVec);
196
197                     return result;
198                     }
199
200             // intersection of three planes. function returns a point.
201             // see http://paulbourke.net/geometry/3planes/ for more information
202             Intersect_3plane.local = 1;
203             function Intersect_3plane(planeA, planeB, planeC){
204
205                             var N1 = planeA.normVec;
206                             var N2 = planeB.normVec;
207                             var N3 = planeC.normVec;
208                             var d1 = planeA.d;
209                             var d2 = planeB.d;
210                             var d3 = planeC.d;
211
212                             // scalar products:
213                             var N2N3 = Cross_product(N2, N3);
214                             var N3N1 = Cross_product(N3, N1);
215                             var N1N2 = Cross_product(N1, N2);
216
217                             // addends in the numerator
218                             var a1 = Scalar_multiplication(N2N3, d1);
219                             var a2 = Scalar_multiplication(N3N1, d2);
220                             var a3 = Scalar_multiplication(N1N2, d3);
221
222                             // the scalar in the denominator
223                             var dScal = Scalar_product(N1, N2N3);
224
225                             //add everything in the nominator up:
226                             var nVec = Vector_add(Vector_add(a1, a2), a3);
227
228                             // devide the vector in the nominator by the scalar in the denominator
229                             var result = Scalar_multiplication(nVec, 1/dScal);
230
231                     return result;
232                     }
```

```
233                 //-------- POINTS TO VECTOR  --------//
234                 // takes two 3d points as input and returns the vector from first to second point
                    // (from a to b).
235                 Vec_from_points.local = 1;
236                 function Vec_from_points(a,b){
237                       var result = new Array();
238
239                             for(var i = 0; i < a.length; i++)
240                                   result[i] = b[i] - a[i];
241
242                       return result;
243                       }
244
245                 //------ VECTOR ADDITION  --------//
246                 function Vector_add(a,b){
247                       var result = new Array();
248
249                             for(var i = 0; i < a.length; i++)
250                                   result[i] = b[i] + a[i];
251                       return result;
252                       }
253
254                 //-------- CROSS PRODUCT  --------//
255                 Cross_product.local = 1;
256                       function Cross_product(a,b){
257                             var result = new Array();
258                                         result[0] = a[1]*b[2] - a[2]*b[1];
259                                         result[1] = a[2]*b[0] - a[0]*b[2];
260                                         result[2] = a[0]*b[1] - a[1]*b[0];
261                             return result;
262                             }
263
264                 //-------- SCALAR PRODUCT  --------//
265                 Scalar_product.local = 1;
266                 function Scalar_product(a,b){
267                       var result = 0.;
268                                   result = a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
269                       return result;
270                       }
271
272                 //-------- SCALAR PRODUCT  --------//
273                 Scalar_multiplication.local = 1;
274                 function Scalar_multiplication(vec,scalar){
275                       var result = new Array();
276                             for(var i = 0; i < vec.length; i++)
277                                   result[i] = vec[i] * scalar;
278                       return result;
279                       }
```

```
280                //-------- ANGLE BETWEEN VECTOR-PLANE  --------//
281                Angle_vec_plane.local = 1;
282                function Angle_vec_plane(pl, vec){
283                       var result;
284                              var angle_pNorm_vec = Angle_vec_vec(pl.normVec, vec);
285                              result = Math.PI/2 - angle_pNorm_vec;
286                       return result;
287                       }
288
289                //-------- ANGLE BETWEEN TWO VECTORS  --------//
290                Angle_vec_vec.local = 1;
291                function Angle_vec_vec(vec1, vec2){
292                              var result;
293                              var vec1Abs = Vector_abs(vec1);
294                              var vec2Abs = Vector_abs(vec2);
295                              var s_prod = Scalar_product(vec1, vec2);
296                              var temp = s_prod / vec1Abs * vec2Abs;
297                              result = Math.acos(temp);
298                       return result;
299                       }
300
301                //-------- ABSOLUTE VALUE OF VECTOR  --------//
302                Vector_abs.local = 1;
303                function Vector_abs(vec){
304                       var result;
305                              result = Math.sqrt(vec[0]*vec[0] + vec[1]*vec[1] + vec[2]*vec[2]);
306                       return result;
307                       }
308
309                //--------//--------//--------//--------//--------//--------//
310                // This is where the communication from Max is handled //
311                //-------- All external from Max calls are handled via the anything()-function:
312                function anything(){
313
314                       switch(messagename){
315                              case "cross_product": {
316                       var VecA = new Array(arguments[0],arguments[1],arguments[2]);
317                       var VecB = new Array(arguments[3],arguments[4],arguments[5]);
318                       outlet(0,Cross_product(VecA, VecB));
319                       break;
320                       }
321                              case "scalar_product": {
322                       var VecA = new Array(arguments[0],arguments[1],arguments[2]);
323                       var VecB = new Array(arguments[3],arguments[4],arguments[5]);
324                       outlet(0,Scalar_product(VecA, VecB));
325                       break;
326                       }
```

```
327                         case "vec_from_points": {
328                 var VecA = new Array(arguments[0],arguments[1],arguments[2]);
329                 var VecB = new Array(arguments[3],arguments[4],arguments[5]);
330                 outlet(0,Vec_from_points(VecA, VecB));
331                 break;
332                 }
333                         case "angle_vec_vec" : {
334                 var VecA = new Array(arguments[0],arguments[1],arguments[2]);
335                 var VecB = new Array(arguments[3],arguments[4],arguments[5]);
336                 outlet(0, Angle_vec_vec(VecA,VecB));
337                 break;
338                 }
339                         case "calc_img_corners": {
340                 var i = 0;
341                 P1 = new Array(arguments[i++],arguments[i++],arguments[i++]);
342                 K11 = new Array(arguments[i++],arguments[i++],arguments[i++]);
343                 K21 = new Array(arguments[i++],arguments[i++],arguments[i++]);
344                 Img_h = arguments[i++];
345                 Img_v = arguments[i];
346                 Calc_img_corners(P1,K11,K21,Img_h,Img_v);
347                 break;
348                 }
349                         default: post(messagename, "is not function name. \n");
350                       } // end of switch
351                 }// end of anything()-function
352
353                 //-------- //-------- //-------- //-------- //-------- //-------- //
354                 // calculate image corners (specific to project „C_stone“)
355
356                 // v is the value of which the plane Exz lies higher and lower than Point P in
                    y-direction:
357                 find_v.local = 1;
358
359                 function find_v(P1, K1, K2, v_Img_Size){
360                         var Ep = new Plane(P1, K1, K2);
361                         // get thie angle of plane Ep to the y-Axis:
362                         var yVec = new Array(0, 1, 0);
363                         var yAngle = Angle_vec_plane(Ep, yVec);
364                         // the vertical image size and the angle make it possible to compute h
365                         var v = (v_Img_Size/2) * Math.cos(yAngle);
366                         var result = v;
367                       return result;
368                       }
```

```
369                 // h is the value of which the plane Exy lies further left and right than Point P in
                    z-direction:
370                 find_h.local = 1;
371                 function find_h(P1, K1, K2, h_Img_Size){
372                             var Ep = new Plane(P1, K1, K2);
373                             // get thie angle of plane Ep to the z-Axis:
374                             var zVec = new Array(0, 0, 1);
375                             var zAngle = Angle_vec_plane(Ep, zVec);
376                             // the vertical image size and the angle make it possible to compute h
377                             var h = (h_Img_Size/2) * Math.cos(zAngle);
378                             var result = h;
379                       return result;
380                       }
381
382                 Calc_img_corners.local = 1;
383                 function Calc_img_corners(P, K1, K2, h_img_size, v_img_size){
384
385                 // Point p represents the center of the image. together with points K1 and K2
386                 // it defines a plane Ep on with also the image corners lie.
387                 // The wanted points B1, B2, B3 and B4 are the corners of the image (tl,tr,br,bl).
388                 // The points are found by intersectiong the plane Ep with planes parallel
389                 // to the XY-plane and the XZ-plane.
390                 // To find where these planes are we use the real world horizontal and vertical
391                 // image size. With this we can dstermine the distance from each work plane (h_shift
392                 // and v_shift) to the
393                 // point P and because they are parallel to the planes XY and XZ easily define
394                 // equations for them. then we can intersect the planes to find the corner points.
395
396                 //-------- (1)
397                 // first we create the projection plane Ep. All planes are intersected with this
                    plane
398                             var Ep = new Plane(P, K1, K2);
399                 // now the workplane HE_m, a plane parllel to the XZ-plane but through P
400                             var y_vec = new Array(0,1,0);
401                             var HE_m = new Plane_norm(P, y_vec);
402                 // these varables are what we are looking for:
403                             var B1 = new Array();
404                             var B2 = new Array();
405                             var B3 = new Array();
406                             var B4 = new Array();
407                 //-------- (2)
408                 // planes Ep and HE_m intersect in a line though P: g
409                             var g = Intersect_2plane(Ep, HE_m);
410                 // as we know, P lies also on that line so we use P as the Lines origin point
411                             g.originPoint = P;
412                 // find points between B1 and B4 (WB1) and also B2 and B3 (WB2)
413                 // by looking along the intersection line in a certain distance (h_img_size / 2)
414                             var temp = distance_on_line(g, h_img_size/2);
415
```

```
416                             var HP_l = new Array();
417                             var HP_r = new Array();
418                             HP_l[0] = temp[0];
419                             HP_l[1] = temp[1];
420                             HP_l[2] = temp[2];
421                             HP_r[0] = temp[3];
422                             HP_r[1] = temp[4];
423                             HP_r[2] = temp[5];
424                 //-------- (3)
425                 // using the found points WB1 and WB2 we create two planes with a normalvector
426                 // going through point P...
427                             var Nl = Vec_from_points(HP_l,P);
428                             var Nr = Vec_from_points(HP_r,P);
429                 // ... and define the workplane WP_XY_left on which the Points B1 and B4 lie
430                 // and the workplane WP_XY_right on which the points B2 and B3 lie
431                             var HE_links = new Plane_norm(HP_l, Nl);
432                             var HE_rechts = new Plane_norm(HP_r, Nr);
433                 //-------- (4)
434                 // Intersect the workplanes HE_links and HE_rechts with the main Plane Ep
435                 // that gives us two lines on which the points B1, B4 and B2, B3 lie
436                             var hg_l = Intersect_2plane(HE_links, Ep);
437                             var hg_r = Intersect_2plane(HE_rechts, Ep);
438                 //-------- (5)
439                 // Now we have to find the points by looking along the Lines
440                 // hg_l and hg_r in the distance v_img_size / 2
441                             hg_l.originPoint = HP_l;
442                             hg_r.originPoint = HP_r;
443                             temp = distance_on_line(hg_l, v_img_size/2);
444                             B1[0] = temp[0];
445                             B1[1] = temp[1];
446                             B1[2] = temp[2];
447                             B2[0] = temp[3];
448                             B2[1] = temp[4];
449                             B2[2] = temp[5];
450                             temp = distance_on_line(hg_r, v_img_size/2);
451                             B3[0] = temp[0];
452                             B3[1] = temp[1];
453                             B3[2] = temp[2];
454                             B4[0] = temp[3];
455                             B4[1] = temp[4];
456                             B4[2] = temp[5];
457                 //-------- (6)
458                             outlet(0, B1, B2, B3, B4);
459                       }
```

## Quellcode calibrateCam.c des CalibrateCam-Max-Externals

Die Implementierung der im Rahmen dieser Arbeit entstandenen *OpenCV*-Externals, welche in Kapitel 2.6.3.1 aufgeführt wurden, soll anhand des Source-Codes des Externals *CalibrateCam* dargestellt werden. Innerhalb der Funktion *calibrateCam_matrix_calc()* (ab Codezeile 570) findet die eigentliche Vorbereitung der Output-Jitter-Matrizen und die Umwandlung der Input-Jitter-Matritzen zu OpenCV-Matrizen statt. Ebenso findet sich innerhalb dieser Funktion der eigentliche Funktionsaufruf der *OpenCV-C*-Funktion *cvCalibrateCamera2()* (siehe Codezeile 719). Alle anderen *OpenCV*-Externals, die im oben genannten Kapitel beschrieben werden, sind nach diesem Prinzip aufgebaut und weisen in erster Linie Unterschiede in der *calibrateCam_matrix_calc()*-Funktion auf.

```
460                 /*
461                 calibrateCam
462                 2012, by Tom Orr
463                 mail@tomorr.info
464                 This File is based on Jean-Marc Pelletier's cv.jit-project.
465
466                 Original licence:
467                 Copyright 2010, Jean-Marc Pelletier
468                 jmp@jmpelletier.com
469                 ++--++--++---++--++--++--++--++--+++--++--++--
470                 This file is part of cv.jit
471                 cv.jit is free software: you can redistribute it and/or modify
472                 it under the terms of the GNU Lesser General Public License as published
473                 by the Free Software Foundation, either version 3 of the License, or
474                 (at your option) any later version.
475
476                 cv.jit is distributed in the hope that it will be useful,
477                 but WITHOUT ANY WARRANTY; without even the implied warranty of
478                 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
479                 GNU Lesser General Public License for more details.
480
481                 You should have received a copy of the GNU Lesser General Public License
482                 along with cv.jit.  If not, see <http://www.gnu.org/licenses/>
483                 */
484                 /*
485                 This file links to the OpenCV library
                    <http://sourceforge.net/projects/opencvlibrary/>
486                 Please refer to the  Intel License Agreement For Open Source Computer Vision Library.
487                 Please also read the notes concerning technical issues with using the OpenCV library
488                 in Jitter externals.
489                 */
490
```

```
491 #ifdef __cplusplus
492 extern "C" {
493 #endif
494 #include "jit.common.h"
495 #ifdef __cplusplus
496 } //extern "C"
497 #endif
498
499 #define CV_NO_BACKWARD_COMPATIBILITY
500
501 #include "cv.h"
502 #include "jitOpenCV.h"
503
504 #define DEFAULT_IMAGE_SIZE_X 1024
505 #define DEFAULT_IMAGE_SIZE_Y 768
506
507 typedef struct _calibrateCam
508 {
509       t_object                        ob;
510       long                            imgSize[2];
511       long                            imgSizeCount;
512 } t_calibrateCam;
513
514 void *_calibrateCam_class;
515
516 t_jit_err           calibrateCam_init(void);
517 t_calibrateCam      *calibrateCam_new(void);
518 void                calibrateCam_free(t_calibrateCam *x);
519 t_jit_err           calibrateCam_matrix_calc(t_calibrateCam *x, void *inputs, void
    *outputs);
520
521 t_jit_err calibrateCam_init(void)
522 {
523       long attrflags=0;
524       t_jit_object *mop,*output,*attr, *input;
525       _calibrateCam_class = jit_class_new("calibrateCam",
      (method)calibrateCam_new,
      (method)calibrateCam_free,
      sizeof(t_calibrateCam),0L);
526
527       //add mop
528       mop = (t_jit_object *)jit_object_new(_jit_sym_jit_mop,3,4);
529
530       // turn off all dimlinks. We want individual matrices at all inputs and
531       //outputs. "dimlink" links the sizes of matrices to the leftmost input
532       // turn off all typelinks
533
534       jit_mop_input_nolink(mop, 1);
535       input = jit_object_method(mop, _jit_sym_getinput,1);
```

```
536                     jit_object_method(input,_jit_sym_ioproc,jit_mop_ioproc_copy_adapt);
537
538                     jit_mop_input_nolink(mop, 2);
539                     input = jit_object_method(mop, _jit_sym_getinput,2);
540                     jit_object_method(input,_jit_sym_ioproc,jit_mop_ioproc_copy_adapt);
541
542                     jit_mop_input_nolink(mop, 3);
543                     input = jit_object_method(mop, _jit_sym_getinput,3);
544                     jit_object_method(input,_jit_sym_ioproc,jit_mop_ioproc_copy_adapt);
545
546                     output = jit_object_method(mop,_jit_sym_getoutput,1);
547                     jit_mop_output_nolink(mop, 1);
548                     output = jit_object_method(mop,_jit_sym_getoutput,2);
549                     jit_mop_output_nolink(mop, 2);
550                     output = jit_object_method(mop,_jit_sym_getoutput,3);
551                     jit_mop_output_nolink(mop, 3);
552                     output = jit_object_method(mop,_jit_sym_getoutput,4);
553                     jit_mop_output_nolink(mop, 4);
554
555                     jit_class_addadornment(_calibrateCam_class,mop);
556
557                     //add methods
558                     jit_class_addmethod(_calibrateCam_class,
                                    (method)calibrateCam_matrix_calc,
                                    "matrix_calc", A_CANT, 0L);
559
560                     //add attributes
561                     attrflags = JIT_ATTR_GET_DEFER_LOW | JIT_ATTR_SET_USURP_LOW;
562                     attr = (t_jit_object *)jit_object_new(
                                    _jit_sym_jit_attr_offset_array,
                                    "imgSize",
                                    _jit_sym_long,
                                    2,
                                    attrflags,
                                    (method)0L,
                                    (method)0L,
                              calcoffset(t_calibrateCam,imgSizeCount),
                     calcoffset(t_calibrateCam,imgSize));
563
564                     jit_class_addattr(_calibrateCam_class, attr);
565                     jit_class_register(_calibrateCam_class);
566                     return JIT_ERR_NONE;
567             }
568
569
570             t_jit_err calibrateCam_matrix_calc(t_calibrateCam *x, void *inputs, void *outputs)
571             {
572                     t_jit_err err=JIT_ERR_NONE;
573
```

```
574                 long in_savelock1, in_savelock2, in_savelock3, out_savelock1, out_savelock2,
                    out_savelock3, out_savelock4;
575                 t_jit_matrix_info in_minfo1, in_minfo2, in_minfo3, out_minfo1, out_minfo2,
                    out_minfo3, out_minfo4;
576                 void *in_matrix1, *in_matrix2, *in_matrix3, *out_matrix_1, *out_matrix_2,
                    *out_matrix_3, *out_matrix_4;
577                 char * in_bp3;
578
579                 // calibration pattern features in the object model space (3*Nor N*3)
580                       CvMat* objPoints = 0;
581                 // 2d projections of object points in the camera view (3*N or N*3)
582                       CvMat* imgPoints = 0;
583                 // 1xM Vector with number of points in each image
584                       CvMat* pointCounts = 0;
585
586                       CvMat*  cameraIntrinsics = cvCreateMat(3, 3, CV_64F);
587                       CvMat*  distCoeffs = cvCreateMat(1, 5, CV_64F);
588                       CvMat* rotMatrix = 0;
589                       CvMat* transMatrix = 0;
590
591                       x->imgSize[0] = DEFAULT_IMAGE_SIZE_X;
592                       x->imgSize[1] = DEFAULT_IMAGE_SIZE_Y;
593
594                       in_matrix1   = jit_object_method(inputs,_jit_sym_getindex,0);
595                       in_matrix2   = jit_object_method(inputs,_jit_sym_getindex,1);
596                       in_matrix3   = jit_object_method(inputs,_jit_sym_getindex,2);
597                       out_matrix_1 = jit_object_method(outputs,_jit_sym_getindex,0);
598                       out_matrix_2 = jit_object_method(outputs,_jit_sym_getindex,1);
599                       out_matrix_3 = jit_object_method(outputs,_jit_sym_getindex,2);
600                       out_matrix_4 = jit_object_method(outputs,_jit_sym_getindex,3);
601
602                       if (x&&in_matrix1&&in_matrix2&&in_matrix3&&out_matrix_1) {
603
604                             in_savelock1 = (long) jit_object_method(in_matrix1,_jit_sym_lock,1);
605                             in_savelock2 = (long) jit_object_method(in_matrix2,_jit_sym_lock,1);
606                             in_savelock3 = (long) jit_object_method(in_matrix3,_jit_sym_lock,1);
607                             out_savelock1 = (long) jit_object_method(out_matrix_1,_jit_sym_lock,1);
608                             out_savelock2 = (long) jit_object_method(out_matrix_2,_jit_sym_lock,1);
609                             out_savelock3 = (long) jit_object_method(out_matrix_3,_jit_sym_lock,1);
610                             out_savelock4 = (long) jit_object_method(out_matrix_4,_jit_sym_lock,1);
611
612                             jit_object_method(in_matrix1,_jit_sym_getinfo,&in_minfo1);
613                             jit_object_method(in_matrix2,_jit_sym_getinfo,&in_minfo2);
614                             jit_object_method(in_matrix3,_jit_sym_getinfo,&in_minfo3);
615                             jit_object_method(out_matrix_1,_jit_sym_getinfo,&out_minfo1);
616                             jit_object_method(out_matrix_2,_jit_sym_getinfo,&out_minfo2);
617                             jit_object_method(out_matrix_3,_jit_sym_getinfo,&out_minfo3);
618                             jit_object_method(out_matrix_4,_jit_sym_getinfo,&out_minfo4);
619
```

```
620                     //___________INPUT MATRICES:
621                     // First Matrix is used for the Object Points (must be a 3*N Matrix)
622             if (in_minfo1.type != _jit_sym_float64)  { err=JIT_ERR_MISMATCH_TYPE;  goto out; }
623             if (in_minfo1.planecount != 1) { err=JIT_ERR_MISMATCH_PLANE; goto out; }
624             if (in_minfo1.dimcount!= 2) { err=JIT_ERR_MISMATCH_DIM; goto out; }
625             if (in_minfo1.dim[0] != 3) { err=JIT_ERR_MISMATCH_DIM; goto out; }
626
627                     // Second matrix holds the image Points (2*N)
628             if (in_minfo2.type != _jit_sym_float64)  { err=JIT_ERR_MISMATCH_TYPE;  goto out; }
629             if (in_minfo2.planecount != 1) { err=JIT_ERR_MISMATCH_PLANE; goto out; }
630             if (in_minfo2.dimcount!= 2) { err=JIT_ERR_MISMATCH_DIM; goto out; }
631             if (in_minfo2.dim[0] != 2) { err=JIT_ERR_MISMATCH_DIM; goto out; }
632
633                     // 1*M Matrix with the number of points per passed image
634             if (in_minfo3.type != _jit_sym_long)  { err=JIT_ERR_MISMATCH_TYPE;  goto out; }
635             if (in_minfo3.planecount != 1) { err=JIT_ERR_MISMATCH_PLANE; goto out; }
636             if (in_minfo3.dimcount!= 2) { err=JIT_ERR_MISMATCH_DIM; goto out; }
637             if (in_minfo3.dim[0] != 1) { err=JIT_ERR_MISMATCH_DIM; goto out; }
638
639                     //___________OUTPUT MATRICES:
640                     // intrinsic camera matrix:
641             out_minfo3.dimcount = 2;
642             out_minfo1.dim[0] = 3;
643             out_minfo1.dim[1] = 3;
644             out_minfo1.planecount = 1;
645             out_minfo1.type = gensym("float64");
646             jit_object_method(out_matrix_1,_jit_sym_setinfo,&out_minfo1);
647
648             out_minfo3.dimcount = 2;
649             out_minfo2.dim[0] = 1;
650             out_minfo2.dim[1] = 5;
651             out_minfo2.planecount = 1;
652             out_minfo2.type = gensym("float64");
653             jit_object_method(out_matrix_2,_jit_sym_setinfo,&out_minfo2);
654
655             out_minfo3.dimcount = 2;
656             out_minfo3.dim[0] = 3;
657             out_minfo3.dim[1] = in_minfo3.dim[1];
658             out_minfo3.planecount = 1;
659             out_minfo3.type = gensym("float64");
660             jit_object_method(out_matrix_3,_jit_sym_setinfo,&out_minfo3);
661
662             out_minfo4.dimcount = 2;
663             out_minfo4.dim[0] = 3;
664             out_minfo4.dim[1] = in_minfo3.dim[1];
665             out_minfo4.planecount = 1;
666             out_minfo4.type = gensym("float64");
667             jit_object_method(out_matrix_4,_jit_sym_setinfo,&out_minfo4);
668
```

```
669                 ////////////////////////////////////////////////////////////
670                 ////////////////////////////////////////////////////////////
671
672                             // make sure all the matrix infos variables are set properly
673                 jit_object_method(in_matrix1,_jit_sym_getinfo,&in_minfo1);
674                 jit_object_method(in_matrix2,_jit_sym_getinfo,&in_minfo2);
675                 jit_object_method(in_matrix3,_jit_sym_getinfo,&in_minfo3);
676                 jit_object_method(out_matrix_1,_jit_sym_getinfo,&out_minfo1);
677                 jit_object_method(out_matrix_2,_jit_sym_setinfo,&out_minfo2);
678                 jit_object_method(out_matrix_3,_jit_sym_setinfo,&out_minfo3);
679                 jit_object_method(out_matrix_4,_jit_sym_setinfo,&out_minfo4);
680
681                 objPoints = cvCreateMat(in_minfo1.dim[1], 3, CV_64FC1);
682                 imgPoints = cvCreateMat(in_minfo2.dim[1], 2, CV_64FC1);
683                 pointCounts = cvCreateMat(in_minfo3.dim[1], 1,  CV_32SC1);
684
685                             // cameraIntrinsics Mat is already created
686                             // distCoeffs Mat is already created
687                 rotMatrix = cvCreateMat(out_minfo3.dim[1], 3, CV_64F);
688                 transMatrix = cvCreateMat(out_minfo3.dim[1], 3, CV_64F);
689
690                 cvJitter2CvMat(in_matrix1, objPoints);
691                 cvJitter2CvMat(in_matrix2, imgPoints);
692                             //cvJitter2CvMat(in_matrix3, pointCounts);
693                 cvJitter2CvMat(out_matrix_1, cameraIntrinsics);
694                 cvJitter2CvMat(out_matrix_2, distCoeffs);
695                 cvJitter2CvMat(out_matrix_3, rotMatrix);
696                 cvJitter2CvMat(out_matrix_4, transMatrix);
697
698                             // copy data from in_matrix4 (type long) to pointCount (type int 32)
699                 jit_object_method(in_matrix3,_jit_sym_getdata,&in_bp3);
700
701                 int i=0;
702                        for (i=0;i<in_minfo3.dim[1];i++)
703                        {
704                               int val = (int)*(long *)in_bp3;
705                               cvSetReal1D(pointCounts, i, val);
706                        }
707                  in_minfo3.dim[1]);
708                             // Initial camera Matrix:
709                 cvSetReal2D(cameraIntrinsics, 0, 0, x->imgSize[0]);
710                 cvSetReal2D(cameraIntrinsics, 1, 1, x->imgSize[1]);
711                 cvSetReal2D(cameraIntrinsics, 0, 2, x->imgSize[0]/2.);
712                 cvSetReal2D(cameraIntrinsics, 1, 2, x->imgSize[1]/2.);
713                 cvSetReal2D(cameraIntrinsics, 2, 2, 1.);
714
715                 int flags = 0;
716                 flags = CV_CALIB_USE_INTRINSIC_GUESS | CV_CALIB_ZERO_TANGENT_DIST | CV_CALIB_FIX_K1 |
                    CV_CALIB_FIX_K2 | CV_CALIB_FIX_K3;
```

```
717
718                 // // // // // //  CALIBRATE CAMERA FUNCTION:
719                 cvCalibrateCamera2( objPoints,
720                                     imgPoints,
721                                     pointCounts,
722                                     cvSize(x->imgSize[0], x->imgSize[1]),
723                                     cameraIntrinsics,
724                                     distCoeffs,
725                                     rotMatrix,
726                                     transMatrix,
727                                     flags);
728
729                       } else {
730                             return JIT_ERR_INVALID_PTR;
731                       }
732                 out:
733                       jit_object_method(out_matrix_4,gensym("lock"),out_savelock4);
734                       jit_object_method(out_matrix_3,gensym("lock"),out_savelock3);
735                       jit_object_method(out_matrix_2,gensym("lock"),out_savelock2);
736                       jit_object_method(out_matrix_1,gensym("lock"),out_savelock1);
737                       jit_object_method(in_matrix1,gensym("lock"),in_savelock1);
738                       jit_object_method(in_matrix2,gensym("lock"),in_savelock2);
739                       jit_object_method(in_matrix3,gensym("lock"),in_savelo ck3);
740
741                       if(imgPoints) cvReleaseMat(&(imgPoints));  // free Memory
742                       if(objPoints) cvReleaseMat(&(objPoints));  // free Memory
743                       if(pointCounts)     cvReleaseMat(&(pointCounts));
744                       if(transMatrix) cvReleaseMat(&(transMatrix));
745                       if(rotMatrix) cvReleaseMat(&(rotMatrix));
746
747                       return err;
748                 }
749                 t_calibrateCam *calibrateCam_new(void)
750                 {
751                       t_calibrateCam *x;
752                       if (x=(t_calibrateCam *)jit_object_alloc(_calibrateCam_class))
753                       {
754                             x->imgSize[0] = DEFAULT_IMAGE_SIZE_X;
755                             x->imgSize[1] = DEFAULT_IMAGE_SIZE_Y;
756                       } else {
757                             x = NULL;
758                       }
759                       return x;
760                 }
761
762                 void calibrateCam_free(t_calibrateCam *x)
763                 { // empty
764                 }
```