Inhaltsverzeichnis

1 Einleitung

1.1 Ausganssituation

Drei dimensionale Techniken schreiten immer weiter voran. In den letzten Jahren kamen die ersten 3D-Drucker auf den Markt und mittlerweile sind diese soweit ausgereift, dass einige Modelle sogar für den privaten Verbraucher erschwinglich sind. Zusammen mit den 3D-Druckern reift auch die Technik im Bereich des 3D-Scannens voran. Genau wie 3D-Drucker gibt es mittlerweile auch viele verschiedene 3D-Scanner. Waren die Scanner vor einigen Jahren noch sperrig, so sind sie jetzt handlich und kompakt.

Diese Fortschritte in der Technik bieten für viele Branchen große Vorteile. Im Maschinenbau können nun Prototypen von Maschinen und dessen Bauteile gescannt und mit einem 3D-Drucker vervielfältigt werden. Das erleichtert den Entwicklungsprozess, da Prototypen nun nicht mehr manuell hergestellt werden müssen.

Gebäude und Räume können nun gescannt werden, um diese zu vermessen oder zu gestalten. Dies kommt dem Bereich der Architektur zu gute.

Multimediale Bereiche profitieren am meisten von dieser Entwicklung. 3D Modelle von Menschen müssen nun nicht mehr manuell modelliert werden, sondern können einfach gescannt werden. Dies erspart erhebliche Kosten und Arbeitszeit in der Entwicklung von Filmen und Videospielen.

Auch in der Medizin kommen diese Techniken zum Einsatz. Besonders dann, wenn Körperteile vermessen werden müssen. Durch das 3D-Scannen können beispielsweise Prothesen anhand von 3D-Scans der gesunden Körperteile entwickelt werden. Dadurch können Prothesen individueller und schneller gestaltet werden. Außerdem erspart das Scannen auch hier Arbeitszeit und Kosten. Ohne die 3D-Technik nehmen diese Prozesse unheimlich viel Zeit in Anspruch.

Dasselbe gilt auch für die Orthesenanpassung. Patienten müssen mühselig vermessen werden, um die Orthesen exakt und individuell anzupassen. Dies erfordert viel Geduld vom Patienten und dem Orthopädietechniker. Auch in diesem Bereich kann die 3D-Technik aushelfen, in dem es den Vermessungsprozess beschleunigt. Somit erspart man dem Patienten und dem Orthopädietechniker unnötige Strapazen.

1.2 Zielsetzung

Am Ende dieser Arbeit soll ein 3D-Scanner fertiggestellt sein, der es einem Orthopädietechniker erlaubt, z.B. das Bein des Patienten drei dimensional zu scannen.

Es muss also ein Scanner entwickelt werden, der kompakt genug ist, um in der Hand geführt werden zu können. Außerdem muss der Scanner dafür sorgen, dass einzelne Körperteile des Patienten komplett gescannt werden können. Das heißt der Scanner muss 360° Scans von Objekten ermöglichen.

Die Software des Scanners sollte außerdem intuitiv und leicht verständlich gestaltet sein. Somit muss sich der Orthopädietechniker nicht langwierig in die Software einarbeiten.

Der Scanvorgang sollte nicht länger dauern als eine manuelle Vermessung. D.h. der Vorgang sollte nur einige Minuten in Anspruch nehmen.

1.3 Gliederung der Arbeit

Die Arbeit beginnt mit dem Abschnitt „Grundlagen“. In diesem Abschnitt werde ich die Grundlagen von 3D-Scannern erläutern und erklären, wie sie funktionieren. Außerdem werde ich hier auch den Stand der Technik darstellen und einen Überblick über die verschiedenen Scan Methoden verschaffen. Zusätzlich zu den Methoden werden hier auch bereits bestehende Scansysteme vorgestellt.

Im nächsten Abschnitt „Material und Methode“ werde ich verschiedene Methoden der Softwareentwicklung vorstellen und anwenden. Des Weiteren werde ich hier die von mir verwendete Programmiersprache und Entwicklungsumgebung vorstellen, sowie erörtern warum genau diese ausgewählt wurden. Anschließend stelle ich die Grundlagen der von mir gewählten Programmiersprachen dar, um dem Leser eine Basis für das Verstehen des Quellcodes zu bieten. Zusätzlich werde ich hier auch näher auf die von mir verwendete „Asus Xtion Pro Live“ Kamera detailliert eingehen. In diesem Abschnitt wird auch unter Anwendung der verschiedenen Methoden ein Grundgerüst für die Software entstehen. Dieses werde ich dann im nächsten Abschnitt als Hilfe nehmen, um die Software zu entwickeln.

Im Abschnitt „Implementation“ werde ich schließlich auf die Implementation der Software eingehen. Dazu werde ich die Umsetzung der einzelnen Klassen präsentieren und im Detail erklären wie die Methoden dieser Klassen funktionieren.

Innerhalb des letzten Abschnittes „Diskussion und Ausblick“ werde ich schließlich die Arbeit zusammenfassen und evaluieren. Im letzten Absatz dieses Abschnittes werde ich einen Ausblick auf die Zukunft der Software geben.

2 Grundlagen

2.1 Grundlegende Funktionsweise von 3D-Scannern

Ein 3D-Scanner scannt die Umgebung bzw. ein Objekt auf das es Gerichtet ist, und digitalisiert diese in Form von 3D-Modellen. Um dies zu bewerkstelligen, müssen Tiefeninformationen der Umgebung gesammelt werden. Obwohl jeder 3D-Scanner auf eine andere Art und Weise scannt, haben alle 3D-Scanner eine gemeinsame Funktionsweise. In jedem Fall muss ein Medium von einem Emitter[1] ausgesendet werden. Dieses Medium trifft dann das Objekt was gescannt werden soll und prallt daran ab. Dieses abgeprallte Medium wird anschließend von einem Detektor registriert. Der Detektor sammelt dann Informationen über dessen Flugbahn, bzw. über dessen Reflektion. Diese Informationen werden anschließend analysiert. Das Ergebnis der Analyse liefert die Tiefenwerte, bzw. Informationen über die Entfernung eines bestimmten Punktes von dem Emitter, siehe Abbildung 1.

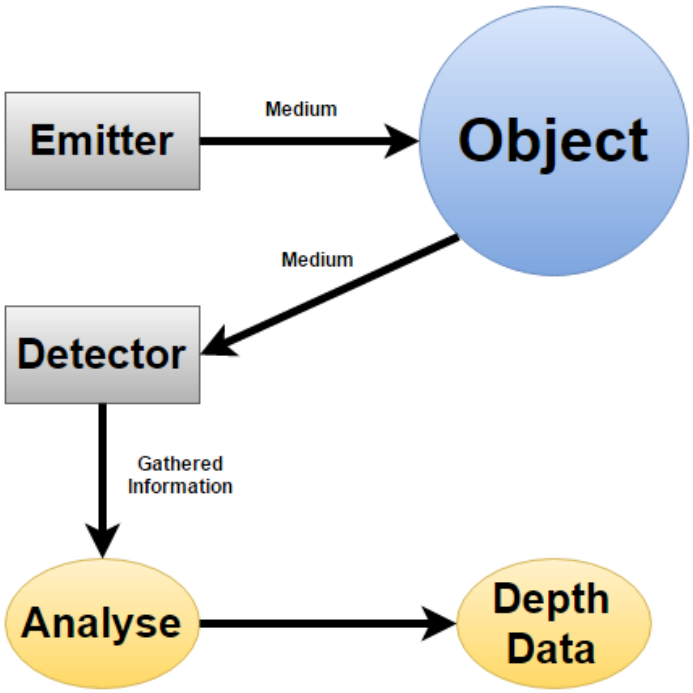

Abbildung 1. Diagramm zur allgemeinen Funktionsweise von 3D-Scannern.

Die Art und Weise wie der Emitter funktioniert, das Medium aussieht, oder die Informationen analysiert werden, hängen wiederrum von dem jeweiligen Scanner ab.

[1] Englisch für Sender.

2.2 Tiefenwertbilder

Alle 3D-Scanner erzeugen auf die eine oder andere Weise Tiefenwerte. Einige Scanner nutzen Tiefenwertbilder um diese Tiefenwerte anzuzeigen. Dabei wird wie mit normalen RGB-Kamera[2] auch, ein Bild aufgenommen. Der Unterschied zu RGB-Bildern liegt jedoch darin, dass die einzelnen Pixel keine Farbenwerte enthalten, sondern Tiefenwerte. Diese Tiefenwerte geben an wie weit sich der Punkt, den das Pixel in der 3D-Umgebung darstellt, von der Kamera entfern ist. Diese Bilder allein sind jedoch in keiner Weise drei Dimensional. Sie dienen lediglich als Speichermöglichkeit um die ermittelten Punkte festzuhalten, siehe Abbildung 2.

Abbildung 2. Tiefenwertbild einer Drachenfigur[3].

Je heller ein Pixel auf dem Tiefenwertbild ist, desto näher muss sich der Punkt, den dieser Pixel repräsentiert an der Kamera befinden. Die Farben sagen dabei jedoch nichts über die Tiefenwerte aus, es geht dabei allein um die Helligkeit. In der Abbildung besitzt das Tiefenwertbild verschiedene Weißtöne bis hin zu Schwarz. Es könnte aber auch genauso gut verschiedene Grüntöne bis hin zu Schwarz enthalten.

2.3 Pointcloud

Nachdem Scannen, muss ein Scanner die ermittelten Daten visualisieren. Da alle 3D-Scanner auf irgendeine Weise Tiefenwerte sammeln, bietet es sich an, diese einfach im drei dimensionalen Raum zu visualisieren. Dies nennt sich dann eine „Pointcloud“, da das Ergebnis aussieht wie eine Wolke aus vielen einzelnen Punkten. Diese einzelnen Punkte werden Vertex[4], bzw. Vertices[5] im Plural, genannt. Je mehr Vertices in der Wolke zu sehen sind, umso höher ist das Modell aufgelöst und somit detailreicher. Abbildung 3. zeigt eine beispielhafte Pointcloud.

[2] RGB steht für Red, Green, Blue. RGB Bilder stellen alle Farben durch unterschiedliche Mischverhältnisse aus Rot, Grün und Blau dar.

[3] Quelle: [CAMO] camo.githubusercontent.com

[4] Englisch für Eckpunkt. So werden Punkte im 3D-Raum genannt.

[5] Plural von Vertex.

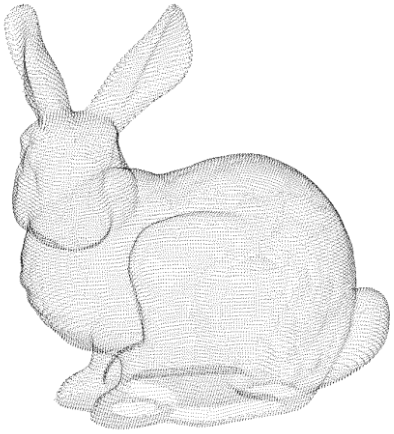

Abbildung 3. Pointcloud des "Stanford Bunnys[6]".[7]

Diese Darstellung ist jedoch kein massives Modell wie man es von einem 3D-Scan erwarten würde. Für die meisten Anwendungen reicht es jedoch aus, da hier bereits vermessungstechnische Informationen wie z.B. der Umfang, die Breite oder die Höhe des Modells bereits vorhanden sind. Falls es jedoch trotzdem nötig ist ein massives Modell darzustellen, weil das Modell beispielsweise in einem 3D-Drucker gedruckt werden soll, kann man dies durch weitere Algorithmen im Nachhinein ergänzen. Dabei muss ermittelt werden welche Vertices eine Fläche im Modell bilden sollen. Dieser Vorgang ist recht kompliziert und wird deshalb an dieser Stelle nicht näher erläutert. Nachdem dies ermittelt wurde, werden die Vertices mit Kanten, bzw. Edges[8], verbunden. Diese Prozedur nennt sich „3D-Reconstruction“. Ein fertiges, solides Model ist in der Abbildung 4 zu sehen.

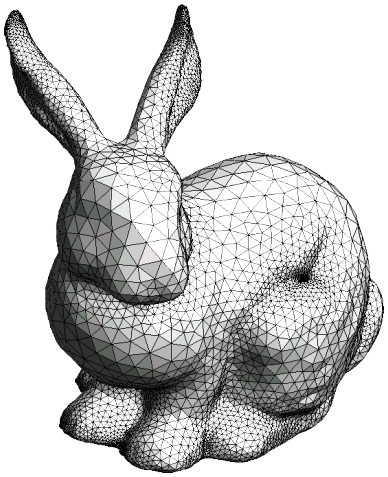

Abbildung 4. Aus der Pointcloud rekonstruiertes Modell.[9]

[6] Gängigstes 3D Testmodel. Wurde 1994 von Marc Levoy und Greg Turk an der Stanford Universität erstellt.
[7] Quelle: http://waldyrious.net/learning-holography/img/stanford-bunny-points.png (abgerufen am 8.10.2015)
[8] Englisch für Kante. So nennt sich die Verbindung von zweier Vertices
[9] Quelle: http://www.cs.mun.ca/~omeruvia/philosophy/images/BunnyWire.gif (abgerufen am 8.10.2015)

2.4 Stand der Technik

Da das 3D-Scannen immer mehr an Bedeutung in der Forschung und in der Industrie gewinnt, gibt es zurzeit viele verschiedene Wege um Objekte drei dimensional einzuscannen. Diese Wege unterscheiden sich sowohl in ihrer Effizient, Qualität als auch in ihren Kosten. Welche dieser vielen Möglichkeiten man Wählt, hängt sehr von dem Einsatzgebiet und der Aufgabe des Scanners ab. Im Folgenden werde ich drei der bewährtesten Methoden vorstellen und miteinander vergleichen um die optimalste Wahl zu treffen. Zusätzlich zur Wahl des Scanners, muss noch ein geeignetes Dateiformat zum Speichern des 3D-Modells gewählt werden.

Der 3D-Scanner der verwendet werden soll, wird in der Medizin Einsatz finden, um den Prozess der Orthesenanfertigung zu optimieren und effizienter zu gestalten. Durch das 3D-Scannen kann somit einfach das Bein des Patienten gescannt werden, und der Großteil der Vermessungen kann dann präzise am 3D-Modell stattfinden.

Wir suchen nach einem Scanner der nicht zu teuer ausfallen sollte, jedoch trotzdem möglichst präzise Scannt. Außerdem sollte das Scannen auch schnell durchgeführt werden, um den Patienten möglichst angenehm zu vermessen.

2.4.1 Stereoskopie

Die einfachste und älteste Form des 3D-Scannens bzw. von 3D-Aufnahmen ist die Stereoskopie. Bei der Stereoskopie werden zwei Kameras verwendet. Diese werden in einer fixen Distanz von einander aufgestellt. Anschließend nehmen dann beide Kameras synchron Aufnahmen von demselben Objekt auf, jedoch von anderen Perspektiven. Dies ist in der Abbildung 5. dargestellt.

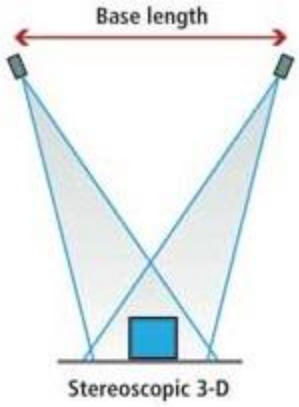

Abbildung 5. Aufbau und Funktionsweise einer Stereoskopiekamera.[10]

Diese Methode lehnt sich an die Funktionsweise des menschlichen Auges an, um Informationen über den drei dimensionalen Raum vor den Kameras zu sammeln. Der Standpunkt eines Objektes im Raum kann durch den Versatz des Objektes auf beiden Bildern ermittelt werden. Je weiter die Distanz des Objektes auf beiden Bildern ist, umso näher muss sich das Objekt vor der Kamera befinden.

[10] Quelle:http://www.vision-systems.com/content/dam/VSD/print-articles/2012/06/leadf5-1206vsd.jpg (abgerufen am 7.10.2015)

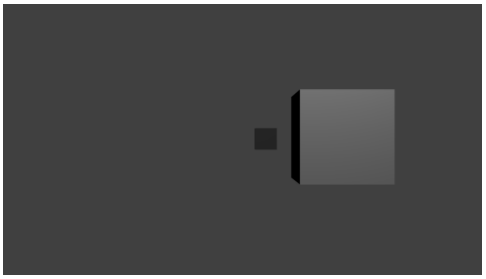

Abbildung 6. Perspektive der linken Kamera.

Abbildung 7. Perspektive der rechten Kamera.

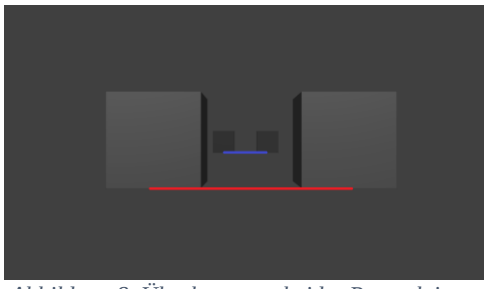

Abbildung 8. Überlagerung beider Perspektiven.

Auf den Bildern wurde eine Szene in einem drei dimensionalem Raum aufgenommen, in dem zwei Quader hintereinander platziert sind. Ein Quader befindet sich recht nahe an den Kameras, während der zweite Quader weiter entfernt ist. Das erste Bild zeigt die Perspektive der linken Kamera, das zweite Bild die der rechten. In dem dritten Bild wurden die ersten beiden Bilder überlagert. Es ist klar zu erkennen, dass der Versatz des vorderen Quaders, dargestellt durch die rote Linie, in dem dritten Bild größer ist, als der des hinteren, dargestellt durch die blaue Linie.

Problematisch ist jedoch die Analyse. Für den Menschen ist es auf den Bildern klar erkennbar welche Objekte sich nahe der Kamera befinden, nicht aber für den Computer oder den Scanner. Somit ist es keine triviale Aufgabe, Tiefenwerte über die Umgebung zu erlangen. Damit Tiefenwerte aus den Bildern ermittelt werden können, müssten jetzt weitere Bildverarbeitungen stattfinden, um Objekte zu detektieren. Die Detektion allein wäre jedoch nicht ausreichend, es müsste auch noch dasselbe Objekt im anderen Bild gefunden werden, damit der Versatz berechnet werden kann. Dies ist jedoch extrem mühselig und ineffizient.

Diese Methode ist weit verbreitet in der Unterhaltungsindustrie, wie z.B. in Filmen oder Videospielen, da hier keine Analyse über die Tiefenwerte benötigt werden. Hier müssen lediglich 3D-Bilder bzw. Aufnahmen visuell dargestellt werden. Dafür bietet die Stereoskopie eine schnelle und günstige Möglichkeit, weil nur zwei Kameras als Material benötigt werden. Eine weitere Analyse der Bilder ist nicht notwendig.

Somit wird deutlich, dass obwohl diese Methode weit verbreitet ist, nicht als 3D-Scanner in Frage kommt, da sie nicht effizient genug arbeitet und eine manuelle Nachbearbeitung erfordert.

2.4.2 Time of Flight

Diese Methode dient, nur zum 3D-Scannen und vermessen von Objekten. Daher funktioniert sie, anders als die Stereoskopie, wie in den Grundlagen des 3D-Scannens beschrieben. Eine Lichtquelle dient hierbei als Emitter, Infrarotlicht als Medium und ein Infrarotlichtsensor als Detektor. Als Analyse-Einheit dient ein Timer. Der Emitter strahlt einen hoch energetischen Infrarotlichtpuls aus, registriert den Zeitpunkt und schickt diesen zum Timer. Das Licht wird am Objekt reflektiert und vom Sensor detektiert, der Sensor registriert wiederum die Ankunftszeit und sendet diese ebenfalls zum Timer. Der Timer ermittelt dann die Differenz der Zeiten und erhält dadurch die Dauer, die das Licht unterwegs war. Mit dieser Zeitangabe und zusammen mit der Lichtgeschwindigkeit, die mit c=3*10^8 m/s Konstant ist, kann der Scanner nun ermitteln wie weit sich der Aufprallpunkt des Lichtstrahls von dem Sensor befinden muss. Die Positionen der Aufprallpunkte können nun in einer Pointcloud drei dimensional visualisiert werden.

Der Aufbau dieser Methode ist in der Abbildung 9 visualisiert.

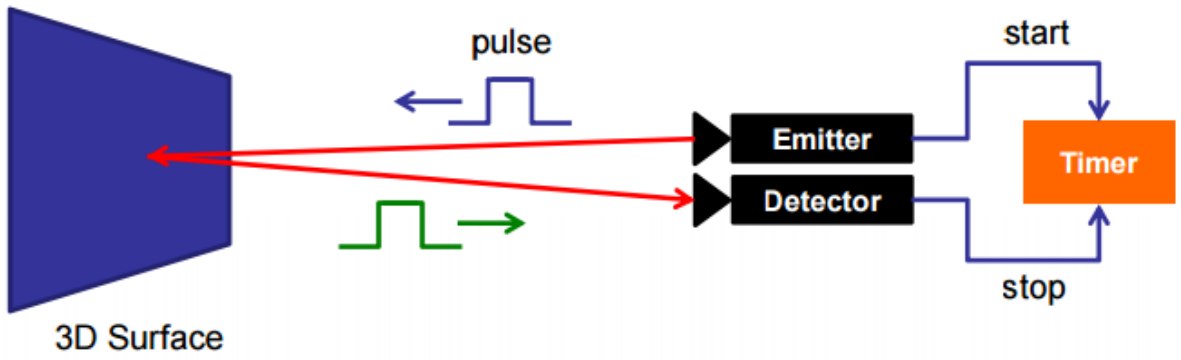

Abbildung 9. Aufbau und Funktionsweise eines Time-of-Flight Scanners.[11]

Diese Methode ist sehr performant, da sie keine manuelle Nachbearbeitung erfordert. Die Tiefenwerte müssen nicht nach einer Aufnahme extrahiert werden, sondern liegen sofort vor. Somit ist ein schneller Scan gewährleistet. Außerdem hat diese Methode den Vorteil, dass sie nur eine Kamera braucht.

Problematisch ist jedoch die Streuung des Lichts, da es nicht immer exakt im selben Winkel reflektiert wird. Somit können Messfehler entstehen. Außerdem kann die Struktur und die Beschaffenheit der Oberfläche des zu scannenden Objektes die Messung beeinflussen. Eine glatte spiegelnde Oberfläche streut das Licht weiter, als eine matte raue Oberfläche. Wiederrum schlucken matte raue Oberflächen mehr von dem Licht als, glatte spiegelnde Oberflächen. Dies hat zur Folge, dass man einen Lichtpuls der stark genug ist aussenden muss, um dem vorzubeugen.

Trotz ihrer Nachteile bietet diese Methode eine sehr gute, und recht präzise Möglichkeit zum 3D-Scannen.

[11] [VINA2011] Time-of-Flight and Kinect Imaging, S.11

2.4.3 Structured Light

Diese Methode verwendet wie die Time-of-Flight Methode, die Eigenschaften des Lichts, jedoch auf eine andere Art und Weise. Der Aufbau ist ebenfalls ähnlich wie bei der Time-of-Flight Methode. Eine Lichtquelle dient hierbei wieder als Emitter. Als Medium kommt hier ebenfalls Infrarotlicht zum Einsatz. Der Detektor ist bei dieser Methode ebenfalls eine Kamera, die Infrarotlicht detektieren kann. Der Unterschied zur Time-of-Flight Methode besteht darin, dass die Tiefenwertbestimmung nicht über die Flugdauer des Lichts bestimmt wird.

Der Emitter strahlt ein bekanntes Lichtmuster aus. Dieses Muster wird auf das zu scannende Objekt projiziert. Der Detektor macht anschließend eine Aufnahme von dem Objekt mit dem Lichtmuster. Ein beispielhaftes Lichtmuster ist in der Abbildung 10. zu sehen.

Abbildung 10. Light Speckle Muster.[12]

Diese Aufnahme wird nun auf verschiedene Eigenschaften des Lichtmusters hin analysiert. Diese Eigenschaften liefern dann die Entfernung eines Punktes auf der Aufnahme, zur Kamera. In dem Beispiel auf der Abbildung 9 wurde ein Lichtpunktmuster verwendet.

Das Muster auf der Aufnahme wird anschließend analysiert und auf Veränderungen geprüft. Verzerrungen, Helligkeit und Größe der Punkte in dem Muster liefern Informationen zur Entfernung zur Kamera.

Je kleiner ein Punkt in dem Muster nach der Projektion ist, desto weiter muss er von der Kamera entfernt sein. Das gleiche gilt ebenfalls für die Helligkeit, je dunkler der Punkt nach der Projektion ist, umso weiter muss sich dieser von der Kamera befinden. Zusammen mit den Veränderungen der benachbarten Punkte, können die Lücken zwischen den Punkten ebenfalls bestimmt werden.

Diese Methode ist besonders zum 3D-Scannen von Objekten geeignet, bei denen es nicht auf Details ankommt. Je detailreicher ein Objekt ist, umso feiner sind die Verzerrungen der Punkte, wodurch diese schwerer zu detektieren sind. Sehr feine Details können daher nur schwer bis gar nicht vermessen werden.

Außerdem ist diese Methode nicht von hoch präzisen Messwerkzeugen abhängig, wie z.B. die Time-of-Flight Methode, bei der die Flugdauer der Lichtstrahlen so präzise wie möglich gemessen werden muss. Dies hat zur Folge, dass diese Art von Scannern günstiger sind als die Time-of-Flight Scanner. Wiederrum bedeutet das auch, dass diese Methode langsamer ist als die Time-of-Flight Methode, weil hier die Aufnahmen analysiert werden müssen.

[12] Quelle: http://gmv.cast.uark.edu/wp-content/uploads/2012/07/KinectIR.png (abgerufen am 8.10.2015)

2.5.1 Polygon File Format

Es gibt viele verschiedene Dateiformate um 3D-Scans bzw. Pointclouds zu speichern. Als gängigstes Format zum Speichern von Pointlcouds gilt jedoch das ply-Format, da dieses Format speziell für den Einsatz mit 3D-Scannern entwickelt wurde. Somit bietet dieses Format, im Gegensatz zu anderen Formaten, die komfortabelste Möglichkeit Pointclouds zu speichern, siehe Abbildung 11.

```
ply
format ascii 1.0           { ascii/binary, format version number }
comment made by anonymous  { comments are keyword specified }
comment this file is a cube
element vertex 8           { define "vertex" element, 8 in file }
property float32 x         { vertex contains float "x" coordinate }
property float32 y         { y coordinate is also a vertex property }
property float32 z         { z coordinate, too }
element face 6             { there are 6 "face" elements in the file }
property list uint8 int32 vertex_index
                           { "vertex_indices" is a list of ints }
end_header                 { delimits the end of the header }
0 0 0                      { start of vertex list }
0 0 1
0 1 1
0 1 0
1 0 0
1 0 1
1 1 1
1 1 0
4 0 1 2 3                  { start of face list }
4 7 6 5 4
4 0 4 5 1
4 1 5 6 2
4 2 6 7 3
4 3 7 4 0
```

Abbildung 11. Beispiel einer ply-Datei.[13]

Eine ply Datei besteht aus einem Header und einem Hauptteil. Der Header enthält Grundinformationen des Modells. Zunächst muss im Header festgelegt werden, dass es sich um eine ply Datei handelt. Dies geschieht in der ersten Zeile einfach durch „ply“. In der nächsten Zeile wird durch den Befehl „format“ bestimmt in welchem Zeichensatz die Datei codiert werden soll. In dem Beispiel wird die ascii[14] codieren verwendet. Kommentare können durch „comment“ eingefügt werden. Als nächstes muss die Anzahl der Vertices die für die Pointcloud verwendet werden müssen, folgen. Dies geschieht durch den Aufruf „element vertex“ gefolgt von der Anzahl der Vertices. In dem Beispiel sind es acht. Die nächsten drei Zeilen geben an, in welchem Datentyp die x, y und z Positionen der Vertices gespeichert werden sollen, was durch den Befehl „property [Datentyp] [Koordinate]“ erfolgt. Im Beispiel haben alle drei Koordinaten den Datentyp float32. Die nächste Angabe ist für eine Pointcloud irrelevant, da hier die Anzahl der Faces bestimmt wird. Wenn jedoch ein massives Modell gewünscht ist, muss diese Information vorhanden sein. Auf die Anzahl der Faces folgt dann eine Liste in denen die Indizes der Vertices gespeichert werden, die zu einer Face gehören. In dem Beispiel ist das in der zehnten Zeile zu sehen. Hat man alle Informationen die für das Modell benötigt werden angegeben, kann der Header durch den Befehl „end_header“ geschlossen werden.

Nun beginnt der Hauptteil. In diesen werden lediglich die Koordinaten aller Vertices zeilenweise eingetragen. Zu beachten ist jedoch, dass genauso viele Vertices verwendet werden wie vorher angegeben wurden. In dem Beispiel wurden acht Vertices im Header angegeben. Also müssen im

[13] Quelle: http://people.sc.fsu.edu/~jburkardt/data/ply/ply.html (abgerufen am 9.10.2015)

[14] American Standard Code for Information Interchange. Eine der gängigsten Zeichencodierungen.

Hauptteil auch acht Vertices folgen. Falls mehr oder weniger als acht Vertices angegeben worden wären, würde es zu einem Fehler führen, wenn die Datei angezeigt würde.

Hat man alle Vertices eingefügt, müssen nun sofern im Header angegeben, die Faces folgen. Die Faces werden direkt im Anschluss an die Vertices drangehängt. Eine Face beginnt mit der Angabe wie viele Vertices die jeweilige Face haben soll. Bei dreieckigen Flächen würde man hier also eine drei angeben, und bei viereckigen eine vier. Im Beispiel werden Vierecke als Faces[15] benutzt, also beginnen alle Faces mit einer vier. Danach folgen die Indizes der Vertices, die diese Vierecke formen. Die Indizes sind dabei die Positionen der Vertices in der sie sich in der vorherigen Liste befinden. Im Beispiel hätte also der Vertex mit der Position 0 0 0 den Index 0, während der Vertex mit der Position 1 1 0 den Index 7 hätte. Dabei ist zu beachten, dass die Indizierung wie häufig in der Programmierung, bei null beginnt.

2.6 Bereits bestehende Scansysteme

Es existiert bereits eine Vielzahl von verschiedenen Scansystemen. Die meisten Systeme sind jedoch große sperrige, festinstallierte Vorrichtungen. Diese Systeme würden sich also weniger für den Einsatz in einem Sanitätshaus oder dem privaten Gebrauch eignen.

Daher werden in diesem Kapitel zwei Beispielsysteme vorgestellt, die portabel bzw. handlich sind.

2.6.1 Artec Eva

Die „Artec Eva“ ist ein „Structured Light“ hand-held Scanner, daher sehr kompakt und mobil. Ursprünglich kommt dieser Scanner aus dem medizinischen Bereich, was sich auch in der Scangenauigkeit von 0,1 mm wiederspiegelt. Als Lichtmuster kommt hier ein Streifenmuster zum Einsatz. Dabei werden Lichtstreifen auf das zu scannende Objekt geworfen und deren Verformung analysiert[16]. Um 3D-Modelle erstellen zu können, muss dieser Scanner jedoch mit einem Rechner verbunden sein, was die Mobilität leicht einschränkt.

Um eine 3D-Aufnahme zu machen, muss die „Artec Eva“ um das zu scannende Objekt bewegt werden, bis die „Artec Eva“ das Objekt aus genug Perspektiven erfassen konnte, ähnlich wie eine Aufnahme mit einer Videokamera. Alternativ kann das zu scannende Objekt um sich selbst gedreht werden, während die „Artec Eva“ das Objekt aufnimmt.

Die Qualitäten des Scanners, spiegeln sich auch in dessen Preis von ca 13.000€ wieder.

[15] Englisch für Fläche. Faces bilden sich aus mindestens drei Edges.

[16] Siehe 2.4.3. Structured Light

Abbildung 12. Es wird ein 3D-Modell von Präsident Obama angefertigt.[17]

In der Abbildung 12. ist gut zu erkennen, wie Aufnahmen mit der „Artec Eva“ entstehen. Die Kameras werden so lange um das Objekt bewegt, bis ein fertiges Modell generiert wurde.

2.6.2 Fuel 3D

Dieser Scanner kommt auch Ursprünglich aus dem medizinischen Bereich. Genau wie die „Artec Eva“, ist auch dieser Scanner ein „Structured Light“ hand-held Scanner. Somit bietet auch dieser Scanner große Mobilität. Das verwendete Muster entspricht ebenfalls dem Streifenlichtmuster der „Artec Eva“. In der folgenden Abbildung ist der sehr kompakte „Fuel 3D“ Scanner zu sehen.

Abbildung 13. Der "Fuel 3D" Scanner.[18]

Der Unterschied zwischen den beiden Systemen, besteht in der Weise wie die Aufnahmen getätigt werden. Während bei der „Artec Eva“ wie mit einer Videokamera eine durchgehende Aufnahme

[17] Quelle: http://www.sariki.es/wp-content/uploads/2015/03/retrato-3D-obama-con-escáner-artec-EVA-e1427103221214.png (abgerufen am 18.10.2015)
[18] Quelle: http://www.minifablab.nl/wp-content/uploads/2013/04/scanify.png (abgerufen am 18.10.2015)

gemacht wird, werden mit der „Fuel 3D“ mehrere Einzelaufnahmen gemacht. Diese Einzelaufnahmen werden dann zu einem 3D-Modell zusammengefügt. Abgesehen davon, muss die Kamera bei der Aufnahme kalibriert werden. Dies geschieht durch eine Schablone, die neben das Objekt gelegt wird. Diese Schablone wird dann auf den Aufnahmen von der „Fuel 3D“ analysiert. Die nächste Abbildung zeigt die Anwendung der Schablone.

Abbildung 14. Die Schablone wurde in die Szene gelegt, um die Kamera zu kalibrieren.[19]

Da vom Hersteller Empfohlen wird, keine komplizierten Strukturen aufzunehmen, kann davon ausgegangen werden, dass die Messgenauigkeit geringer ist, als bei der „Artec Eva“. Eine genaue Angabe der Messgenauigkeit konnte ich leider nicht auf der Website des Herstellers finden.

Der Preis liegt mit 1200 bis 1500€[20] weit unter dem Preis der „Artec Eva“.

[19] [NITZ2014] 3D-Druck, S.155

[20] Quelle: http://www.artec3d.com/de/hardware/artec-eva/ (abgerufen am 18.10.2015)

3. Material und Methode

3.1. ASUS Xtion PRO Live

Die „ASUS Xtion PRO Live“ ist die Kamera, die im Rahmen dieser Arbeit zur Verfügung gestellt wurde. Beeinflusst wurde sie durch die erste Generation von der Microsoft Kinect Kamera. Beide Kameras funktionieren dabei nahezu identisch, da beide auf Primesense[21] Technologie beruhen. Diese Kameras werden Hauptsächlich in multimedialen Anwendungen genutzt. Der Fokus liegt dabei auf Motion-Sensing[22] um Bewegungen des Körpers oder Gesten der Hand zu erkennen, um damit Anwendungen zu steuern.

Abbildung 15. Microsoft Kinect[23]

Abbildung 16. Asus Xtion PRO Live[24]

Auf den Abbildungen 15. und 16. ist klar zu erkennen, dass beide Kameras gleich aufgebaut sind. Von links nach rechts sieht man bei beiden Kameras den Infrarotlicht-Projektor, eine RGB Kamera und den Infrarotlicht-Sensor.

Glücklicherweise benutzen diese Art von Kameras, Tiefenwerte um die Umgebung zu analysieren und die Bewegungen der Benutzer zu detektieren. Dadurch kann man diese Kameras auch als 3D-Scanner nutzen.

Die Tiefenwerte werden hierbei durch das Structured Light Verfahren bestimmt. Der Infrarotlicht-Projektor sendet hierbei ein bekanntes Lichtmuster aus und der Infrarotlicht-Sensor detektiert die Projektion.

Wie bereits beschrieben sind diese Art von Kameras kostengünstig und somit der breiten Masse zugänglich, da sie auch hauptsächlich in der Unterhaltungselektronik zum Einsatz kommen. Deswegen eignet sich diese Kamera gut für einen 3D-Scanner, der in der Orthesenanpassung benutzt werden soll. Sanitätshäuser die diesen Scanner nutzen wollen, müssen daher keine teure Investition in spezialisierte Hardware tätigen.

[21] Ehemals Israelische Firma, spezialisiert auf 3D-Erkennung. Wurde am 24.11.2013 von Apple aufgekauft.
[22] Englisch für Bewegungserkennung
[23] Quelle: http://www.codeproject.com/KB/audio-video/317974/kinect.JPG (abgerufen am 9.19.2015)
[24] Quelle: https://www.asus.com/3D-Sensor/Xtion_PRO_LIVE/ (abgerufen am 9.10.2015)

Folgende Tabelle zeigt die Eigenschaften der ASUS Kamera als 3D-Scanner, im Vergleich zu den zwei bereits vorgestellten Scansystemen „Artec Eva“ und „Fuel 3D“.

	Artec Eva	Fuel 3D	Asus Xtion Pro Live
Genauigkeit	Sehr gut	Gut bis Sehr gut	Befriedigend
Geschwindigkeit	Befriedigend	Befriedigend	Befriedigend
Mobilität	Sehr gut	Sehr gut	Befriedigend
Preis	ca. 13000€	ca. 1500€	ca. 150€

Tabelle 1. Die drei vorgestellten Systeme im Vergleich

In der Tabelle ist zu erkennen, dass die „Artec Eva“ und die „Fuel 3D“ sehr gute Leistungen liefern, jedoch vergleichsweise Teuer sind. Die „Asus Xtion“ Kamera hingegen liefert brauchbare Ergebnisse und ist dabei mit 150€ für den Privatgebrauch angemessen.

Alle Systeme sind theoretisch auf gleichem Niveau mobil, d.h. alle Geräte können frei bewegt werden. Die Asus Kamera hat jedoch den Nachteil, dass es nicht als hand-held Scanner konzipiert ist, sondern als eine stationäre Kamera. Somit ist es komfortabler die „Artec Eva“ und die „Fuel 3D“ zu bewegen.

Die Nachteile der Asus Kamera werden jedoch eindeutig von dem Preis und der Leistung die sie letztendlich erbringen muss, wettgemacht.

Ein weiterer Vorteil dieser Primesense Kameras ist die vorhandene Dokumentation. Wegen der hohen Popularität des Microsoft Kinect und dessen Kompatibilität mit Windows, konnten viele Benutzer des Kinect eigene Anwendungen schreiben. Um die Primesense Kameras zu fördern, wurde die Bibliothek OpenNI von Primesense entwickelt. Durch diese Biibliothek können nun alle Benutzer, die Programmierkenntnisse besitzen, die Kamera in ihren Programmen ansteuern.

3.2 Java

Es gibt eine Vielzahl verschiedener Programmiersprachen, daher ist es wichtig sich für die Programmiersprache zu entscheiden, die sich am besten für die Problemstellung eignet. Programmiersprachen lassen sich grundlegend in zwei verschiedene Sprachgruppen unterteilen, prozedurale und objektorientierte Sprachen.

Prozedurale Sprachen sind maschinennahe Sprachen. Sie sind dadurch gekennzeichnet, dass Programme aus einer Vielzahl von Befehlen bestehen. Diese Befehle werden Zeilenweise aneinandergereiht und ausgeführt. Zur Verbesserung der Übersicht sowie um Redundanzen im Code zu vermeiden, kommen Prozeduren zum Einsatz. Eine Prozedur stellt ein Paket aus Befehlen dar. Dieses Paket wird mit einem Namen gekennzeichnet und kann an späteren Stellen im Code aufgerufen werden. Beim Aufruf einer Prozedur werden alle Befehle innerhalb dieses Pakets ausgeführt. Außerdem lösen Programme die durch prozedurale Sprachen erzeugt werden, nur einzelne bzw. simplere Probleme, wie beispielsweise mathematische Berechnungen. Zusätzlich sind diese Programme meist nicht interaktiv, bzw. erwarten keine, oder nur wenige Eingaben vom Benutzer zur Programmlaufzeit.
Zu den prozeduralen Programmiersprachen zählen C, Pascal, Basic sowie Assembler. Diese Art von Sprachen bieten den Vorteil, dass sie für gewöhnlich schneller sind als objektorientierte Sprachen, da sie maschinennäher arbeiten. Somit wird der Code schneller in Maschinensprache umgewandelt und vom Rechner ausgeführt. Dadurch sind prozedurale Sprachen für Programme geeignet, die für Berechnungen konzipiert sind und weniger für Anwendungen.

Objektorientierte Sprachen legen einen großen Wert auf Abstraktion. Programme in der OOP[25] besteht aus vielen Einzelteilen. Diese Einzelteile nennen sich Objekte. Der Code der Software wird, nach Aufgabenbereich getrennt und in einzelnen Objekten ausgeführt. Somit übernimmt jedes Objekt die für sie bestimmte Aufgabe. Dies hat zur Folge, dass anders als bei prozeduralen Sprachen, der Code in der OOP nicht strikt von oben nach unten durchlaufen wird. Es wird eher im Code umhergesprungen. Die Zusammenarbeit und Vernetzung der Objekte sorgen für die Funktion der Software. Diese Abstraktion führt dazu, dass der Code übersichtlicher wird und Redundanzen gespart werden. Somit ist es für den Entwickler komfortabel die Software zu schreiben. Ein weiterer Vorteil der OOP ist, dass Programme leichter im Team bearbeitet werden können, da jederzeit Objekte ausgetauscht oder neu eingefügt werden können, ohne dass der gesamte Code geändert werden muss. Diese Art von Programmierung eignet sich besonders gut für Anwendungen, da hier, durch die Benutzerinteraktion und -oberfläche komplexe Codestrukturen entstehen. Die Abstraktion in Objekte hat jedoch den Nachteil, dass der Code stärker interpretiert werden muss, bzw. länger braucht um in Maschinensprache übersetzt zu werden. Dies hat lange Lade-, und Ausführzeiten zur Folge. Somit eignet sich diese Sprache nicht für hochkomplizierte Berechnungen, sowie Programme in denen es auf die Laufzeit ankommt.

Die Entscheidung für eine Programmiersprache, in dem die Scananwendung geschrieben werden soll, fällt eindeutig auf eine objektorientierte Programmiersprache, da die Scananwendung eine Benutzeroberfläche, sowie Interaktionen bieten muss. Problematisch ist dabei jedoch die Laufzeit, da der Nutzer nicht mehrere Minuten auf Ergebnisse warten sollte. Glücklicherweise sind die Datensätze die bei einem Scan anfallen, bei weitem nicht so groß als das eine prozedurale Sprache nötig wäre.

Die gängigsten objektorientierten Sprachen sind dabei Java und C#. Diese Sprachen unterscheiden sich kaum in ihrer Syntax und sind somit nahezu identisch. Der Vorteil von Java ist jedoch, dass die Sprache auf fast allen multimedialen Geräten unterstützt wird. Ein weiterer Vorteil von Java gegenüber C# ist, dass Java auf Android Geräten unterstützt wird. Da der Scanner später auch auf Smartphones und Tablets laufen soll und diese, abgesehen von Apple Produkten, nahezu alle Android Betriebssystem haben, fällt die Entscheidung hierbei auf Java.

3.3. Entwicklungsumgebung

Nachdem man sich für eine Sprache entschieden hat, muss nun eine Entwicklungsumgebung ausgewählt werden. Theoretisch könnten alle Computerprogramme in beliebigen Texteditoren geschrieben werden. Das Problem dabei ist jedoch, dass das Programm in Maschinensprache übersetzt werden muss, was normale Editoren nicht können. Deswegen ist eine Entwicklungsumgebung nötig. Diese Entwicklungsumgebungen bieten die Möglichkeit den Code in Maschinensprache umzuwandeln und auszuführen. Außerdem sind Entwicklungsumgebungen speziell für das Programmieren ausgelegt und bieten dem Entwickler viele hilfreiche Tools[26], wie zum Beispiel das automatische Vervollständigen von bestimmten Codestrukturen, um dem Entwickler redundante Schreibarbeit abzunehmen. Entwicklungsumgebungen werden auch kurz IDE[27] genannt.

Die zwei größten IDEs die für das Programmieren in Java genutzt werden sind Eclipse und Netbeans. Für welche man sich letztendlich entscheidet, ist reine Geschmackssache, da beide dieselben Grundfunktionalitäten bieten.

In diesem Fall werde ich mich also für die IDE Eclipse entscheiden, da ich durch mein Studium die meiste Erfahrung mit dieser IDE sammeln konnte.

[25] Abkürzung für „Object Oriented Programming“
[26] Englisch für Werkzeuge
[27] Abkürzung für „Integrated Development Environment“

3.4 Java Grundkonzepte und Begriffe

Im Folgenden werden die wichtigsten Grundkonzepte sowie Schlüsselbegriffe aus Java vorgestellt, um dem Leser eine Grundlage für das Verstehen von Java-Code zu verschaffen. Dabei begrenze ich mich auf die von mir im Projekt verwendeten Grundkonzepte und Begriffe.

Klassen

Um gleichartige Objekte zusammenzufassen und Redundanzen zu vermeiden, werden Klassen angelegt. Diese Klassen sind eine Abstraktionsebene höher als Objekte und dienen als Schablone für verschiedene Instanzen von Objekten.

Äpfel, Orangen und Bananen sind Früchte. Dabei wäre „Frucht“ die Klasse, und Äpfel, Orangen und Bananen, die Objekte die aus der Klasse „Frucht“ erzeugt werden. Es wäre verschwenderisch für Äpfel, Orangen und Bananen unterschiedlichen Code zu schreiben, da alle im Grunde Früchte sind und sich nur in ihren Attributen unterscheiden. Die folgende Abbildung zeigt die Syntax einer leeren Klasse in Java.

```
public class Frucht {

}
```

Abbildung 17. Die leere Klasse "Frucht".

Public und Private

Um die Struktur der OOP zu bewahren, kann man die Schlüsselwörter „private“ und „public“ verwenden. Klassen, Methoden und Attribute können durch diese Schlüsselwörter gekennzeichnet werden. Das Schlüsselwort „private“ sorgt dabei dafür, dass das gekennzeichnete Objekt, nicht außerhalb der Klasse in der es definiert wurde, sichtbar bzw. ansprechbar ist. Das Schlüsselwort „public“ hingegen, sorgt für das Gegenteil. So gekennzeichnete Objekte sind im ganzen Projekt sichtbar. Um für eine gute Struktur zu sorgen, sollte sich der Entwickler darum bemühen, diese Schlüsselwörter sinnvoll zu nutzen. Objekte, die nicht für das gesamte Projekt von Relevanz sind oder Objekte die nicht unkontrolliert verändert werden dürfen, sollten mit „private“ gekennzeichnet sein. Dies nennt sich „Geheimnisprinzip“.

Instanziieren

Der Prozess, in dem ein konkretes Objekt aus einer Klasse erstellt wird. Wenn man einen Apfel der Klasse „Frucht“ erstellt, so instanziiert man.

Variablen und Datentypen

Variablen dienen als Behälter für Informationen und Daten. Diese Variablen müssen immer mit Datentypen gekennzeichnet werden. Nach der Deklaration, kann eine gekennzeichnete Variable keine anderen Werte, die einen anderen Datentyp als die Variable haben, mehr aufnehmen. Die Syntax sieht dabei wie folgt aus:

DATEITYP VARIABLENNAME;

Eine Deklaration einer Variablen die Objekte der Klasse „Frucht“ beinhalten kann, würde wie folgt aussehen:

Frucht eineFrucht;

Datentypen

In Java werden viele verschiedene Datentypen, mit denen Objekte deklariert werden können, unterstützt. Hier werde ich jedoch nur die für das Projekt relevanten Datentypen aufzählen und kurz erläutern.

- void: leerer/kein Datentyp.
- string: Zeichenketten Bsp: string name;
- int: Ganzzahlen Bsp: int zahl
- double: Gleitkommazahlen Bsp: double pi = 3.14156;
- float: wie double, nimmt aber mehr Dezimalstellen als double
- short: wie double, nimmt aber weniger Dezimalstellen als double

Instanz

Ein konkretes Objekt einer Klasse. Klassen an sich können nicht aktiv benutzt werden. Diese stellen nur eine Abstraktion der Objekte die aus der Klasse erzeugt werden können dar. Es muss also erst ein Objekt der Klasse erzeugt werden. Das Objekt kann anschließend benutzt werden. Durch das Schlüsselwort „new“ gefolgt von dem Klassennamen mit einem Klammerpaar, kann eine Instanz erzeugt werden. Der Variable „eineFrucht“, in der Abbildung 18, wird eine Instanz der Klasse „Frucht“ zugewiesen.

```
public static void main(String args[])
{
    Frucht eineFrucht =new Frucht();
}
```

Abbildung 18. Eine Instanz der Klasse Frucht wird erzeugt.

Attribute

Attribute stellen die Variablen und Konstanten von Klassen dar.

Die Attribute von der Klasse „Frucht“ könnten beispielsweise „Farbe“, „Form“, „Größe“ und „Reifezeit“ sein. Abbildung 19. zeigt die Deklaration der Attribute der Klasse Frucht.

```
public class Frucht {

    String farbe;
    int groesse;
    String form;
}
```

Abbildung 19. Die Klasse Frucht wurde um drei Attribute erweitert.

Funktionen / Methoden

Methoden einer Klasse führen bestimmte Prozeduren aus. Der Unterschied zwischen Funktionen und Methoden liegen dabei in ihrem Rückgabewert. Funktionen geben etwas zurück, während Methoden lediglich etwas ausführen. Trotzdem müssen alle Methoden genau wie die Funktionen einen Rückgabewert besitzen. Die Syntax sieht dabei wie folgt aus:

[public oder private] DATENTYP METHODNNAME();

Die Klasse „Frucht“ könnte die Methode wachse() haben. Bei Aufruf dieser Methode könnte die Methode dafür sorgen, dass das Attribut Größe um ein Zentimeter zu inkrementieren. Eine Funktion gibGroesse() könnte dabei abrufen wie groß die Frucht geworden ist. Dafür muss die auszugebende Variable in der Funktion mit dem Schlüsselwort „return“ gekennzeichnet werden. Beide Beispiele sind auf der Abbildung 20 zu sehen.

```
public class Frucht {

    String farbe;
    int groesse;
    String form;

    public void wachse()
    {
        groesse=groesse+1;
    }

    public int gibGroesse()
    {
        return groesse;
    }
}
```

Abbildung 20. Die Klasse Frucht wurde um zwei Methoden erweitert.

Der Aufruf der Methoden einer Klasse erfolgt durch die instanziierten Objekte. Siehe Abbildung 21.

```
public static void main(String args[])
{
    Frucht eineFrucht =new Frucht();
    eineFrucht.wachse();
}
```

Abbildung 21. Aufruf einer Methode.

Wichtig ist zu wissen, dass wenn die Methoden nicht durch konkrete Instanzen aufgerufen werden, der Code in den Methoden nicht ausgeführt wird.

Parameter

Es ist auch möglich Funktionen und Methoden Parameter zu geben. Über diese Parameter können beim Aufruf der Methoden Werte übergeben werden. Die Methoden können dann in ihren Prozeduren auf die übergebenen Werte zurückgreifen. Diese Parameter werden wie normale Variablen in das Klammerpaar der Methode geschrieben.

Der Methode „wachse()“ könnte der Parameter „zeitInTagen“ übergeben werden. Die Methode „wachse()“ würde dann zu „wachse(int zeitInTagen)“ werden. Diese Methode würde dann die Frucht anhand der übergebenen Zeit in Tagen wachsen lassen. Die Erweiterung der „wachse()“ Methode ist in der Abbildung 22 zu sehen.

```
public class Frucht {

    String farbe;
    int groesse;
    String form;
    int wachstumsFaktor;

    public void wachse(int zeitInTagen)
    {
        groesse=groesse+zeitInTagen*wachstumsFaktor;
    }

    public int gibGroesse()
    {
        return groesse;
    }
}
```

Abbildung 22. Die Methode "wachse" wurde parametrisiert.

Parametrisierte Methoden werden genauso aufgerufen wie normale Methoden, mit dem Unterschied, dass ein Parameterwert erwartet wird. Dieser Wert wird einfach in dem Klammerpaar im Anschluss des Aufrufes übergeben, wie auf der Abbildung 23. zu sehen.

```
public static void main(String args[])
{
    Frucht eineFrucht =new Frucht();
    eineFrucht.wachse(5);
}
```

Abbildung 23. Aufruf einer parametrisierten Methode.

Konstruktor

Der Konstruktor einer Klasse ist die Methode, die eine Instanz der Klasse erzeugt. Falls nicht explizit ein Konstruktor erstellt wird, wird implizit ein Standard Konstruktor erstellt. Will man bestimmte Aktionen während der Instanziierung ausführen, so kann ein eigener Konstruktor erstellt werden. Diese Methode lässt sich ebenfalls, wie normale Methoden, parametrisieren. Dadurch können beispielsweise, die Attribute einer Instanz schon bei der Instanziierung manipuliert werden. Zu beachten beim Erstellen des Konstruktors ist, dass der Name des Konstruktors exakt dem Namen der Klasse entsprechen muss. Außerdem darf der Konstruktor kein Rückgabetyp aufweisen. In der folgenden Abbildung ist ein parametrisierter Konstruktor zu sehen.

```
public class Frucht {

    private String name;

    public Frucht(String uebergebenerName)
    {
        name=uebergebenerName;
    }
}
```

Abbildung 24. Ein Konstruktor mit einem Parameter.

Der parametrisierte Konstruktor wird genauso aufgerufen wie der Standard Konstruktor im Abschnitt „Instanziieren“. Der einzige Unterschied ist, dass ein Wert für den Parameter angegeben werden muss. Dies wird in der nächsten Abbildung deutlich.

```
public static void main(String args[])
{
    Frucht banane= new Frucht("Banane");
}
```

Abbildung 25. Aufruf eines Konstruktors mit einem Parameter.

ArrayList

Die ArrayList ist eine von Java vorgegebene Datenstruktur. In ihr können Objekte in Form einer Liste gespeichert werden. Neue Objekte werden dabei als letztes Objekt in die Liste eingefügt. Außerdem bietet die Datenstruktur Methoden an um die Länge der Liste zu ermitteln oder Elemente aus der Liste zu löschen. Wird ein Element aus einer Position gelöscht, so wird die Liste an der Stelle von der das Element gelöscht wurde, wieder zusammengefügt. D.h. es können beliebige Elemente aus der Liste gelöscht werden, ohne dass die Datenstruktur zerstört wird. Die ArrayList ist dabei eine generische Datenstruktur, d.h. die Struktur kann beliebige Objekte aufnehmen, sie ist also nicht von vorneherein an eine bestimmte Klasse gebunden. Dabei muss jedoch darauf geachtet werden, dass wenn die ArrayList für eine Klasse bestimmt wurde, müssen alle eingefügten Objekte von derselben Klasse sein. Man bestimmt eine ArrayList für eine Klasse in dem man die folgende Syntax für die Instanziierung einer ArrayList nutzt:

new ArrayList<KLASSENNAME>();

In den Spitzenklammern muss der Klassenname angegeben werden, zu dem die einzufügenden Objekte gehören.

In der Folgenden Abbildung ist ein kurzes Beispiel einer ArrayList in einem Programm zu sehen.

```
public static void main(String args[])
{
    Frucht banane= new Frucht("Banane");
    Frucht apfel= new Frucht("Apfel");
    Frucht birne= new Frucht("Birne");

    ArrayList<Frucht> listeAusFruechten=new ArrayList<Frucht>();

    listeAusFruechten.add(banane);
    listeAusFruechten.add(apfel);
    listeAusFruechten.add(birne);

    //Ausgabe 3
    listeAusFruechten.size();

    //Das als erstes eingefügte Objekt wird
    //in "eineFrucht"gespeichert
    Frucht eineFrucht=listeAusFruechten.get(0);

    //Ausgabe "Banane"
    eineFrucht.getName();
}
```

Abbildung 26. Beispielhafte Anwendung einer ArrayList.

Array

Ein Array, auch Feld genannt, ist eine Datenstruktur, die ähnlich wie die ArrayList, Objekte speichert. Anders als die Liste jedoch, ist das Array weniger flexibel. Obwohl diese Struktur weniger flexibel ist als die Liste, bietet diese Struktur einige wichtige Vorteile. Ein Vorteil ist die Geschwindigkeit. Da das Array sich nicht um die Reihenfolge und Ordnung von Objekten kümmern muss, funktioniert das Löschen und das Einfügen wesentlich schneller. Außerdem nimmt das Array weniger Arbeitsspeicher ein, da die Größe eines Arrays im Gegensatz zur ArrayList, nicht variabel ist. Wenn ein Array mit der Größe drei definiert wurde, so muss diese Größe über den Lebenszeitraum des Arrays hin, immer drei bleiben. Wenn ein Objekt aus dem Array gelöscht wird, so bleibt dieses Feld dann leer. Diese Lücke wird also nicht wie bei der ArrayList automatisch geschlossen, sondern bleibt leer, bis das Feld wieder befüllt wird. Es ist also Vorsicht geboten, wenn man Daten aus einem Array löscht und später wieder auf das Feld zugreift.

Das Array ist besonders dann nützlich, wenn man Wert auf Performanz legt oder aber die zu speichernde Objektmenge besonders klein ist. Wenn man beispielsweise weiß, dass man durchgehend ein Vektor mit drei Einträgen braucht, so würde sich ein Array mit der Größe drei eignen. Eine ArrayList wäre in diesem Fall reine Speicherverschwendung, da die Vorzüge einer ArrayList bei einer so kleinen Datenmenge nicht ins Gewicht fällt.

Der Zugriff auf bestimmte Felder im Array folgt über Indizierung. Um ein Array zu definieren verwendet man folgende Syntax:

KLASSE[] einFeld = new KLASSE[ARRAYGRÖSSE];

Wenn ein doppeltes Array gewünscht ist, beispielsweise um Tabellen oder Matrizen darzustellen, so kann man einfach statt „[]“ ein doppeltes Klammerpaar „[][]“ nutzen.

In der folgenden Abbildung ist eine Beispielhafte Anwendung eines Arrays dargestellt.

```
public static void main(String args[])
{
    //Packung Früchte die im Supermarkt angeboten werden.
    //Diese Packungen haben immer 5 Früchte.
    Frucht[] einePackungFruechte=new Frucht[5];

    //Die Packung wird mit fünf Äpfeln gefüllt
    einePackungFruechte[0]=new Frucht("Apfel");
    einePackungFruechte[1]=new Frucht("Apfel");
    einePackungFruechte[2]=new Frucht("Apfel");
    einePackungFruechte[3]=new Frucht("Apfel");
    einePackungFruechte[4]=new Frucht("Apfel");

    //Ein Kunde Kauft diese Packung, und isst den 2. Apfel
    //(index 1, da man ab 0 zählt. "null" stellt ein leeres Objekt dar)
    einePackungFruechte[1]=null;

    //Ausgabe: "Apfel", da in diesem Feld ein Apfel liegt.
    einePackungFruechte[2].getName();

    //Fehler, da dieser Apfel gegessen wurde und nicht mehr existiert.
    einePackungFruechte[1].getName();
}
```

Abbildung 27. Beispielhafte Anwendung eines Arrays.

Schleifen

Um einen Vorgang der immer wieder ausgeführt werden soll zu automatisieren, werden Schleifen, auch Loops genannt, verwendet. In Java gibt es drei verschiedene Loops, die while-Schleife, die for-Schleife und foreach- Schleife.

In der for- Schleife wird eine Variable mit einem Startwert initialisiert und so lange manipuliert, bis diese einen bestimmten, definierten Endwert erreicht. Bei jedem Schritt, wird der Code in der Schleife einmal durchlaufen. Die folgende Abbildung stellt eine for-Schleife dar.

```
public static void main(String args[])
{
    //Packung Früchte die im Supermarkt angeboten werden.
    //Diese Packungen haben immer 5 Früchte.
    Frucht[] einePackungFruechte=new Frucht[5];

    //Die Packung wird mit fünf Äpfeln gefüllt
    for(int i=0;i<5;i++)
    {
        einePackungFruechte[i]=new Frucht("Apfel");
    }


}
```

Abbildung 28. Eine For-Schleife

Die while-Schleife wird solange durchlaufen, bis eine bestimmte Bedingung erfüllt ist. Um dies zu verdeutlichen wird die Schleife aus der vorherigen Abbildung, in der nächsten Abbildung in eine while-Schleife umgeformt.

```
public static void main(String args[])
{
    //Packung Früchte die im Supermarkt angeboten werden.
    //Diese Packungen haben immer 5 Früchte.
    Frucht[] einePackungFruechte=new Frucht[5];

    //Variable zum Zählen der bereits verpackten Früchte
    int eingepackteFruechte=0;

    //Die Packung wird mit fünf Äpfeln gefüllt
    while(eingepackteFruechte<5)
    {
        einePackungFruechte[eingepackteFruechte]=new Frucht("Apfel");

        //Verpackte Früchte um 1 hochzählen
        eingepackteFruechte++;
    }


}
```

Abbildung 29. Eine While-Schleife

Die letzte Schleife ist die foreach-Schleife. Diese Schleife iteriert solange über eine iterierbare Datenstruktur, bis alle Objekte in der Datenstruktur einmal durchlaufen worden sind. Dabei wird in der

Schleife eine Variable initialisiert, die in jedem Schritt mit dem aktuellen Objekt befüllt wird. Diese Variable steht dann in der Schleife zur Verfügung. Die nächste Abbildung zeigt wie eine foreach-Schleife aussehen könnte.

```
public static void main(String args[])
{

    ArrayList<Frucht> fruechte=new ArrayList<Frucht>();

    //Liste mit 10 Äpfeln befüllen
    for(int i=0;i<10;i++)
    {
        fruechte.add(new Frucht("Apfel"));
    }


    //Foreach-Schleife
    //Jede Frucht in der Liste einmal durchlaufen
    for(Frucht aktuelleFrucht : fruechte)
    {
        //Gibt den Namen der aktuellen Frucht zurück
        aktuelleFrucht.getName();
    }
}
```

Abbildung 30. Eine Foreach-Schleife

if-Abfrage

Will man bestimmte Bedingungen oder Zustände prüfen, so kann man Gebrauch von der if-Kontrollstruktur machen. Dabei wird ein Zustand geprüft und je nach Ausgang der Prüfung wird „true“ oder „false“ zurückgegeben. Falls „true“ zurückgegeben wird, so wird der folgende Codeblock ausgeführt. Die if-Abfrage ist nach Bedarf auch mit einem „oder“ bzw. „else“ verknüpfbar. Dabei wird ein alternativer Codeblock ausgeführt, falls die if-Abfrage „false“ zurückgibt. Die folgende Abbildung stellt eine if-Abfrage verknüpft mit einem else dar.

```
ArrayList<Frucht> aepfel=new ArrayList<Frucht>();
ArrayList<Frucht> bananen=new ArrayList<Frucht>();

ArrayList<Frucht> fruechte=new ArrayList<Frucht>();

//Früchte Liste mit Äpfeln und Bananen befüllen
fruechte.add(new Frucht("Banane"));
fruechte.add(new Frucht("Apfel"));
fruechte.add(new Frucht("Banane"));
fruechte.add(new Frucht("Banane"));
fruechte.add(new Frucht("Apfel"));
fruechte.add(new Frucht("Banane"));

//Foreach-Schleife
//Jede Frucht in der Liste einmal durchlaufen
for(Frucht aktuelleFrucht : fruechte)
{
    //falls die aktuelle Frucht eine Banane ist,
    //füge die Frucht zur Bananenliste hinzu
    if(aktuelleFrucht.getName()=="Banane")
    {
        bananen.add(aktuelleFrucht);
    }
    //ansonsten muss es ein Apfel sein,
    //füge die Frucht zur Apfelliste hinzu
    else
    {
        aepfel.add(aktuelleFrucht);
    }
}
```

Abbildung 31. Anwendung einer if-Abfrage.

3.5 Requirements Engineering

Jede Software durchläuft in seiner Entwicklung einen Zyklus. Dieser Zyklus wird Software-Lebenszyklus genannt und besteht aus fünf verschiedenen Phasen. Zu diesen Phasen gehören Spezifikation, Entwurf, Implementierung, Installation und Betrieb. Die ersten zwei Phasen werden in diesem Kapitel behandelt. Der komplette Lebenszyklus ist in der Abbildung 32. zu sehen.

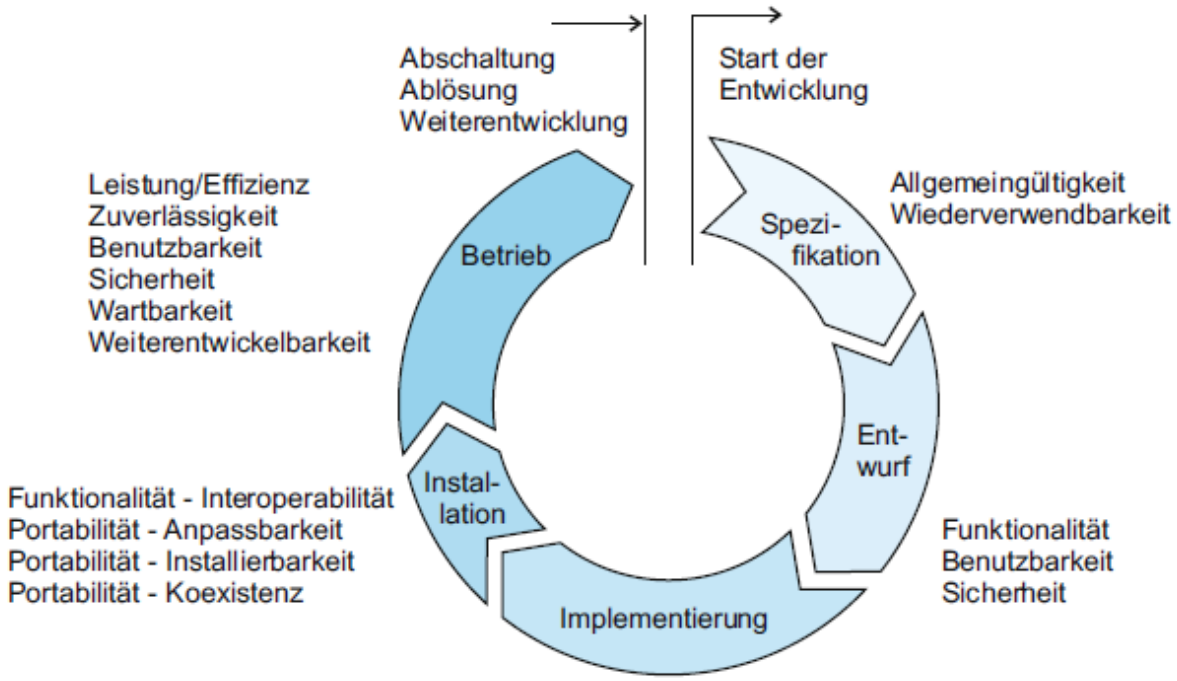

Abbildung 32. Lebenszyklus einer Software[28]

In der Spezifikationsphase wird festgehalten was von der Software gefordert wird, bzw. was die Software leisten soll[29]. Dazu legt man ein Lastenheft bzw. ein Pflichtenheft an. In diesem werden dann alle Punkte notiert, die der Kunde als wichtig erachtet. Normalerweise würde so ein Lastenheft während eines Gespräches mit dem Auftraggeber entstehen. In diesem Fall werde ich jedoch selbst die wichtigsten Punkte die von der Software gefordert werden zusammenfassen, ausgehend von der Aufgabenstellung.

3.5.1 Lastenheft

Die Software soll später bei der Orthesenanpassung genutzt werden. Um die Vermessung von Patienten zu erleichtern und zu beschleunigen, soll ein 3D-Scanner zum Einsatz kommen. Der Prototyp des Scanners soll über eine Drehscheibe, kleinere Objekte scannen können. Diese Drehschreibe wird mit einem Microcontroller ausgestattet sein, über den man die Position der Drehscheibe abrufen kann. Dieser Prototyp soll am Lehrstuhl für Produktionsentwicklung der Ruhr-Universität-Bochum zum Einsatz kommen.

Aus diesen Informationen lässt sich folgendes Lastenheft zusammenstellen:

Ziele
/LZ10/ Das Sanitätshaus soll in die Lage versetzt werden 3D-Scans von Patienten anfertigen zu können.
/LZ20/ Durch den Scanner soll die Orthesenanpassung beschleunigt und verbessert werden.
Rahmenbedingungen
/LR10/ Die Anwendung soll in der finalen Version in Sanitätshäusern zum Einsatz kommen.
/LR20/ Die Anwendung sollte leicht zu bedienen sein.
/LR30/ Zielgruppe für den Prototyp ist der Lehrstuhl für Produktionsentwicklung

[28] [BALZ2011] Lehrbuch der Softwaretechnik – Entwurf, Implementierung, Installation und Betrieb, S. 1
[29] [BALZ2009] Lehrbuch der Softwaretechnik - Basiskonzepte und Requirements Engineering, vgl. S.434

Funktionale Anforderungen
/LF10/ Die Anwendung soll 360° Scans ermöglichen
/LF20/ Die Anwendung soll unter Windows laufen
/LF30/ Die Anwendung soll die gescannten Modelle speichern können

Qualitätsanforderungen
/LQ10/ Das Scannen sollte so schnell wie möglich funktionieren
/LQ20/ Das Speichern sollte so schnell wie möglich funktionieren
/LQ30/ Der Prototyp mit der Drehscheibe sollte so wenig Benutzereingaben wie möglich erfordern

Dieses Lastenheft bietet nun dem Softwareentwickler die Möglichkeit, während der Entwurfsphase der Software, alle Ziele im Überblick zu haben. Dadurch ist gewährleistet, dass das Endprodukt genau das Liefert was der Kunde erwartet.

3.6 Entwurf

In der Entwurfsphase wird die Funktionsweise und der Aufbau der Software geplant. Zunächst muss der Entwickler sich im Klaren sein wie seine Software grundsätzlich arbeiten soll. Im Anschluss müssen die Klassen geplant und skizziert werden. Außerdem muss festgehalten werden, auf welche Weise die Klassen untereinander agieren und in welcher Beziehung sie zu einander stehen. Dies kann man durch ein Klassendiagram darstellen. Zusätzlich sollte sich der Entwickler Gedanken um den Kontrollfluss der Software machen, bzw. wie die Software auf verschiedene Eingaben reagiert oder welche Angaben vom Benutzer erwartet werden, um bestimmte Aktionen auszulösen. Um diesen Kontrollfluss darzustellen, werden Flussdiagramme verwendet.

3.6.1 Skizzierung des Scanvorgangs

Der Scan soll durch Betätigung einer Schaltfläche im Fenster der Anwendung erfolgen. Das Problem dabei ist jedoch, dass ein 360° Scan gewünscht ist. Die Kamera kann jedoch nur ein Tiefenbild bzw. ein Scan von dem was in dessen Blickwinkel liegt machen. D.h. dass beispielsweise die hintere Fläche von dem Objekt was gescannt werden soll, nicht mitgescannt wird. Es muss also eine Möglichkeit gefunden werden dies zu umgehen.

Eine Möglichkeit wäre, die Kamera um das Objekt was gescannt werden soll zu bewegen. Dabei entsteht jedoch das Problem, dass die Software wissen muss wie weit man sich um das Objekt bewegt hat. Zusätzlich müsste noch ermittelt werden ob man sich während der Bewegung dem Objekt genähert hat oder nicht. Prinzipiell ist dies durch Digitale Bildverarbeitung möglich, in dem man beispielsweise nach jedem Frame, den aktuelle Frame mit dem vorherigen vergleicht und analysiert, welche Änderungen stattgefunden haben. Diese Methode wird auch bei größeren Projekten erfolgreich angewandt. Für einen einzelnen Entwickler ist dies jedoch viel zu Aufwendig und würde den Rahmen dieser Arbeit sprengen.

Benötigt wird also eine Methode bei der die Kamera an eine fixierte Position hat. Wenn die Kamera fixiert ist, muss sich also das Objekt um seine eigene Achse drehen. Dabei schießt die Kamera durchgehend einzelne Tiefenwertbilder und fügt diese zusammen. Nach jeder Aufnahme muss das Bild natürlich ausgehend von der Position des Objektes rotiert werden. D.h. wenn sich das Objekt um 30° dreht, muss die Aufnahme auch im 3D-Raum um 30°, um dieselbe Achse rotiert werden. Anschließend müssten die Aufnahmen nur noch im 3D-Raum zusammengefügt werden. Das Problem bei dieser Methode ist es zu ermitteln um welchen Winkel sich das Objekt seit der letzten Aufnahme gedreht hat. Eine Möglichkeit wäre es eine Vorrichtig zu benutzen, bei dem sich das Objekt auf einer drehbaren Plattform mit markanten Markierungen befindet. Die Kamera müsste dann in einer fixierten Entfernung auf das Objekt zeigen. Ein beispielhafter Aufbau ist in der Abbildung 33. zu sehen.

Abbildung 33. Eine Drehplattform mit Markierungen[30]

Nun könnte man das Objekt zusammen mit den Markierungen rotieren lassen und in bestimmten abständen Aufnahmen machen. Dabei müsste man dann wie in der vorherigen Methode in jeder Aufnahme analysieren wie weit sich die Markierungen bewegt haben. Daher, dass die Markierungen in diesem Beispiel markant genug sind, können diese nun recht einfach detektiert und analysiert werden. Das Problem der Digitalen Bildverarbeitung besteht jedoch weiterhin.

Die dritte Alternative wäre eine Drehplattform die mit einem Mikrocontroller versehen ist zu nutzen. Mit dieser Plattform könnte man das Objekt wie in der vorherigen Methode automatisch drehen lassen und währenddessen in bestimmten Zeitabständen Aufnahmen machen. Der Unterschied zur vorherigen Methode besteht darin, dass die Ermittlung der Drehwinkel wesentlich einfacher ist. Hierbei muss der Mikrocontroller einfach nach der Information angefragt werden. Mit dieser Information können die Aufnahmen dann dementsprechend rotiert werden. Ein beispielhafter Scanner, der diese Methode nutzt, ist in der Abbildung 34. zu sehen.

Die dritte Methode ist also die Effektivste, da hier keine Bildverarbeitung nötig ist.

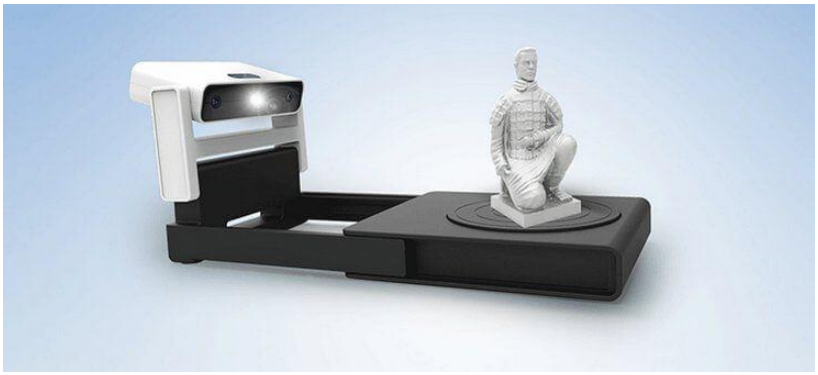

Abbildung 34. Beispiel einer Drehplattform mit Mikrocontroller Steuerung[31]

Für den Einsatz in Sanitätshäusern sollte der Scanner jedoch auch ohne Drehplattform nutzbar sein, da es hier z.B. vorkommen kann, dass man gezielt nur eine Gliedmaße scannen muss. Für diesen Fall sollte die Anwendung also auch manuelle Eingabemöglichkeiten bieten, um die Position des Objektes anzugeben.

Wenn beispielsweise nur ein Bein gescannt werden soll, könnte ein frontaler Scan vom Bein gemacht werden. Nun müsste der Patient sich einmal um 90° im Uhrzeigersinn drehen und der Benutzer der

[30] Quelle: https://hackadaycom.files.wordpress.com/2011/08/kinect-3d-turntable.png (abgerufen am 10.10.2015)
[31] Quelle: http://www.3d-grenzenlos.de/wp/wp-content/uploads/2015/01/einscan_S-3d_scanner.jpg (abgerufen am 10.10.2015)

Software müsste eingeben, dass sich das Objekt um 90° im Uhrzeigersinn gedreht hat. Dieser Vorgang ist beliebig oft um beliebige Winkel wiederholbar. Der Vorgang müsste also solange wiederholt werden, bis man das Bein aus genug Perspektiven aufgenommen hat, um aus den Aufnahmen ein 3D-Scan zu erstellen.

Die geringste Anzahl an Aufnahmen, sollte vier betragen. Diese Aufnahmen sollten jeweils Aufnahmen von dem Objekt nach einer 0°, 90°, 180° und 270° Drehung sein.

3.6.2 MVC-Muster

Jede Software hat eine bestimmte grundlegende Architektur, die bereits grob angibt wie die Software gebaut sein sollte. Man sollte also nicht versuchen das Rad neu zu erfinden, sondern sich einfach an das Architekturmuster halten, was die zu entwickelnde Software am besten wiederspiegelt. Über die Jahre sind dadurch viele verschiedene Architekturmuster endstanden. Wenn man sich näher über all diese Architekturen informieren will, so empfiehlt sich das Buch „Design Patterns. Elements of Reusable Object-Oriented Software“ geschrieben von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. An dieser Stelle soll es jedoch genügen, das MVC[32]-Muster zu betrachten.

Die zu entwickelnde Software soll eine unter Windows benutzbare interaktive Anwendung sein. D.h. die Anwendung muss visuell auf dem Bildschirm zu erkennen sein und dem Benutzer Eingaben ermöglichen. Ebenfalls sollen gesammelte Daten, bzw. in unserem Fall ein Bild dargestellt werden, nämlich die Tiefenwertaufnahme der Kamera.

Diese Beschreibung der Software ähnelt sehr stark dem MVC-Muster. Bei diesem Muster gibt es immer ein „Model[33]“ eine „View[34]“ und ein „Controller[35]“ die als Klassen dargestellt werden. Die Model-Klasse stellt die Daten die visualisiert werden müssen, bzw. die Daten die gespeichert werden sollen dar. In unserem Falle wäre das, die Tiefenwertaufnahme, bzw. der Scan. Die View-Klasse stellt die optische Repräsentation der Anwendung dar, also das Fenster in unserer Software. Die Controller-Klasse übernimmt die Verarbeitung der Eingaben in der View. Dieser Vorgang ist auf dem Diagramm in der Abbildung 35. zu sehen.

[32] Abkürzung für Model View Controller
[33] Englisch für Modell
[34] Englisch für Ansicht
[35] Englisch für Regler

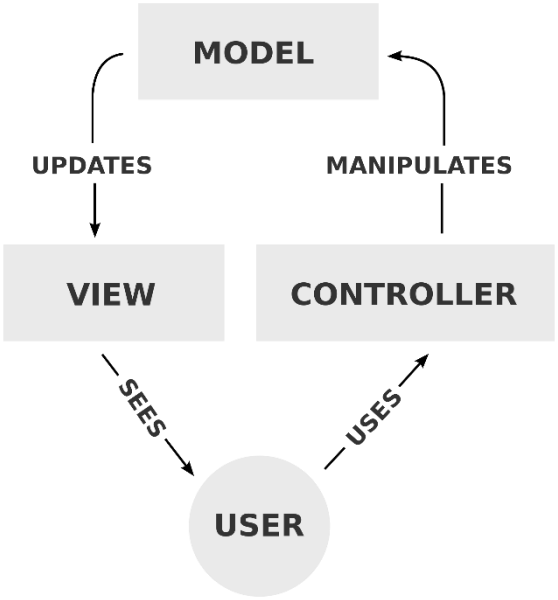

Abbildung 35. Diagramm einer MVC-Architektur[36]

Da unsere Software dem MVC-Muster ähnelt, wäre es also ratsam die Software nach diesem Muster zu erstellen. Das spart Arbeit, da man sich keine Gedanken mehr um die Grundstruktur der Software machen muss. Unsere Software wird also auf irgendeine Art und Weise die drei Kernteile „Model“, „View“ und „Controller“ beinhalten.

3.6.3 Klassen

Eine Software besteht aus vielen Einzelteilen. Diese Einzelteile nennen sich Klassen. Der Code der Software wird, nach Aufgabenbereich getrennt und in einzelne Klassen gekapselt. Somit übernimmt jede Klasse die für sie bestimmte Aufgabe. Die Zusammenarbeit und Vernetzung der Klassen formen dann die Software. Dies nennt sich das Prinzip der „Trennung von Zuständigkeiten“[37] in der Softwaretechnik. Durch die Trennung in einzelne Aufgabenteile, ist die Software besser strukturiert und somit leichter zu lesen bzw. zu verstehen. Außerdem reduziert man dadurch redundante Arbeiten, da es sein kann, dass eine Teilaufgabe in einer Software, in einer anderen Software auf genau dieselbe Art und Weise wiederauftaucht. In diesem Fall kann man dann die Klassen aus dem ersten Projekt einfach wiederverwenden. Durch die Trennung in Klassen entsteht ebenfalls eine bessere Wartbarkeit, weil mögliche Änderungen am Code nur an einer Stelle vorgenommen werden müssen.

Die in diesem Projekt verwendeten Klassen können grob in fünf Gruppen unterteilt werden. Das MainFrame sowie das CustomPanel sind zuständig für die Anzeige und die Nutzereingaben der Software. Der Controller übernimmt die Steuerung der Software. Vektor3 und Matrix3 sind zuständig für die Berechnungen. Die Klassen Scan und Scanner sind für das eigentliche Scannen verantwortlich. Der Assembler übernimmt das Speichern der Modelle.

Das MainFrame erbt von der Klasse JFrame[38] und ist somit das Hauptfenster der Anwendung. Hier werden alle Steuerelemente der Anwendung angezeigt. Außerdem dient diese Klasse als Schnittstelle

[36] Quelle: https://upload.wikimedia.org/wikipedia/commons/thumb/a/a0/MVC-Process.svg/2000px-MVC-Process.svg.png (abgerufen am 12.10.2015)
[37] [BALZ2011], Lehrbuch der Softwaretechnik – Entwurf, Implementierung, Installation und Betrieb, S.31
[38] JFrame ist die Basisklasse in Java, die zur Darstellung von Anwendungsfenstern dient.

zwischen dem Controller und dem Nutzer. Die Eingaben die der Nutzer im MainFrame tätigt, werden dem Controller übermittelt.

Der Controller steuert die gesamte Software. Hier werden die Methoden der Klassen je nach den Eingaben, die der Controller vom MainFrame erhält, aufgerufen. Ein Controller ist nötig um das Prinzip der „Trennung von Zuständigkeiten“ aufrecht zu erhalten, da die Klassen sich untereinander nicht kennen und jeweils nur eine Aufgabe übernehmen können.

Das CustomPanel erbt von der Klasse JPanel[39] und ist dafür spezialisiert Bilder anzuzeigen. Diese Aufgabe könnte auch einfach der JPanel übernehmen, jedoch ist es komplizierter mit dem JPanel Bilder anzuzeigen. Also wird der Code um Bilder in einem JPanel anzuzeigen in den CustomPanel verlagert, um so den Hauptteil des Codes übersichtlicher zu halten.

Die Klassen Vektor3 und Matrix3 stellen wie aus den Namen erkenntlich Vektoren und Matrizen dar. Diese Klassen sind nötig, da es möglich sein muss die Aufnahmen zu rotieren, damit 360° Scans erzeugt werden können. Um dies zu ermöglichen braucht man Drehmatrizen, die mit der Klasse Matrix3 erzeugt werden können. Die Vektor3 Klasse stellt die einzelnen Vertices dar, die dann durch eine Multiplikation mit einer Drehmatrix rotiert werden können. Wenn alle Vertices einer Pointcloud auf diese Weise rotiert werden, rotiert folglich die gesamte Aufnahme.

Die Klasse Scanner erzeugt zusammen mit der Klasse Scan die Aufnahme. Der Scanner steuert durch die OpenNI Bibliothek die Kamera und erzeugt dabei Tiefenwertbilder. Diese Tiefenwertbilder werden Zeilenweise durchlaufen und dabei alle Tiefenwerte in Vertices umgewandelt. Anschließend wird eine Instanz der Klasse Scanner erzeugt und mit den Vertices befüllt. Diese Instanz stellt nun die Aufnahme dar.

Die Assembler Klasse ist dafür zuständig, die einzelnen Scans, die aus verschiedenen Perspektiven erzeugt wurden, zu einem vollständigen 3D-Scan zu vereinen.

Die Klassen „MainFrame“, „Scan“ und „Controller“ stellen dabei die Kernklassen des MVC-Musters, „View“, „Model“ und „Controller“ dar.

Nachdem alle Klassen skizziert worden sind, müssen in einem Klassendiagramm dessen Attribute, Methoden und Funktionen, sowie deren Verbindungen untereinander dargestellt werden. Dieses Diagramm benutzt der Softwareentwickler als Grundgerüst für sein Programm.

[39] JPanel ist die Basisklasse in Java, die zur Darstellung und Gruppierung verschiedener Inhalte dient.

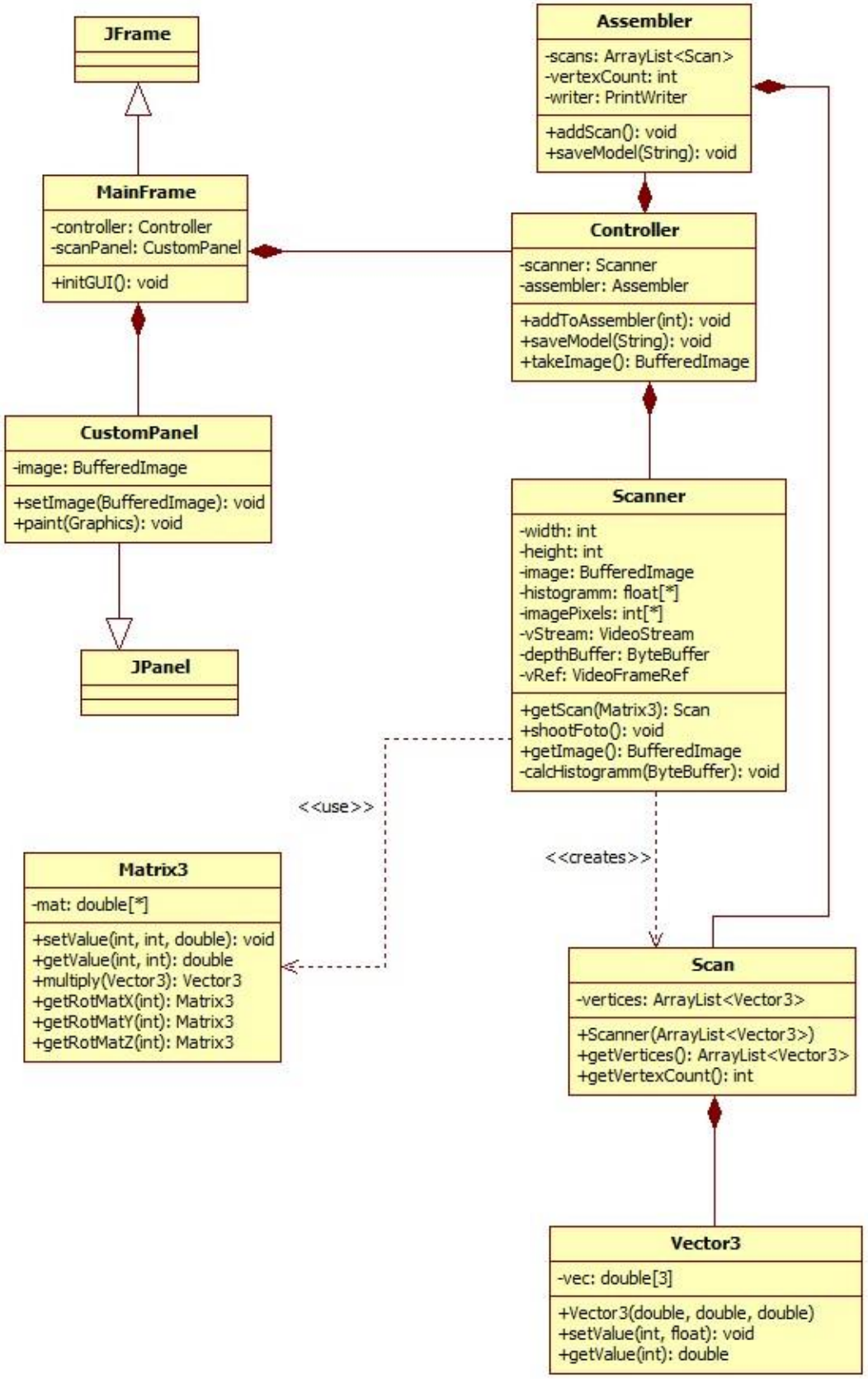

Abbildung 36. Klassendiagramm der zu entwickelnden Software

Die einzelnen Kästchen im Diagramm stellen die Klassen der Software dar. Im oberen Feld ist der Klassenname zu sehen. Im zweiten Feld stehen die Attribute der Klassen und im letzten Feld die Operationen der Klassen. Dabei werden Globale Attribute und Operationen durch ein „+“ Zeichen

eingeleitet, die Privaten durch ein „-“ Zeichen. Die Notation der Attribute besteht aus dem Namen des Attributs sowie dessen Datentyp getrennt durch ein „:“. Die Operationen werden auf dieselbe Weise notiert, nur mit dem Unterschied, dass am Ende der Rückgabedatentyp der Operation steht.

Die Verbindungen zwischen den Klassen geben deren Beziehung untereinander an. In dem Diagramm wurden drei verschiedene Verbindungen genutzt.

Ein Pfeil mit weißer Spitze gibt eine Vererbung an. Wenn zwei Klasen durch so einen Pfeil mit einander verbunden sind bedeutet es, dass die Klasse von dem der Pfeil ausgeht, von der Klasse zu dem der Pfeil zeigt, erbt.

Ein Pfeil mit gestrichelter Linie gibt eine Abhängigkeit an. Dabei hängt die Klasse aus dem der Pfeil ausgeht von der Klasse ab, auf die der Pfeil Zeigt. Diese Pfeile werden meist beschrieben um weitere Informationen über die Abhängigkeiten zu geben. Im Diagramm ist z.B. zu sehen, dass eine Abhängigkeit zwischen der Klasse Scanner und Scan besteht. Zusätzlich ist durch die Beschriftung <<creates>> zu erkenn, dass die Abhängigkeit dadurch entsteht, dass die Klasse Scanner eine Instanz der Klasse Scan erzeugt.

Ein Pfeil mit gefüllter Raute als Spitze gibt eine Komposition an. Eine Komposition gibt dabei an, dass die Existenz eines Objektes von der eines anderen abhängt. Zum Beispiel kann ein Raum ohne Gebäude nicht existieren. Beispielsweise ist im Diagramm zu sehen, dass keine Instanz einer Klasse, ohne eine Instanz der MainFrame Klasse existieren könnte.

Auf die Implementation und die Erklärung der Funktionen, sowie Attribute der Klassen, wird im Abschnitt „Programmverlauf im Detail“ eingegangen, nachdem der Programmfluss im Flussdiagramm dargestellt wurde. Dies führt zu einem besseren Verständnis der Programmstruktur.

4 Implementation

4.1 Flussdiagramm

Durch ein Flussdiagramm kann der Kontrollfluss einer Software dargestellt werden. In ihr wird festgehalten welche Aktionen und Eingaben des Benutzers, welche Reaktionen der Software auslöst. Daraus wird ersichtlich, wie das Anwendungsfenster aufgebaut sein sollte und welche Kontrollelemente, wie z.B. Schaltflächen und Textfelder, nötig sind.

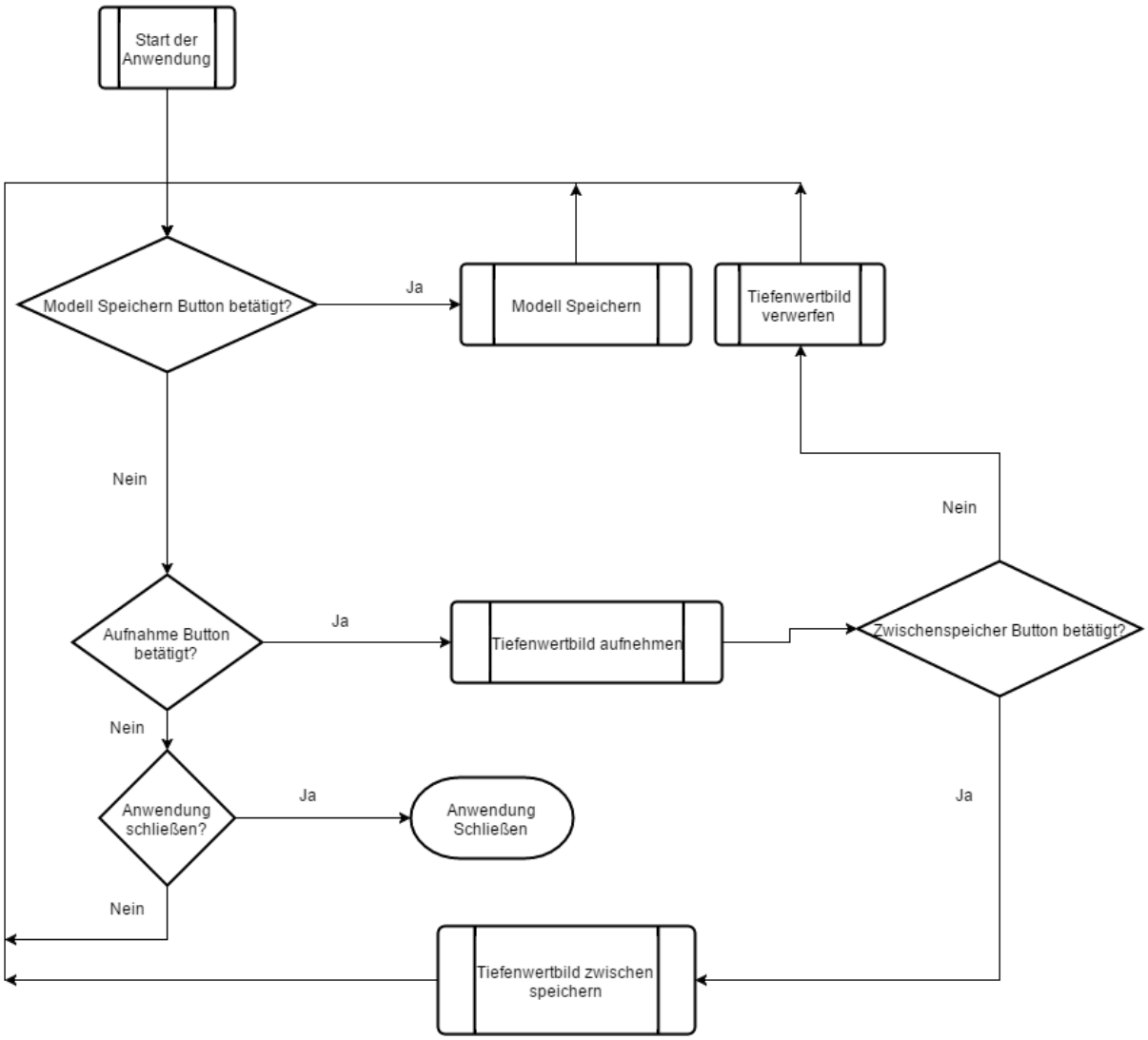

Abbildung 37. Flussdiagramm der Scan-Software

Ein Flussdiagramm besteht im Wesentlichen aus drei Bausteinen, Entscheidungen, Prozesse und Terminale. Entscheidungen werden durch Rauten dargestellt. Diese Entscheidungen bieten Verzweigungen die nach vorausgesetzten Bedingungen eingeschlagen werden. Diese Entscheidungen sind häufig Fragen die mit Ja oder Nein beantwortet werden können. Die Verzweigungen führen dann zu weiteren Elementen. Prozesse werden durch Rechtecke mit abgerundeten Ecken dargestellt und zeigen einen Prozess an, der im Programm vollzogen wird. Die Prozesse haben immer einen Ein- und einen Ausgang. Terminale, dargestellt durch Ovale, stellen den Endpunkt des Flusses dar.

Das in der Abbildung 37 dargestellte Flussdiagramm, beginnt mit dem Start der Anwendung. Nach dem Start wird geprüft ob der Modell-Speichern Button getätigt wurde. Falls ja wird das Modell gespeichert.

Nach dem Speichern kann man weitere Scans aufnehmen, da die Software an dieser Stelle nicht endet. Hat man den Modell-Speicher Button nicht betätigt, so wird geprüft ob der Aufnahme-Button betätigt wurde. Falls ja, wird eine Aufnahme getätigt und ein Tiefenwertbild gespeichert. Danach hat man die Wahl den Zwischenspeicher-Button zu tätigen. Falls man diesen betätigt, wird das Bild zwischengespeichert und man kann weitere Bilder aufnehmen, da man zum Anfang des Programms geleitet wird. Falls man den Zwischenspeicher-Button jedoch nicht tätigt, so wird das zuletzt gespeicherte Bild zum Überschreiben vom nächsten Bild freigegeben und somit quasi verworfen. Die letzte Entscheidung ist, ob die Anwendung geschlossen werden soll. Wenn sie geschlossen werden soll, gelangt man bei der Terminale, die das Ende des Programmlaufs darstellt und die Anwendung schließt sich.

Es ist zu beachten, dass die Reihenfolge der ersten drei, untereinander folgenden Entscheidungen in der Abbildung, austauschbar sind. Es könnte also genauso gut erst überprüft werden ob der Aufnahme-Button betätigt wurde. Die Reihenfolge der Entscheidungen in der Abbildung sorgt lediglich für eine bessere Übersicht des Diagramms.

4.2 Anwendung des Scanners

Anhand des Flussdiagramms und des Klassendiagramms, kann nun eine beispielhafte Anwendung der Software dargestellt werden.

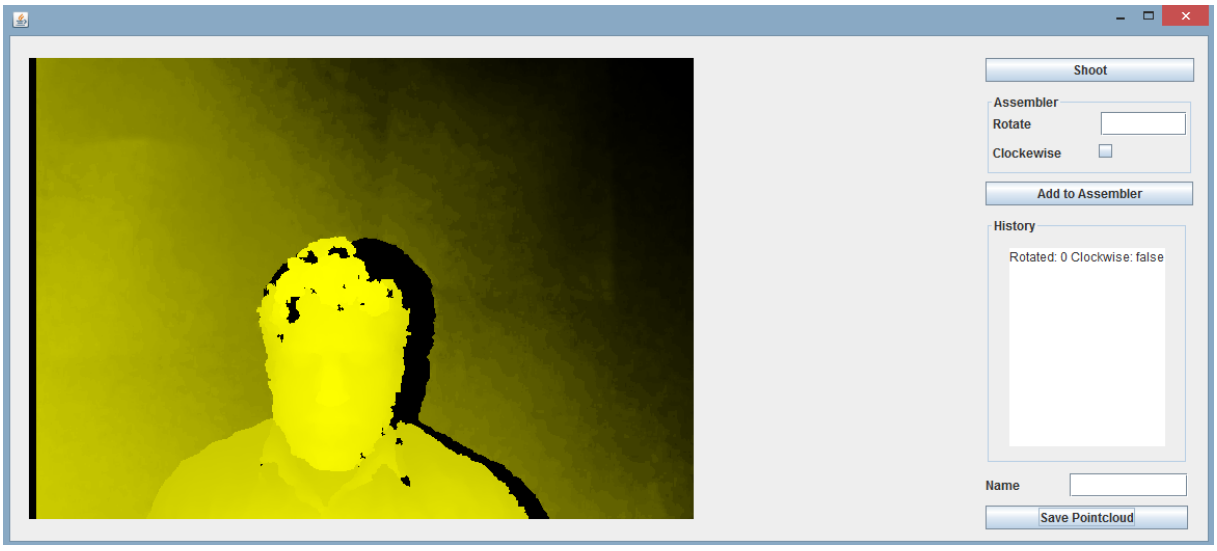

Abbildung 38. Die Benutzeroberfläche der Software.

Auf der Abbildung 38. ist die Benutzeroberfläche zu sehen. Im Hauptbereich des Fensters ist eine Vorschau des Scans, bzw. das Tiefenwertbild des Scans zu erkennen. Die Rechte spalte im Fenster beinhaltet alle Steuerelemente.

Um einen Scan anzufertigen, muss die Kamera zunächst auf das zu scannende Objekt gerichtet werden. In der Abbildung habe ich mein Gesicht gescannt. Also musste ich die Kamera zunächst auf mein Gesicht richten. Anschließend kann der Benutzer per Klick auf den „Shoot“-Button[40], ein Bild

[40] Englisch für Knopf/Schaltfläche

aufnehmen. Das Ergebnis wird daraufhin im Vorschaubereich angezeigt. Ist der Nutzer mit dem Ergebnis unzufrieden, so kann er ohne weiteres erneut den „Shoot“-Button anklicken, woraufhin die alte Aufnahme durch eine neue ersetzt wird.

Ist der Benutzer zufrieden, so kann er angeben ob die Pointcloud, die durch das Tiefenwertbild erzeugt würde, rotiert werden soll. Dafür gibt der Benutzer im Textfeld „Rotate“ in der rechten Spalte ein, um wie viel Grad das Objekt rotiert werden muss. Es kann auch angegeben werden, ob das Objekt im- oder gegen den Uhrzeigersinn rotiert werden soll. Dazu muss ein Häkchen in die „Clockwise“-Checkbox[41] gesetzt werden, falls das Objekt im Uhrzeigersinn rotieren soll, ansonsten bleibt die Box frei. In der Abbildung, wollte ich eine Frontalaufnahme meines Gesichts haben. Da mein Geischt ohnehin frontal positioniert war und keine Rotierung nötig war, musste ich als Winkel null eingeben.

Sind alle Eingaben getätigt, so kann der „Add to Assembler“-Button betätigt werden. Dadurch wird die Aufnahme, zusammen mit den eingegeben Werten verarbeitet und zwischengespeichert. Die Informationen, der bereits gespeicherten Aufnahmen, werden im Bereich „History“ in der rechten Spalte festgehalten. Dadurch weiß man immer, welche Aufnahmen bereits getätigt wurden und welche noch fehlen. In der Abbildung ist zu sehen, dass die zuletzt gespeicherte Aufnahme um null Grad gegen den Uhrzeigersinn rotiert wurde. Wenn ich eine 360° Aufnahme meines Kopfes anfertigen müsste, so wüsste ich nun, dass noch Aufnahmen fehlen, die um 90°, 180° und 270° rotiert wurden.

Der Benutzer kann nun beliebig viele weitere Aufnahmen tätigen, die für den anzufertigenden Scan nötig sind. Wurden alle Aufnahmen getätigt, so müssen alle zwischengespeicherten Aufnahmen in einem 3D-Modell gespeichert werden. Dazu muss zunächst der Dateiname des Modells eingegeben werden. Dieser wird über das Textfeld „Name“ eingegeben. Anschließend kann der Speichervorgang durch den „Save Pointcloud“-Button gestartet werden. Dies erzeugt eine Pointcloud des Scans und speichert diese in einem 3D-Modell in Form einer ply Datei ab.

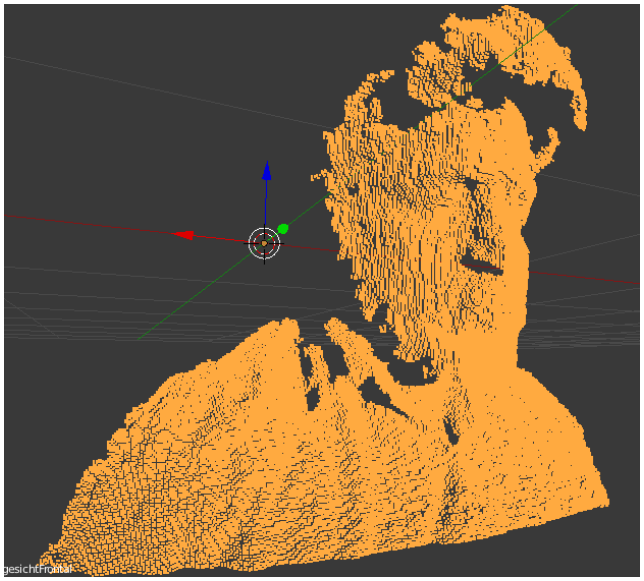

Abbildung 39. Die, mit der Software erzeugte Pointcloud.

[41] Checkboxen können aktiviert oder deaktiviert werden. Im aktiven Zustand liefern sie den Wert true, im deaktivierten Zustand den Wert false

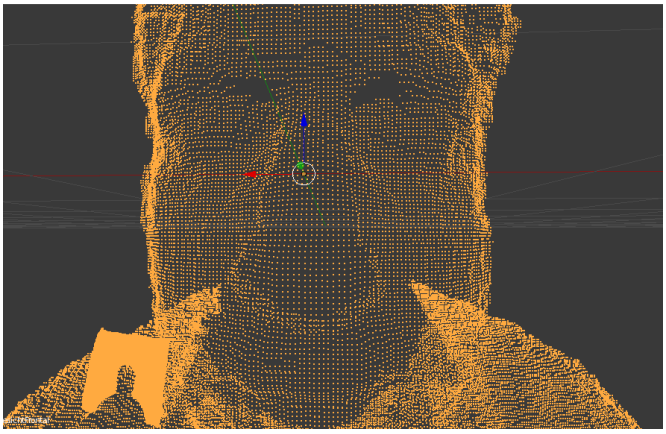

Abbildung 40. Pointcloud im detail.

In den Abbildungen 39 und 40 ist nun die fertige Pointcloud, die durch das Tiefenwertbild aus der Vorschau entstanden ist, zu sehen.

4.3 Umsetzung der Klassen

In diesem Abschnitt, werden die wichtigsten Prozesse die im Hintergrund laufen und somit für den Benutzer unsichtbar sind, erläutert. Zusätzlich wird grob die Implementierung der Klassen aus dem Klassendiagramm dargestellt. Dabei wird vorausgesetzt, dass dem Leser bewusst ist, wofür die einzelnen Klassen dienen, siehe „3.2.3 Klassen“.

4.3.1 Vector3 und Matrix3

Um die Rotation von Vertices zu ermöglichen, müssen Matrizen sowie Vektoren benutzt werden. Die Vector3 Klasse, stellt dabei einen drei dimensionalen Vektor dar. Die Implementation dieser Klasse ist sehr simpel und trivial. Um die einzelnen Koordinaten eines Vertex zu speichern, besitzt die Vector3 Klasse ein drei dimensionales Array. Um die einzelnen Felder mit Werten zu belegen, besitzt diese Klasse simple Getter und Setter[42]. Abgesehen davon ist die „toString()“ Methode, die wichtigste Methode. Diese Methode gibt die drei Koordinaten eines Vertex als Zeichenkette aus. In der Assembler Klasse wird weiter auf diese Methode eingegangen. Die Abbildung 41. zeigt die „toString()“ Methode.

```
@Override
public String toString()
{
    return vec[0]+" "+vec[1]+" "+vec[2];
}
```

Abbildung 41. toString() Methode der Klasse Vertex3.

[42] Getter und Setter Methoden dienen dazu, Attribute von Klassen abzufragen (Getter), sowie Attribute mit Werten zu belegen (Setter)

Die Matrix3 Klasse ist komplexer als die Vector3 Klasse. Genau wie die Vector3 Klasse, besitzt auch diese Klasse simple Getter und Setter um Werte in der Matrix zu setzen. Die Hauptfunktion dieser Klasse besteht darin, Drehmatrizen zu bieten. Um drei dimensionale Punkte rotieren zu können, müssen Vektoren die diese Punkte darstellen, mit den Drehmatrizen multipliziert werden. Also muss die Matrix3 Klasse eine Methode zum Multiplizieren bieten.

Eine drei dimensionale Drehung um die X-Achse um den Winkel α würde dabei wie folgt aussehen.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & cos \propto & -sin \propto \\ 0 & sin \propto & cos \propto \end{pmatrix}$$

Da es mühselig wäre, jedes Mal diese Matrix manuell im Code zu erzeugen, bietet die Matrix3 Klasse die Methode „getRotMatX(int degrees)“. Diese Methode nimmt den gewünschten Drehwinkel α entgegen, erzeugt damit automatisch die Drehmatrix und gibt diese zurück. Da die Methode als „static“[43] deklariert ist, handelt es sich um eine Klassenmethode. Somit kann die Methode „getRotMatX(int degees)“ aufgerufen werden, ohne eine Instanz der Klasse Matrix3 erstellt zu haben. Das hat den Vorteil, dass eine Variable sofort mit dem Rückgabewert der Methode gefüllt werden kann.

```
public static Matrix3 getRotMatX(int degrees)
{
    double cosAlpha =Math.cos(Math.toRadians(degrees));
    double sinAlpha =Math.sin(Math.toRadians(degrees));

    Matrix3 tmp=new Matrix3();

    tmp.setValue(0, 0, 1);
    tmp.setValue(1, 1, cosAlpha);
    tmp.setValue(1, 2, sinAlpha*-1);
    tmp.setValue(2, 1, sinAlpha);
    tmp.setValue(2, 2, cosAlpha);

    return tmp;
}
```

Abbildung 42. Implementierung der "getRotMatX" Methode.

In der Abbildung 42. ist zu sehen, dass zunächst die zwei Variablen cosAlpha und sinAlpha angelegt werden. In diesen Variablen werden die cos(α) und sin(α) Werte aus der Drehmatrix festgehalten. Dazu wird die Java Klasse „Math“ und dessen cos und sin Methoden benutzt. Danach wird eine temporäre Matrix erstellt und der Variable „tmp“ zugewiesen. Anschließend werden alle Werte der Matrix entsprechend der Drehmatrix um die X-Asche aufgefüllt und die fertige Matrix wird ausgegeben.

Analog zu dieser Methode existieren auch Methoden die Drehmatrizen um die Y- oder Z-Achse liefern.

Die Multiplikation der Matrix mit einem Vektor wird von der „multiply(Vector3 vec)“ übernommen. Dieser Methode wird ein Vektor, der mit der Matrix multipliziert werden soll, über einen Parameter übergeben. Dieser Vektor wird anschließend mit der Matrix multipliziert und der daraus resultierende neue Vektor wird zurückgegeben.

[43] Schlüsselwort um Klassenmethoden, bzw. Klassenattribute zu deklarieren. Diese Methoden und Attribute beziehen sich dabei nicht auf einzelne Instanzen einer Klasse, sondern für die Klasse selbst.

In der folgenden Abbildung ist die Implementation der „multiply(Vector3 vec)“ Methode zu sehen.

```
public Vector3 multiply(Vector3 vec)
{
    //Den Parameter zwischenspeichern
    Vector3 tmpVec=new Vector3();

    //Hier wird jede Zeile der Matrix mit dem Vektor tmpVec multipliziert
    for(int i=0;i<3;i++)
    {
        float result=0;

        for(int j=0;j<3;j++)
        {
            result+=mat[i][j]*vec.getValue(j);
        }

        tmpVec.setValue(i, result);
    }
    return tmpVec;
}
```

Abbildung 43. Implementierung der "multiply" Methode.

Um Matrizen mit Vektoren zu multiplizieren, muss zunächst sichergestellt werden, dass die Spaltenanzahl der Matrix, der Größe des Vektors entspricht. Trifft dies zu, sind Matrix und Vektor multiplizierbar. Dieser Schritt entfällt bei der Implementierung, da beide für den drei dimensionalen Raum konzipiert sind. Das Vektor Matrix Produkt ist immer ein weiterer Vektor. Die Einträge dieses Vektors entsprechen der zeilen- und spaltenweisen Multiplikation, der Komponenten von dem Ausgangsvektor und der Matrix.

Beispiel:

$$\begin{pmatrix} 3 & 0 & 2 \\ 4 & 2 & 5 \\ 6 & 2 & 3 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 3*1+0*2+2*3 \\ 4*1+2*2+5*3 \\ 6*1+2*2+3*3 \end{pmatrix} = \begin{pmatrix} 9 \\ 23 \\ 19 \end{pmatrix}$$

In der Methode wird der übergebene Parameter zunächst in einer Variable gespeichert. Wie im Beispiel zu sehen ist, muss die Matrix nun zeilen- und spaltenweise durchlaufen werden. Dabei müssen die entsprechenden Komponenten der Matrix und des Ausgangsvektors multipliziert und die Ergebnisse aufsummiert werden. Dazu wird eine doppelte for-Schleife genutzt. Die erste for-Schleife, dient dabei dem zeilenweisen Durchlauf. Am Anfang dieser Schleife wird das Zwischenergebnis, welches in der Variable „result“ festgehalten wird, jedes Mal auf null gesetzt, da hier immer ein neuer Eintrag im Ergebnisvektor beginnt.

In der zweiten Schleife wird die Matrix spaltenweise durchlaufen. D.h. man geht von links nach rechts und von oben nach unten durch die Matrix. Dabei gibt die Variable i die Zeilenposition und j die Spaltenposition des aktuellen Wertes in der Matrix an. In jedem Schritt der inneren Schleife wird der Wert der sich an der Position [i,j] in der Matrix befindet, mit dem Wert der sich an der Position [j] im Ausgangsvektor befindet, multipliziert. Das Produkt beider Werte wird auf die Variable „result“ hinzuaddiert.

Wurden alle Werte in einer Zeile durchlaufen, d.h. wenn die innere Schleife terminiert, wird der Wert an der i. Position im Ergebnisvektor eingetragen. Wenn auch die äußere Schleife terminiert, so ist der Vorgang abgeschlossen und der Ergebnisvektor „tmpVec“ kann zurückgegeben werden.

4.3.2 Scan

Die Klasse Scan stellt die einzelnen Aufnahmen dar, in dem jeder Pixel in einem Tiefenwertbild als ein Vektor dargestellt und gespeichert wird. Da dies pro Aufnahme nur einmal geschehen soll, bzw. da es nicht nötig ist diese Scans im Nachhinein zu manipulieren, reicht es aus die Vertices im Konstruktor zu übergeben.

Um die Vertices zu speichern wird hier die Variable „vertices“ vom Typ ArrayList<Vektor3> benutzt. Im Konstruktor wird dieser Variable dann der übergebene Parameter zugewiesen, wie in der Abbildung 44. zu sehen.

```
public Scan(ArrayList<Vector3> vertices)
{
    this.vertices=vertices;
}
```

Abbildung 44. Konstruktor der Klasse Scan.

Abgesehen von dem Konstruktor besitzt die Klasse zwei Getter Methoden. Eine dieser Methoden ist die „getVertexCount“ Methode. Diese Methode gibt die Anzahl der gespeicherten Vertices an.

Die zweite Methode ist die „getVertices“ Methode. Diese Methode gibt die gespeicherte Liste an Vertices aus.

4.3.3 Scanner

Der Scanner generiert die Scans, d.h. in dieser Klasse werden die Aufnahmen erstellt. Dazu wird die Asus Kamera in Zusammenhang mit zwei Klassen aus der OpenNi Bibliothek genutzt. Eine dieser Klassen ist die VideoStream Klasse. Diese Klasse stellt den Datenstrom der Kamera dar. Man kann diesen entweder schließen oder öffnen. Falls der Datenstrom geschlossen wurde, hört die Kamera auf Daten zu erzeugen, andernfalls nimmt er weiterhin Daten auf.

Durch die zweite Klasse VideoFrameRef erhält man Zugang auf die Daten, die durch den Fluss erzeugt werden. In unserem Fall sind das die Tiefenwertdaten.

Die Klasse Scanner bietet vier Methoden, „shootFoto“, „calcHistogramm“, „getImage“ sowie „getScan()“.

Die Methode „shootFoto()“ erzeugt eine Aufnahme. Diese Aufnahme wird auf zwei unterschiedliche Weisen gespeichert. Einmal als BufferedImage, also als Bild und einmal als ein ByteBuffer. Ein ByteBuffer ist eine Datenstruktur, in denen Werte als Bytes eingereiht werden können. Diese Werte werden listenartig gespeichert.

Um überhaupt eine Aufnahme tätigen zu können, müssen erstmal Voreinstellungen und Initialisierungen durchgeführt werden. Auf diese werde ich an dieser Stelle jedoch nicht näher eingehen. Für nähere Erläuterungen sei auf die Kommentierung im Quellcode verwiesen.

Wurden alle Vorbereitungen im Code getroffen, so existieren Instanzen von VideoFrameRef sowie VideoStream. In dem Quellcode heißen die Instanzen „vRef“ und „vStream“. Der Code, der die Aufnahme erzeugt, sieht wie folgt aus.

```
vRef=vStream.readFrame();
depthBuffer=vRef.getData().order(ByteOrder.LITTLE_ENDIAN);

width=vRef.getWidth();
height=vRef.getHeight();

imagePixels=new int[width*height];

calcHistogram(depthBuffer);
depthBuffer.rewind();
int pos = 0;

while(depthBuffer.remaining() > 0)
{
    int depth = (int)depthBuffer.getShort() & 0xFFFF;
    short pixel = (short)histogramm[depth];
    imagePixels[pos] = 0xFF000000 | (pixel << 16) | (pixel << 8);
    pos++;
}

image=new BufferedImage(width, height,BufferedImage.TYPE_INT_RGB);
image.setRGB(0, 0, width, height, imagePixels, 0, width);

vStream.stop();
```

Abbildung 45. Codeblock zum Erzeugen einer Aufnahme mit der Asus Kamera.

Zunächst wird ein Frame aus dem Datenstrom gelesen und dem vRef zugewiesen. vRef enthält nun die Daten der Aufnahme. Diese müssen nun extrahiert und in eine für uns nutzbare Form gebracht werden. D.h. die Daten der Aufnahme müssen aufbereitet werden. Dazu werden die Daten aus dem Frame gelesen und der Variable „depthBuffer“ als ein ByteBuffer übergeben. In diesem Buffer sind nun alle Daten der Aufnahme in linearer Form enthalten.

Die lineare Form ist dabei jedoch problematisch, da hierbei nicht klar ist, welche Daten in der Liste welchem Pixel der Aufnahme zuzuweisen sind. Um dies zu umgehen liest man zunächst die Dimensionen der Aufnahme aus. Dafür nutzt man die getWidth() und die getHeight() Methoden der Klasse VideFrameRef. Die Dimensionen werden dann in den jeweiligen Variablen festgehalten, im Code sind das die width und height Variablen. Anschließend wird durch die Methode calcHistogramm ein Histogramm berechnet. Das Histogramm ist dabei ein Array, in dem alle Tiefenwerte und ihre Häufigkeit im Bild gespeichert wurden. Dieses Histogramm wird für die visuelle Darstellung des Tiefenwertbildes benötigt. Die Funktionsweise von „calcHistogram“ wird später erklärt. In der nächsten while Schleife werden die Tiefenwerte aus dem Buffer ausgelesen und deren Farbwerte ermittelt. Die Tiefenwerte werden einfach aus dem Buffer als Byte ausgelesen in einen Integer umgewandelt und in der Variable depth gespeichert. Der Farbwert, wird durch den Tiefenwert depth aus dem Histogramm ausgelesen und in der Variable Pixel gespeichert. Anschließend wird der Farbwert des Pixels umgeformt und in das Array imagePixels eingefügt. Dies wird solange wiederholt, bis der Buffer leer ist.

Wenn alle Daten auf diese Weise verarbeitet worden sind, so wird ein neues BufferedImage aus den vorher gespeicherten Dimensionen und dem Array imagePixels erzeugt. Dieses BufferedImage wird in der Variable image gespeichert. In image ist nun das Tiefenwertbild für die Vorschau gespeichert.

Im Anschluss kann der Datenfluss durch „vSteam.stop();“ geschlossen werden, da der Fluss bis zur nächsten Aufnahme nicht mehr benötigt wird.

Die „calcHistogram“ Methode berechnet aus einem ByteBuffer ein Histogramm. Ein Histogramm gibt die Verteilung von Werten an, bzw. wie oft welcher Wert vorkommt. In der folgenden Abbildung ist der Quellcode dieser Methode zu sehen.

```
private void calcHistogram(ByteBuffer depthBuffer)
{
    histogramm = new float[vStream.getMaxPixelValue()];
    int points = 0;
    while (depthBuffer.remaining() > 0) {
        int depth = depthBuffer.getShort() & 0xFFFF;
        if (depth != 0) {
            histogramm[depth]++;
            points++;
        }
    }

    for (int i = 1; i < histogramm.length; i++) {
        histogramm[i] += histogramm[i - 1];
    }

    if (points > 0) {
        for (int i = 1; i < histogramm.length; i++) {
            histogramm[i] = (int) (256 * (1.0f - (histogramm[i] / (float) points)));
        }
    }
}
```

Abbildung 46. Implementierung der "calcHistogram" Methode.

Die Grundidee ein Histogramm zu erstellen besteht daraus, ein Array anzulegen, das so groß ist wie der größte Pixelwert der im Bild vorkommt und diese zu befüllen.

Um das Array zu befüllen, werden die Daten, in diesem Fall der ByteBuffer, der in der Variable depthBuffer gespeichert ist, durchzulaufen und den Wert der an der x. Stelle im Array ist um 1 zu erhöhen, wobei x dem Tiefenwert entspricht und der Inhalt des Feldes mit dem Index x, dessen Häufigkeit angibt. Wenn man beispielsweise die Tiefenwertfolge [5,2,10,30] im ByteBuffer hätte, so würde man das erste Element aus dem Buffer untersuchen und feststellen, dass der Wert 5 beträgt. Man würde jetzt also das 5. Feld im Array um 1 inkrementieren. Als nächstes käme die 2, nun müsste man das 2. Feld im Array um 1 inkrementieren. Bei der 10 müsste das 10. Feld und bei der 30, das 30. Feld im Array inkrementiert werden. Dies alles geschieht in der ersten While schleife im Code.

In der letzten Schleife wird die Struktur so umgeformt, dass statt den absoluten Häufigkeiten, ein Farbwert entsprechend der Prozentualen Häufigkeit eines Tiefenwertes angegeben wird. Das heißt, würde ein Tiefenwert nur 1% des Bildes ausmachen, so würde diesem Tiefenwert ein dunkler Farbton zugewiesen. Wenn ein Tiefenwert 50% des Bildes ausmachen würde, so würde diesem Tiefenwert ein sehr heller Farbton zugewiesen werden.

Außerdem ist anzumerken, dass die Methode als „private“ gekennzeichnet ist, da keine andere Klasse diese Funktionalität in Anspruch nehmen müsste. Somit ist es besser diese Methode im globalen Bereich unsichtbar zu halten.

Die Methode „getScan(Matrix3 rotMat)“ erzeugt anhand des zuvor erzeugten ByteBuffers und der übergebenen Drehmatrix einen Scan. Dazu wird lediglich der ByteBuffer durchlaufen und dabei jeder Wert in einen Vektor umgewandelt. Diese Vektoren werden anschließend mit der Drehmatrix rotiert und in einer Liste gespeichert. Der Quellcode ist in der nächsten Abbildung zu sehen.

```
public Scan getScan(Matrix3 rotMat)
{
    ArrayList<Vector3> vertices=new ArrayList<Vector3>();
    int vertexCounter=0;
    int hCounter=height;
    Vector3 vertex=new Vector3();

    depthBuffer.rewind();

    while(depthBuffer.remaining()>0)
    {
        vertexCounter++;
        float value =depthBuffer.getShort() ;

        vertex.setValue(0, (int)vertexCounter-width/2);
        vertex.setValue(1, (int)hCounter-height/2);
        vertex.setValue(2,(int)value/10);

        if(value>0)
        {
            vertices.add(rotMat.multiply(vertex));
        }

        if(vertexCounter>width-1)
        {
            hCounter--;
            vertexCounter=0;
        }

    }
    return new Scan(vertices);
}
```

Abbildung 47. Implementierung der "getScan" Methode.

Am Anfang der Methode wird eine Liste erzeugt um die Vertices darin speichern zu können. Außerdem wird die Breite der zuvor aufgenommenen Aufnahme in einem Zähler, hier „hCounter“ benötigt, um zu merken wann ein Zeilenumbruch im Scan stattfinden muss. Dies ist nötig, da der ByteBuffer Daten linear speichert. Zusätzlich wird noch ein zweiter Zähler, „vertexCounter“ benötigt, der die bereits in einer Zeiler verarbeiteten Vertices zählt. Außerdem wird ein Vektor erzeugt in dem die Koordinaten des aktuellen Vertex gespeichert werden.

Das Problem besteht darin, die X-, Y- und Z-Koordinaten der Vertices zu bestimmen. Die Z-Koordinate entspricht den Tiefenwerten aus dem Buffer. Diese können einfach ausgelesen werden. Die X- und Y-Koordinaten müssen manuell bestimmt werden.

Um diese beiden Koordinaten zu bestimmen, kann man gebrauch aus den Dimensionen der Aufnahme machen. Wenn bekannt ist wie Hoch und Breit die Aufnahme ist, so weiß man auch wie viele Vertices pro Zeile in das Bild passen. Ist eine Tiefenwertaufnahme beispielsweise 10x10 Pixel groß, so ist bekannt, dass pro Zeile 10 Vertices existieren müssen. Dasselbe gilt auch für die Spalten.

Zwei Zähler müssen also jedes Mal, wenn ein Wert aus dem Buffer gelesen wird, mitzählen an welcher Position sich der aktuelle Vertex gerade im Bild befindet. Ein Zähler müsste festhalten in welcher Zeile und der andere Zähler in welcher Spalte, sich der aktuelle Vertex befindet. Diese Zähler sind „hCounter“ für die Zeilen und „vertexCounter“ für die Spalten.

Bei jedem Durchlauf der Schleife wird ein Wert aus dem Buffer gelesen. Gleichzeitig wird der vertexCounter hochgezählt. Ansschließen wird geprüft ob der vertexCounter größer ist als die Breite des Bildes. Falls ja, heißt es, dass der vertexCounter am Bildrand angekommen ist, bzw. dass eine Zeile am Ende ist und eine neue anfängt, also wird der vertexCounter dann auf null gesetzt. Gleichzeitig wird

der hCounter um 1 verringert, da eine Zeile bereits abgearbeitet wurde. Auf diese Weise entspricht die Y-Koordinaten des Punktes dem hCounter und die X-Koordinate dem vertexCounter.

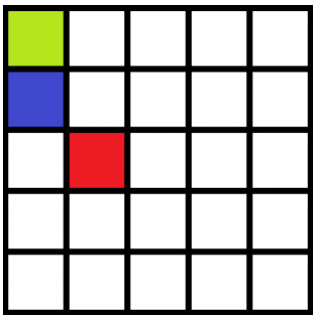

Abbildung 48. Beispielhaftes Scanszenario, in dem die ausgelesenen Werte, einzelnen Pixeln zugewiesen werden müssen.

In der Abbildung 48. ist ein Beispiel mit einem 5x5 Bild zu sehen. Die Höhe des Bildes entspricht also 4, genau wie die Breite des Bildes (in der Informatik wird ab 0 gezählt). Der hCounter wird gleich der Höhe des Bildes, bzw. 4, gesetzt. Der erste Wert aus dem Buffer wird ausgelesen. Da bisher noch keine Werte ausgelesen wurden, hat der vertexCounter den Wert 0. Der Erste Wert gehört also an die Position [hCounter, vertexCounter] bzw. [4,0]. Die Position [4,0] entspricht also der grünen Zelle. Anschließend würde der zweite Wert ausgelesen werden und der vertexCounter würde um 1 hochgezählt werden. Folglich wäre die Position für den zweiten Wert [4,1]. Dies würde so weitergehen bis zum sechsten Wert. Beim sechsten Auslesen würde der vertexCounter auf 5 erhöht werden. 5 ist dabei größer als die Breite des Bildes 4, also würde der vertexCounter zurückgesetzt werden auf 0 und gleichzeitig würde der hCounter sich um 1 verringern und auf 3 springen. Das heißt der 6. Wert würde in die Position [3,0] gehören, bzw. die blaue Zelle. Der 12. Wert aus dem Buffer würde folglich in die rot markierte Zelle gehören.

4.3.4 Assembler

Der Assembler sammelt alle Scans und fügt sie anschließend als eine fertige 3D-Datei im ply Format zusammen. Diese Scans werden in einer ArrayList<Scan> gesammelt. Um die Scans entgegen zunehmen, bietet diese Klasse die Methode „addScan(Scan scan)“. In der Methode wird der übergebene Scan in die Liste eingefügt und die Gesamtzahl der Vertices wird hochgezählt. Diese Information wird später benötigt um die ply Datei zu generieren. Der Quellcode ist auf der folgenden Abbildung zu sehen.

```
public void saveModel(String name) throws FileNotFoundException,
        UnsupportedEncodingException {

    //Erstellung der .ply Datei mit der Variable "name" als Dateiname
    writer = new PrintWriter(name + ".ply", "UTF-8");

    //Erstellung des Headers mit allen nötigen Informationen für das 3D-Model
    writer.println("ply");
    writer.println("format ascii 1.0");
    writer.println("element vertex " + vertexCount);//Einfügen der Gesamtanzahl der Vertices
    writer.println("property float x");
    writer.println("property float y");
    writer.println("property float z");
    writer.println("end_header");

    //Durch die gespeicherten Scans iterieren
    for (Scan s : scans)
    {
        //Vertexliste des aktuellen Scans merken
        ArrayList<Vector3> vertices = s.getVertices();

        //Durch die gemerkte Vertexliste iterieren
        for (Vector3 v : vertices)
        {
            //3D Position des aktuellen Vertex als String in die .ply Datei schreiben
            writer.println(v.toString());
        }
    }

    //Writer schließen
    writer.close();
}
```

Abbildung 49. Implementierung der "saveModel" Methode.

Die Methode „saveModel(String name)“ sorgt für das Speichern des Models. In dieser Methode wird zunächst ein PrintWriter erzeugt. Diese Klasse ist eine Java Klasse, sie bietet die Möglichkeit Dateien zu erzeugen und zu beschreiben.

Nachdem eine Datei erzeugt wurde, wird zunächst der Header der ply Datei geschrieben. Anschließend folgt eine doppelte for-Schleife. Die erste Schleife iteriert dabei über alle Scans und hält die Liste der Vertices des aktuellen Scans fest. Die zweite Schleife iteriert über diese Vertex Liste und schreibt die Koordinaten der einzelnen Vertices zeilenweise in die Datei.

Wenn beide Schleifen terminieren, ist die Datei fertig beschrieben und der PrintWriter kann geschlossen werden.

4.3.5 Controller

Der Controller beinhaltet den Assembler sowie den Scanner und steuert diese. Der Controller bietet lediglich drei simple Methoden, da die einzige Aufgabe dieser Klasse darin besteht, dem MainFrame eine kompakte Schnittstelle zu den anderen Klassen zu bieten.

Die erste Methode ist die „addToAssembler(int degrees)“ Methode. Durch diese Methode können die durch den Scanner erzeugten Scans im Assembler zwischengespeichert werden. Diese Methode nimmt eine Winkelgröße in Grad entgegen und erzeugt aus dieser Angabe zusammen mit der statischen Methode „getRotMatX(int degrees)“ der Klasse Matrix3 eine Drehmatrix. Danach holt sich der Controller einen Scan vom Scanner indem er die Matrix dem Scanner übergibt. Diesen Scan leitet er wiederrum in die „addScan(Scan scan)“ Methode des Assemblers, damit dieser den Scan zwischenspeichert.

Die zweite Methode ist die „saveModel(String filename)“ Methode. Diese dient dazu den Assembler davon in Kenntnis zu setzen, dass er alle Scans in einer Datei speichern soll. Dafür nimmt die Methode den gewünschten Dateinamen als Parameter entgegen und leitet diesen weiter in die „saveModel(String filename)“ Methode des Assemblers.

Die letzte Methode „takeImage()“ dient dazu, das Vorschaubild aus dem Scanner zu entnehmen. Dazu wird der Scanner aufgefordert eine neue Aufnahme zumachen. Anschließend wird durch die „getImage()“ Methode des Scanners, das Vorschaubild gefordert und zurückgegeben.

Die komplette Implementierung dieser Methoden ist in der folgenden Abbildung zu sehen.

```
public void addToAssembler(int degrees)
{
    Matrix3 rotMat= Matrix3.getRotMatZ(degrees);
    Scan scan= scanner.getScan(rotMat);
    assembler.addScan(scan);
}

public void saveModel(String fileName)
{
    try {
        assembler.saveModel(fileName);
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (UnsupportedEncodingException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

public BufferedImage takeImage()
{
    scanner.shootFoto();
    return scanner.getImage();
}
```

Abbildung 50. Implementation Controller Klasse.

4.3.6 MainFrame

Diese Klasse stellt die Benutzeroberfläche dar. Neben einer Vielzahl an Steuerelementen besitzt diese Klasse eine Instanz des Controllers. Die einzige Aufgabe dieser Klasse besteht darin, die Eingaben des Nutzers in den Controller weiterzuleiten, damit dieser die Daten an die zuständigen Klassen weiterleiten kann.

Die Implementierung der Steuerelemente weiter zu erläutern, wäre zu langwierig und redundant. Daher sei an dieser Stelle auf den beiliegenden Quellcode verwiesen.

5 Diskussion und Ausblick

5.1 Zusammenfassung

Im Rahmen dieser Arbeit, sollte ein 3D-Scanner in Verbindung mit der „Asus Xtion Pro Live“ Kamera entwickelt werden. Dieser Scanner soll in erster Linie in Sanitätshäusern zur Optimierung von Orthesen genutzt werden. Dazu wurde zunächst der Stand der Technik betrachtet und geklärt inwiefern bzw. wieso die „Asus Xtion Pro Live“ Kamera als 3D-Scanner in Frage kommt.

Anschließend wurden Vorbereitungen für die Entwicklung der Software getroffen. Es musste entschieden werden welche Programmiersprache und welche IDE zum Einsatz kommen sollten. Dabei fiel die Entscheidung auf die Programmiersprache Java und die IDE Eclipse.

Das Softwaregrundgerüst wurde durch verschiedenen Methoden aus dem Software Engineering geplant und skizziert. Als Grundgerüst kam dabei das MVC-Pattern zum Einsatz. Mit dieser Entscheidung ausgestattet, wurden die einzelnen Klassen im Detail geplant und entworfen.

Abschließend wurden die Klassen implementiert, bzw. deren Implementation vorgestellt und erklärt.

5.2 Evaluation

Es lässt sich sagen, dass die Aufgabe letztlich hinreichend erfüllt wurde.

Bei der Entwicklung ist eine Anwendung entstanden, die es ermöglicht 360° Scans von Objekten anzufertigen, so wie es entsprechend der Aufgabenstellung gefordert war.

Problematisch war jedoch die Realisierung der 360° Scans. Die Kamera war von sich aus bereits in der Lage Tiefenwertbilder aus einer bestimmten Perspektive aufzunehmen. Es mussten lediglich die Daten aus der Aufnahme gelesen und umgeformt werden. Dazu war eine gewisse Einarbeitungszeit in die Bibliothek OpenNi nötig, um dessen Klassen richtig verstehen und nutzen zu können. Nach dieser Einarbeitungszeit war ich in der Lage 3D-Aufnahmen aus einer einzigen Perspektive anzufertigen. Dies entsprach jedoch nicht der Forderung von 360° Scans. Ich musste also einen Weg finden diese 360° Scans ohne viel Programmieraufwand zu realisieren.

Nach langem Versuchen bin ich zu dem Schluss gekommen, dass es die einfachste Möglichkeit wäre, einzelne Aufnahmen aus verschiedenen Perspektiven, zu einer 360° Aufnahme zu vereinen. Nachdem ich die Software unter Anwendung dieser Idee zu Ende geschrieben hatte, war ich lange Zeit unsicher, ob diese Variante ausreichend wäre.

Anschließend habe ich angefangen die Arbeit zu schreiben. Während der Recherche ist mir aufgefallen, dass der „Fuel 3D“ Scanner auf eine ähnliche Weise funktioniert wie der Scanner den ich geschrieben habe.

Dadurch kann ich abschließend sagen, dass die Aufgabenstellung hinreichen erfüllt sein sollte.

5.3 Ausblick

Die Anwendung ist bereits Nutzbar. Das heißt jedoch bei weitem nicht, dass sie fertig ist. Die Software soll in Zukunft auf Android basierten Betriebssystem Lauffähig sein, um die Mobilität des Scanners zu steigern.

Das heißt die gesamte Anwendung muss auf Android portiert werden. Dazu muss zunächst eine Android Anwendung geschrieben werden. Anschließend muss der Code des Scanners in diese Anwendung

eingebettet werden. Von Vorteil ist dabei, dass der Code des Scanners in Java geschrieben wurde. Das erleichtert den Prozess der Portierung.

Außerdem muss dafür gesorgt werden, dass die Asus Kamera eine Verbindung zu Android Geräten aufbauen kann und auch von der Android Anwendung aus ansprechbar ist.

In der finalen Version soll dieser Scanner zusammen mit einer Android Anwendung in Sanitätshäusern zum Einsatz kommen.

Abgesehen von der Android Version, wird auch an einer anderen Version gearbeitet. Diese Version verbindet den Scanner mit einer Drehscheibe auf dem das zu scannende Objekt gedreht wird. Dadurch sollen 360° Scans von Objekten automatisch aufgenommen werden können, ohne dass der Benutzer manuell mehrere Aufnahmen machen muss. Dazu muss die Software leicht abgeändert werden. Da ich bereits bei der Entwicklung wusste, dass an dieser Version gearbeitet wird, habe ich die Software so geschrieben, dass eine Umstellung problemlos möglich sein sollte.

Literaturverzeichnis

[BALZ2009] Helmut Balzert: „Lehrbuch der Softwaretechnik - Basiskonzepte und Requirements Engineering“ (2009)

[BALZ2011] Helmut Balzert: „Lehrbuch der Softwaretechnik - Entwurf, Implementierung, Installation und Betrieb“ (2011)

Ian Sommerville: „Software Engineering 9“ (2010)

Julie Reece: „Trends in 3D Scan-to-Print Applications“ (23.3.2015) – URL: http://mcortechnologies.com/trends-3d-scan-print-applications-blog/ (abgerufen am 17.10.2015)

Larry Li: „Time-of-Flight Camera – an Introduction“ (2014) – URL: http://www.ti.com/lit/wp/sloa190b/sloa190b.pdf (abgerufen am 8.10.2015)

[NITZ2014] Stefan Nitz: „3D-Druck“ (2014)

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: „Design Patterns: Elements of Reusable Object-Oriented Software“ (1994)

[VINA 2011] Victor Castaneda, Nassir Navab: „Time-of-Flight and Kinect Imaging“ (2011) – URL: http://campar.in.tum.de/twiki/pub/Chair/TeachingSs11Kinect/2011-DSensors_LabCourse_Kinect.pdf (abgerufen am 8.10.2015)