F A C H H O C H S C H U L E W E D E L

V I R T U A L R E A L I T Y P R A K T I K U M

Dokumentation

Eingereicht von:	Aykut Özdemir Fritz-Lindemann-Weg 4 21031 Hamburg Tel. (040) 210 56 782
Matrikelnummer:	cgt101381
Erarbeitet im:	6. Semester
Abgegeben am:	23.10.2017
Referent (FH Wedel):	M. Sc. Marcus Riemer Fachhochschule Wedel Feldstraße 143 22880 Wedel Tel. (04103) 80 48 - 68

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings-Verzeichnis

Abkürzungsverzeichnis

Abb. Abbildung

Abs. Abschnitt

bzw. beziehungsweise

Dat. Datei

List. Listing

S. Seite

s. siehe

Tab. Tabelle

Z. Zeile

z.B. zum Beispiel

1 Zielsetzung und Vorgehensweise

Im Rahmen des Virtual Reality-Praktikums sollte eine Augumented Reality-Applikation entwickelt werden, die einem Endbenutzer die Möglichkeit bietet mithilfe eines Balles Münzen einzusammeln und dabei gleichzeitig Hindernissen zu umgehen. Für die Steuerung des Balls wird die Tastatur verwendet. Mit einer Webcam werden eine Anzahl an bestimmten Bildinformationen, sogenannte Image Targets, getrackt. Der Endnutzer soll in der Lage sein diese Image Targets selbst auszuwählen. Es können Image Targets entweder mittels der Webcam geschossen oder über das Dateisystem geladen werden. Dem Nutzer wird die Verwendung von aussagekräftigen und viereckigen Bildern empfohlen, die der Endbenutzer dann zufällig auf einen einfarbigen Hintergrund positionieren kann. Der einfarbige Hintergrund dient zur einfachen Erkennung der Image Targets. Über den Image Targets sollen sowohl Münzen, die eingesammelt werden können, als auch Hindernisse als 3D-Objekte gerendert werden. Hindernisse stellen horizontal zum Boden stehende und im Uhrzeigersinn rotierende Balken dar. Ziel des Spiels ist es, mithilfe des steuerbaren Balles sämtliche Münzen einzusammeln und folglich den Hindernissen auszuweichen.

Zu Beginn der Projekts war eine mobile Android-App geplant, da dort sowohl die Kamera, als auch die Steuerung in einem Smartphone gekapselt wäre. Zusätzlich wird die Entwicklung von Android-Anwendungen mithilfe der Open Source-Bibliothek OpenCV unterstützt. Erste Tests haben ergeben, dass Android-Anwendungen mit OpenCV mangelnde Performance aufweisen. Sogar von OpenCV mitgelieferte Samples, die auf einem Samsung Galaxy Note Edge getestet wurden, liefen nicht flüssig. Aus diesen Gründen fiel die Wahl der Programmiersprache auf C++, da hierfür im Rahmen vergangener Praktika schon Erfahrung gesammelt wurde und C++-Anwendungen in der Performance besser abschneiden.

Für die Detektion von Image Targets wird die Bibliothek OpenCV verwendet. Mittels von Feature-Detektoren[1], Descriptor-Extraktoren[2] und Descriptor-Matchern[3] werden bestimmte Bildinformationen in Bildszenen detektiert. Im Rahmen der Entwicklung und erster Tests hat sich dieser Ansatz als ungünstig und als nicht kontrollierbar herausgestellt. Oftmals wurden Bildmerkmale inkorrekt gepaart (s. Abb. 1) und die Gründe für die falsche Paarung waren nur schwer nachvollziehbar.

Abb. 1: Beispiel für das Finden von falschen Paaren mit dom DescriptorMatcher von OpenCV[4]

[1] *cv::FeatureDetector Class Reference* 2017.
[2] *Class DescriptorExtractor* 2017.
[3] *cv::DescriptorMatcher Class Reference* 2017.

Zusätzlich ist die Einwirkung auf den Paarungs- und Suchprozess schwierig, da diese Funktionen von OpenCV stammen, also einer vorgefertigten Bibliothek. Aus diesem Grund wurde auf die Verwendung von Image Targets verzichtet. Anstelle von Image Targets sollten nun mit sogenannten ArUco Markern (s. Abb. 2) gearbeitet werden. ArUco Marker bilden binäre Bitmasken, die ein eigenständiges Detektieren ermöglichen, da keine aufwendigen Bildmerkmale gesucht werden müssen, sondern fest definierte binäre Bildinformationen.

Vor der Kernentwicklung wurde zuerst eine Recherche nach essentiellen Quellen betrieben, um einen ersten Blick zu erhalten, wie die Detektion von ArUco Markern realisiert werden könnte. Die wichtigsten Quellen, die im Rahmen der Recherche gefunden wurden, bilden das Buch „Mastering OpenCV with Practical Computer Vision Projects“[5] von Daniel Lélis Baggio, die auf OpenCV basierende Bibliothek „ArUco“[6] und die Videoreihe „OpenCV Basics“ zur Kamerakalibrierung mit OpenCV von George Lecakes auf Youtube[7]. Anstelle die externe Bibliothek ArUco zu nutzen, wurde innerhalb der Recherche evaluiert, welche Bestandteile der Bibliothek für einen eigenen Suchalgorithmus verwendet werden könnte. Der Grund für die Entscheidung bildet die bessere Kontrolle einzelner Prozessschritte. Im Vergleich zum Image Target-Ansatz wäre die Detektion von Markern kontrollierbarer und folglich die Problembehandlung simpler. Zusätzlich wäre die einfache Anwendung einer externen Bibliothek, anstelle der selbständigen Entwicklung, fernab des Ziels dieses Praktikums. Eine selbständige Entwicklung stärkt das Verständnis für Anwendungen, die eine externe Bibliothek anbietet.

Für die Darstellung und das Rendern von virtuellen 3D-Objekten wurde OpenGL verwendet. Aus zeitkritischen Gründen musste die Entscheidung gefällt werden, dass sämtliche 3D-Objekte als Kugeln bzw. Bälle dargestellt werden weil hierdurch eine sehr einfache Kollisionserkennung möglich ist, die für das Einsammeln von Münzen benötigt wird. Dargestellte Bälle unterscheiden sich semantisch in der Farbe:

Grün: Steuerbarer Ball des Spielers

Gelb: Bälle, die eingesammelt werden müssen (Münze)

Rot: Bälle, die nicht mit dem grünen Ball berührt werden dürfen (Hindernisse)

Bei aufwendigeren Objekten wäre die Verwendung von Bounding Boxes nötig gewesen, um eine Kollisionserkennung realisieren zu können. Dieser Ansatz war nichtsdestotrotz aus zeitkritischen Gründen nicht mehr realisierbar gewesen. Im Rahmen der Implementierung der Kollisionserkennung traten zusätzliche Komplikationen auf, die eine Realisierung des Features nicht ermöglichten. Sämtliche Gründe und Informationen werden im Abs. 2.5 näher erläutert.

Als Versionsverwaltung und Repository wurde GitHub[8] verwendet.

[4] *Example for bad matches with DescriptorMatcher* 2017.

[5] *Mastering OpenCV with Practical Computer Vision Projects - Step-by-step tutorials to solve common real-world computer vision problems for desktop or mobile, from augmented reality and number plate recognition to face recognition and 3D head tracking* 2017.

[6] *ArUco: a minimal library for Augmented Reality applications based on OpenCV* 2017.

[7] *OpenCV Basics 15-21* 2017.

[8] Das Projekt ist unter https://github.com/aoezd/Virtual-Reality-Praktikum aufrufbar.

2 Implementierung

In diesem Kapitel wird der technische Kern der Applikation vorgestellt. Der Fokus liegt hierbei auf der Detektion und Validierung von Markern in einer Bildszene, sowie der Lagebestimmung von Markern und dem Zeichnen von 3D-Objekten auf Markern mit OpenGL. Abschließend wird erläutert wie das Programm auszuführen ist worauf Vorschläge zur Problembehebung folgen. Wie zuvor im Abs. 1 beschrieben, bilden das Buch „Mastering OpenCV with Practical Computer Vision Projects“[9] von Daniel Lélis Baggio, die auf OpenCV basierende Bibliothek „ArUco“[10] und die Videoreihe „OpenCV Basics“ zur Kamerakalibrierung mit OpenCV von George Lecakes auf Youtube[11] wichtige Quellen im Rahmen der Entwicklung der Applikation.

2.1 Detektion von Markern

Marker speichern binäre Bildinformationen auf einer quadratischen Ebene ab. Die Detektion und Validierung von Markern, in gegebenen Bildsequenzen, bilden einen essentiellen Bestandteil der Applikation. Aus Gründen der Einfachheit wird nur das 7x7-Format unterstützt. Hierbei ist zu beachten, dass jeder Marker einen schwarzen Rand besitzt und folglich die eigentlichen Daten im inneren 5x5-Block gespeichert sind.

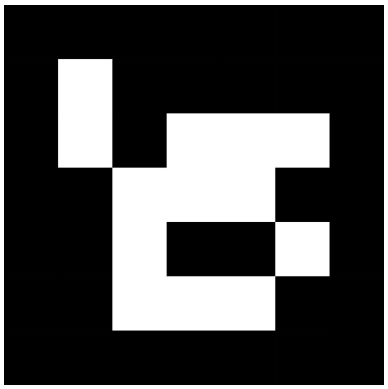

Abb. 2: Beispiel für einen binären Marker im 7x7-Format

Die Abb. 2 zeigt ein Beispiel für einen Marker mit binären Bildinformationen im 7x7-Format. Ausgewählte Marker wurden auf Papier gedruckt und laminiert, da laminiertes Papier eine flache Ebene garantiert und demzufolge das Suchen in Bildszenen vereinfacht. Im Rahmen der Entwicklung wurde versucht den Suchalgorithmus so simpel wie möglich zu halten, um eine nachträgliche unkomplizierte Wartung zu gewährleisten.

[9] *Mastering OpenCV with Practical Computer Vision Projects - Step-by-step tutorials to solve common real-world computer vision problems for desktop or mobile, from augmented reality and number plate recognition to face recognition and 3D head tracking* 2017.
[10] *ArUco: a minimal library for Augmented Reality applications based on OpenCV* 2017.
[11] *OpenCV Basics 15-21* 2017.

Der Suchalgorithmus hat folgenden allgemeinen Ablauf:

1. Das Finden von geschlossenen und viereckigen Konturen in der Bildszene
2. Die gefundenen Konturen im Bild auf eine 2D-Fläche transformieren
3. Prüfung, ob es sich um einen gesuchten Marker handelt

Für die Detektion von Markern wird eine Webcam benötigt. Jedes von der Webcam aufgezeichnete Bild wird zu Beginn in ein Graustufenbild umgewandelt, da gesuchte Marker binäre Informationen speichern und damit schwarzweiß sind. Durch ein Binarisierungsverfahren des Graustufenbilds wird jedes Pixel in schwarz oder weiß umgewandelt. Dieser Prozessschritt wird für das Suchen von Konturen benötigt. Aufgrund der dynamischen Belichtung in Bildszenen, wird auf ein absolutes Binarisierungsverfahren verzichtet und ein adaptives Verfahren verwendet. Beim adaptiven Binarisierungsverfahren wird ein Schwellenwert für einen kleinen Bildbereich berechnet. Durch den Erhalt von unterschiedlichen Schwellwerten für verschiedene Bildregionen sind bessere Ergebnisse für Bilder mit unterschiedlicher Beleuchtung möglich (s. Abb. 3).[12]

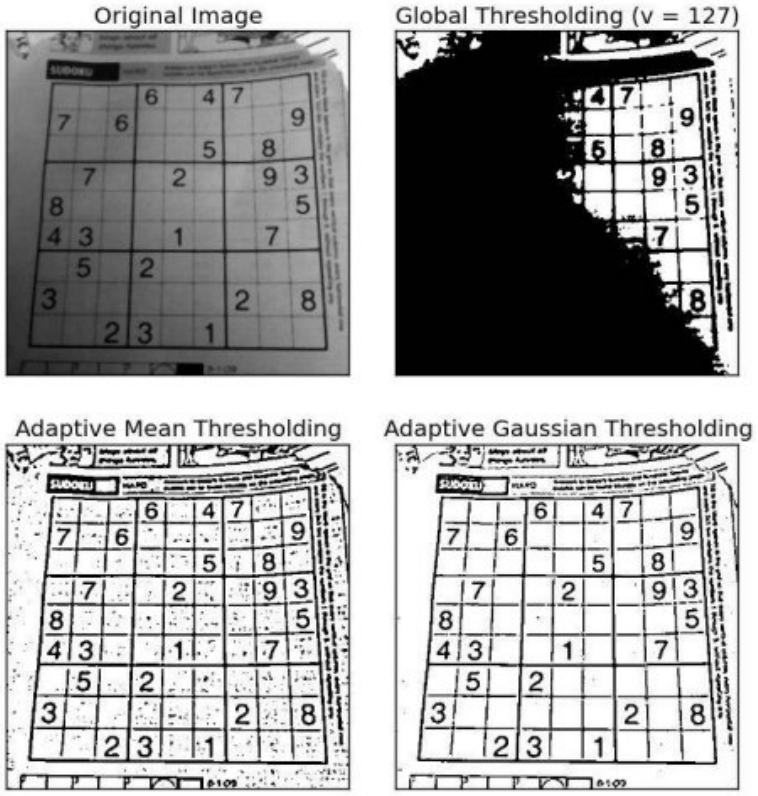

Abb. 3: Unterschiede zwischen absoluten und adaptiven Binarisierungsverfahren in Bezug auf dynamische Beleuchtung[13]

[12] *Adaptive Thresholding Description* 2017.
[13] *Adaptive Thresholding Example* 2017.

Mithilfe des binären Szenenbilds ist die Findung von existierenden Konturen realisierbar. Hierfür wird die Funktion `findContours()`[14] von OpenCV verwendet. Die von der Funktion gefundenen Konturen werden anschließend anhand der Anzahl an Punkten evaluiert und entsprechend gefiltert, da für den Suchalgorithmus zu kleine Konturen, wie beispielsweise einzelne Blöcke innerhalb von Markern, uninteressant sind. Dies bedeutet, dass Konturen verworfen werden, die eine zu kleine Anzahl an Punkten aufweisen. Der nächste Schritt des Suchalgorithmus bildet die Evaluierung der gefundenen Konturen. Die Funktion `findCandidates()` (s. List. 1) prüft hierfür, ob die gefundenen Konturen grundsätzlich einen Marker darstellen könnten.

```
1  void findCandidates(int minSideEdgeLength, const std::vector<std::vector<cv::↩
       Point>> &contours, std::vector<std::vector<cv::Point>> &detectedQuads) {
2    std::vector<cv::Point> approxQuad;
3    for (size_t i = 0; i < contours.size(); i++) {
4      if (approximateQuad(contours[i], approxQuad)) {
5        if (getShortestEdgeLength(approxQuad) > minSideEdgeLength) {
6          cv::Point p1 = approxQuad[1] - approxQuad[0];
7          cv::Point p2 = approxQuad[2] - approxQuad[0];
8          if ((p1.x * p2.y) - (p1.y * p2.x) < 0.0) {
9            std::swap(approxQuad[1], approxQuad[3]);
10         }
11         if (!approxQuadExists(detectedQuads, approxQuad)) {
12           detectedQuads.push_back(approxQuad);
13         }
14       }
15     }
16   }
17 }
```

List. 1: Die Funktion `detectormarkerbased.cpp/findCandidates();` evaluiert, ob die übergebenen Konturen grundsätzlich einen Marker darstellen könnten

Aus jeder Kontur wird zu Beginn versucht mittels der Funktion `approximateQuad()` ein Polygon aus genau vier Punkten zu bilden, da ein Marker immer quadratisch ist (s. List. 1, Z. 6). Eine Kontur wird zuerst mithilfe der OpenCV-Funktion `approxPolyDP()`[15] auf das kleinstmögliche Polygon approximiert und anschließend geprüft, ob das approximierte Polygon aus genau vier Punkten besteht und konvex[16] ist (s. List. 2, Z. 3-4).

[14] *Structural Analysis and Shape Descriptors - findContours* 2017.
[15] *Structural Analysis and Shape Descriptors - approxPolyDP* 2017.
[16] Ein geschlossenes Polygon ist konvex.

```
1 bool approximateQuad(const std::vector<cv::Point> &contour, std::vector<cv::↩
    Point> &approxQuad) {
2   cv::approxPolyDP(contour, approxQuad, contour.size() * 0.05, true);
3   return approxQuad.size() == QUAD_EDGE_COUNT && cv::isContourConvex(approxQuad)↩
      ;
4 }
```

List. 2: Die Funktion `detectormarkerbased.cpp/approximateQuad();` approximiert aus einer gegeben Kontur ein Quadrat bzw. ein konvexes Polygon aus genau vier Punkten darstellt

Anschließend wird geprüft, ob die kleinste Kantenlänge des approximierten Polygons groß genug ist, um beispielsweise kleine Quadrate innerhalb eines Markers auszuschließen (s. List. 1, Z. 8). Damit sämtliche Polygone, die einen gefundenen Marker repräsentieren, konsistent in ihrer Repräsentation bleiben, wird gegebenenfalls die Reihenfolge der Punkte umsortiert, sodass diese immer gegen den Uhrzeigersinn aufgelistet werden (s. List. 1, Z. 10-15). Bevor das quadratische Polygon gespeichert wird, prüft die Funktion `approxQuadExists()` (s. List. 1, Z. 16), ob nicht schon ein ähnliches Polygon gefunden wurde. Der Grund für diesen letzten Filter bildet die Tatsache, dass im Rahmen der OpenCV-Funktion `findContours()` redundante aber doch minimal unterschiedliche Konturen geliefert werden, die letztendlich die gleiche Bildregion repräsentieren (s. Abb. 4).

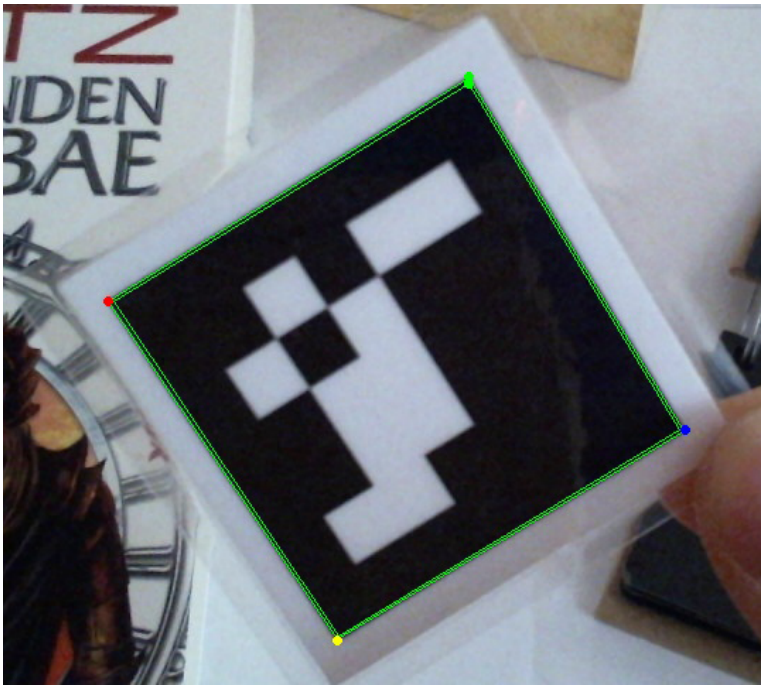

Abb. 4: Beispiel für den Fund von redundanten Konturen

Bevor die Evaluierung von detektierten, möglichen Markern starten kann, müssen die Bildinformationen innerhalb von gefundenen Polygonen auf eine zweidimensionale Ebene projiziert werden, um einen Informationsvergleich zu gewährleisten. Letztendlich wird versucht die perspektivische Projektion zu entfernen, um eine frontale Ansicht des Rechteckbereichs im Polygon

zu erhalten.[17] Hierfür wird zuerst die jeweilige perspektivische Projektionsmatrix mittels der OpenCV-Funktion `getPerspectiveTransform()`[18] berechnet. Die Funktion `getPerspectiveTransform()` berechnet aus vier korrespondierenden Punktpaaren eine Transformationsmatrix[19] (s. Abb. 5).

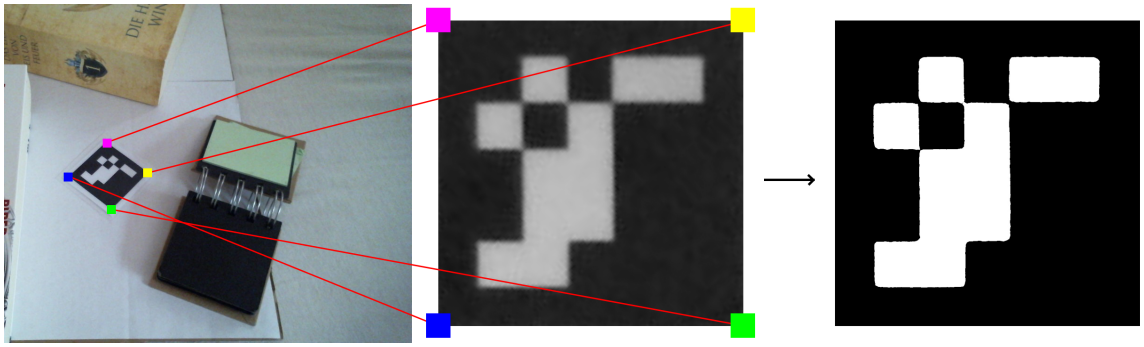

Abb. 5: Beispiel für die Entzerrung der Perspektive anhand von Punktpaaren für die Binarisierung mittels des Otsu-Algorithmus

Die Punktpaare bestehen einerseits aus den vier Punkten des gefundenen Polygons und andererseits aus den Eckpunkten des gesuchten quadratischen Markers. Die berechnete Transformationsmatrix wird nun der OpenCV-Funktion `warpPerspective()`[20] zur Verfügung gestellt, die anschließend die Verzerrung aus dem gewünschten Bildbereich entfernt und das Ergebnis als Bild speichert. Abschließend wird das Bild mithilfe des Otsu-Algorithmus[21] binarisiert. Dieser Algorithmus geht von einem bimodalen Bild[22] aus und findet den Schwellenwert, bei dem die Streuung innerhalb der dadurch bestimmten Klassen möglichst klein, zwischen den Klassen gleichzeitig aber möglichst groß ist.[23] Die Abb. 6 zeigt den Unterschied zwischen dem Ergebnis eines einfachen Binarisierungsverfahren und dem Otsu-Algortihmus, der vor allem an Ecken deutlich erkennbar ist.

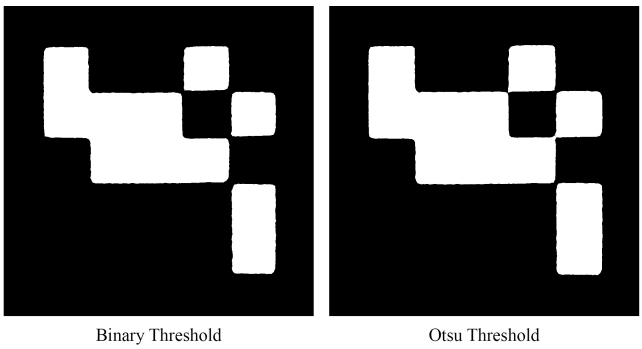

Abb. 6: Vergleich zwischen dem Ergebnis eines einfachen Binarisierungsverfahren und dem Otsu-Algortihmus

[17] Vgl. Baggio u. a. 2012, Seite (S.). 70.
[18] *Geometric Image Transformations - getPerspectiveTransform* 2017.
[19] Ebd.
[20] *Geometric Image Transformations - warpPerspective* 2017.
[21] *Adaptive Thresholding Description* 2017.
[22] Ein Bild ist bimodal, wenn dessen Histogramm genau zwei Spitzen aufweist.
[23] Otsu 1979.

2.2 Evaluierung von detektierten Markern

Sämtliche in der Bildszene gefundene und binarisierte Marker werden zu Beginn geprüft, ob diese einen schwarzen Rand aufweisen. Wie die Abb. 2 beschreibt, besteht ein Marker immer aus einem schwarzen Rand. Hierfür wird das quadratische Bild gleichmäßig in 7x7-Blöcke aufgeteilt und untersucht, ob die prozentuale Anzahl an schwarzen Pixel im Block größer ist als ein ausgewählter Schwellenwert. Dieser prozentuale Schwellenwert ist zur Laufzeit manipulierbar. Wie eine gleichmäßige Aufteilung aussehen könnte, beschreibt die Abb. 7.

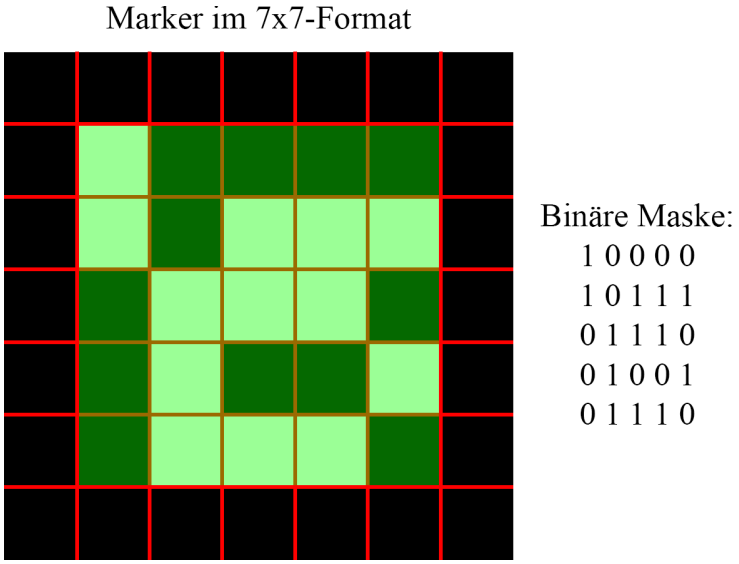

Abb. 7: Gleichmäßige Aufteilung von Blöcken innerhalb eines Markers und die dazugehörige Bitmaske

Falls der gefundene quadratische Bildbereich einen schwarzen Rand aufweist, wird anschließend der Informationsgehalt des möglichen Markers, im Quellcode „ID“ genannt, berechnet. Wie zu Beginn des Abs. 2.1 beschrieben, wird im Rahmen der Applikation nur das 7x7-Format unterstützt. Aus diesem Grund wird der Informationsgehalt eines Markers immer aus 5x5-Blöcken bestehen. Zunächst wird aus dem Bild eine binäre Maske gebildet, die im Grunde eine 5x5-Matrix aus Nullen und Einsen besteht und die eigentliche ID des Markers darstellt. Wie auch bei der zuvor beschriebenen Überprüfung des schwarzen Rands, wird für die Erzeugung der binären Matrix der innere Bildbereich des Markers gleichmäßig in 5x5-Blöcke aufgeteilt und anschließend untersucht, ob die prozentuale Anzahl an weißen Pixeln in einem Block größer ist als ein ausgewählter Schwellenwert. Dieser Schwellenwert ist zur Laufzeit der Applikation manipulierbar. Aus der generierten binären Maske berechnet die Funktion `computeId()` eine eindeutige ID. Hierfür werden jegliche Reihen der Maske hintereinander positioniert und in ein `unsigned long long` umgewandelt (s. List. 3, Z. 5-12).

```cpp
uint64_t computeId(const cv::Mat &bitMask) {
  std::bitset<64> bits;
  int k = 0;
  for (int y = bitMask.rows - 1; y >= 0; y--) {
    for (int x = bitMask.cols - 1; x >= 0; x--) {
      bits[k++] = bitMask.at<uchar>(y, x);
    }
  }
  return bits.to_ullong();
}
```

List. 3: Die Funktion `detectormarkerbased.cpp/computeId();` konvertiert aus einer gegebenen binären Maske eine eindeutige ID

Im Rahmen der Initialisierung des Programms werden die IDs für alle gesuchten bzw. validen Marker berechnet und für die Validierung der im Bild gesuchten Marker gespeichert. Die IDs der gefundenen Marker werden innerhalb der Validierung mit denen der validen Marker verglichen und gegebenenfalls maximal drei mal rotiert, da die Marker in der Bildszene unterschiedliche Ausrichtungen aufweisen können (s. List. 4, Z. 5-23).

```cpp
bool isValidMarker(Marker &marker) {
  unsigned int rotationCount = 0;
  cv::Mat bitMask = marker.bitMask;
  do {
    bool valid = false;
    uint64_t id = computeId(bitMask);
    if (id == defaultMarkers[MARKER_TYPE_COIN].id) {
      marker.type = MARKER_TYPE_COIN;
      valid = true;
    }
    // ... Check if marker is player, obstacle
    if (valid) {
      marker.rotationCount = rotationCount;
      marker.id = id;
      return true;
    }
    bitMask = rotate90deg(bitMask, false);
    rotationCount++;
  } while (rotationCount < 4);
  return false;
}
```

List. 4: Die Funktion `detectormarkerbased.cpp/isValidMarker();` überprüft, ob der übergebene Marker gesucht wird bzw. valide ist

Bevor der Rendering-Prozess ausgeführt werden kann, muss die Lage eines gefundenen Markers bestimmt werden, um ein 3D-Objekt korrekt über den Markern in der Szene zu rendern. Dieses Verfahren wird im kommenden Abschnitt näher erläutert.

2.3 Kamerakalibrierung und Lagebestimmung von Markern

Grundsätzlich bezeichnet die Kamerakalibrierung ein Verfahren, das die Parameter einer Kamera bestimmen und das für die Entzerrung des Bilds zuständig ist, das durch eine Objektiv-Kamera verzerrt wurde.[24] Zusätzlich kann mittels einer Kamerakalibrierung eine Beziehung zwischen den natürlichen Einheiten der Kamera, also den Pixeln, und den realen Einheiten, z.B. Millimeter, geschaffen werden.[25] Die Kamerakalibrierung innerhalb dieser Applikation wird ausschließlich mithilfe der Bibliothek OpenCV realisiert. Hierfür wird die OpenCV-Funktion `calibrateCamera()` benötigt, die anhand von mehreren unterschiedlichen Ansichten eines Kalibriermusters (s. Abb. 8) die intrinsischen und extrinsischen Kameraparameter findet.

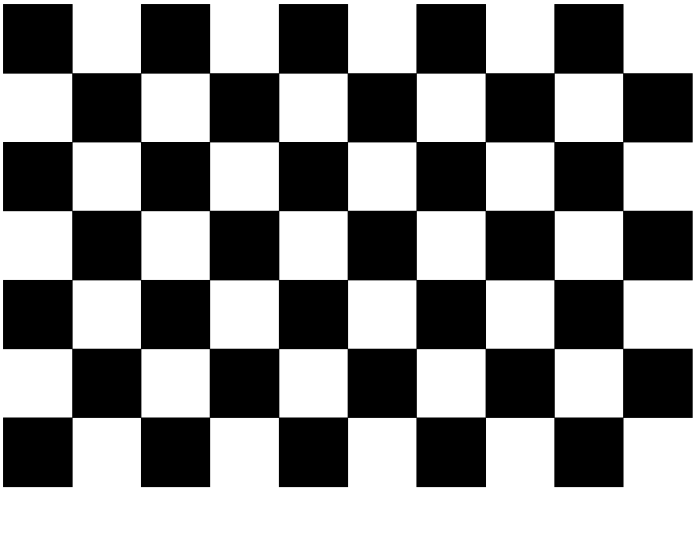

Abb. 8: Das Kalibriermuster, das für die Kamerakalibrierung mit OpenCV benötigt wird[26]

Beide Kameraparameter sind folgendermaßen beschrieben:

- **Intrinsische Kameraparameter** beschreiben die Beziehung zwischen dem Weltkoordinatensystem und dem Kamerakoordinatensystem.[27] Im OpenGL-Kontext sind die intrinsischen Parameter mit der Kameramatrix bzw. Viewmatrix vergleichbar. Die intrinsischen Parameter bestehen aus der Brennweite, Bildmittelpunkt und der Pixelskalierung.[28]
- **Extrinsische Kameraparameter** „beschreiben die Position und Orientierung der Kamera im Raum.“[29] Hierzu gehört einerseits die Beschreibung einer Verschiebung der Kamera zum Weltkoordinatenursprung durch einen Translationsvektor, andererseits die Beschreibung der Drehung um die drei Eulerwinkel durch einen Rotationsvektor.[30]

[24] Hofmann 2017.
[25] *Camera calibration With OpenCV* 2017.
[26] *Camera Calibration Chessboard Pattern* 2017
[27] Hofmann 2017.
[28] Ebd.
[29] Ebd.
[30] Ebd.

Für Augmented Reality-Anwendungen ist der Kalibrierungsprozess der Kamera ein wichtiger Bestandteil, da es die perspektivische Transformation und die Verzerrung des Ausgabebilds durch das Objektiv beschreibt. Um ein akzeptables Nutzererlebnis erreichen zu können, müssen Objekte im Augmented Reality-Kontext mit derselben perspektivischen Projektion visualisiert werden.[31]

Die Applikation ermöglicht dem Endnutzer eine Kamerakalibrierung durchzuführen (s. List. 5, Z. 8), zu berechnen (s. List. 5, Z. 4) und zu laden (s. List. 5, Z. 6). Insgesamt muss mindestens einmal eine Kamerakalibrierung durchgeführt werden. Alle drei Techniken werden in den nächsten Abschnitten beschrieben. Bevor eine Kalibrierung gestartet werden kann, muss das Kalibriermuster (s. Abb. 8) vom Endbenutzer ausgedruckt und die Kantenlänge eines Felds in Metern ermittelt werden.

```
1  // cc = Camera Calibration Settings
2  if (cc.cameraName.empty()) {
3    if (calibrationImages.size() > 0) {
4      return computeCameraCalibration(cc, calibrationImages);
5    } else if (fileExists(DEFAULT_CC_FILEPATH + DEFAULT_CC_CAMERA_NAME + ↩
         DEFAULT_CC_FILE_EXTENSION)) {
6      return loadCameraCalibration(cc, DEFAULT_CC_FILEPATH + ↩
           DEFAULT_CC_CAMERA_NAME + DEFAULT_CC_FILE_EXTENSION);
7    } else {
8      return startCameraCalibration(cc);
9    }
10 }
```

List. 5: Ein Ausschnitt der Funktion `camera.cpp/initializeCamera();` die überprüft, auf welche Art und Weise die Kamerakalibrierung durchgeführt werden soll

2.3.1 Berechnung der Kamerakalibrierung zur Laufzeit

Falls die Applikation ohne Pfadangabe einer Kalibrierungsdatei (s. Abs. 2.3.3) oder der Pfadangabe eines Ordners (s. Abs. 2.3.2), der Bilder von Ansichten eines Kalibriermusters speichert, aufgerufen wird, startet die Kamerakalibrierung zur Laufzeit automatisch. Die Live-Kamerakalibrierung wird von der Funktion `camera.cpp/startCameraCalibration()` realisiert. Innerhalb dieses Abschnitts wird mit Verweisen auf bestimmte Dateizeilen gearbeitet, da die Funktion `startCameraCalibration()` mehr als 80 Zeilen lang ist.

[31] Vgl. Baggio u. a. 2012, S. 76.

Abb. 9: Terminalausgaben beim Start einer Live-Kamerakalibrierung

Die Abb. 9 zeigt einen Ausschnitt der Terminalausgaben beim Start der Kalibrierung, in der der Nutzer aufgefordert wird globale Kerninformationen einzugeben (s. Dat. `camera.cpp`, Z. 181-198). Ein essentieller Teil bildet hierbei die Angabe der realen Maße der zu detektierenden Objekte (Kantenlänge eines Felds im Kalibriermuster und Kantenlänge eines Markers), da diese für die Projektion von 2D-Bildpunkten zu korrespondierenden 3D-Objektpunkten benötigt werden. Anschließend wird ein OpenCV-Fenster geöffnet und sämtliche Eingangsbilder der Webcam ausgespielt (s. Dat. `camera.cpp`, Z. 204-205, 224 und 238). In diesem Teil der Kalibrierung wird der Nutzer aufgefordert, das ausgedruckte Kalibriermuster in unterschiedlichen Ansichten in die Kamera zu halten. Falls das Kalibriermuster mithilfe der OpenCV-Funktion `findChessboardCorners()`[32] erkannt wurde (s. Dat. `camera.cpp`, Z. 226), werden die gefundenen Kreuzungen farblich mittels der OpenCV-Funktion `drawChessboardCorners()`[33] gekennzeichnet (s. Dat. `camera.cpp`, Z. 231) und der Nutzer ist in der Lage das Bild mit der Leertaste zu bestätigen (s. Abb. 10). Nach einer Bestätigung vom Nutzer, werden die 2D-Bildpunkte der gefundenen Kreuzpunkte in einem Vektor abgespeichert (s. Dat. `camera.cpp`, Z. 249).

[32] *Camera Calibration and 3D Reconstruction - findChessboardCorners* 2017.
[33] *Camera Calibration and 3D Reconstruction - drawChessboardCorners* 2017.

Abb. 10: Darstellung der Bildausgabe beim Fund des Kalibiermusters im Rahmen der Kamerakalibrierung

Für eine gute Kamerakalibrierung wird eine Anzahl von mindestens 50 Bildern empfohlen. Sämtliche bestätigte Bilder werden zur Laufzeit im Projektunterorder „`Images/CameraCalibration/`“ zwischengespeichert (s. Dat. `camera.cpp`, Z. 251) und können in einem späteren Zeitpunkt wiederverwendet werden (s. Abs. 2.3.2. Der Endbenutzer kann anschließend mit der Eingabetaste die Anzahl an gemachten Bildern bestätigen und die Kamerakalibrierung starten. An dieser Stelle werden alle 2D-Bildpunkte der gefundenen Kreuzungen im Kalibriermuster der Funktion `camera.cpp/calibrateCamera()` weitergeleitet, die daraufhin die eigentliche Kamerakalibrierung berechnet und das Ergebnis als Referenzparameter zurückliefert (s. List. 6). Die vom Nutzer eingegebene Kantenlänge eines Feldes im Kalibriermuster wird nun verwendet, um das Kalibriermuster dreidimensional nachzustellen (s. List. 6, Z. 6). Die 2D-Bildpunkte und die dazugehörigen 3D-Punkte werden der OpenCV-Funktion `calibrateCamera()` bereitgestellt, die dann daraus die intrinsischen und extrinsischen Kameraparameter berechnet (s. List. 6, Z. 8).

```
1 void calibrateCamera(CameraCalibration &cc, const std::vector<std::vector<cv::↩
    Point2f>> &chessboardImageSpacePoints)
2 {
3   std::cout << "Compute camera calibration..." << std::endl;
4   std::vector<cv::Mat> rVec, tVec;
5   std::vector<std::vector<cv::Point3f>> worldSpaceCornerPoints(1);
6   createKnownBoardPosition(DEFAULT_CC_CHESSBOARD_SIZE, cc.↩
      chessboardRealTileEdgeLength, worldSpaceCornerPoints[0]);
7   worldSpaceCornerPoints.resize(chessboardImageSpacePoints.size(), ↩
      worldSpaceCornerPoints[0]);
8   cv::calibrateCamera(worldSpaceCornerPoints, chessboardImageSpacePoints, ↩
      DEFAULT_CC_CHESSBOARD_SIZE, cc.cameraMatrix, cc.distanceCoefficients, rVec↩
```

```
        , tVec);
}
```

List. 6: Die Funktion `camera.cpp/calibrateCamera();` berechnet anhand von übergebenen 2D-Bildpunkten eines Kalibriermusters die intrinsischen und extrinsischen Kameraparameter

Abschließend werden die berechneten Daten mittels der Funktion `camera.cpp/saveCameraCalibration()` in eine lokale Kalibrierungsdatei im Projektunterordner „`Resources/`“ abgespeichert, die beim Start der Applikation mitgeliefert werden kann (s. Abs. 2.3.3). Wie solch eine Kalibrierungsdatei für diese Applikation strukturiert ist, wird im List. 7 beschrieben und der Tab. 1 näher erläutert.

2.3.2 Berechnung der Kamerakalibrierung aus lokalen Bildern

Wie zuvor im Abs. 2.3.1 beschrieben, werden bestätigte Bilder, die ein gefundenes Kalibiermuster aufweisen, zwischengespeichert und können wiederverwendet werden. Der Endnutzer muss lediglich beim Start der Applikation den Pfad zum Ordner mitliefern, in dem die zwischengespeicherten Bilder liegen (s. Abs. 3). Es wird empfohlen ein Ordner anzugeben, der nur Bilder beinhaltet und keine anderen Dateiformate. Letztendlich wird der im Abs. 2.3.1 beschriebene Prozess wiederholt, mit dem Unterschied, dass die Kamerakalibrierung direkt ausgeführt und aus den gelieferten Bildern berechnet wird. In diesem Fall wird beim Aufruf der OpenCV-Funktion `findChessboardCorners()`, die das Kalibriermuster innerhalb der mitgelieferten Bilder findet, auf den Parameter `CV_CALIB_CB_FAST_CHECK` verzichtet, da im Vergleich zur Live-Kalibrierung die erhaltenen Bilder einer Webcam nicht dargestellt werden müssen und folglich ein schnelles finden des Kalibriermusters im Bild nicht notwendig ist.

2.3.3 Laden der Kamerakalibrierung aus einer Datei

Der Endnutzer kann im Rahmen der Ausführung der Applikation eine Datei mitliefern, die eine schon zuvor berechnete Kamerakalibrierung beschreibt (s. Abs. 3). Die Sicherung der Eigenschaften einer Kamerakalibrierung ermöglicht die Ausführung der Applikation, ohne eine erneute Berechnung der Kalibrierung durchführen zu müssen. Dateien, die eine Kamerakalibrierung für diese Applikation speichern, weisen die Dateiendung “`.ccc`“ auf. Das List. 7 zeigt die Struktur einer beispielhaften Kalibrierungsdatei. Zugleich beschreibt die Tab. 1 den semantischen Inhalt einzelner Zeilen der Datei.

Zeile	Beschreibung
1	Der Name der Kamera.
2 - 3	Die Dimension des Kalibriermusters (Schachbrett) für die Kamerakalibrierung mit OpenCV. Hierbei ist zu beachten, dass nicht nicht die Anzahl an Felder gemein sind, sondern die Anzahl an Kreuzungen von vier Feldern.
4	Die Länge der Kante eines Markers in Meter.
5	Die Länge der Kante eines Felds im Kalibriermuster (Schachbrett) in Meter.
6 - 7	Die Dimension der Kameramatrix.
8 - 16	Der Inhalt der Kameramatrix reihenweise aufgelistet.
17 - 18	Die Dimension des Verzerrungsvektors.
19 - 22	Der Inhalt des Verzerrungsvektors reihenweise aufgelistet.

Tab. 1: Beschreibung des semantischen Inhalts einzelner Zeilen einer Kalibrierungsdatei

```
1  Logitech720p
2  9
3  6
4  0.04
5  0.025
6  3
7  3
8  1097.31
9  0
10 639.544
11 0
12 1101.1
13 384.564
14 0
15 0
16 1
17 5
18 1
19 0.0877583
20 -0.0279325
21 0.00206762
22 -0.00514454
23 -0.393854
```

List. 7: Struktur einer Datei, die eine Kamerakalibrierung speichert und die Dateiendung “`.ccc`“ aufweist

2.3.4 3D-Rekonstruktion

Damit ein virtuelles 3D-Objekt in einer Bildszene platziert werden kann, muss die Objektposition in Bezug auf die Kamera bekannt sein. Im Rahmen der Applikation wird eine euklidische Transformation im kartesischen Koordinatensystem verwendet, um die Objektposition darzustellen. Die Position des Markers in 3D (Model Space) und seine entsprechende Projektion in 2D (Window Space) wird durch die folgende Gleichung[34] dargestellt:

$$p' = C[R|T]p$$

oder [35]

$$s \begin{bmatrix} p'_x \\ p'_y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

Wo:

- p': Projektion vom Punkt p im Window Space
- C: Kameramatrix bzw. Matrix aus intrinsischen Parametern
- f_x, f_y: Brennweite in Pixeln
- c_x, c_y: Bildmittelpunkt in Pixeln
- $[R|T]$: Euklidische Transformation bzw. Matrix aus extrinsischen Parametern
- p: Punkt im Weltkoordinatensystem

Nach der Durchführung der Detektion der Marker und der Kamerakalibrierung sind zum einen die genauen Positionen der Marker im Window Space bekannt, zum anderen sind die in- und extrinsischen Kameraparameter ermittelt. Beide Informationen werden anschließend für die Approximation einer euklidischen Transformation zwischen der Kamera und einem Marker im dreidimensionalen Raum verwendet. Das List. 8 zeigt die Funktion `estimatePosition()`, die die gesuchte euklidische Transformation zwischen der Kamera und einem Marker im dreidimensionalen Raum bestimmt.

```
1 void estimatePosition(Marker &marker, const CameraCalibration &cc) {
2   cv::Mat raux, taux;
```

[34] Vgl. Baggio u. a. 2012, S. 76.
[35] *Camera Calibration and 3D Reconstruction* 2017.

```
3   cv::Mat rotationVector;
4   cv::Mat_<float> translationVector;
5   cv::solvePnP(getMarker3DPoints(cc.markerRealEdgeLength), marker.points, cc.↩
        cameraMatrix, cc.distanceCoefficients, raux, taux);
6   raux.convertTo(rotationVector, CV_32F);
7   taux.convertTo(marker.translationVector, CV_32F);
8   cv::Rodrigues(rotationVector, marker.rotationMatrix);
9   marker.rotationMatrix = marker.rotationMatrix.inv();
10  marker.translationVector = -marker.translationVector;
11 }
```

List. 8: Die Funktion `detectormarkerbased.cpp/estimatePosition();` bestimmt die Lage eines Markers im dreidimensionalen Raum im Bezug auf die Kamera

Die Funktion `estimatePosition()` wird für jeden detektierten Marker aufgerufen und die extrinsischen Parameter des Markers im Bezug auf die Kamera approximiert. Den Kern der Funktion bildet der Aufruf der OpenCV-Funktion `solvpnp()` (s. List. 8, Z. 5), die die Lage eines Objekts aus 2D- und 3D-Punktpaaren findet.[36] Hierfür benötigt die OpenCV-Funktion die Eckpunkte des Markers im dreidimensionalen Raum, die dazu korrespondierenden Eckpunkte des Markers im Window Space und die Kameramatrix. Wie im Abs. 2.3.1 beschrieben, wird der Nutzer zu Beginn der Kamerakalibrierung aufgefordert die reale Kantenlänge eines Markers in Metern anzugeben (s. Abb. 9, Z. 8). Mithilfe der eingegebenen Kantenlänge ermittelt die Funktion `getMarker3DPoints()` die Punkte des Markers im dreidimensionalen Raum. Letztendlich werden die Eckpunkte eines Markers gleichmäßig um den Mittelpunkt verteilt (s. List. 9).

```
1 std::vector<cv::Point3f> getMarker3DPoints(const float &markerRealEdgeLength) {
2   return {cv::Point3f(markerRealEdgeLength * -0.5f, markerRealEdgeLength * 0.5f,↩
        0.0f),
3           cv::Point3f(markerRealEdgeLength * -0.5f, markerRealEdgeLength * -0.5f↩
              , 0.0f),
4           cv::Point3f(markerRealEdgeLength * 0.5f, markerRealEdgeLength * -0.5f,↩
               0.0f),
5           cv::Point3f(markerRealEdgeLength * 0.5f, markerRealEdgeLength * 0.5f, ↩
              0.0f)};
6 }
```

List. 9: Die Funktion `detectormarkerbased.cpp/getMarker3DPoints();` bestimmt die Position eines Markers im dreidimensionalen Raum

Als Ergebnis liefert die OpenCV-Funktion `solvepnp()` die entsprechenden extrinsischen Parameter (Rotations- und Translationsvektor) für einen bestimmten Marker. Damit im Rahmen des Rendering-Prozesses eine einfache Erzeugung der Modelview-Matrix für einen Marker gewährleistet ist, wird sowohl der Rotationsvektor in eine Rotationsmatrix umgewandelt (s. List. 8, Z. 8), als auch die Rotationsmatrix und der Translationsvektor invertiert, da die OpenCV-Funktion `solvepnp()` die Kameraposition in Bezug auf die Lage des Markers im dreidimensio-

[36] *Camera Calibration and 3D Reconstruction - solvePnP* 2017.

nalen Raum findet und im OpenGL-Kontext eine Transformation der 3D-Markerposition in den View Space benötigt wird.

2.4 Rendern von 3D-Objekten auf detektierten Bitmasken

Die endgültige Bildausgabe ist mittels OpenGL realisiert und in drei Phasen eingeteilt:

1. Initialisierung des OpenGL-Kontexts und starten der Ereignisverarbeitungsschleife von OpenGL mit `glutMainLoop()`[37]
2. Die durch die Detektion von Markern verarbeitete Bildszene mittels einer orthogonalen Projektion auf ein flache Ebene rendern
3. Aktuelle Markerpositionen als Modelview-Matrix laden, virtuelles 3D-Objekt zeichnen und mittels einer perspektivischen Matrix in den Window Space projizieren

Nach der erfolgreichen Registrierung sämtlicher Callback-Funktionen, startet die Ereignisverarbeitungsschleife von OpenGL. Innerhalb der Ereignisverarbeitungsschleife spielen zwei Callback-Funktionen, `timerCallback()` (s. Dat. `rendering3d.cpp`, Z. 543) und `drawCallback()` (s. Dat. `rendering3d.cpp`, Z. 421) eine wichtige Rolle. In der `timerCallback()`-Funktion wird die Detektion von Markern angestoßen, gefundene Marker in einem globalen Vektor abgespeichert, die verarbeitete Bildszene global gesichert und letztlich die Szene im OpenGL-Kontext gezeichnet. Die detektierten Marker und die aktuelle Bildszene müssen als globale Variablen deklariert werden, damit diese in sämtliche Callback-Funktionen zur Verfügung stehen. Innerhalb der Funktion `drawCallback()` wird zu Beginn die verarbeitete Bildszene als Hintergrund gezeichnet. Dies wird mittels der Funktion `drawBackground()` (s. Dat. `rendering3d.cpp`, Z. 333) realisiert.

```
1  void drawBackground(void) {
2    glEnable(GL_TEXTURE_2D);
3    glGenTextures(1, &bgTexId);
4    // ...
5    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, bg.cols, bg.rows, 0, GL_BGR_EXT, ↩
         GL_UNSIGNED_BYTE, bg.data);
6    const GLfloat bgTexVertices[] = {0, 0, static_cast<float>(bg.cols), 0, 0, ↩
         static_cast<float>(bg.rows), static_cast<float>(bg.cols), static_cast<↩
         float>(bg.rows)};
7    const GLfloat bgTexels[] = {1, 0, 1, 1, 0, 0, 0, 1};
8    glMatrixMode(GL_PROJECTION);
9    glLoadMatrixf(makeOrthographic(bg.cols, bg.rows).data);
10   // ...
11   glVertexPointer(2, GL_FLOAT, 0, bgTexVertices);
12   glTexCoordPointer(2, GL_FLOAT, 0, bgTexels);
13   // ...
14   glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

[37] *3.1 glutMainLoop* 2017.

```
15   // ...
16 }
```

List. 10: Die Funktion `rendering3d.cpp/drawBackground();` zeichnet den Inhalt des aktuell verarbeiteten Bilds auf eine 2D-Ebene, die orthogonal zu Kamera steht

Hierfür muss die Kamera mittels einer orthogonalen Projektion (s. List. 10, Z. 8-9) rechtwinklig vor eine 2D-Ebene, die genau so groß ist wie die verarbeitete Bildszene (s. List. 10, Z. 6), positioniert werden. Anschließend wird die Bildszene als Textur auf die 2D-Ebene gezeichnet (s. List. 10, Z. 14). Die im List. 10 genannten Variablen `bg` und `bgTexId` bilden globale Variablen, die sowohl die verarbeitete Bildszene, als auch die dazugehörige Textur im OpenGL-Kontext speichern.

Nachdem die Bildszene als Hintergrund in den Framebuffer geschrieben wurde, müssen anschließend virtuelle 3D-Objekte an den gefundenen Markerpositionen gezeichnet werden. Hierfür wurde die Funktion `drawAR()` (s. Dat. `rendering3d.cpp`, Z. 282) implementiert.

```
1  void drawAR(void) {
2    glEnable(GL_DEPTH_TEST);
3    glDepthFunc(GL_LESS);
4    glMatrixMode(GL_PROJECTION);
5    glLoadMatrixf(makePerspective(cc.cameraMatrix.at<double>(0, 0),
6      cc.cameraMatrix.at<double>(1, 1),
7      cc.cameraMatrix.at<double>(0, 2),
8      cc.cameraMatrix.at<double>(1, 2),
9      CAMERA_WIDTH, CAMERA_HEIGHT,
10     0.1f, 10000.0f).data);
11   glMatrixMode(GL_MODELVIEW);
12   glLoadIdentity();
13   if (dm.size() > 0) {
14     for (size_t i = 0; i < dm.size(); i++)  {
15       Mat4 transformation = makeMatRows(
16         dm[i].rotationMatrix(0, 0), dm[i].rotationMatrix(0, 1), dm[i].↩
             rotationMatrix(0, 2), 0.0f,
17         dm[i].rotationMatrix(1, 0), dm[i].rotationMatrix(1, 1), dm[i].↩
             rotationMatrix(1, 2), 0.0f,
18         dm[i].rotationMatrix(2, 0), dm[i].rotationMatrix(2, 1), dm[i].↩
             rotationMatrix(2, 2), 0.0f,
19         dm[i].translationVector(0), dm[i].translationVector(1), dm[i].↩
             translationVector(2), 1.0f);
20       glLoadMatrixf(&transformation.data[0]);
21       // ... Draw functions
22     }
23   }
24   glDisable(GL_DEPTH_TEST);
```

```
25 }
```

List. 11: Die Funktion `rendering3d.cpp/drawAR();` zeichnet über gefundenen Markern 3D-Objekte

Im Rahmen der Initialisierung des OpenGL-Kontextes wurde der Tiefentest bewusst nicht eingeschaltet, da Objekte mit unterschiedlichen Projektionen gezeichnet werden. Beim eingeschlatenen Tiefentest würde demzufolge der zuvor gezeichnete Hintergrund immer im Vordergrund liegen und die nachfolgenden 3D-Objekte verdecken, da der Hintergrund mittels einer orthogonalen Projektion gezeichnet wurde und folglich direkt vor der Kamera positioniert ist. Aus diesem Grund wird der Tiefentest nur im Rahmen der Zeichnung der 3D-Objekte eingeschaltet (s. List. 11, Z. 2-3 und 24). Um 3D-Objekte zeichnen zu können, benötigen wir eine perspektivische Projektion in Bezug auf den intrinsischen Kameraparametern, die die Kamerakalibrierung ergeben hat (s. List. 11, Z. 4-10). Die Funktion `makePerspective()` (s. List. 12) erzeugt hierfür anhand der übergebenen intrinsischen Parameter eine perspektivische Transformationsmatrix im OpenGL-Kontext, die Koordinaten im View Space in den Window Space überführt.

```
1 Mat4 makePerspective(const float &fx, const float &fy, const float &cx, const ↩
      float &cy, const int &w, const int &h, const float &near, const float &far)
2 {
3   return makeMatRows(
4     -2.0f * fx / w, 0.0f, 0.0f, 0.0f,
5     0.0f, 2.0f * fy / h, 0.0f, 0.0f,
6     2.0f * cx / w - 1.0f, 2.0f * cy / h - 1.0f, (-far - near) / (far - near), ↩
          -1.0f,
7     0.0f, 0.0f, -2.0f * far * near / (far - near), 0.0f);
8 }
```

List. 12: Die Funktion `algebra.cpp/makePerspective();`, die anhand von intrinsischen Parametern eine perspektivische Projektionsmatrix im OpenGL-Kontext erzeugt

Wie im List. 9 beschrieben, wurden die dreidimensionalen Eckpunkte eines Markers um den Mittelpunkt gesetzt. Im Rahmen der 3D-Rekonstruktion (s. Abs. 2.3.4) wurden diese 3D-Punkte auf die korrespondierenden 2D-Bildpunkte der gefundenen Marker projiziert. Die daraus resultierende Rotationsmatrix bzw. resultierender Translationsvektor werden nun benötigt, um den entsprechenden Mittelpunkt des Markers im Model Space zu finden. Die Kombination der beiden extrinsischen Parameter bildet die Modelview-Matrix des jeweiligen Markers. Bei der Erzeugung der Modelview-Matrix muss berücksichtigt werden, dass Matrizen im OpenCV-Kontext reihenweise und im OpenGL-Kontext spaltenweise gespeichert werden. Folglich ist eine Transponierung der Matrix notwendig. Anschließend wird die Transformationsmatrix als Modelview-Matrix geladen und entsprechende Zeichenroutinen von OpenGL können ausgeführt werden, um 3D-Objekte auf Markern zu zeichnen (s. Abb. **??**).

TODO BILD VOM ENDPRODUKT

2.5 Kollisionserkennung von 3D-Objekten

Aus diversen Gründen konnte eine Kollisionsberechnung nicht implementiert werden. Aus diesem Grund wird in diesem Abs. beschrieben warum die Implementierung der Kollisionserkennung nicht möglich war und welche Lösungsansätze getestet wurden. Alle Code-Ausschnitte der getesteten Lösungsansätze sind nicht im abgegebenen Projektquellcode enthalten. Nichtsdestotrotz werden die wichtigsten Codebeispiele in diesem Abs. erläutert.

Der Plan vor der Implementierung der Kollisionserkennung bildete die Verwaltung eines weiteren Translationsvektor, der die Position des Spielerobjekts bewegt (s. Dat. `rendering3d.cpp`, Z. 26). Bevor die Modelview-Matrix des Markers für die Zeichnung geladen werden würde, müsste der neue Translationsvektor lediglich auf den extrinsischen Translationvektor des jeweiligen Markers addiert werden. Grafisch betrachtet war eine korrekte Manipulation der Spielerobjektposition möglich, aber keine Kollisionserkennung mit anderen Objekten. Demzufolge wäre das Einsammeln von Münzen nicht realisierbar. Der Grund hierfür liegt in der Tatsache, dass jeder Marker sein eigenes Koordinatensystem besitzt und folglich sämtliche 3D-Makerobjekte in keinster Weise in Relation stehen. Das liegt daran, dass in Rahmen der 3D-Rekonstruktion (s. Abs. 2.3.4) sämtliche Mittelpunkte der 2D-Polygone bzw. Konturen von Markern auf den 3D-Mittelpunkt des Koordinatensystems projiziert wurden. In einer Bildszene mit zwei Markern würde sofort eine Kollision erkannt werden, da beide 3D-Objekte der Marker mathematisch im Mittelpunkt ihrer Koordinatensysteme liegen. Dieser Ansatz ist im abgegebenen Projektquellcode wiederzufinden.

Die erfolgversprechendste Lösung bildete das Hinzufügen eines neuen Marker-Typs, der das Zentrum des Spielfelds darstellt. Das globale Ziel bildet, dass sich sämtliche Marker im gleichen Koordinatensystem befinden, um eine Relation zwischen einzelnen Markern zu schaffen. Mit genau einem Marker, der das Zentrum des Koordinatensystems darstellt, gibt es die Möglichkeit Objekte zu definieren die alle im gleichen Koordinatensystem liegen. Folglich wäre eine Kollisionserkennung möglich. Nichtsdestotrotz ist dieser Ansatz nur dann möglich wenn in der Bildszene nur ein Marker bzw. ein Koordinatensystem existiert.

Ein globales Teilziel bildet die gleichzeitige Verwendung von verschiedenen Marker, die untereinander interagieren können. Der zuvor beschriebene Ansatz mit der Deklaration eines neuen Marker-Typs ist eine weitere Lösung denkbar. Hierbei wird die inverse 3D-Rekonstruktion des neuen Marker-Typs verwendet, um die 2D-Bildpunkte anderer Marker ins Koordinatensystem des neuen Marker-Typs zu transferieren. Folglich wären sämtliche Marker im gleichen Koordinatensystem bzw. im Koordinatensystem des neuen Marker-Typs. Hierfür müsste der Algorithmus wie folgt angepasst werden:

1. Die Suche von Markern im der Bildszene bleibt erhalten
2. Ein neuer Marker-Typ wird definiert, der das Zentrum des Spielfeld darstellt und nur einmal im Bild vorkommen darf
3. Die 3D-Rekonstruktion bzw. Ermittlung der extrinsischen Parameter wird nur für den

neuen Marker-Typ durchgeführt und global gespeichert

4. Die ermittelten extrinsischen Parameter des neuen Marker-Typs werden für eine inverse 3D-Rekonstruktion verwendet, um 2D-Bildpunkte von weiteren gefundenen Markern in das Koordinatensystem des zentralen Markers zu transformieren und anschließend eine Zeichnung durchzuführen

Dieser Ansatz wurde in groben Zügen implementiert, um den allgemeinen Prozess testen zu können. Für die inverse 3D-Rekonstruktion wird die Funktion `get3DPoint()` (s. List. 13) benötigt, die einen übergebenen Marker im View Space in die korrespondierende 3D-Koordinate im Model Space des zentralen Markers umwandelt.

```
1  void get3DPoint(const Marker &marker, cv::Point3f realPoint) {
2    cv::Mat imagePoint = cv::Mat::ones(3, 1, cv::DataType<double>::type);
3    imagePoint.at<double>(0, 0) = marker.points[0].x;
4    imagePoint.at<double>(1, 0) = marker.points[0].y;
5    cv::Mat m = rotationMatrix.inv() * cc.cameraMatrix.inv() * uvPoint;
6    cv::Mat m2 = rotationMatrix.inv() * translationVector;
7    double s = m2.at<double>(2, 0);
8    s /= m.at<double>(2, 0);
9    cv::Mat wcPoint = rotationMatrix.inv() * (s * cc.cameraMatrix.inv() * ↩
         imagePoint - translationVector);
10   realPoint.x = wcPoint.at<double>(0, 0);
11   realPoint.y = wcPoint.at<double>(1, 0);
12   realPoint.z = 0.0f;
13 }
```

List. 13: Die Funktion `get3DPoint();`, die einen übergebenen Marker im View Space mittels einer inversen Transformation in den Model Space des zentralen Markers transformiert und nicht Bestandteil des Projektquellcodes ist

Letztendlich muss die im Abs. 2.3.4 beschriebene Gleichung in folgende umgewandelt werden, um einen 2D-Bildpunkt in einen dazugehörigen 3D-Punkt umzurechnen:

TODO GLEICHUNG WIEDERFINDEN UND BESCHREIBEN... BESCHREIBUNG FÜR S FEHLT

Für einen grundsätzlichen Test wurde versucht ein Eckpunkt eines gefundenen Markers im View Space in den Model Space des zentralen Markers umzuwandeln. Wie die Abb. zeigt, liegt die graue Kugel, die den BLAUEN? 2D-Eckpunkt des Markers im dreidimensionalen Raum darstellen soll, nicht im korrespondierenden 2D-Punkt des Markers. Beim Versuch die Position des gefundenen Markers zu verschieben, bewegt sich die graue 3D-Kugel in die korrekte Richtung mit, obwohl sie dennoch nicht im korrespondieren 2D-Bildpunkt liegt. Der Grund für den Versatz der Position des 3D-Objektes könnte an der Verzerrung des Bildes liegen, die durch die Kamera erzeugt wurde, wobei dies nur eine Vermutung ist und aus zeitkritischen Gründen nicht näher geprüft werden kann.

TODO ABBILDUNGEN VOM TEST

3 Bedienung der Applikation

Der gesamte Projektquellcode ist auf GitHub öffentlich verfügbar[38] und kann als ZIP-Datei in ein beliebiges Ordner heruntergeladen werden. Es wird empfohlen unter einem Linux-System zu testen, da im Projekt ein Makefile mitgeliefert wird. Das List. 14 beschreibt die Projektstruktur.

```
- Application/
  -- Header/
    --- // Wie Source-Ordner nur mit Header-Dateien
  -- Images/
    --- CameraCalibration/
      ---- // Bilder mit Kalibriermuster, die im Rahmen der Live-Kalibrierung ↩
          gemacht wurden
    --- chessboard.png // Kalibriermuster
    --- markerCoin.jpg // Standard-Marker für die Mönze
    --- markerObstacle.jpg // Standard-Marker für die Hindernisse
    --- markerPlayer.jpg // Standard-Marker für den Spieler
  -- Resources/
    --- Logitech720p.ccc // Kalibrierungdatei die im Rahmen der Entwicklung ↩
        verwendet wurde
  -- Source/
    --- ImageDetection/
      ---- camera.cpp
      ---- detectormarkerbased.cpp
    --- Logging/
      ---- logger.cpp
    --- Rendering3D/
      ---- rendering3d.cpp
    --- Utilities/
      ---- algebra.cpp
      ---- argparse.cpp
      ---- imageio.cpp
      ---- utils.cpp
    --- main.cpp
  -- Makefile
- Test/
  -- Header/
    --- catch.h // Header-only test framework für C++ Applikationen; https://↩
        github.com/philsquared/Catch
  -- Source/
    -- // Wie ../Application/Source-Ordner nur mit C++-Dateien die getestet ↩
        wurden
```

List. 14: Die Beschreibung der allgemeinen Projektstruktur

Um die Applikation starten zu können, muss ein Terminal im Projektunterordner „Application/“ geöffnet und mit dem Kommando „`make`“ die Applikation gebaut werden. Je nachdem

[38]Das Projekt ist unter https://github.com/aoezd/Virtual-Reality-Praktikum aufrufbar.

welche Version von OpenCV/OpenGL installiert ist bzw. wo diese Bibliotheken im jeweiligen System installiert sind, kann es sein, dass Pfade zu Header-Dateien im Quellcode angepasst werden müssen. Nachdem die Applikation erfolgreich gebaut wurde, kann diese mit den folgenden Eingabeparametern gestartet werden:

- `-h, --help`: Spielt die Bedienungsanleitung auf der Konsole aus
- `-mc, --markercoin [Pfad zum Markerbild]`: Definiert den Marker für das Objekt einer Münze. Standardmäßiger Marker: „`Application/Images/markerCoin.jpg`“
- `-mo, --markerobstacle [Pfad zum Markerbild]`: Definiert den Marker für das Objekt eines Hindernisses. Standardmäßiger Marker: „`Application/Images/markerObstacle.jpg`“
- `-mp, --markerplayer [Pfad zum Markerbild]`: Definiert den Marker für das Objekt des Spielers. Standardmäßiger Marker: „`Application/Images/markerPlayer.jpg`“
- `-ccc, --cameracalibrationcompute [Pfad zum Ordner mit Kalibrierungsbildern]`: Berechnet die Kamerakalibrierung anhand der Bilder die im übergebenen Ordner
- `-ccl, --cameracalibrationload [Pfad zur Kalibrierungsdatei]`: Lädt die Kamerakalibrierung aus der übergebenen Datei.

Falls die Applikation ohne die Angabe eines Eingabeparameters ausgeführt wird, startet eine Live-Kamerakalibrierung (s. Abs. 2.3.1) automatisch und als Standard definierte Marker werden geladen. Mithilfe der Taste „`h`“ können zur Laufzeit weitere Informationen ausgespielt werden (siehe BILD TODO).

3.1 Systemeinstellungen zurzeit der Entwicklung

Die Applikation wurde auf einem System mit folgenden Eigenschaften entwickelt und getestet:

Komponente	**Beschreibung**
`System`	Linux
`Betriebssystem`	Xubuntu 16.04
`Grafikkarte`	NVIDIA GeForce GTX 970
`Physischer Speicher`	16 GB
`Grafikkartenspeicher`	3.5 GB
`OpenGL-Version`	TODO
`OpenCV-Version`	TODO

Tab. 2: Beschreibung der Systemeinstellungen zurzeit der Entwicklung

4 Literaturverzeichnis

4.1 Literatur

Baggio, Daniel Lélis u. a. (2012). *Mastering OpenCV with Practical Computer Vision Projects - Step-by-step tutorials to solve common real-world computer vision problems for desktop or mobile, from augmented reality and number plate recognition to face recognition and 3D head tracking*. 1. Aufl. Birmingham, UK: Packt Publishing Ltd. ISBN: 978-1-84951-782-9. URL: http://www.inc.eng.kmutt.ac.th/inc161/project/opencv/Mastering_opencv.pdf.

Otsu, Nobuyuki (1979). »A threshold selection method from grey level histograms«. In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-9.1, S. 62–66.

4.2 Internetquellen

3.1 glutMainLoop (2017). URL: https://www.opengl.org/resources/libraries/glut/spec3/node14.html (besucht am 21. 10. 2017).

Adaptive Thresholding Description (2017). URL: http://docs.opencv.org/3.2.0/d7/d4d/tutorial_py_thresholding.html (besucht am 10. 10. 2017).

Adaptive Thresholding Example (2017). URL: http://docs.opencv.org/3.2.0/ada_threshold.jpg (besucht am 10. 10. 2017).

ArUco: a minimal library for Augmented Reality applications based on OpenCV (2017). URL: https://www.uco.es/investiga/grupos/ava/node/26 (besucht am 17. 10. 2017).

Camera Calibration and 3D Reconstruction (2017). URL: https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.htm (besucht am 21. 10. 2017).

Camera Calibration and 3D Reconstruction - drawChessboardCorners (2017). URL: https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#drawchessboardcorners (besucht am 21. 10. 2017).

Camera Calibration and 3D Reconstruction - findChessboardCorners (2017). URL: https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#findchessboardcorners (besucht am 21. 10. 2017).

Camera Calibration and 3D Reconstruction - solvePnP (2017). URL: https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html (besucht am 19. 10. 2017).

Camera Calibration Chessboard Pattern (2017). URL: https://docs.opencv.org/2.4/_downloads/pattern.png (besucht am 18. 10. 2017).

Camera calibration With OpenCV (2017). URL: https://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html (besucht am 18. 10. 2017).

Class DescriptorExtractor (2017). URL: https://docs.opencv.org/java/2.4.9/org/opencv/features2d/DescriptorExtractor.html (besucht am 20. 10. 2017).

cv::DescriptorMatcher Class Reference (2017). URL: https://docs.opencv.org/3.2.0/db/d39/classcv_1_1DescriptorMatcher.html (besucht am 20. 10. 2017).

cv::FeatureDetector Class Reference (2017). URL: https://docs.opencv.org/ref/2.4/d9/df0/classcv_1_1FeatureDetector.html (besucht am 20. 10. 2017).

Example for bad matches with DescriptorMatcher (2017). URL: http://answers.opencv.org/upfiles/1428488468101951.jpg (besucht am 20. 10. 2017).

Geometric Image Transformations - getPerspectiveTransform (2017). URL: https://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html (besucht am 17. 10. 2017).

Geometric Image Transformations - warpPerspective (2017). URL: http://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html (besucht am 10. 10. 2017).

Hofmann, Tim (2017). *Geometrische Kamerakalibrierung*. URL: http://www.mi.hs-rm.de/~schwan/Projects/CG/CarreraCV/doku/intrinsisch/intrinsisch.htm (besucht am 18. 10. 2017).

Mastering OpenCV with Practical Computer Vision Projects - Step-by-step tutorials to solve common real-world computer vision problems for desktop or mobile, from augmented reality and number plate recognition to face recognition and 3D head tracking (2017). URL: http://www.inc.eng.kmutt.ac.th/inc161/project/opencv/Mastering_opencv.pdf (besucht am 17. 10. 2017).

OpenCV Basics 15-21 (2017). URL: http://youtu.be/HNfPbw-1e_w (besucht am 20. 10. 2017).

Structural Analysis and Shape Descriptors - approxPolyDP (2017). URL: http://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html (besucht am 10. 10. 2017).

Structural Analysis and Shape Descriptors - findContours (2017). URL: http://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html (besucht am 10. 10. 2017).

5 Eidesstattliche Erklärung

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

Ort, Datum	Unterschrift (Vor- und Nachname)