Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Entwurf eines virtuellen Umkleideraums mittels Microsoft Kinect und OpenGL

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Vorgelegt von: Daniel Schröder – 1960363
am: 28.10.2015 in Hamburg

Studiengang: Media Systems
Fakultät: Design, Medien und Information
Department: Medientechnik

Erstbetreuer: Prof. Dr. Andreas Plaß
Zweitbetreuer: Prof. Dr. Edmund Weitz

Zusammenfassung

Im Rahmen meiner Bachelorarbeit habe ich mich mit dem Zusammenspiel verschiedener Schnittstellen zur Anwendungsprogrammierung beschäftigt, mit dem Ziel eine Grundlage und einen Ausgangspunkt für eine virtuelle Darstellung eines Umkleideraums zu schaffen.
Dabei liegt der Fokus nicht unbedingt auf der Komplexität und Beschaffenheit des finalen Programms, sondern vielmehr ruht die Aufgabe auch darin, drei unterschiedliche Programmierschnittstellen funktionell zu verbinden: So wird die Microsoft Kinect v1 als Kamera eingesetzt, dessen geliefertes Bild später von der Rendering-Bibliothek OpenGL weiterverarbeitet wird. Darüber hinaus wird auf das Qt-Framework zur generellen Darstellung der grafischen Oberfläche und Bedienung des Menüs zurückgegriffen.
Nach einer Einleitung in das Szenario folgt eine konzeptionelle Annäherung an die drei thematischen Säulen dieser Arbeit, bei der zum Teil auf eine allgemeine Einführung nicht verzichtet werden kann, damit entscheidende Bestandteile der anschließenden Konzeptfindung sowie Programmierumsetzung verständlich erscheinen.

Abstraction

The subject of my bachelor thesis deals with the possibilities of merging different software interfaces, aiming at the creation of a starting point for a virtual dressing room. The underlying task is not solely to design a complex and feature-rich conclusive product, but rather the challenging process of combining three diverse APIs into one functional program: Microsofts first Kinect device is used to capture image stream data, while the rendering library OpenGL is implemented to process the frames afterwards. In order to obtain a working graphical user interface, Qt acts the fundamental core framework.
After an introduction to the scenario follows a conceptual approach to the three thematic pillars of the thesis, where a general approach is mandatory, so that crucial parts of the subsequent specification of concept and programming appears comprehensible.

Inhaltsverzeichnis

1 Einleitung

Die Einleitung in die Bachelor-Thesis beginnt mit einer Darlegung der Motivation, die einerseits die Motivation aus der Schilderung des Szenarios heraus zeigt, aber auch anschließend die persönliche Motivation für die Wahl dieses Themas erklärt.

Die Einleitung mündet in einer knappen Vorstellung der Hardware- und Software-Spezifikationen, die den Rahmen für das Projekt festlegen.

1.1 Motivation

Das Kaufen von Kleidung in Geschäften vor Ort sowie auch online ist ein alltägliches Ereignis.

Während die Online-Kundschaft gezwungen ist, sich auf die Größenangaben der Hersteller zu verlassen, wählt der Offline-Kunde oftmals den Weg des Anprobierens, um festzustellen, ob das ausgewählte Produkt seinen Körpermaßen, beziehungsweise den geschmacklichen Vorstellungen entspricht. Bei diesem Prozess können jedoch folgende Probleme auftreten:

- Durch die begrenzte Anzahl der Umkleideräume können sich, insbesondere zu Haupteinkaufszeiten, Warteschlangen bilden, welche
  - den Kunden möglicherweise davon abhalten, sich einzureihen und damit überhaupt etwas zu kaufen, was wiederum eine entgangene Umsatzchance für das Unternehmen bedeutet.
- Viele Kaufhäuser setzen, um dem soeben erwähnten Problem entgegenzuwirken, eine Grenze für die erlaubte Menge an Kleidungsstücken, die mit in die Kabine genommen werden dürfen, was unter Umständen dazu führen kann, dass der Kunde aus der Menge der ursprünglich ausgewählten Zielmenge an Produkten lediglich eine Teilmenge als potenziellen Kauf auswählt.

Ebenso ist in Betracht zu ziehen, dass die Kundenzufriedenheit durch diesen eventuell problematischen Ablauf leidet, und somit auch die generelle Bereitschaft etwas zu kaufen, sinken kann.

Verallgemeinert man das Szenario, führt die nicht vorhandene Skalierbarkeit seitens der Kaufhäuser dazu, dass beide Seiten, Kundschaft und Unternehmen, sich in einer Verlustsituation befinden, welche mit dem Einsatz erschwingbarer Technologie durchaus vermeidbar wäre.

Im Rahmen meines Studiums hatte ich bereits Berührungspunkte mit Computer Vision-Bibliotheken wie OpenCV oder aber auch erste Erfahrungen mit WebGL über die Bibliothek three.js gesammelt. Da ich seit geraumer Zeit im Besitz der Kinect bin, entstand eines Tages die Idee, all diese Elemente in einem Projekt zu verbinden und eine Anwendung entwickeln, die eventuell sogar vom Grundgedanken her auf realistische Systeme anwendbar wäre.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es, eine funktionsfähige Anwendung zu entwerfen, die eine Basis für komplexere Software-Lösungen des unter Kapitel 1.1 beschriebenen Problems darstellt und die in der Theorie nach Belieben erweitert werden könnte.
Die grundlegenden Anforderungen an die Anwendungen sind somit:

- Erfassen (zunächst) einer Person, die sich vor dem Kameraobjekt befindet
- Auslesen der Raumkoordinaten, um eine Projektion zu ermöglichen
- Das Ermöglichen einer Interaktion zwischen Person und Programm
- Ausgabe einer Anzeige, auf der die Person zu sehen ist
- Projektion von Kleidungstexturen auf die Person

Ausgehend von diesen Rahmenbedingungen, ist die Auswahl an den unterschiedlichen, zur Verfügung stehenden Software-Bibliotheken sowie Hardware-Spezifikationen bereits eingeschränkt.

1.3 Hardware- und Software-Spezifikationen

Damit eine Realisierung der Anforderung überhaupt möglich ist, muss sichergestellt sein, dass

die einzelnen Komponenten des Projekts miteinander kompatibel sind.

Der erste Schritt ist sinnvollerweise die Wahl der Hardware, sprich in diesem Fall der Kamera.

Da es eine Kamera mit einem zusätzlichem Tiefensensor sein musste und ich bereits im Besitz der Kinect war, konnte diese Entscheidung schnell getroffen werden.

Die Software-Spezifikationen sind derweil:

- IDE:
  - Qt Creator 3.4.2
- Schnittstellen:
  - Kinect SDK v1.8
  - Qt 5.5
- Compiler:
  - MSVC 2010 32-bit

Das anschließende Kapitel beschäftigt sich mit der Analyse der einzelnen Komponenten dieser Arbeit.

2 Analyse

Nachdem die Einleitung einen groben Eindruck vermittelt hat, so soll sich dieses Kapitel mit einer näheren Betrachtung der einzelnen Schnittstellen beschäftigen.

Hierzu wird zuerst die Kinect und die zugehörige API kurz vorgestellt, gefolgt von einer groben Vorstellung der Open Graphics Library. Den dritten Part stellt die Analyse des Qt-Frameworks dar, welches vor Allem auf die für diese Anwendung nützlichen Elemente untersucht wird.

2.1 Die Kinect

Die Microsoft Kinect v1 (v1, Version 1 auf Grund der Tatsache, dass im Jahr 2013 mit der Xbox One die Version 2 erschienen ist; im Anschluss lediglich mit *Kinect* referenziert), erschien im November 2010 für die Xbox 360 und wurde ursprünglich am 1. Juni 2009 auf der Spielemesse E3 unter dem damaligen Titel *Project Natal* der Öffentlichkeit vorgestellt.

Bereits innerhalb kürzester Zeit wurde die Kinect laut Guiness World Records zu dem sich am schnellsten verkaufenden elektronischen Gerät.[1]

Zu einem Einstiegspreis von damals 149USD wurde der breiten Masse der Zugang zu einer damals in diesem Preissegment nicht vor findbaren Technologie ermöglicht.

2.1.1 Technische Eckdaten

Die Kinect besitzt neben einem RGB-Sensor zusätzlich einen Tiefensensor und eine Infraroteinheit, welche sich aus *Emitter* und *Receiver* zusammensetzt (siehe Abbildung 1: Technische Einheiten der Kinect).

Kommt die Infraroteinheit zum Einsatz, so sendet der *Emitter* Licht aus und der *Receiver* empfängt daraufhin die eingehenden Reflektionen, sodass vor dem Sensor sich befindliche Objekte sichtbar gemacht werden können.

1 Vgl. Catuhe, David (2012): „Programming with the Kinect for Windows Software Development Kit“, S. 3

Darüber hinaus befinden sich an der unteren Seite der Kinect vier Mikrofone, die unter anderem zur Sprachsteuerung eingesetzt werden können, sowie ein motorisierter Neigungsstand, der ein Neigen von bis zu [+27, -27] Grad ermöglicht.

Der 2011 erschienene Kinect for Windows-Sensor muss, im Unterschied zur Xbox 360-Version, nicht über eine externe Stromversorgug angeschlossen werden und verfügt über einen *Near Mode*, der für einen Abstand von 0.4m – 0.8m geeignet ist (die Xbox 360-Version misst erst ab 0.8m Abstand genau).

Abbildung 1: Technische Einheiten der Kinect

2.1.2 Die Kinect SDK

Nachdem die Kinect bereits kurze Zeit nach ihrer Veröffentlichung gehackt wurde und mit der Zeit diverse Bibliotheken wie OpenNI, NITE oder libfreenect erschienen, brachte Microsoft mit der Kinect SDK im Frühjahr 2012 ein eigenes Software Development Kit für ihr Gerät auf den Markt.

Die aktuellste Version ist die Version 1.8, welche in diesem Projekt eingesetzt wird.

2.2 OpenGL

Die Open Graphics Library ist vor Allem als Synonym für eine plattformunabhängige 3D-API für den Desktopbereich bekannt. Dabei ist OpenGL schon lange keine einzelne API mehr, sondern hat sich über die Jahre viel mehr zu einer API-Familie entwickelt: Neben dem altbekannten OpenGL für den Desktop hat sich mittlerweile auch OpenGL ES *(OpenGL Embedded Systems)* zu einem Standard für 3D-Programmierung für den mobilen Markt, soll heißen Tablets, Smartphones aber auch exotischeren Umgebungen wie Set-Top-Boxen oder Autos, entwickelt.[2]

Darüber hinaus rückt WebGL, eine 3D-API, die auf OpenGL ES basiert, immer mehr in den Vordergrund der Programmierung in Browserumgebungen, indem eine plattformübergreifende Entwicklung schneller und visuell beeindruckender Inhalte ermöglicht wird.

2 Vgl. Cozzi, Riccio (2012): „OpenGL Insights“: S. xxi [Vorwort]

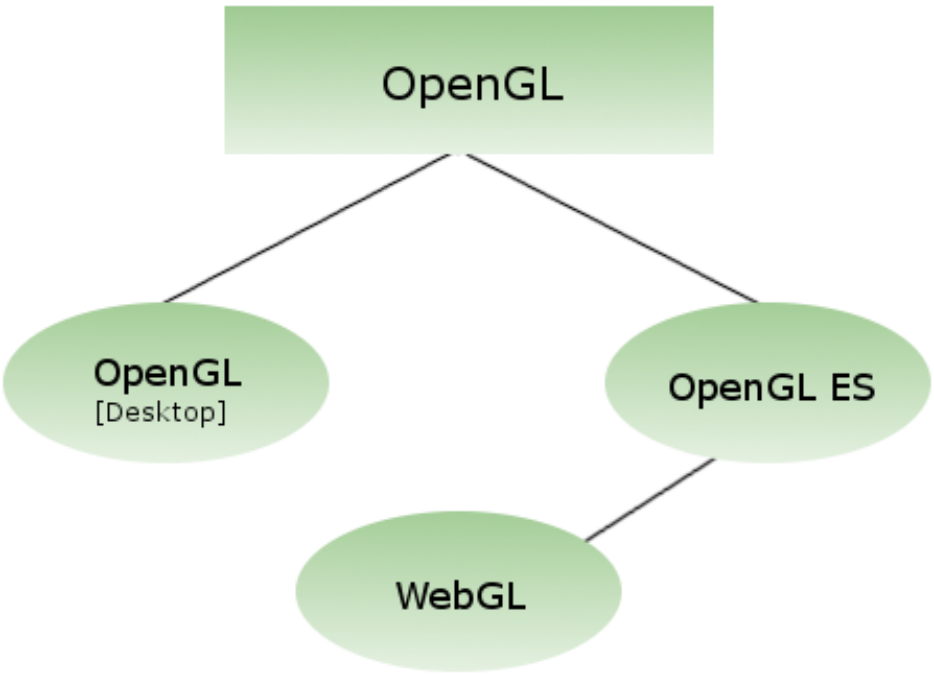

Abbildung 2: Die OpenGL-Familie

Im Anschluss wird knapp das Grundprinzip der 3D-Programmierung und der Werdegang des Rendering-Mechanismus von OpenGL dargelegt, um die Entscheidung für die Verwendung einzelner Bausteine aus dem gewaltigen Baukasten, den OpenGL zur Verfügung stellt, für die Anwendung dieser Arbeit zu untermauern und verständlich zu machen.

2.2.1 Grundlagen

2.2.1.1 Der dreidimensionale Raum

Der den Menschen alltäglich begleitende dreidimensionale Raum ist mathematisch gesehen durchaus als eine Herausforderung zu betrachten. Eine ausführliche Kenntnis der

zugrundeliegenden Koordinatensysteme ebenso wie der damit gekoppelten mathematischen Operationen ist in der 3D-Computergrafik von großer Bedeutung[3]. Kern dieser Operationen ist der Umgang mit Abbildungsmatrizen, die vor Allem zur Transformation von Punkten beziehungsweise Vektoren eingesetzt werden.

Für eine ganzheitliche Betrachtung der Abbildungsabläufe ist es erforderlich, sich mit homogenen Koordinaten und dem Umgang mit 4x4-Matrizen auseinanderzusetzen. Die Mathematik hinter diesen Vorgängen ist allerdings für den Kontext dieser Arbeit nicht von direkter Relevanz und würde den Rahmen der ursprünglichen Aufgabenstellung sprengen. Daher gilt: Es wird lediglich akzeptiert und „hingenommen“, dass jegliche Drehungen, Scherungen und Translationen, also Transformationen jeglicher Art, über die Rechnungen mit homogenen Koordinaten durchgeführt werden.

2.2.1.2 Der Abbildungsvorgang

Objekte beginnen in OpenGL ihr Leben stets im Ursprung eines sogenannten Weltkoordinatensystem. Sollen Objekte an eine bestimmte Position und Orientierung in einer Szene gebracht werden, so geschieht dies über die sogenannte Modellmatrix. Sind Objekte platziert, so muss noch eine Umrechnung erfolgen, denn die Szene wird noch nicht aus der Sicht des Beobachters dargestellt. Man stelle sich einen Tisch in der Mitte des Raumes vor, den der Beobachter von seinem Standpunkt in der Zimmertür noch nicht sehen kann.

Mit Hilfe einer Kameramatrix wird die Szene auf die Sicht des Beobachters umgerechnet. Daraufhin findet eine Projektion auf eine zweidimensionale Fläche statt, wobei die Tiefeninformation beibehalten wird. Im Anschluss erfolgt das sogenannte Clipping, welches alle Objekte entfernt, die außerhalb eines bestimmten Bereichs liegen (vergleiche 2.2.1.3). Man stelle sich vor, der Beobachter sieht aus dem Fenster eines Hochhauses, von wo ihm eine gewaltige Szene zu Füßen liegt – ein enormer Aufwand für eine Berechnung. Wenn allerdings der Sichtbarkeitsbereich sinnvoll begrenzt ist, kann die Zahl der Objekte deutlich reduziert werden.

Nach dem Clipping-Vorgang liegen die dreidimensionalen Device-Koordinaten fest. Hierbei

3 Vgl. Virag, Gerhard (2012): „Grundlagen der 3D-Programmierung“, S. 37

handelt es sich um normalisierte Koordinaten im Wertebereich [-1, 1], die einen Würfel darstellen. Durch die folgende Viewport-Transformation werden diese *Normalized Device Coordinates* (NDC) auf die Dimensionen des Bildschirmausschnitts umgerechnet.

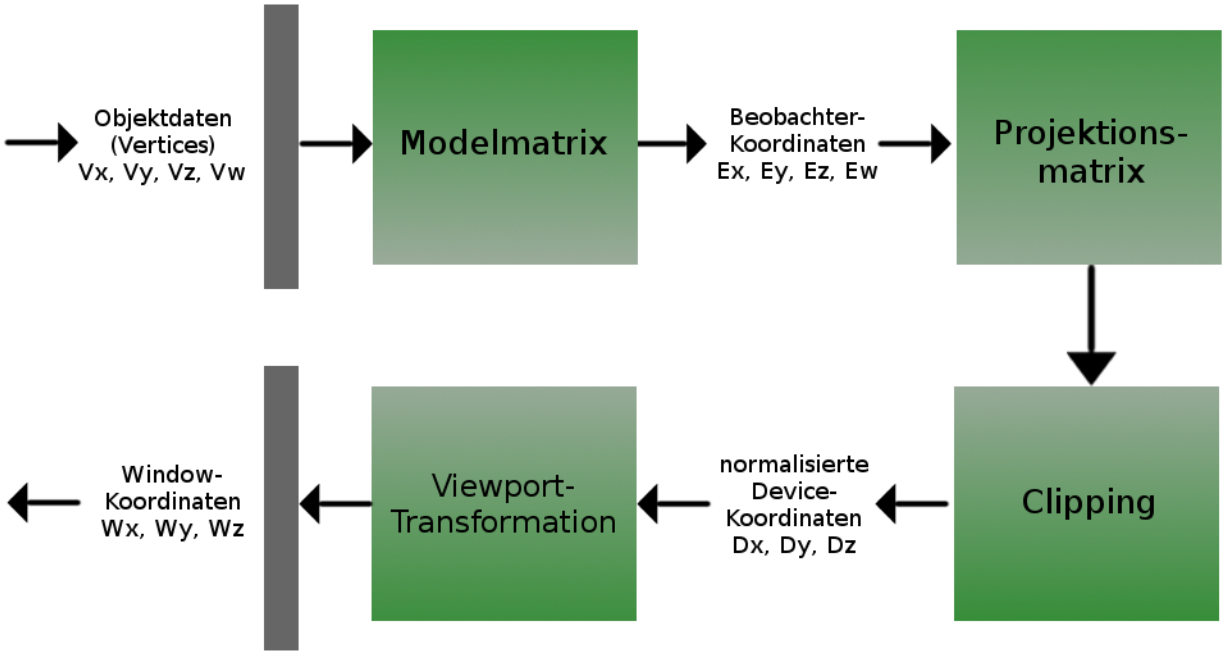

Abbildung 3: Der Weg von Objektdaten zu Fensterkoordinaten

2.2.1.3 Die Projektionstypen

Es gibt grundsätzlich zwei Arten der Projektion:

- Parallele Projektion
- Perspektivische Projektion

Der bekannteste Vertreter der parallelen Projektion ist die orthogonale Projektion. Entscheidend bei der orthogonalen Projektion ist die Tatsache, dass alle Objekte für den Betrachter gleich groß erscheinen, unabhängig davon, wie weit sie vom Betrachter entfernt sind. Das Clipping-Volumen

der orthogonalen Projektion ist ein Quader (siehe Abbildung 4).

Diese Art der Projektion eignet sich für 2D-Abbildungen, bei denen es nicht notwendig ist, realistische räumliche Eindrücke zu simulieren.

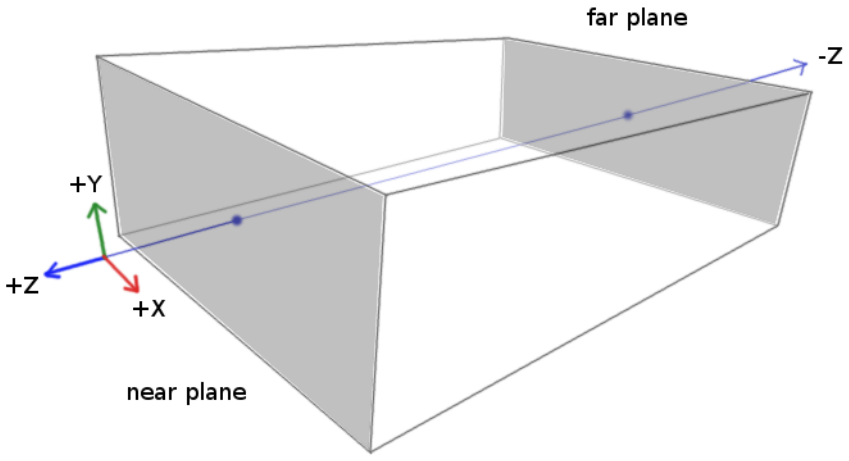

Abbildung 4: Orthogonale Projektion

Die perspektivische Projektion imitiert die Sichtweise, wie sie dem Menschen aus dem Alltag bekannt ist – Objekte erscheinen für den Beobachter mit zunehmender Distanz kleiner. Dabei ist das Clipping-Volumen kein Quader mehr, sondern ein Pyramidenstumpf (engl. *Frustum,* siehe Abbildung 5).

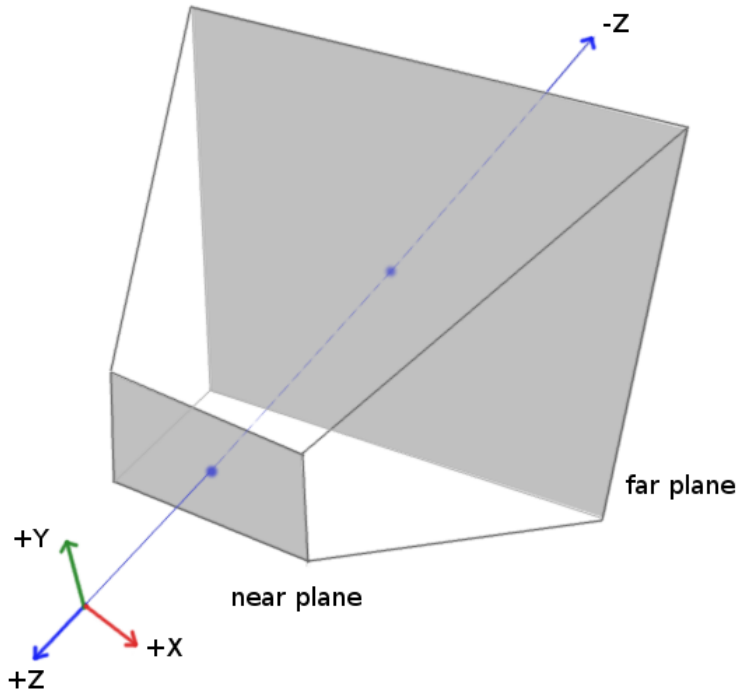

Abbildung 5: Perspektivische Projektion (Frustum)

Schlussfolgerung:

Für eine Darstellung von 2D-Texturen (wie in diesem Projekt) ist die Verwendung der orthogonalen Projektion ausreichend. Sobald ein räumlicher Eindruck möglichst realistisch dargestellt werden soll, ist der Einsatz einer perspektivischer Projektion jedoch unabdingbar.

2.2.1.4 Shader

Mit OpenGL 2.0 wurde die Shaderprogrammierung eingeführt und Shader haben seither eine fundamentale Rolle in der OpenGL-Pipeline. Shader sind eigene Programme, die in einer eigenen, C-ähnlichen Sprache, der *OpenGL Shading Language* (GLSL), geschrieben werden.

Shader laufen auf der Grafik-Hardware und dienen somit auch der Entlastung der CPU.

Die zwei wichtigsten und grundlegendsten Shader sind dabei der *Vertex Shader* und *Fragment*

Shader.

Vertex-Shader

Der Vertex-Shader übernimmt alle Matrixtransformationen für eingehende Vertices und bestimmt die Endposition von Vertices.

Fragment-Shader

Der Fragment-Shader ist zuständig für die Färbung der Pixel.

Schlussfolgerung:

Für die Anwendung sollten Shader genutzt werden, da sie die hohe Leistung der Grafikkarten nutzen können.

2.2.2 Übergabe von Objekten

Um Objekte, definiert durch ihre Mesh- und Materialdaten, an OpenGL zu übergeben, gibt es mehrere Möglichkeiten. Die modernste und schnellste Übergabemethode sind Vertex Buffer Objects (VBO)[4].

VBOs sind das mächtigste Konzept, das OpenGL zur Verfügung stellt. Dabei sind die Vertex-Daten in einem Buffer auf OpenGL-Seite hinterlegt, nachdem sie einmal in den Buffer geladen wurden – genauer genommen residieren sie im schnellen Speicher der Grafikkarte und daher ist der Zugriff auf sie für OpenGL die denkbar schnellste Variante.

Schlussfolgerung:

Um für die Anwendung eine moderne OpenGL-Programmierung zu gewährleisten, sollten Vertex Buffer Objects verwendet werden.

4 Vgl. Virag, Gerhard (2012): „Grundlagen der 3D-Programmierung“, S.166

Die Analyse von OpenGL ist hiermit abgeschlossen und es folgt die Analyse des Qt-Framework.

2.3 Das Qt-Framework

Das Qt-Framework bietet dem Anwender eine weite Palette an Optionen, um eine grafische Benutzeroberfläche zu entwerfen. Sinn dieses Abschnitts ist es, anhand der Zielsetzung die geeigneten Klassen für dieses Programm zu bestimmen und sich auch mit dem daraus ableitenden Programmfluss, etwa bezüglich Events und Callbacks zu beschäftigen.

2.3.1 Fensterwahl

Für die Ausgabe eines Hauptfensters existieren in Qt viele Möglichkeiten. Dabei gibt es verschiedene Subtypen, die allesamt von *QWindow*, der Basisklasse für Fenster, abstammen.

Die Recherche in der Dokumentation ergab eine enge Auswahl an zwei Optionen, die für den Anwendungszweck geeignet sind.

QMainWindow

Die *QMainWindow*-Klasse stellt ein Fenster zu Verfügung, welches nach Belieben mit weiteren Bauteilen, wie zum Beispiel Widgets oder Buttons ausgerüstet werden kann. Somit können mehrere Anzeigen gleichzeitig eingerichtet werden und es kann über eine Menüleiste oder Buttons eine Interaktion mit dem Benutzer hergestellt werden – *QMainWindow* besitzt ein Layout-Framework (siehe Abbildung 6), das weiter ausgebaut werden kann:

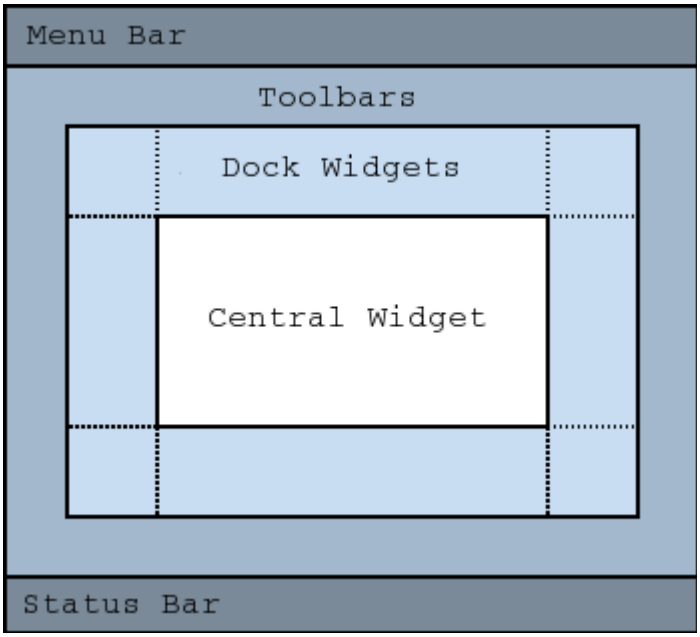

Abbildung 6: Layout von QMainWindow

QOpenGLWindow

Mit dem *QOpenGLWindow* stellt Qt einen Klasse zur Verfügung, die darauf ausgelegt ist, in einem Fenster einen OpenGL-Kontext darzustellen. Problematisch ist dabei, dass der Kontext das gesamte Fenster ausfüllt; im Endeffekt fungiert es also wie ein fensterweites, einziges Widget, ohne die Möglichkeit, weitere Elemente dem Fenster hinzuzufügen.

Abbildung 7 zeigt ein *QOpenGLWindow*, in dem ein gefärbtes Dreieck vor schwarzem Hintergrund gezeichnet wird.

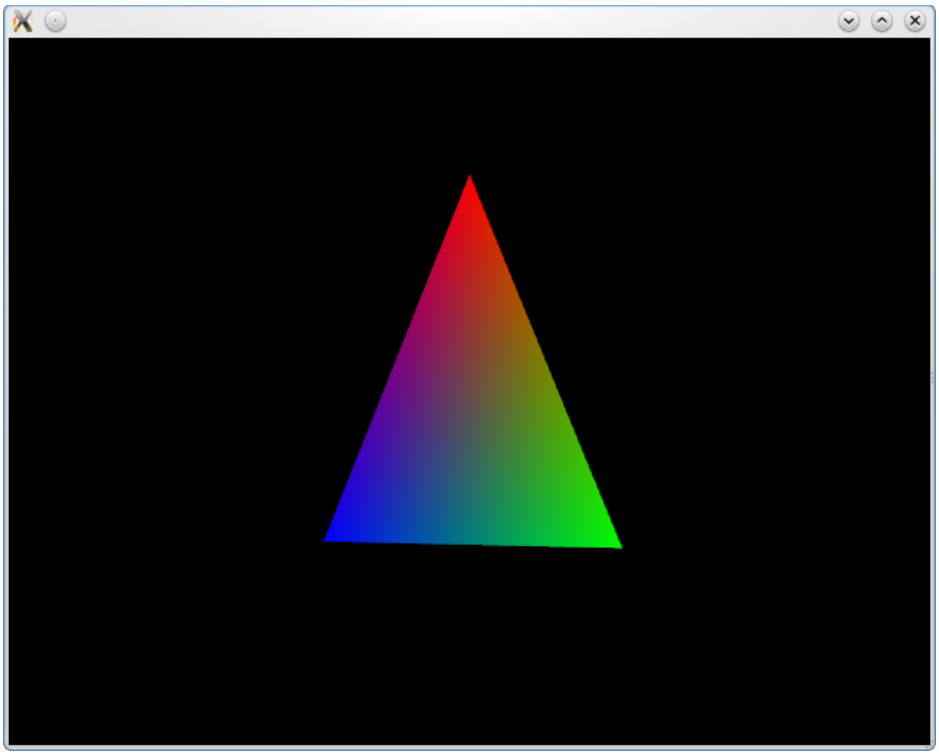

Abbildung 7: Beispielhafte Implementation von QOpenGLWindow

Schlussfolgerung:

Für die Desktop-Applikation dieser Arbeit eignet sich besonders das *QMainWindow*, da es nach eigenem Ermessen mit diversen GUI-Elementen ausgestattet werden kann.

Für eine eventuelle Portierung auf ein Kaufhaussystem käme das *QOpenGLWindow* in Frage, da für dieses System keine Maus-Tastatur-Interaktion von Nöten ist, die Elemente wie Buttons voraussetzt, und es sich nur um die Anzeige von OpenGL-Kontext handelt.

2.3.2 Image-Container

Um Bilddaten zu verarbeiten und Bilder anzuzeigen bietet Qt vier verschiedene Typen an:

QImage

Der Datentyp *QImage* wurde speziell designed und optimiert für die Ein- und Ausgabe, das bedeutet das Einlesen und Schreiben. Es erlaubt direkten Pixelzugriff und -manipulation, und unterstützt dabei viele Bildformate.

QPixmap

QPixmap wurde dahingehend optimiert, um Bilder anzuzeigen. Gewöhnlich wird dazu ein Bild vom Typ *QImage* in ein *QPixmap* konvertiert, nachdem die Bilddaten die gewünschten Modifizierungen durchlaufen hat. Soll ein Bild jedoch nur angezeigt werden, ohne das direkter Zugriff auf Rohdaten nötig ist, kann eine Bilddatei auch direkt in ein *QPixmap* geladen werden.

QBitmap

Der Bilddatentyp *QBitmap* ähnelt *QPixmap*, bis auf die Tatsache, dass er nur 1-bit Farbtiefe liefert. Somit eignet sich *QBitmap* besonders für die Kreation von Cursor-Objekten (mit Hilfe von *QCursor*) oder Bildmasken – prinzipiell überall dort, wo eine Unterteilung in Schwarz oder Weiß genügt.

QPicture

Die vierte und letzte Klasse, die Qt zur Handhabung mit Bilddaten bereitstellt, ist *QPicture*. Instantiiert man ein Objekt dieser Klasse, so kann mit Hilfe Zeichenbefehle der *QPainter*-Klasse manuell in dieses Objekt gemalt werden (wie Ellipsen oder Kreise). Darüber hinaus können auch alle Inhalte, die in die anderen drei Bilddatentypen geladen werden können, auch in *QPicture* geladen werden – *QPicture* kennt keine Limitierungen für seinen Kontext.

Schlussfolgerung:

Um die Pixelinformationen des von der Kinect gesendeten Bildes zu verwerten, muss auf den Typ *QImage* zurückgegriffen werden, da nur dieser Datentyp ein Schreiben und Lesen auf dieser Ebene ermöglicht.

Damit diese Bildinformationen final angezeigt werden können, muss dieses *QImage* im Anschluss in den Typ *QPixmap* umgewandelt werden.

2.3.3 Widget-Container

Um ein Bild in einem bestimmten Bereich eines Fensters anzeigen zu können, muss auf einen Klassentyp zurückgegriffen werden, der es ermöglicht, innerhalb eines Fenster einen Bildrahmen zu schaffen.

Für den Zweck dieser Arbeit kommen dabei vor Allem zwei Typen in Frage:

QLabel

Die Klasse *QLabel* ist per Defintion kein Widget, sondern stammt von der Klasse *QFrame* ab. *QLabel* ermöglicht das Anzeigen von Text- oder Bildinformationen. Dazu enthält es die Funktionen *setText()*, *setPixmap()* und *setMovie()*, um entweder ein Text, Bild oder Video anzuzeigen.

QOpenGLWidget

Die *QOpenGLWidget*-Klasse erbt von *QWidget* und ist ein Widget, das zum Rendern von OpenGL-Grafiken entworfen wurde.

Um OpenGL-Inhalt anzuzeigen, sind dabei beim Ableiten einer Unterklasse generell drei virtuelle Funktionen zu überschreiben:

- *paintGL()*:
    - Diese Funktion rendert die OpenGL-Szene und wird immer dann aufgerufen, wenn das Widget aktualisiert werden muss.
- *resizeGL()*:
    - *resizeGL()* setzt OpenGL-Parameter wie Projektion und Fenstermaße und wird aufgerufen, sobald das Widget eine externe Größenveränderung erfährt, etwa durch

Veränderung der Fenstergröße, und beim Erstellen des Widgets, da alle neu kreierten Widgets beim erstmaligen Anzeigen ein Größenanpassungs-Event auslösen.

- *initializeGL()*:
    - Der Inhalt dieser Funktion stellt die nötigen Ressourcen und Beschaffenheiten, die einmalig initialisiert werden müssen, zur Verfügung. Hierunter fallen zum Beispiel das Initialisieren der Shader oder das Laden von Texturen. *initializeGL()* wird exakt einmal aufgerufen bevor *resizeGL()* und *paintGL()* erstmalig aufgerufen werden.

Um ein manuelles Neuzeichnen der Szene auszulösen, das heißt ein Auslösen abseits von *paintGL()*, kann die von von *QWidget* geerbte Funktion *update()* genutzt werden. Diese verzeichnet eine Aktualisierung der Anzeige und eignet sich besonders im Einsatz mit Timer-gesteuerten Animationen (beispielsweise über die Klasse *QTimer*).

QOpenGLWindow und *QOpenGLWidget* erlauben das Benutzen verschiedener Versionen und Profile von OpenGL, solange die Betriebsplattform diese unterstützt. Für den Fall, dass in einem Fenster mehrere Instanzen von *QOpenGLWidget* untergebracht sind, wird vorausgesetzt, dass alle Instanzen das gleiche Format benutzen. Um Ungereimtheiten vorzubeugen, sollte daher in der main()-Funktion über den Aufruf von *QSurfaceFormat::setDefaultFormat()* das gewünschte Format festgelegt werden[5].

Schlussfolgerung:

Die *QLabel*-Klasse eignet sich offenbar dafür, ein *QPixmap*-Objekt, welches durch ein *QImage* gespeicherte Daten der Kinect enthält, über *setPixmap()* anzuzeigen.

Daneben kann das *QOpenGLWidget* dazu eingesetzt werden, um das eingehende Bild der Kinect in eine OpenGL-Textur umzuwandeln und anzuzeigen.

5 Vgl. http://doc.qt.io/qt-5/qopenglwidget.html#details , letzter Aufruf: 17.10.2015

2.3.4 Die OpenGL-Wrapper-Klassen

Qt bietet neben den zuvor erwähnten *QOpenGLWindow* und *QOpenGLWidget* eigene Wrapper-Klassen an, die native OpenGL-Kommandos verkapseln und den Versuch darstellen, für eine Programmierung von OpenGL-Kontext in einem Qt-Framework einheitlich im Qt-Kanon bleiben zu können und OpenGL zu abstrahieren. Eine Kompatibilität zu nativen OpenGL-Befehlen ist dennoch gegeben – es existiert kein Zwang, die Wrapper-Klassen einzusetzen.

Im Anschluss werden die Wichtigsten dieser Vertreter aufgelistet.

QOpenGLFunctions

Mit der *QOpenGLFunctions*-Klasse bietet Qt einen plattformübergreifenden Zugang zu der OpenGL ES 2.0 API. Die Erwähnung dieser Klasse mag zunächst merkwürdig erscheinen, schließlich ist die Zielumgebung der Applikation der Desktop. Der Sinn dahinter ist, dass OpenGL ES 2.0 vor Allem aus Performance-Gründen angesichts der (noch) meist relativ schwachen Hardware der Zielsysteme (zum Beispiel Smartphones) eine Untermenge der OpenGL-Funktionen unterstützt und zwar lediglich jene, die nicht zu den als *deprecated,* also als überholt markierten Funktionen gehören. Dadurch können nur OpenGL-Funktionen benutzt werden, die plattformübergreifend funktionieren und Teil der modernen OpenGL-Richtlinien sind.

Die einfachste Methode, *QOpenGLFunctions* in den Programmkontext miteinzubeziehen, ist die Integration als Superklasse:

```
1     class GLWidget : public QOpenGLWidget,
2                          protected QOpenGLFunctions
3     {
4         // ...
5     }
```

Listing 1:Vererbung durch QOpenGLFunctions

Um den Effekt der Klasse zu aktivieren, bedarf es noch den Aufruf der Memberfunktion *initializeOpenGLFunctions()*:

```
1     void GLWidget::paintGL()
2     {
3         initializeOpenGLFunctions();
4
5         // ...
6     }
```

Listing 2: Aktivierung innerhalb der paintGL()-Funktion

Die *paintGL()*-Funktion kann nun auf alle OpenGL ES 2.0 Funktionen zurückgreifen.

QOpenGLBuffer

Um Vertex Buffer Objekte zu formen und zu verwalten wurde mit der *QOpenGLBuffer*-Klasse ein Qt-interner Objekttyp dafür geschaffen.

Die Implementation eines solchen Buffer-Objekts soll anhand von Listing 3 kompakt aufgezeigt werden:

```
1         // Buffertyp bestimmen
2         QOpenGLBuffer myBuffer(QOpenGLBuffer::VertexBuffer);
3         // Bufferobjekt initiieren
4         myBuffer.create();
5         // Nutzungsmuster bestimmen
6         myBuffer.setUsagePattern(QOpenGLBuffer::StaticDraw);
7         // Bufferobjekt an aktuellen OpenGL-Kontext binden
8         myBuffer.bind();
9         // Speicherplatz allozieren; optional inklusive Datenfüllung
10        myBuffer.allocate( (const void *) data,
11                           (int) numberOfBytes);
12        // oder:
13        myBuffer.allocate((int) numberOfBytes);
14        // Bufferobjekt aus dem aktuellen OpenGL-Kontext lösen
15        myBuffer.release();
```

Listing 3: Beispiel eines Vertexbuffers anhand des QOpenGLBuffer-Wrappers

Für die Festlegung des Typs ist neben der Option *QOpenGLBuffer::VertexBuffer* noch die enumeration *QOpenGLBuffer::IndexBuffer* verfügbar. Mit einem Index-Buffer ist es möglich, Objekte mit Hilfe der Funktion *glDrawElements()* zu zeichnen, welche eine Indizierung der Vertices benötigt.

Das Nutzungsmuster des Buffers, in Listing 3 mit *QOpenGLBuffer::StaticDraw* versehen, optimiert den Buffer auf eine bestimmte Auslegung: Sollen beispielsweise bloß einmal Vertex-Koordinaten eines Objekts übergeben werden, die sich im Laufe der Anwendung nicht ändern, so eignet sich der Parameter *QOpenGLBuffer::StaticDraw* als Argument[6].

Die Funktion *allocate()* kann neben der Anzahl der zu reservierenden Bytes ein zweites Argument, und zwar ein Datenobjekt mit übergeben bekommen. Dabei wird der Inhalt von *data* über die Anzahl [*numberOfBytes*] Bytes initialisiert. Wird *allocate()* erneut aufgerufen, werden die zuvor enthaltenden Daten des Buffers überschrieben.

QOpenGLShaderProgram / QOpenGLShader

Die *QOpenGLShaderProgram*-Klasse sowie die *QOpenGLShader*-Klasse bilden zusammen eine robuste Kombination für die Shaderverwaltung in auf Qt basierenden OpenGL-Programmen.

Das Vorgehen hierbei besteht aus zwei Schritten: Erstens, das Initiieren eines Shader-Programms und zweitens das Hinzufügen von Shadern zu diesem Shader-Programm. Listing 4 demonstriert ein Beispiel.

```
1        QOpenGLShaderProgram myShaderProgram;
2        QOpenGLShader myShader(QOpenGLShader::Vertex);
3        myShader.compileSourceFile(filename);
4        myShaderProgram.addShader(&myShader);
5        myShaderProgram.link();
6        myShaderProgram.bind();
```

Listing 4: Beispiel eines Shaderprogramms unter Qt

Listing 4 zeigt das Kompilieren des Shaders über das Aufrufen einer externen Datei (filename). Dies ist nicht der einzige Weg, einen Shader dem Shaderprogramm hinzuzufügen. Weitere Möglichkeiten sind derweil:

- *myShader.compileSourceCode(source)*:
    - Dieser Funktion wird ein Source-Code des Typs *char, QString* oder *QByteArray* übergeben. Der Source-Code des Shaders kann dabei auch direkt als Argument

6 Weitere Parameter sind einsehbar unter: http://doc.qt.io/qt-5/qopenglbuffer.html#UsagePattern-enum

übergeben werden. Diese Methode hat den Nachteil, dass sich der Shader-Code in einem auskommentierten String befindet und somit keine Syntax-Fehler erkannt werden können.

- *myShaderProgram.addShaderFromSourceFile(type, filename)*:
  - Dem Shader-Programm wird direkt ein Shader über eine externe Datei hinzugefügt. Der Shader-Typ wird über *type* festgelegt (zum Beispiel *QOpenGLShader::Fragment*). Dieser Weg stellt eine Abkürzung dar, da kein eigenes Shader-Objekt angelegt werden muss.
- *myShaderProgram.addShaderFromSourceCode(type, source)*:
  - Dem Shader-Programm wird direkt ein Shader über einen Source-Code hinzugefügt. Der Parameter *type* bestimmt den Shader-Typ und *source* kann vom Typ *QString*, *QByteArray* oder *char* sein. Dieser Weg stellt eine Abkürzung dar, da kein eigenes Shader-Objekt angelegt werden muss.

Durch das Hinzufügen der Shader zu dem Shader-Programm sind die Shader jedoch noch nicht aktiv. Erst durch den Aufruf von *link()* werden die Shader verlinkt und die Shader-Pipeline gestartet.

Da es möglich ist, mehrere Shader-Programme zu entwerfen und in einem Programm zu vewenden, muss das jeweilige Shader-Programm über die Funktion *bind()* an den aktuellen OpenGL-Kontext explizit gebunden werden.

QOpenGLTexture

Mit der *QOpenGLTexture*-Klasse bietet Qt eine interne Verkapselung von OpenGL-Texturen an und erleichtert den Umgang mit den diversen Eigenschaften und Vorgaben, die eine OpenGL-Textur betreffen kann.

Dabei sieht der typische Ablauf, von einem Objekt dieser Klasse Gebrauch zu machen, wie folgt aus:

- Instantiieren eines Textur-Objekts unter Berücksichtigung des Zielobjekts

- Eigenschaften bezüglich der Speicheranforderungen bestimmen, wie etwa Format und Dimensionen
- Speicher allozieren
- Pixel-Daten hochladen (optional)
- Filter-Eigenschaften und Border-Optionen festlegen (optional)
- Die Textur rendern

Listing 5 zeigt ein Beispiel dieses Prozesses:

```
1     // Objekt initiieren
2     QOpenGLTexture *pTexture =
3             new QOpenGLTexture(QOpenGLTexture::TextureRectangle);
4     // Format der Texture festlegen
5     pTexture->setFormat(QOpenGLTexture::RGBAFormat);
6     // Speicher allozieren
7     pTexture->allocateStorage(QOpenGLTexture::RGBA,
8                               QOpenGLTexture::UInt32);
9     // Pixeldaten hochladen
10    pTexture->setData(QOpenGLTexture::RGBA,
11                      QOpenGLTexture::UInt32, &data[0]);
12    // Minificaton- und Magnification-Filter festlegen
13    pTexture->setMinificationFilter(QOpenGLTexture::Nearest);
14    pTexture->setMagnificationFilter(QOpenGLTexture::Linear);
15    // Wrap-Mode festlegen
16    pTexture->setWrapMode(QOpenGLTexture::Repeat);

17    // Textur binden
18    if(pTexture->isCreated())
19        pTexture->bind();
20
21        // ... process ...

22    // Textur lösen
23    if(pTexture->isBound())
24        pTexture->release();
```

Listing 5: Beispiel einer QOpenGLTexture-Verwendung

Sobald über *allocateStorage()* der Speicher reserviert wurde, sind die Rahmeneigenschaften der Textur nicht mehr veränderbar. In Listing 5 sind das die Parameter *QOpenGLTexture::RGBA* und

QOpenGLTexture::UInt32, die das Format und den erwarteten Datentyp bestimmen. Fortan müssen diese Parameter von *setData()* eingehalten werden, ansonsten können keine neuen Pixel-Daten (in diesem Beispiel durch einen Zeiger auf das erste Bit von *data* gekennzeichnet) in die Textur hochgeladen werden.

Eine weitere Stärke der *QOpenGLTexture*-Klasse besteht darin, Texturen direkt aus Bilddateien heraus zu erstellen. Dafür ist allein der Einsatz eines alternativen Konstruktors nötig:

```
1     QOpenGLTexture *pTexture =
2             new QOpenGLTexture(QImage(":/texture.png").mirrored());
```

Listing 6: Laden einer QImage-Datei in die Textur

Während in einer nativen OpenGL-Umgebung eine weitere Bibliothek zum Laden einer Bilddatei nötig wäre, kann Qt hier seine Stärken als umfassendes Framework ausspielen.

Die Funktion *mirrored()* sorgt für einen komfortablen Nebeneffekt: Da *QImage* und OpenGL über entgegengesetzte Richtungen der Y-Achse verfügen, muss das geladene Bild vertikal gespiegelt werden – *mirrored()* tut genau dies. Somit muss beim Anlegen der Textur-Koordinaten nicht mehr umgedacht werden.

Schlussfolgerung:

Qt bietet offenbar eine Reihe von Mitteln an, die nützlich sein könnten, um einen OpenGL-Kontext unter Einbehaltung des Qt-Programmierstils zu ermöglichen und zu abstrahieren.

Während *QOpenGLBuffer*, *QOpenGLShader* und *QOpenGLShaderProgram* das Verwalten von Positions- und Textur-Koordinaten abdecken, können mit Hilfe von *QOpenGLTexture* bequem die notwendigen Bilddateien für die Kleidungstexturen geladen und eingesetzt werden.

2.3.5 Das Signal- und Slot-Prinzip

Für die Kommunikation zwischen Objekten greift Qt auf ein eigenes Signal- und Slot-Konzept zurück, welches inzwischen auch in anderen Programmbibliotheken Verwendung gefunden hat.

Klassische Rückruffunktionen (*Callbacks)* werden eingesetzt, um eine Funktion unter bestimmten Bedingungen innerhalb einer anderen Funktion aufzurufen. Dabei wird typischerweise ein Zeiger zu einer Funktion A als Argument an Funktion B übergeben. Funktion B kann nun über das Auftreten eines Events berichten, indem Funktion A innerhalb von Funktion B ausgelöst wird (siehe Abbildung 6).

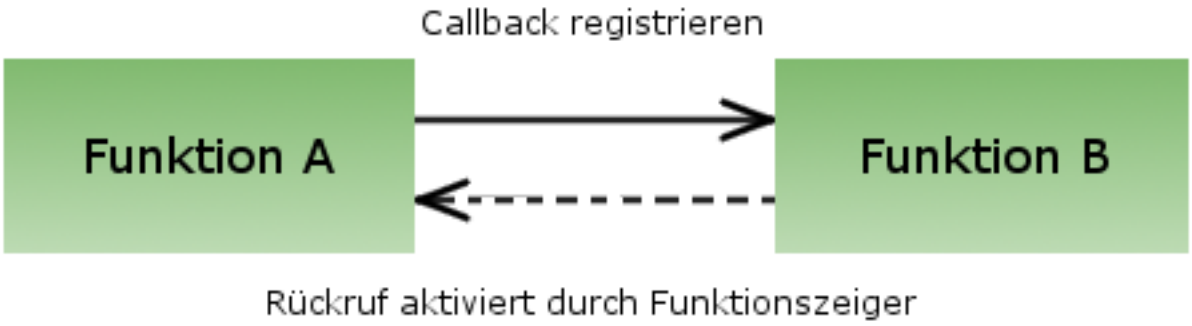

Abbildung 8: Callback-Funktionsweise

Dieses Prinzip hat grundsätzlich zwei Mankos:

1. Callbacks sind nicht typsicher[7]: Es kann nicht sichergestellt werden, dass zur Laufzeit die Rückruffunktion mit den korrekten Argumenten aufgerufen wird.
2. Die Rückruffunktion ist stark an die sie registrierende Funktion gebunden, da diese Funktion wissen muss, welcher Callback aufgerufen werden soll.

Qt bietet mit dem Signal- und Slot-Konzept einen alternativen Ansatz, um eine Verbindung und die Möglichkeit einer Ereignisverwaltung zwischen Objekten herzustellen.

7 Vgl. Wietzke, Joachim (2012): „Embedded Technologies: Vom Treiber bis zur Grafik-Anwendung“, S.196

Ein Signal wird gesendet, sobald ein spezielles Ereignis auftritt. Die Widgets in Qt haben eine Reihe an vordefinierten Signalen (beispielsweise ein Signal wie *clicked()* bei einem Button, der mit der Maus angeklickt wurde), dessen ungeachtet ist es gangbar, ein selbst entworfenes Widget abzuleiten und eigene Signale und Slots hinzuzufügen.

Hinter einem Slot verbirgt sich eine Funktion, die ausgeführt wird, sobald das mit dem Slot verbundene Signal ausgelöst wurde.

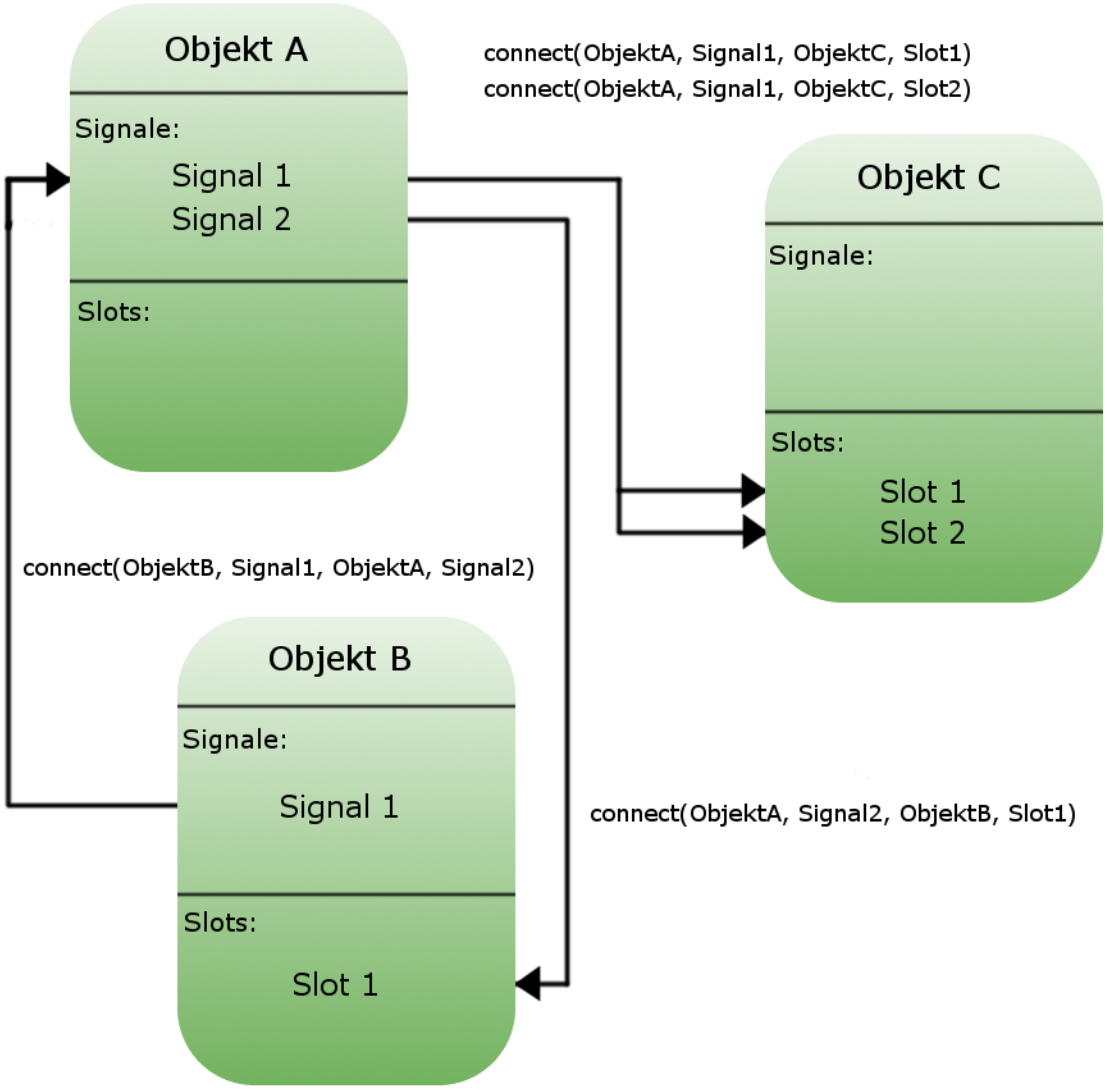

Abbildung 9: Beispiel eines Signal- und Slot-Systems

Der Signal- und Slot-Mechanismus ist typsicher: Der Mechanismus setzt voraus, dass die Signatur eines Signals und die Signatur des angebundenen Slots *identisch* sind. Dadurch kann gewährleistet werden, dass es nie zu einer Verletzung der Typsicherheit kommt – der Compiler würde eine fehlende Übereinstimmung sofort aufdecken.

Außerdem sind Signal und Slot lose miteinander verbunden: Eine Klasse, die ein Signal sendet, hat keinerlei Informationen darüber, welcher Slot das Signal empfängt. Die Intention, ein Signal zu senden, besteht lediglich darin, dass eine Veränderung des Objekts für andere Objekte interessant sein *könnte*. So kann ein Slot auch als reguläre Memberfunktion eingesetzt werden, ohne tatsächlich ein Signal zu empfangen. Mit dieser Technik können voneinander unabhängige Komponenten geschaffen werden und es entsteht somit eine wahre Informationskapselung.

Darüber hinaus kann eine unbegrenzte Anzahl an Signalen einem Slot zugeordnet werden, ebenso wie ein Signal einer unbegrenzten Anzahl an Slots zugeteilt werden kann.

Abbildung 9 zeigt beispielhafte Verknüpfungen von Signalen und Slots und demonstriert eine weitere Eigenschaft der Signale, und zwar die Verbindung eines Signals zu einem anderen Signal. Durch diese Signalverkettung ist es möglich, durch die Auslösung eines Signals ein weiteres Signal auszulösen, sodass sich flexible und komplexe Ereignis-Mechanismen erschaffen lassen.

Schlussfolgerung:

Der Signal- und Slot-Mechanismus eignet sich ideal um eine Kommunikation zwischen der Kinect und den Qt-Objekten herzustellen. Die Kinect kann dabei ein Signal aussenden, sobald ein neues Bild verfügbar ist und schickt die Daten an einen Slot eines *QLabel* beziehungsweise eines *QWidget*.

2.4 Fazit

In diesem Kapitel konnten eine Reihe von Schlussfolgerungen getroffen werden, die den Aufbau und das Konzept der Programmierarbeit bestimmen, von den nötigen OpenGL-Grundlagen bis hin zu den Mitteln, OpenGL erfolgreich in Qt integrieren zu können.

Jedoch bedarf es noch einer Einteilung und Zuordnung in Klassen, die sinnvoll und überlegt zusammengestellt werden sollten, sodass die aus der Analyse gewonnenen Erkenntnisse in die Tat umgesetzt werden können.

Im nächsten Kapitel wird demnach das Konzept dargelegt, welches die Klassen aufstellt und Klarheit über den konkreten Programmablauf verschaffen soll.

3 Konzept

Aufbauend auf dem vorgehenden Kapitel wird mit diesem Kapitel das grundlegende Konzept der Programmierarbeit besprochen. Dabei werden die Klassen bestimmt und ihre verschiedenen Aufgaben erläutert. Darüber hinaus müssen noch einige offene Fragen, wie etwa die Wahl der Texturen, geklärt werden.

3.1 Entscheidung für 2D-Texturen

Zu Beginn der Bearbeitung bestand die Idee noch darin, für die Kleidungstexturen 3D-Models zu entwerfen mit der Hilfe eines Programms wie Blender. Dieser Ansatz musste jedoch schnell verworfen werden, da die Einarbeitung in die umfangreiche Materie die Zeit für eine komplexe Modellentwerfung nicht zuließ.

Somit fiel der Entschluss, für eine Demonstration der Funktionsweise auf 2D-Texturen zurückzugreifen, die mit dem Laden von Bilddateien umgesetzt werden.

Um die Dateien für den Einsatz innerhalb der Anwendung vorzubereiten, musste den Bilddateien ein Alpha-Kanal hinzugefügt werden (siehe Abbildung 10), damit die Texturen später erfolgreich übereinander gelegt werden können.

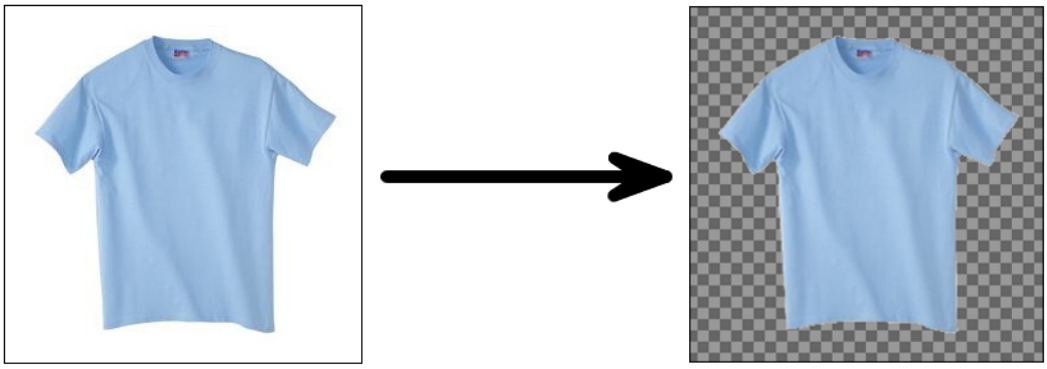

Abbildung 10: Hinzufügen eines Alpha-Kanals

3.2 Entscheidung für die orthogonale Projektion

Mit der Entscheidung für die 2D-Texturen fiel auch die Entscheidung für die Art der Projektion. Da die orthogonale Projektion für die Darstellung von 2D-Texturen ausreicht und keine 3D-Modelle zum Einsatz kommen, soll die orthogonale Projektion dem Anspruch dieser Anwendung genügen.

Im Folgenden wird gezeigt, unter welchen Bedingungen die Projektion stattfinden soll. Diese Bedingungen sind ein Resultat der Überlegungen, die angestellt werden müssen, wenn die Pixel-Koordinaten der Kinect in OpenGL-Koordinaten konvertiert werden sollen.

3.2.1 Umwandlung der Kinect-Koordinaten

Die Bildkoordinaten des Kinect-Bildes sind Pixel-basiert beziehungsweise Meter-basiert:

- X-Koordinaten:
  - Wertebereich von 0 bis (Bildbreite minus 1)
- Y-Koordinaten:

- Wertebereich von 0 bis (Bildhöhe minus 1)

- Z-Koordinate:
  - Wertebereich von 0 Millimeter bis 4000 Millimeter

Eine naheliegende Methode, diese Koordinaten in handhabbare OpenGL-Koordinaten umzuwandeln, wäre die Division des Werts durch die Bildbreite oder Bildhöhe beziehungsweise der maximal wahrnehmbaren Tiefe:

- X-Koordinaten:
  - Pixel-Koordinate dividiert durch Bildbreite → Wertebereich von 0 bis 1

- Y-Koordinaten:
  - Pixel-Koordinate dividiert durch Bildhöhe → Wertebereich von 0 bis 1

- Z-Koordinate:
  - Anzahl der Millimeter dividiert durch 4000 Millimeter

Mit Hilfe dieser Vorgehensweise ergibt sich der Aufbau der Projektion für die geplante Anwendung.

3.2.2 Aufbau der Projektion

Wie unter Kapitel 2.2.1.3 (Die Projektionstypen) beschrieben, wird das Clipping-Volumen der orthogonale Projektion durch die *near plane* und *far plane* aufgespannt. Sollen sich alle Werte der Koordinaten in dem Intervall [0, 1] befinden, so ergibt sich für die Projektion dieser Anwendung folgendes Bild:

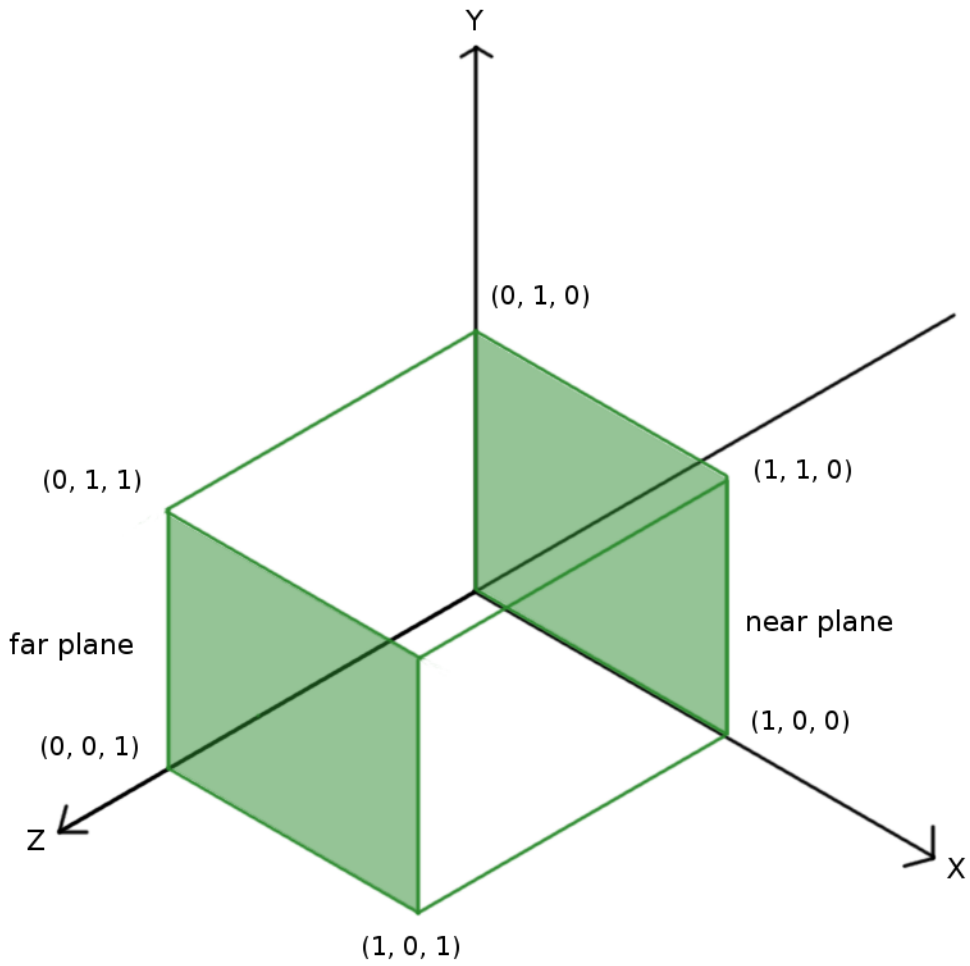

Abbildung 11: Projektion für den OpenGL-Kontext

Damit ergibt sich eine Blickrichtung in positive Z-Richtung. Objekte mit einer Z-Koordinate von 1 sind vom Betrachter weit entfernt, während Objekte mit einer Z-Koordinate von 0 die dem Betrachter am nächsten, sichtbaren Objekte darstellen.

3.2.3 Das sukzessive Zeichnen

Da weit entfernte Objekte sich dem Z-Wert von 1 nähern, müssen die Z-Werte der auf den Kinect-Hintergrund projizierten Kleidungstexturen sich zwischen den Werten 0 und 1 bewegen.

Der Farbbild-Hintergrund der Kinect stellt dabei die *far plane* dar. Beim Rendern wird etappenweise zunächst die Farbe der 2D-Textur über den Befehl *glClearColor()* gelöscht, dann der Hintergrund gezeichnet und daraufhin die Kleidungstextur gezeichnet.

Abbildung 12 veranschaulicht diesen Prozess.

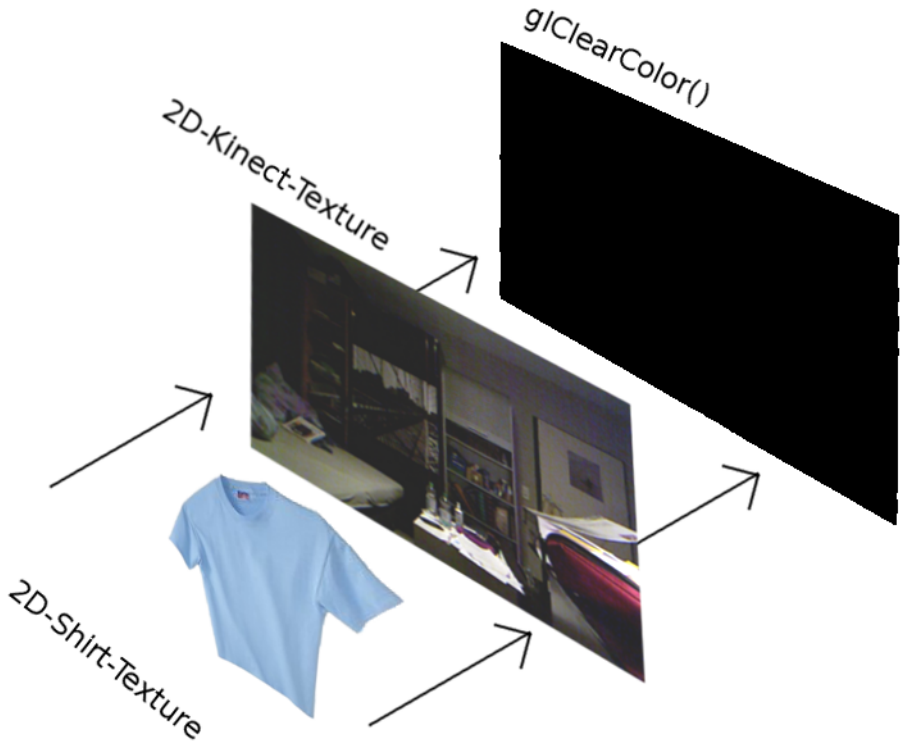

Abbildung 12: Das sukzessive Zeichnen während des Render-Vorgangs

3.3 Architektur

Wie bei jeder Software-Lösung ist es auch hier durchaus sinnvoll, sich Gedanken über die generelle Struktur des Programms zu machen und gegebenenfalls sich an etablierte Paradigmen anzulehnen und deren Philosophie und Richtlinien in die eigene Umsetzung mit einwirken zu lassen.

Im Folgenden werden jene Prinzipien, die in dieser Anwendung zum Einsatz kommen, überschaubar erläutert.

3.3.1 DRY-Prinzip

Das DRY-Prinzip („don't repeat yourself“, auch unter „once and only once“ bekannt)[8] besagt, dass Redundanz möglichst vermieden werden sollte.
Bei der Programmierung tritt Redundanz beispielsweise in Form von Code-Duplizierungen im Quellcode auf. Folgt man dem DRY-Prinzip nun an dieser Stelle, sind Änderungen, wie zum Beispiel das Ändern einer Variable, nur an einer Stelle vorzunehmen und müssen nicht im gesamten Code-Kontext wiederholt neu gesetzt werden.
Dadurch wird nicht nur der Quelltext übersichtlicher, sondern auch besser wartbar.

3.3.2 SRP-Prinzip

Unter dem SRP-Prinzip („Single-Responsibility-Prinzip“)[9] ist ein Prinzip zu verstehen, dass den Programmierer dahingehend lenken soll, möglichst keine Klassen mit mehreren Verantwortlichkeiten zu entwerfen. Klassen sollen, soweit es der Programmkontext zulässt, eine fest definierte Aufgabe haben, die idealerweise intuitiv aus ihrer Namensgebung abzuleiten ist.

Darüber hinaus sollte die Klasse alleinig Funktionen besitzen, die zur unmittelbaren Erfüllung dieser Aufgabe beitragen.

3.4 Klassenentwurf

Aus den Erkenntnissen der Analyse (Kapitel 2) und den Überlegungen bezüglich der Software-Architektur (Kapitel 3.3) können nun die Klassen und ihre individuellen Funktionen konkretisiert werden.

8 Vgl. http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod , letzter Aufruf: 08.10.2015
9 Vgl. http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod , letzter Aufruf: 08.10.2015

3.4.1 KinectEngine

Die Klasse *KinectEngine* soll verantwortlich sein für die Steuerung der Kinect. Sie muss dafür sorgen, dass die Kinect funktionsfähig eingesetzt werden kann und die notwendigen Daten der Kinect gesendet werden können.

Für diese Aufgabe muss *KinectEngine* eine von *QThread* abgeleitete Klasse sein, da Qt keinen Eingriff in die die main()-Schleife (*Qapplication::exec()*) zulässt[10]. Deswegen bedarf es eines externen, weiteren Threads, mit dem die Kinect betrieben werden kann.

3.4.2 VideoWidget

Mit der Klasse *VideoWidget* soll eine Klasse zur Verfügung gestellt werden, die von der Kinect gesendete Bilddaten anzeigen kann, ohne dabei auf OpenGL zurückzugreifen. Sie dient der erstmaligen Erkundung der Kinect-Sensoren, sodass das ordnungsgemäße Funktionieren der Hardware sichergestellt werden kann, bevor mit OpenGL eine weitere API (und damit potentielle Fehlerquelle) hinzugezogen wird.

3.4.3 MainWindow

Bevor überhaupt etwas angezeigt werden kann, wird ein Fenster benötigt. Die Klasse *MainWindow* kümmert sich um das Bereitstellen eines Fensters und setzt die Funktionsweise weitere User Interface-Elemente wie beispielsweise Buttons.

3.4.4 GLWidget

10 Vgl. http://doc.qt.io/qt-5/qapplication.html#exec , letzter Aufruf: 25.10.15

Die Klasse *GLWidget* ist gewissermaßen das OpenGL-Pendant zu der Klasse *VideoWidget*. Sie soll es ermöglichen, die Bilddaten mit Hilfe von OpenGL anzuzeigen und ferner weitere Prozesse, wie das Zeichnen von Kleidungstexturen zu vollbringen.

3.4.5 GeometryEngine

Die Klasse *GeometryEngine* arbeitet eng mit der Klasse *GLWidget* zusammen. Über *GeometryEngine* werden die OpenGL-Texturen gezeichnet und statische Buffer initialisiert und geladen – also Buffer, die nicht-dynamische Werte speichern, wie zum Beispiel die Position der Hintergrundtextur (das Kinect-Bild).

Die Konzeptfindung ist an dieser Stelle fertig und es folgt nun die Implementation.

4 Programmierung

Das Arbeiten mit der Kinect innerhalb einer Qt-Umgebung ergab sich als ein iterativer Prozess.
So teilt sich die Programmierung und Realisierung der Aufgabenstellung grob in zwei Bereiche:

1. Das Erkunden der RGB- und Tiefensensoren und eine Darstellung des Bilds anhand des Datentyps *QPixmap* in einem *VideoWidget*.
2. Weitergabe der Kinectdaten an ein *GLWidget*, wo die Rendering-Pipeline von OpenGL die Daten weiterverarbeitet und die Ausgabe in einer 2D-Textur resultiert.

Diese Einteilung rührt daher, dass zum Einen ein Qt-Projekt nur ein Layout *(.ui*-File) besitzen kann, und zum Anderen die in Qt erhaltene Klasse *QOpenGLWidget*, von der *GLWidget* erbt, den gesamten Fensterkontext (über *QSurfaceFormat*) auf OpenGL optimiert, sodass es bei einer parallelen Anzeige von sowohl OpenGL-Kontext als auch Wiedergabe von QPixmaps (Qt's internem, regulären Bildformat für die Bildanzeige) zu Darstellungsproblemen kommt.
Somit wurden im Qt Creator zwei voneinander unabhängige Projekte angelegt, sodass ein sauberes und geordnetes Arbeiten möglich war.

4.1 Konventionen

Um den Code leserlich und übersichtlich zu halten, werden Membervariablen mit dem Präfix „m“ versehen und Zeiger mit dem Präfix „p“. Ist eine Membervariable gleichzeitig ein Zeiger, so wird der Präfix „mp“ verwendet.

4.2 Kinect und VideoWidget

Zunächst sollen die von der Kinect gesendeten Daten so gekapselt werden, dass sie in einem *QLabel*-Container angezeigt werden können. Dazu eignet sich ein Umwandeln in ein *QImage*, welches dann im *VideoWidget* in ein *QPixmap* konvertiert wird und angezeigt wird.

Um jedoch überhaupt an Daten zu gelangen, muss die Kinect erst initialisiert werden.

4.2.1 Farbbildstream

Die Klasse *KinectEngine* ist zuständig für die Initialisierung der Kinect sowie jedwede Ansteuerung des Geräts. Bevor Bilddaten von der Kinect empfangen werden können, muss zuerst ein Initialisierungsprozess stattfinden.

4.2.1.1 Initialisierung

Die Funktion *initKinect()* übernimmt dabei die Rolle für das Starten der Hardware und das Zugreifen auf die verschiedenen Sensoren. Die ersten Schritte beinhalten dabei das Finden eines angeschlossenen Kinect-Device:

```
1      HRESULT result;
2      int numberOfSensors;
3      result = NuiGetSensorCount(&numberOfSensors);
4      if(FAILED(result))
5      {
6          qDebug() << "No kinect decive was found!";
7          return;
8      }
```

Listing 7: Anzahl der angeschlossenen Sensoren ermitteln

Die Funktion *NuiGetSensorCount()* ermittelt die Anzahl angeschlossener Kinect-Geräte und schreibt diese in die Variable *numberOfSensors*. Der Datentyp HRESULT ist ein Windows-internes Format für ein Ergebnis-Handle und kommt hierbei fortan durch die lokale Variable *result* zum Einsatz – sollte *NuiGetSensorCount()* FAILED zurückgeben, so wird der Initialisierungsvorgang abgebrochen und es kommt über zu einer Fehlerausgabe.

Im C++-Interface der Kinect SDK sind sämtliche, die Sensoren betreffenden Funktionen mit dem Präfix „Nui“ versehen (zum Beispiel ***Nui****SensorCount()*). Das Akronym NUI bedeutet ausgeschrieben *Natural User Interface* und bezieht sich auf die möglichen Interaktionen durch den Benutzer (etwa Gesten, Sprache).

Scheitert der obige Vorgang nicht, so besteht der nächste Schritt in *initKinect()* darin, die

angeschlossenen Sensoren anzusprechen. Dafür kommt die Funktion *NuiCreateSensorByIndex()* zum Einsatz:

```
1       if(numberOfSensors != 0)
2       {
3           result = NuiCreateSensorByIndex(0, &mpSensor);
4           if(FAILED(result))
5           {
6               qDebug() << "Could not create sensor!";
7               return;
8           }
9       }
```

Listing 8: Erzeugen eines Kinect-Interfaces

Hier wird die Kinect mit dem Index 0 (sprich die erste gefundene Instanz von *NuiSensorCount()*) über die Membervariable *mpSensor* logisch erzeugt. *mpSensor* ist vom Typ *INuiSensor*, welcher ein Interface darstellt, über das sämtliche NUI-Funktionen aufgerufen werden können.

Der Vorteil darin kommt in dem Kontext dieser Arbeit nicht zum Einsatz, denn das INuiSensor-Interface ermöglicht das Referenzieren diverser Kinect-Sensoren, wobei im Folgenden nie mehr als eine Kinect benutzt wird. So können theoretisch in einem Programm multiple Kinect-Geräte über das mehrfache Aufrufen von *NuiCreateSensorByIndex()* kreiert werden:

```
1           for(int i = 0; i < numberOfSensors; i++)
2           {
3               result = NuiCreateSensorByIndex(i, &mpSensors[i]);
4               if(FAILED(result))
5               {
6                   qDebug() << "Could not create sensor „ << i << „ !";
7                   return;
8               }
9           }
```

Listing 9: Theoretisches Erzeugen mehrere Kinect-Interfaces

Die unterschiedlichen Zeiger ermöglichten daraufhin einen Zugriff auf je ein Kinect-Device, unter der Annahme, dass *mpSensors* ein Array von INuiSensor-Interfaces darstellt:

```
1       // ...
2       mpSensors[0]->NuiXXX(); // call any function for first Kinect
3       mpSensors[1]->NuiXXX(); // call any function for second Kinect
4       // ...
```

Listing 10: Theoretische Verwendung mehrerer Kinect-Interfaces

Für den jetzigen Zweck, eine Desktop-Applikation als Protoyp zu schaffen, ist es also irrelevant, ob mit dem INuiSensor-Interface gearbeitet wird oder die NUI-Funktionen nicht über einen Zeiger aufgerufen werden. Da sich die Portierung auf ein Kaufhaussystem mit eventuell mehreren Exemplaren jedoch als Zielsystem im Hinterkopf befindet, ist es nicht verkehrt, das Interface bereits an dieser Stelle zu implementieren, sodass dann beim Einsatz zusätzlicher Geräte bloß die *mpSensor*-Kennzeichnung im Code auf den gewünschten Index geändert werden muss und somit einige Kinect-Instanzen gleichzeitig in einem Programm gesteuert werden können.

Nachdem ein entsprechender Sensor über *NuiCreateSensorByIndex()* erzeugt wurde, ist dieser allerdings noch nicht gestartet. Die eigentliche Initialisierung findet über den Aufruf von *NuiInitialize()* statt.

```
1           DWORD sensorFlags = NUI_INITIALIZE_FLAG_USES_COLOR;
2           result = mpSensor->NuiInitialize(sensorFlags);
3           if(FAILED(result))
4           {
5               qDebug() << "Kinect initilization failed!";
6               return;
7           }
```

Listing 11: Initialisieren der Kinect

Der von *NuiInitialize()* erwartete Parameter DWORD ist erneut ein Windows-internes Format für den Datentyp unsigned int. An dieser Stelle wird darüber entschieden, mit welchen aktiven Funktionen der Sensor hochgefahren werden soll. Da zunächst nur der RGB-Stream sichtbar sein soll, wird *sensorFlags* mit der Konstanten NUI_INITIALIZE_FLAG_USES_COLOR initialisiert.

Weitere, möglich Konstanten sind indes:

- NUI_INITIALIZE_FLAG_USES_DEPTH
- NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX
- NUI_INITIALIZE_FLAG_USES_AUDIO
- NUI_INITIALIZE_FLAG_USES_SKELETON

Auf die Bedeutung dieser Konstanten wird, obgleich sie vermutlich durch die Bezeichnungen offenkundig ist, an entsprechender Stelle eingegangen, sobald die jeweilige Konstante eingesetzt wird.

Die Kinect wird nun mit aktiviertem RGB-Stream gestartet. Der letzte Schritt in *initKinect()* ist das explizite Öffnen des Image-Streams:

```
1         result = mpSensor->NuiImageStreamOpen(
2             NUI_IMAGE_TYPE_COLOR,                // 1. Argument
3             NUI_IMAGE_RESOLUTION_640x480,        // 2. Argument
4             0,                                   // 3. Argument
5             2,                                   // 4. Argument
6             NULL,                                // 5. Argument
7             &mColorStreamHandle);                // 6. Argument
8         if(FAILED(result))
9         {
10            qDebug() << "Color stream could not be opened!";
11            return;
12        }
```

Listing 12: Öffnen des Image-Streams

Die Funktion *NuiImageStreamOpen()* erwartet sechs Argumente[11]:

1. NUI_IMAGE_TYPE eImageType:
   - Die für eImageType gültigen Werte hängen davon ab, wie *sensorFlags* initialisiert wurde (siehe Listing …). Aktuell wurde die Kinect nur mit dem Color-Flag gestartet, in sofern hat eImageType in diesem Fall den Wert NUI_IMAGE_TYPE_COLOR.
2. NUI_IMAGE_RESOLUTION eResolution:
   - eResolution spezifiert die Auflösung für die Art des Image-Streams, die zuvor unter

11 Vgl. https://msdn.microsoft.com/en-us/library/jj663864.aspx , letzter Aufruf: 10.10.2015

eImageType angegeben wurde (gegenwärtig NUI_IMAGE_TYPE_COLOR). Für den RGB-Color-Stream gibt drei verschiedene Auflösungen:

- NUI_IMAGE_RESOLUTION_320x240
- NUI_IMAGE_RESOLUTION_640x480
- NUI_IMAGE_RESOLUTION_1280x960

Die Auflösung 1280x960 Pixel wäre zwar für die Qualität des Bildes ideal, doch unter dieser Einstellung ist kein Streaming unter garantierten 30 Bildern pro Sekunde möglich. Da es bei der Anwendung vor allem auch um Stabilität und Performance geht, wird der Funktion der Wert NUI_IMAGE_RESOLUTION_640x480 übergeben, sodass sich die Framerate der Kinect sich nicht zum eventuellen Flaschenhals der Wiedergabeleistung entwickelt.

3. DWORD dwImageFrameFlags:
   - Über diesen Wert können bestimmte Flags für die eingehenden Frames eingestellt, beispielsweise ein minimaler oder maximaler Tiefenwert zum Filtern der Aufnahme, oder das Aktivieren des *Near Modes* (gilt nur für *Kinect for Windows*).

     Für den derzeitigen Kontext sind diese Flags irrelevant und der Wert wird daher auf 0 gesetzt.
4. DWORD dwFrameLimit:
   - Der hier angegebene Wert stellt die Anzahl der Frames dar, die die Kinect-Laufzeitumgebung buffern soll.

     Der von Microsoft empfohlene Wert liegt bei 2 und wird hier dementsprechend verwendet.
5. HANDLE hNextFrameEvent:
   - Wird an dieser Stelle ein gültiger Handle übergeben, so wird der nächste Frame erst eingefordert, sobald dieser Handle manuell zurückgesetzt wurde. Dieser Mechanismus eignet sich vor allem in Multithread-Umgebungen, bei denen es notwendig ist, einen Reset an einer bestimmten Stelle auszulösen, um Stabilität zu gewährleisten.

Die Notwendigkeit eines solchen Handles hat sich für diese Anwendung als nicht gegeben herauskristallisiert und wird daher auf NULL gesetzt.

6. HANDLE *phStreamHandle:
   - Bei dem übergebenen Wert muss es sich um einen Wert ungleich NULL handeln, denn hier muss eine Referenz zu einem gültigen Stream-Handle angegeben werden, sodass bei späterer Abrufung neuer Frames die jeweilige Zuordnung zu einem geöffneten Stream stimmt. Wird hier ein Parameter von NULL angegeben, so können aus diesem Stream keine Daten abgerufen werden, obwohl er geöffnet ist.

     Deswegen wird eine Referenz der Membervariable *mColorStreamHandle* übergeben.

Der Funktionsinhalt von *initKinect()* ist damit abgeschlossen und der Initialisierungsprozess für den Farbbild-Stream somit beendet.

Der nächste Schritt beinhaltet das Implementieren der aus *QThread* geerbten Funktion *run()*, in der mit Hilfe einer Schleife das wiederholte Empfangen neuer Bilddaten möglich ist.

4.2.1.2 Bildschleife

Die Funktion *run()* der Klasse *KinectEngine* bedarf zunächst eine Schleife und eine Bedingung für die Ausführung dieser Schleife:

```
1     void KinectEngine::run()
2     {
3         while(!mStopped){
4
5                 // ... Process ...
6         }
7     }
```

Listing 13: Laufbedingung der while-Schleife der Thread-Funktion run()

Die Membervariable *mStopped* kann nur über die Funktion *stop()* auf *false* gesetzt werden. Dies tritt ein, sobald der Thread angehalten wird.

Um ein Bild aus dem Farbbild-Stream der Kinect zu bekommen, ist der Inhalt der while-Schleife in diesem Fall nur die Funktion *processColor():*

```
1     void KinectEngine::run()
2     {
3         while(!mStopped){
4             processColor();
5         }
6     }
```

Listing 14: Aufruf von processColor()

Den Kern der Funktion *processColor()* bildet eine Variable *colorFrame* vom Typ NUI_IMAGE_FRAME, eine Struktur (englisch struct), die insbesondere die Aufgabe hat, Informationen über ein Farb- oder auch Tiefenbild zu speichern:

```
1     void KinectEngine::processColor()
2     {
3         NUI_IMAGE_FRAME colorFrame;
4         HRESULT result;
5         result = mpSensor->NuiImageStreamGetNextFrame(
6                             mColorStreamHandle, 0, &colorFrame);
7         // continue ...
8     }
```

Listing 15: Schreiben des nächsten Frames in colorFrame

Wie Listing 15 zeigt, wird über die Funktion *NuiImageStreamGetNextFrame()* der nächste Frame angefordert. Dabei werden die Daten in die Variable *colorFrame* geschrieben und über die Variable *mColorStreamHandle* kann sichergestellt werden, dass es sich um die Daten des Farbsensors handelt.

Wie zuvor in dem Initialisierungsprozess wird das Resultat des Aufrufs von *NuiImageStreamGetNextFrame()* in die lokal Variable *result* geschrieben. Ist der Aufruf erfolgreich, so folgt der Zugriff auf *colorFrame* über das Interface INuiFrameTexture:

```
1     void KinectEngine::processColor()
2     {
3         // continue ...
4
5         if(SUCCEEDED(result))
6         {
7             INuiFrameTexture* pFrameTexture =
                                        colorFrame.pFrameTexture;

8             NUI_LOCKED_RECT lockedRect;
9             pFrameTexture->LockRect(0, &lockedRect, NULL, 0);

10            // continue ...
11        }
12    }
```

Listing 16: Erzeugung einer Bildtextur anhand von colorFrame

Das Interface *INuiFrameTexture* repräsentiert ein Objekt, welches Bilddaten enthält, die ähnlich einer Direct3D-Textur sind[12]. Mit Hilfe dieses Interfaces wird der Zugang zu diversen Funktionen gewährt, mit denen der Zugriff und Manipulation auf den aktuellen Bild-Buffer ermöglicht wird. Listing 16 demonstriert, wie zunächst das Interface über *pFrameTexture* implementiert wird und der Zeiger daraufhin der in der Struktur *colorFrame* enthaltenden Eigenschaft des Typs *INuiFrameTexture* zugeordnet wird (vergleiche Listing 21, Definition der Struktur NUI_IMAGE_FRAME).

Mit Hilfe dieser Zuordnung ist nun eine Verbindung zu dem aktuellen Bild-Buffer hergestellt. Um die Daten für eine Anzeige vorzubereiten, bedarf es einer weiteren Struktur der Kinect SDK – NUI_LOCKED_RECT. Diese Struktur definiert die Oberfläche eines Rechtecks und dient der Speicherung der Bits des aktuellen Bild-Buffers[13].

Über die Funktion *LockRect()* wird der Lese- und Schreibzugang des Buffers gesperrt, sodass sichergestellt ist, dass zu diesem Zeitpunkt das Objekt von keinem anderen Ort aus verfügbar ist. An dieser Stelle ist anzumerken, dass Qt mit den Klassen *QMutex* und *QMutexLocker* einen eigenen Mechanismus zur Verfügung stellt, der verhindern kann, dass Objekte zu einem Zeitpunkt mehrere Zugriffe erhalten[14]. Dank *LockRect()* ist der Einsatz dieser Mittel hierbei

12 Vgl. https://msdn.microsoft.com/en-us/library/nuisensor.inuiframetexture.aspx , letzter Aufruf: 21.10.15
13 Vgl. https://msdn.microsoft.com/en-us/library/nuiimagecamera.nui_locked_rect.aspx , letzter Aufruf: 21.10.15
14 Vgl. http://doc.qt.io/qt-5/qmutex.html#details , letzter Aufruf: 22.10.15

jedoch nicht notwendig.

Nach dieser Sicherstellung kann nun der Inhalt des Bild-Buffers kopiert werden. Dazu wird der Typ *QImage* eingesetzt und es wird über den Befehl *emit* das Signal *sendColorImage()* gesendet:

```
1     void KinectEngine::processColor()
2     {
3         // ...

4         if(SUCCEEDED(result))
5         {
6             // ...
7             if(lockedRect.Pitch != 0){
8                 QImage image((const unsigned char*) lockedRect.pBits,
9                                 mWidth, mHeight, QImage::Format_RGB32);
10                emit sendColorImage(image);
11            }

12            // continue ...
13        }
14    }
```

Listing 17: Einkapseln der Bild-Daten in ein QImage

Die Eigenschaft *Pitch* enthält die Anzahl der Bytes in einer Reihe – ist diese ungleich 0, das heißt enthält sie Daten, werden die Bits von *lockedRect* in das QImage *image* geladen. Die Breite und Höhe von *image* wird über *mWidth* und *mHeight* übergeben; da der Stream mit 640 x 480 Pixeln initialisiert wurde, betragen die jeweiligen Werte 640 und 480. Die Bits werden über einen Zeiger zur oberen, linken Ecke von *lockedRect* übergeben (*lockedRect.pBits*) und das Format muss mit dem Format des aktuellen Image-Streams übereinstimmen – in diesem Fall demnach RGB.

Das Übertragen der Bild-Daten ist damit abgeschlossen und die Ressourcen können gelöst werden:

```
1     void KinectEngine::processColor()
2     {
3         // ...

4         if(SUCCEEDED(result))
5         {
6             // ...
7             pFrameTexture->UnlockRect(0);
8             mpSensor->NuiImageStreamReleaseFrame
9                         (mColorStreamHandle, &colorFrame);
10        }
11    }
```

Listing 18: Lösen der Textur und des Farbbildstreams

Um Daten aus *processColor()* zu erhalten, muss das Signal noch mit einem Slot verbunden werden. Dazu muss erst die Klasse *VideoWidget* implementiert werden. *VideoWidget* benötigt nur die Funktion *setImage()* zum Anzeigen eines *QPixmap*:

```
1     void VideoWidget::setImage(const QImage& qImage){
2         QPixmap pixmap = QPixmap::fromImage(qImage);
3         pixmap = pixmap.scaled(this->size(), Qt::KeepAspectRatio);
4         setPixmap(pixmap);
5     }
```

Listing 19: Erzeugen eines QPixmap über das gesendete QImage

Die Memberfunktion *setImage()* besteht aus drei Schritten:

1. Laden der QImage-Daten in ein QPixmap
2. Das QPixmap-Objekt auf die Label-Größe skalieren
3. Das QPixmap-Objekt anzeigen.

Nachdem Signal und Slot implementiert wurden, muss noch eine Verbindung über den Aufruf von *QObject::connect()* aufgebaut werden. Ein geeigneter Ort dafür ist innerhalb des Konstruktors der Fensterklasse *QMainWindow*:

```
1     MainWindow::MainWindow(QWidget *parent)
2         : QMainWindow(parent)
3         , ui(new Ui::MainWindow)
4         , kinectThread(new KinectEngine)
5     {
6         ui->setupUi(this);
7         this->setWindowTitle("Kinect Color Camera");
8         connect(kinectThread, SIGNAL(sendColorImage(const QImage&)),
9                 ui->colorFrame, SLOT(setImage(const QImage&)));
10        ui->colorFrame->label = "color frame";
11        kinectThread->start();
12    }
```

Listing 20: Implementierung des MainWindow-Konstruktors

Zeile 8 und 9 in Listing 20 zeigen das Herstellen der Verbindung zwischen dem Signal- und Slot. Das Objekt, welches das Signal enthält, ist *kinectThread* (ein Objekt der Klasse *KinectEngine*), während der Slot zu dem Objekt *colorFrame* gehört – der Aufruf *ui->colorFrame* gibt an, dass sich das Label *colorFrame* in dem Layout des Fensters befindet.

Da nun eine Verbindung etabliert wurde, kann der Thread gestartet werden und Daten somit gesendet werden. Abbildung 13 zeigt die Anzeige von *colorFrame*.

```
1     typedef struct _NUI_IMAGE_FRAME {
2         LARGE_INTEGER liTimeStamp;
3         DWORD dwFrameNumber;
4         NUI_IMAGE_TYPE eImageType;
5         NUI_IMAGE_RESOLUTION eResolution;
6         INuiFrameTexture *pFrameTexture;
7         DWORD dwFrameFlags;
8         NUI_IMAGE_VIEW_AREA ViewArea;
9     } NUI_IMAGE_FRAME;
```

Listing 21: Definition der Struktur NUI_IMAGE_FRAME

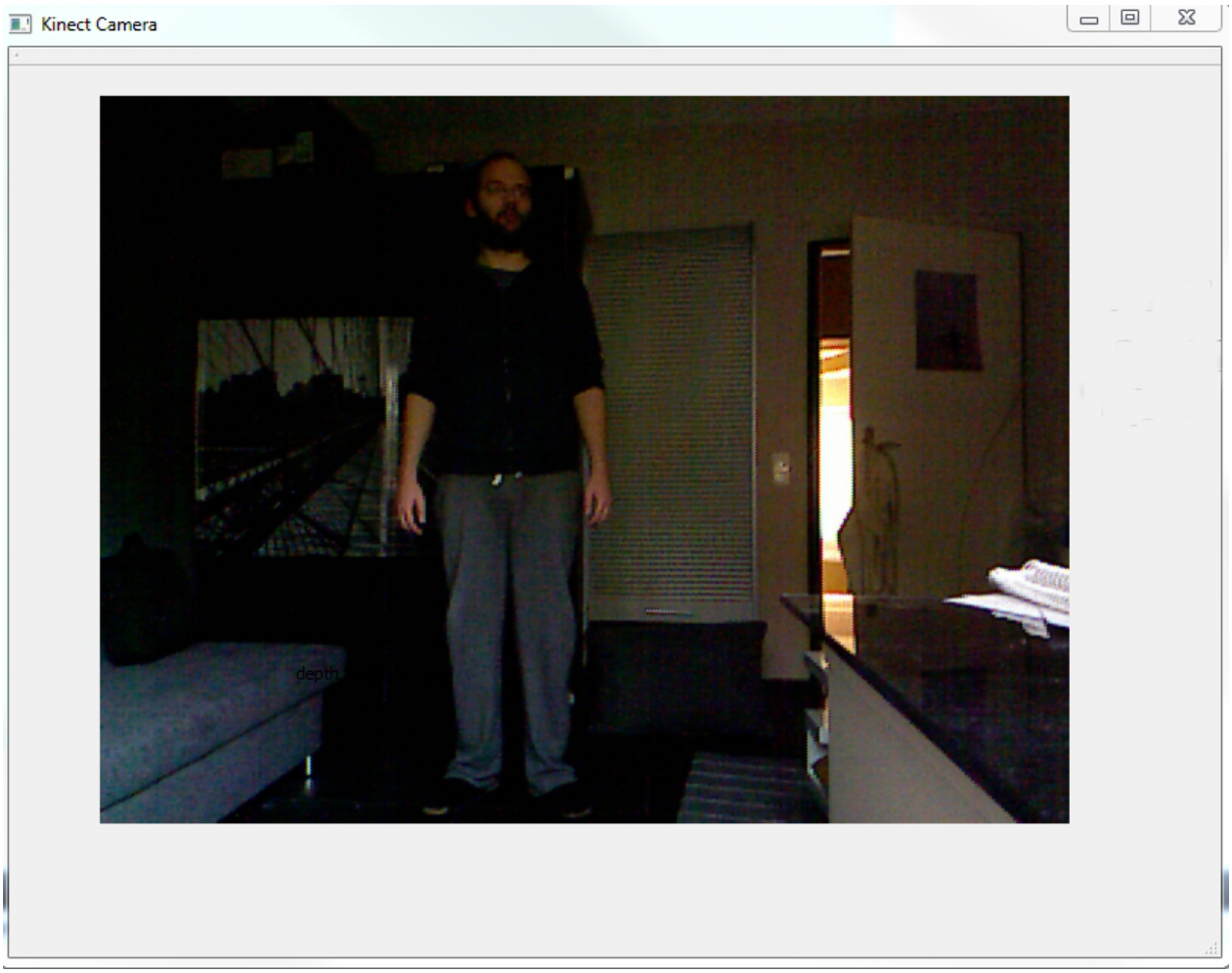

Abbildung 13: Color-Stream-Output

Das Abhandeln des Farbbild-Streams ist hiermit beendet und es kann zu der Behandlung des Tiefensensors übergegangen werden.

4.2.2 Tiefenbild-Stream

Das Einrichten eines Tiefenbild-Streams geschieht analog zu dem Farbbild-Stream. Dazu wird zunächst die Initialisierung dargelegt, allerdings lediglich anhand der Unterschiede zum vorherigen Unterkapitel. Gemeinsamkeiten des Codes, die Farbbild-Stream und Tiefenbild-Stream betreffen, werden nicht weiter erläutert.

4.2.2.1 Initialisierung

Für die Initialisierung müssen ein paar Änderungen in der *initKinect()*-Funktion vorgenommen werden. Zum Einen muss die Variable *sensorFlags* einem anderen Wert zugewiesen werden, da ein anderer Sensor benutzt wird und zum Anderen muss der entsprechende Image-Stream gestartet werden.

```
1     DWORD sensorFlags = NUI_INITIALIZE_FLAG_USES_DEPTH;

2     // ...

3     result = mpSensor->NuiImageStreamOpen(
4         NUI_IMAGE_TYPE_DEPTH,
5         NUI_IMAGE_RESOLUTION_640x480,
6         0,
7         2,
8         NULL,
9         &mDepthStreamHandle);
10    if(FAILED(result))
11    {
12        qDebug() << "Depth stream could not be opened!";
13        return;
14    }
```

Listing 22: Einrichten des Tiefensensors

Analog zum Handle des Farbsensors wird mit der Membervariable *mDepthStreamHandle* der nötige Handle für den Tiefensensor übergeben (siehe Listing 20, Zeile 9).

Die anderen Parameter beim Aufruf von *NuiImageStreamOpen()* bleiben unverändert, bis auf die Konstante NUI_IMAGE_TYPE_DEPTH, die aussagt, dass der Tiefenbild-Stream initialisiert werden soll.

4.2.2.2 Bildschleife

Anstatt der Funktion *processColor()* wird nun die Funktion *processDepth()* in *run()* aufgerufen. Der Inhalt dieser Funktion weist einige Gemeinsamkeiten mit *processColor()* auf, so kommen

erneut die Strukturen NUI_IMAGE_FRAME und NUI_LOCKED_RECT zum Einsatz.

Der erste Schritt besteht abermals in dem Anfordern eines neues Frames:

```
1     void KinectEngine::processDepth()
2     {
3         NUI_IMAGE_FRAME depthFrame;
4         HRESULT result;
5         result = mpSensor->NuiImageStreamGetNextFrame(
6          mDepthStreamHandle, 0, &depthFrame);
7         // continue ...
8     }
```

Listing 23: Schreiben des nächsten Frames in depthFrame

Sollte der Aufruf von *NuiImageStreamGetNextFrame()* erfolgreich gewesen sein, so folgt der Einsatz des Interfaces *INuiFrameTexture*.

```
1     void KinectEngine::processDepth()
2     {
3         // ...
4
5         if(SUCCEEDED(result))
6         {
7             INuiFrameTexture* pFrameTexture =
                      depthFrame.pFrameTexture;
8
9             result = mpSensor
1                   ->NuiImageFrameGetDepthImagePixelFrameTexture
10                     (mDepthStreamHandle, &depthFrame,
11                      false, &pFrameTexture);
12            if(FAILED(result))
13            {
14                qDebug() << "Depth image pixel texture could not be
                                  obtained!";
15                return;
16            }
17        }
18    }
```

Listing 24: Erzeugung einer Bildtextur anhand von depthFrame

Listing 24 zeigt eine Neuheit auf – die Funktion *NuiImageFrameGetDepthImagePixelFrameTexture()* wurde für das Farbbild nicht verwendet.

Diese Funktion bewirkt, dass auf die Tiefeninformationen in Form von der Struktur NUI_DEPTH_IMAGE_PIXEL zugegriffen werden kann; vor dem Effekt der Funktion liegen die Daten einer Ansammlung des Datentyps USHORT (ganze 16-Bit Zahlen ohne Vorzeichen) vor[15]. Der Grund für diese Konvertierung wird im darauffolgenden Abschnitt deutlich.

```
1     void KinectEngine::processDepth()
2     {
3         // ...
4
5         if(SUCCEEDED(result))
6         {
7             // ...
8
9             NUI_LOCKED_RECT lockedRect;
10            pFrameTexture->LockRect(0, &lockedRect, NULL, 0);

11            if(lockedRect.Pitch != 0){
12                BYTE* pRgbCount = mDepthRGBX;
13
14                const NUI_DEPTH_IMAGE_PIXEL* pBufferCount =
15                        reinterpret_cast<const
                            NUI_DEPTH_IMAGE_PIXEL*>(lockedRect.pBits);
16
17                const NUI_DEPTH_IMAGE_PIXEL* pBufferEnd =
18                        pBufferCount + (mWidth * mHeight);
19
20                // ... continue
21            }
22        }
23    }
```

Listing 25: Vorbereitung des Kopier-Vorgangs des Tiefenbuffers

Für die Membervariable *mDepthRGBX* wurde im Konstruktor zuvor für ein Bild mit vier Kanälen Speicher alloziert (*mWidth * mHeight * 4*) . Über den Zeiger *pRgbCount* werden die Buffer-Daten in *mDepthRGBX* geschrieben: Mit Hilfe zweier Zeiger des Typs NUI_DEPTH_IMAGE_PIXEL (in Listing 25 mit *pBufferCount* und *pBufferEnd* bezeichnet), die jeweils einen Start- und Endpunkt eines Kopier-Vorgangs darstellen, können die Tiefendaten extrahiert werden. Die Struktur NUI_DEPTH_IMAGE_PIXEL besitzt nämlich zwei Eigenschaften:

15 Vgl. https://msdn.microsoft.com/en-us/library/nuisensor.inuisensor.nuiimageframegetdepthimagepixelframetexture.aspx , letzter Aufruf: 21.10.15

1. Den Spieler-Index (USHORT *playerIndex*), welcher den Index des Spielers an dem aktuellen Pixel-Wert wiedergibt (es können mehrere Spieler gleichzeitig verfolgt werden und dieser Index dient der Unterscheidungsmöglichkeit). Für die Anzeige eines reinen Tiefenbilds ist dieser Wert irrelevant.
2. Die Tiefe des aktuellen Pixel-Werts (USHORT *depth*). Dieser Wert ist relevant für das Tiefenbild.

Nun kann innerhalb einer Schleife der Kopierprozess gestartet werden:

```
1     void KinectEngine::processDepth()
2     {
3         // ...

4         while(pBufferCount < pBufferEnd){
5
6             USHORT depthValue = pBufferCount->depth;
7             BYTE brightness = (depthValue > 4095 || depthValue == 0) ?
                      0 : 255 - (BYTE)(((float)depthValue / 4095.0f) *
                  255.0f);
8
9             *(pRgbCount++) = brightness;
10            *(pRgbCount++) = brightness;
11            *(pRgbCount++) = brightness;
12
13            ++pRgbCount;
14
15            ++pBufferCount;
16        }
17        // ... Bild senden
18    }
```

Listing 26: Extrahieren und Verarbeiten der Tiefeninformation

Listing 26 zeigt, wie über die Variable *depthValue* auf die zuvor erwähnte Tiefeneigenschaft der Struktur NUI_IMAGE_DEPTH_PIXEL zugegriffen wird. Der Wert *brightness* soll dem Sensor nahe gelegene Objekte einen Weißwert zuweisen, während weit entfernte Objekte schwarz erscheinen sollen. Dafür wird innerhalb einer Bedingung abgefragt, ob der aktuelle Tiefenwert über dem Maximalwert liegt – trifft dies zu, so bekommt der Pixel an dieser Stelle den Wert 0 (schwarz). Allen anderen Werten wird ein Wert zwischen 0 und 255 zugewiesen, sodass eine Interpolation zwischen hellem Weiß und einem Schwarzwert stattfinden kann. Der Toleranzwert von 4095 für die Abfrage rührt daher, dass für die Tiefeninformationen 12 Bits zur Verfügung

stehen. Damit sind [2^{12} = 4096] Werte möglich, die die [4096mm ~ 4m] Tiefenreichweite des Kinect-Sensors widerspiegeln[16].

Wurde *brightness* ermittelt, so müssen diese Werte in die drei Kanäle geschrieben werden (jeweils Blau, Grün und Rot). Daraufhin wird über den Befehl *++pRgbCount* der vierte, reservierte Kanal übersprungen, da augenblicklich kein Alpha-Kanal benötigt wird.

Das Bild kann nun gekapselt und gesendet werden:

```
1     QImage image((const unsigned char*) mDepthRGBX,
2                  mWidth, mHeight, QImage::Format_RGBX8888);
3     emit sendDepthImage(image);
```

Listing 27: Senden des Signals

Die Verbindung von Signal und Slot kann aus dem vorigen Kapitel übernommen werden, da keine Veränderungen diesbezüglich vorgenommen werden müssen.

Abbildung 14 zeigt das empfangene Tiefenbild der Kinect.

16 Vgl. https://msdn.microsoft.com/en-us/library/jj131028.aspx , letzter Aufruf: 24.10.15

Abbildung 14: Depth-Stream-Output

4.2.3 Kombinierte Anzeige

Die Kapitel 4.2.1 und 4.2.2 haben gezeigt, wie der RGB-Sensor beziehungsweise der Tiefen-Sensor der Kinect einzeln genutzt werden kann. Darauf aufbauend bietet es sich an, eine kombinierte Anzeige der beiden Sensoren zu demonstrieren

Die nötigen Schritte hierfür sind trivial und ergeben sich aus der Kombination der bereits erfolgten Arbeit der vorangegangen Unterkapitel:

1. Erweitern der Sensor-Flags innerhalb von *initKinect()*. Sollen mehrere Sensoren initialisiert werden, so sind die entsprechenden Konstanten logisch zu vereinigen (siehe Listing 28).

2. Initialisieren beider Image-Streams (sowohl NUI_IMAGE_TYPE_COLOR als auch NUI_IMAGE_TYPE_DEPTH).
3. Ausführen beider Funktionen *processColor()* und *processDepth()* innerhalb von *run()*.
4. Senden zweier Signale an zwei verschiedene Slots (siehe Listing 29).
5. Anzeige durch zwei Objekte der Klasse *VideoWidget*.

```
1     DWORD sensorFlags = NUI_INITIALIZE_FLAG_USES_COLOR |
2                         NUI_INITIALIZE_FLAG_USES_DEPTH;
```

Listing 28: Vereinigung der Sensor-Flags

```
1     connect(kinectThread, SIGNAL(sendColorImage(const QImage&)),
2                 ui->colorFrame, SLOT(setImage(const QImage&)));
3     connect(kinectThread, SIGNAL(sendDepthImage(const QImage&)),
4                 ui->depthFrame, SLOT(setImage(const QImage&)));
5     ui->colorFrame->label = "color frame";
6     ui->depthFrame->label = "depth frame";
```

Listing 29: Änderungen im Konstruktor von MainWindow

Abbildung 15 zeigt die kombinierte Anzeige von Tiefen- und Farbbild-Stream.

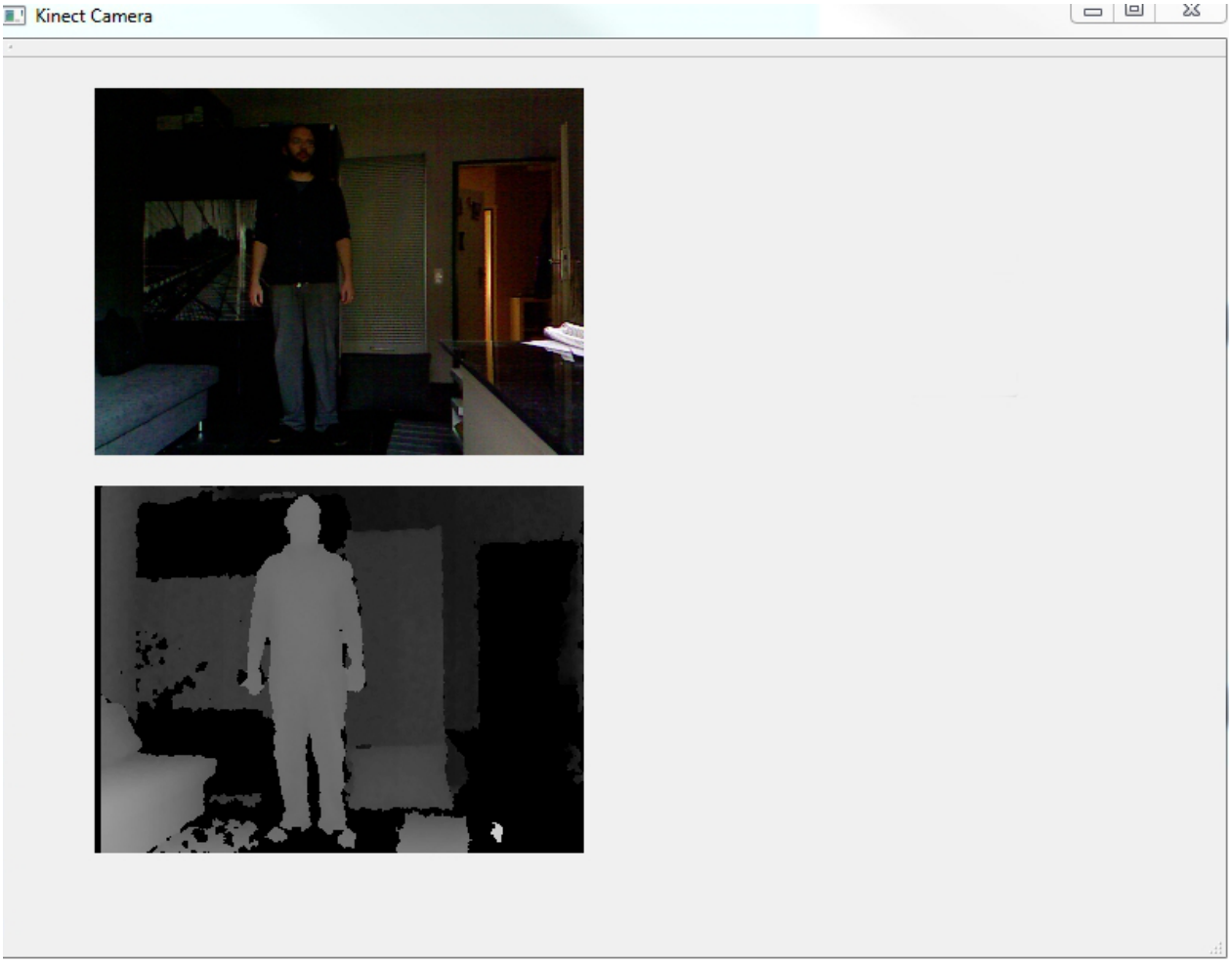

Abbildung 15: Kombinierte Anzeige von Color und Depth

4.3 Ansteuerung des Motors

Mit der Ansteuerung des Kinect-Motors wird dem Benutzer ermöglicht, die Kinect im Bereich von [+27, -27] Grad zu neigen (gemessen an der Horizontalen).

Für die Desktop-Applikation ist es sinnvoll, diese Funktion über eine Interaktion mit Menü-Buttons zu lösen. Dazu müssen zunächst über den QDesigner Button-Objekte in das Fenster integriert werden. In diesem Fall fiel die Wahl auf den Typ *QPushButton*, welcher einen regulären, anklickbaren Button repräsentiert.

Die Implementation des *QPushButton* ist simpel: Listing 30 zeigt, wie das vordefinierte Signal

clicked() des *QPushButton* implementiert wird und die zugehörigen Funktionen von *kinectThread* aufgerufen werden.

```
1     void MainWindow::on_upButton_clicked()
2     {
3         kinectThread->motorTiltUp();
4     }

5     void MainWindow::on_downButton_clicked()
6     {
7         kinectThread->motorTiltDown();
8     }

9     void MainWindow::on_centerButton_clicked()
10    {
11        kinectThread->motorCenter();
12    }
```

Listing 30: Implementation von drei QPushButton-Objekten

Die Funktion *on_buttonName_clicked()* ist dabei der zugehörige Slot zum Signal *clicked()*, welches durch einen Mausklick auf den Button gesendet wird und die jeweilige Funktion in *kinectThread* auslöst.

Die drei Buttons haben unterschiedliche Funktion:

- *Up*-Button:
  - Vergrößert den Winkel um drei Grad.
- *Down*-Button:
  - Verringert den Winkel um drei Grad.
- *Center*-Button:
  - Setzt den Winkel auf den Nullwinkel.

Die Klasse *KinectEngine* enthält die Funktionen, welche auf den Motor zugreifen und über die Slots aufgerufen werden. Listing 31 dokumentiert die Implementation der Funktion *KinectEngine::motorTiltUp()*, welche für die Erhöhung des Winkels zuständig ist.

Die Implementation der zwei anderen Funktionen, *KinectEngine::motorTiltDown()* sowie

KinectEngine::motorCenter(), geschieht analog dazu und kann unter *KinectEngine.cpp* eingesehen werden.

```
1     void KinectEngine::motorTiltUp()
2     {
3         if(!mStopped){
4             if(mAngle + mStep <= NUI_CAMERA_ELEVATION_MAXIMUM){
5                 mAngle += mStep;
6                 mpSensor->NuiCameraElevationSetAngle(mAngle);
7             }
8             else{
9                 qDebug() << "Maximum angle reached!
10                             Cannot tilt up further!";
11                return;
12            }
13        }
14    }
```

Listing 31: Implementation von motorTiltUp()

Die Membervariable *mAngle* enthält den aktuellen Winkel, während *mStep* die Schrittgröße (beträgt hierbei den Wert drei, sodass neun mal der Winkel erhört werden kann[17]), mit der *mAngle* verändert wird. Die Funktion *NuiCameraElevationSetAngle()* setzt den Winkel, falls das Maximum nicht überschritten würde.

Die Anzeige der Buttons ist unter Abbildung 16 zu sehen – die Buttons wurden dem Fenster, welches die kombinierte Anzeige der Image-Streams (4.2.3) enthält, hinzugefügt.

17 Das ergibt einen Winkel von 27 Grad: 9 * 3 = 27. Dies entspricht dem absoluten, maximalen Neigungswinkel des Motors.

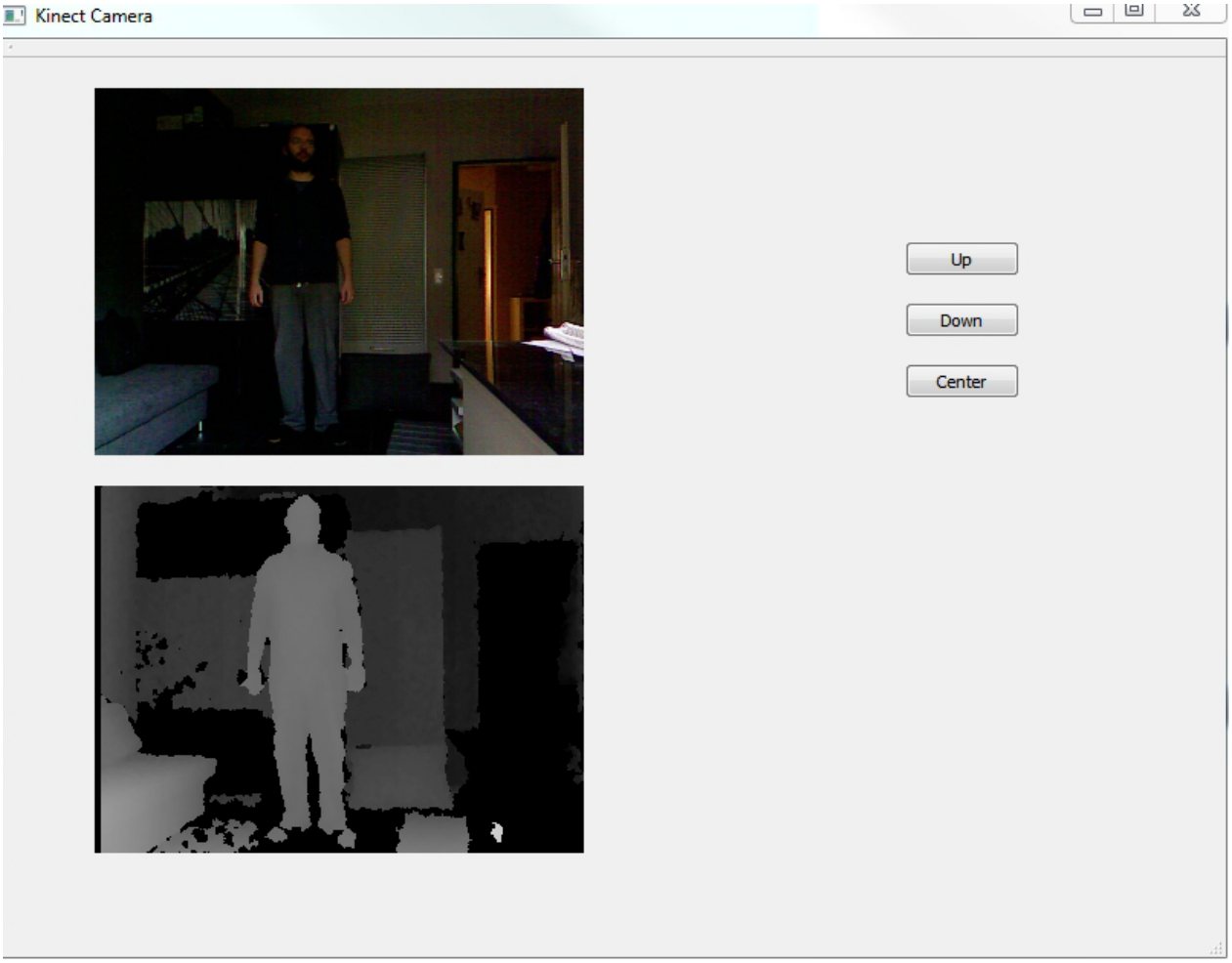

Abbildung 16: Kombinierte Anzeige mit Buttons

4.4 Kinect und GLWidget

Nachdem die vorangegangen Unterkapitel den Programmierteil ohne Verwendung von OpenGL darlegten, soll nun die Implementation von OpenGL über das *GLWidget* erfolgen.

Der Anfang dieses Kapitels besteht in der Übertragung des Farbbildes der Kinect an OpenGL, analog zu Kapitel 4.2.1 . Im Anschluss daran wird der Skelett-Stream der Kinect erstmalig genutzt und mit dessen Hilfe das von der Kinect erfasste Skelett über OpenGL gezeichnet. Der letzte Teil und damit der abschließende Part der Programmierung besteht im Zeichnen der Kleidungstexturen.

4.4.1 Farbbild-Stream

Um innerhalb von *GLWidget* Daten der Kinect empfangen zu können, müssen einige Vorbereitungen getroffen werden:

1. Einstellung der Shader und des Shader-Programms
2. Definieren der Buffer
3. Verbinden neuer Signale und Slots

Diese drei Vorgänge werden im Folgenden dargelegt.

4.4.1.1 Die Shader-Programmierung

Wie in 3.2.1 vermerkt, sollen alle Koordinaten der OpenGL-Projektion zwischen [0, 1] liegen. Aufgabe der Shader ist es nun, mit Hilfe der Buffer die übergebenen Positionen und Texturkoordinaten abzubilden und für jeden Objektpunkt (Vertex) durchzuführen.

Für diesen Zweck reicht es aus, jeweils einen Vertex- und einen Fragment-Shader einzusetzen.

Der Vertex-Shader ist in Abbildung dargestellt.

```
1     uniform mat4 mvp_matrix;
2     layout(location=0) in vec4 in_position;
3     layout(location=1) in vec2 in_texcoord;

4     out vec2 ex_texcoord;

5     void main(){
6
7         gl_Position = mvp_matrix * in_position;
8         ex_texcoord = in_texcoord;
9     }
```

Listing 32: Implementierung des Vertex-Shaders

Über den Vertex-Shader werden über die uniform-Variable *mvp_matrix* alle Matrixabbildungen durchgeführt (Model-View-Projektion). Die Variable *gl_Position* ist vorstellbar als eine Art globale Variable innerhalb des Shaders – sie bestimmt die Endposition für jeden eingehenden Vertex.

Die mit als *layout* markierten Variablen *in_position* und *in_texcoord* stehen für die eingehende Position und Texturkoordinate. Über die Anweisung *layout(location=X)* sind diese Variablen „von außen“ (sprich außerhalb der Shader-Datei, im Quellcode des Hauptprogramms) erreichbar und die Buffer können somit mit Positions- und Texturkoordinaten geladen werden, indem sie dieses Attribut über den Wert *'X'* aktivieren.

Neben diesen beiden eingehenden Variablen besitzt der Vertex-Shader auch eine ausgehende Variable. Diese ist in Listing 32 mit *ex_texcoord* bezeichnet und ihr wird lediglich der eingehende Wert *in_texcoord* zugewiesen – was bedeutet, dass der Vertex-Shader keine weitere Manipulationen an den Texturkoordinaten vornehmen soll und sie unverändert an den Fragment-Shader weitergegeben werden (vergleiche Listing 33).

```
1     uniform sampler2D texture;
2     in vec2 ex_texcoord;

3     layout(location=0) out vec4 out_color;

4     void main()
5     {
6         out_color = texture2D(texture, ex_texcoord);
7     }
```

Listing 33: Implementierung des Fragment-Shaders

Ähnlich wie der Vertex-Shader mit der uniform-Variable *mvp_matrix* (vergleiche Listing 32) besitzt der Fragment-Shader eine uniform-sampler2D-Variable *texture*. Diese wird außerhalb des Shader-Kontext genutzt, um die aktuelle Textur zu binden und sie zu manipulieren.

Über die Funktion *texture2D()*, welche eine vordefinierte Shader-Funktion ist, werden durch die eingehende Variable *ex_texcoord* (aus dem Vertex-Shader stammend) und *texture* für jedes Fragment die Texturwerte interpoliert und mit *out_color* ausgegeben. Da *out_color* über eine *layout-location* verfügt, ist *out_color* die Standardfarbe für jedes Fragment.

Nachdem die Shader kreiert wurden, müssen sie nun noch über ein Shader-Programm geladen und verlinkt werden. Dafür existiert in *GLWidget* die Funktion *initShaders()*:

```
1     void GLWidget::initShaders()
2     {
3         // Load vertex shader
4         if (!mShaderProgram
1           .addShaderFromSourceFile(QOpenGLShader::Vertex,
5                                                  ":/vshader.glsl"))
6         {
7             qDebug() << "Could not compile vertex shader! Exiting
                          program!";
8             close();
9         }

10        // Load fragment shader
11        if (!mShaderProgram
1           .addShaderFromSourceFile(QOpenGLShader::Fragment,
12                                                 ":/fshader.glsl"))
13        {
14            qDebug() << "Could not compile fragment shader! Exiting
                          program!";
15            close();
16        }

17        // Link shader pipeline
18        if (!mShaderProgram.link())
19        {
20            qDebug() << "Could not link shader pipeline! Exiting
                          program!";
21            close();
22        }

23        // Bind shader pipeline
24        if (!mShaderProgram.bind())
25        {
26            qDebug() << "Could not bind shader pipeline! Exiting
                          program!";
27            close();
28        }
29    }
```

Listing 34: Abhandlung der Shader und des Shader-Programms

4.4.1.2 Generierung der Buffer

Mit der Fertigstellung des Shader-Programms können nun die Buffer initialisiert werden.

In der Klasse *GeometryEngine* sind für das Empfangen der Farbbilddaten der Kinect die nötigen Parameter hinterlegt. Die Funktion *GeometryEngine::initKinectPlane*() lädt einen Buffer mit Vertex-Daten des Typs *VertexData* (vergleiche Listing 36). Dies ist eine eigen entworfene Struktur, die über die Datei *structs.h* eingebunden wurde. Mit Hilfe dieser Struktur ist es möglich, die benötigten Vertex-Informationen zu bündeln. In dieser Anwendung werden nur die Position und Texturkoordinaten benötigt, daher enthält *VertexData* nur diese beiden Parameter (siehe Listing 35).

```
struct VertexData

{
    QVector3D position;
    QVector2D texCoord;
};
```

Listing 35: Die Struktur VertexData

```
1     void GeometryEngine::initKinectPlane()
2     {
3         VertexData vertices[] = {
4             {QVector3D(0.0f, 0.0f, 1.0f), QVector2D(0.0f, 1.0f)},  // v0
5             {QVector3D(1.0f, 0.0f, 1.0f), QVector2D(1.0f, 1.0f)},  // v1
6             {QVector3D(1.0f, 1.0f, 1.0f), QVector2D(1.0f, 0.0f)},  // v2
7             {QVector3D(0.0f, 1.0f, 1.0f), QVector2D(0.0f, 0.0f)},  // v3
8         };
9
10        mKinectImageVBO.bind();
11        mKinectImageVBO.allocate(vertices, 4 * sizeof(VertexData));
12        mKinectImageVBO.release();
13    }
```

Listing 36: Die Funktion initKinectPlane()

Die Variable *vertices[]* enthält vier Vertices, jeweils bestehend auf einem Positionswert und einer Texturkoordinate (damit ist *vertices[]* ein sogenanntes *Interleaved Array*, da mehrere Vertex-Informationen in einem Array gespeichert sind). Dabei erfolgte die Zuordnung in der Form, dass die Positionswerte mit einer Y-Koordinate von 0 den Texturkoordinaten mit einem Y-Wert von 1 zugeordnet wurden. Kapitel 2.3.4 referierte unter dem Punkt QOpenGLTexture bereits über die

Tatsache, dass OpenGL Bilder über die untere linke Ecke aufspannt (0, 0), während Formate wie QImage oder auch die Bit-Daten der Kinect den Startpunkt in der oberen linken Ecke (0, 1) festlegen. Dadurch ergibt sich eine vertikale Spiegelung des Bildes. Durch die Zuordnung in *vertices[]* wird dem entgegengewirkt und die Textur sozusagen gespiegelt aufgetragen (vergleiche Abbildung 17).

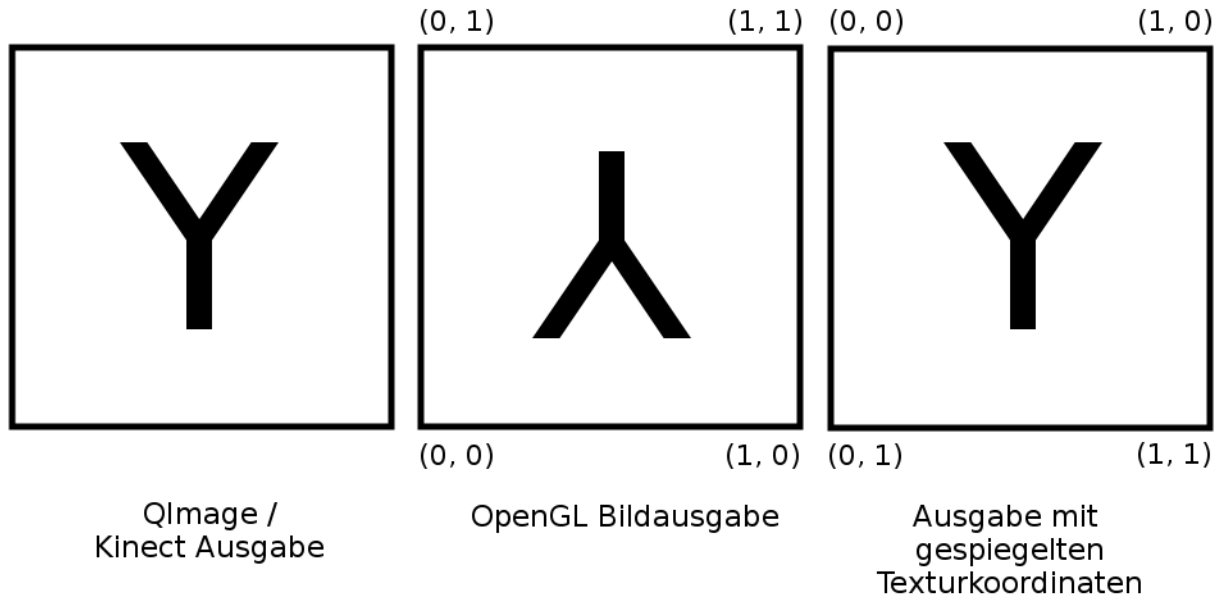

Abbildung 17: Spiegelung des eingehenden Bildes

Die Buffer sind nun initialisiert und es kann begonnen werden, ein Signal der Kinect zu empfangen.

4.4.1.3 Senden und Empfangen des Kinect-Signals

Das Initialisieren der Kinect und das Einstellen des Farbstreams wird aus 4.2.1 übernommen. Listing 37 zeigt, wie innerhalb von *processColor()* das Signal *sendColorStream()* gesendet wird, welches einen Zeiger auf die Bits des Farbbildes an den Slot *updateColorStreamTexture()* übergibt.

```
1     void KinectEngine::run()
2     {
3         while(!mStopped){
4             processColor();
5         }
6     }

7     void KinectEngine::processColor()
8     {
9         // ...
10        if(lockedRect.Pitch != 0){
11            emit sendColorStream((const unsigned char*)
                                        lockedRect.pBits);
12        }
13        // ...
14    }
15
16    MainWindow::MainWindow(QWidget *parent)
17        // ...
18    {
19        ui->setupUi(this);
20        this->setWindowTitle("Kinect Camera with OpenGL");
21        connect(kinectThread,
22        SIGNAL(sendColorStream(const unsigned char*)), ui->glFrame,
23        SLOT(updateColorStreamTexture(const unsigned char*)));
24        kinectThread->start();
25    }
```

Listing 37: Senden des Signals sendColorStream()

Der Slot *updateColorStreamTexture()*, welcher das Signal empfängt, wird in *GLWidget* implementiert.

```
1     void GLWidget::updateColorStreamTexture(const unsigned char*
            videoData)
2     {
3         if(!mTexturesReady)
4         {
5             glGenTextures(1, &mTexture);
6             glActiveTexture(GL_TEXTURE0);
7             glBindTexture(GL_TEXTURE_2D, mTexture);
8             glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR);
9             glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                  GL_LINEAR);
10            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, mWidth, mHeight, 0,
                  GL_BGRA, GL_UNSIGNED_BYTE, &videoData[0]);
11            mTexturesReady = true;
12        }
13        glBindTexture(GL_TEXTURE_2D, mTexture);
14        glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, mWidth, mHeight,
                  GL_BGRA, GL_UNSIGNED_BYTE, &videoData[0]);
15    }
```

Listing 38: Implementierung des Slots updateColorStreamTexture()

Damit eine Bild angezeigt werden kann, muss zunächst eine Textur für diesen Zweck generiert und daraufhin gebunden werden. Listing 38 demonstriert, wie dies über eine Reihe von nativen OpenGL-Befehlen realisiert wird:

- glGenTextures(1, &mTexture):
  - Erzeugt *1* Bezeichner und legt diesen in der Membervariable *mTexture* des Typs GLuint (ein typedef für unsigned int, welcher Plattformunabhängigkeit ermöglicht) ab. Dieser Bezeichner (*mTexture*) ist ab diesem Zeitpunkt für die Verknüpfung durch *glBindTexture()* mit Texturobjekten reserviert[18].
- glActiveTexture(GL_TEXTURE0):
  - Aktiviert die Textureinheit, die als Argument übergeben wurde (hier GL_TEXTURE0). Eine bildhafte Analogie zu dieser Funktion ist ein Bildrahmen, in den ein Bild gehängt werden kann.
- glBindTexture(GL_TEXTURE_2D, mTexture).
  - Bindet einen Texturbezeichner (*mTexture*) an die aktive Textureinheit und bestimmt

18 Vgl. Virag, Gerhard (2012) : „Grundlagen der 3D-Programmierung“, S. 663

beim erstmaligen Bindungsvorgang den Typ der Textur (GL_TEXTURE_2D). In diesem Fall eine zweidimensionale Textur. Diese Funktion wird üblicherweise stets in Kombination mit *glActiveTexture()* aufgerufen. Eine bildhafte Analogie zu dieser Funktion ist ein Bild, welches in einen Bildrahmen gehängt wird.

- glTexParameteri():
  - Setzt bestimmte Parameter für die Textur. In diesem Fall Pixel-Filter.
- glTexImage2D():
  - Spezifiziert ein zweidimensionales Texturbild mit den angegebenen Parametern, wie die Dimensionen des Bildes und dem übergebenen Zeiger auf das erste Bit des empfangenen Kinect-Farbbilds.

Nach Abschluss dieser Befehle wird *mTexturesReady* auf *true* gesetzt, sodass der Prozess nicht erneut durchlaufen wird. Bei jedem erneuten Aufrufen von *updateColorStreamTexture()* wird die Textur lediglich gebunden und mit *glTexSubImage2D()* eine native OpenGL-Funktion aufgerufen, die, im Gegensatz zu *glTexImage2D()* die Parameter für die Textur nicht erneut setzt, sondern nur den Inhalt aktualisiert (die Bits). Somit ist *glTexSubImage2D()* schneller als *glTexImage2D()*.

Nach dem Aufruf von *updateColorStreamTexture()* wird jedoch noch kein Bild angezeigt, da die Daten bisher über keinen finalen Zeichenbefehl gerendert wurden.

4.4.1.4 Rendern der Farbbild-Textur

Die Aufgabe des Renderns übernimmt die Funktion *GeometryEngine::drawKinectPlane()*, die innerhalb von *GLWidget::paintGL()* aufgerufen wird.

```
1     void GLWidget::paintGL()
2     {
3         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
4         glEnable(GL_TEXTURE_2D);
5         glActiveTexture(GL_TEXTURE0);
6         glBindTexture(GL_TEXTURE_2D, mTexture);

7         QMatrix4x4 modelviewMatrix;
8         modelviewMatrix.setToIdentity();
9         mShaderProgram.setUniformValue("mvp_matrix", mProjectionMatrix
                  * modelviewMatrix);
10        mShaderProgram.setUniformValue("texture", 0);

11        if(mTexturesReady)
12            mpGeometries->drawKinectPlane(&mShaderProgram);

13        glBindTexture(GL_TEXTURE_2D, 0);
14        glActiveTexture(GL_TEXTURE0);
15    }

16    void GeometryEngine::drawKinectPlane(QOpenGLShaderProgram *program)
17    {
18        mKinectImageVBO.bind();
19        program->enableAttributeArray(0);
20        program->setAttributeBuffer(0, GL_FLOAT, 0, 3,
            sizeof(VertexData));

21        program->enableAttributeArray(1);
22        program->setAttributeBuffer(1, GL_FLOAT, sizeof(QVector3D), 2,
            sizeof(VertexData));

23        glDrawArrays(GL_QUADS, 0, 4);

24        program->disableAttributeArray(0);
25        program->disableAttributeArray(1);
26        mKinectImageVBO.release();
27    }
```

Listing 39: Render-Vorgang für die 2D-Textur

Zeile 9 und 10 in Listing 39 legen dar, wie nun extern auf die uniform-Variablen der Shader (4.4.1.1 , Die Shader-Programmierung) zugegriffen wird und den Shadern die finale Abbildungsmatrix (*mProjectionMatrix * modelviewMatrix*) und die ID der aktiven Textureinheit (*0*) übergeben wird.

Innerhalb von *GeometryEngine::drawKinectPlane()* wird der Vertex-Buffer *mKinectImageVBO* gebunden und die Attribute des Vertex-Shaders aktiviert. Dazu muss das jeweils erste Argument

von *enableAttributeArray()* und *setAttributeBuffer()* auf eine gültige layout-location des Vertex-Shaders hinweisen – zur Erinnerung genügt ein Blick auf Listing 32 (Seite 60) um zu überprüfen, dass die layout-location mit dem Wert 0 die eingehende Position darstellt und die layout-location mit dem Wert 1 die eingehende Texturkoordinate.

Die Funktion *setAttributeBuffer()* übergibt die Vertex-Daten des Buffers *mKinectImageVBO* an den Shader weiter. Zum genauen Verständnis der zwei verschiedenen Aufrufe in Listing 39 (Zeile 20 und Zeile 22) muss zunächst die Signatur der Funktion analysiert werden.

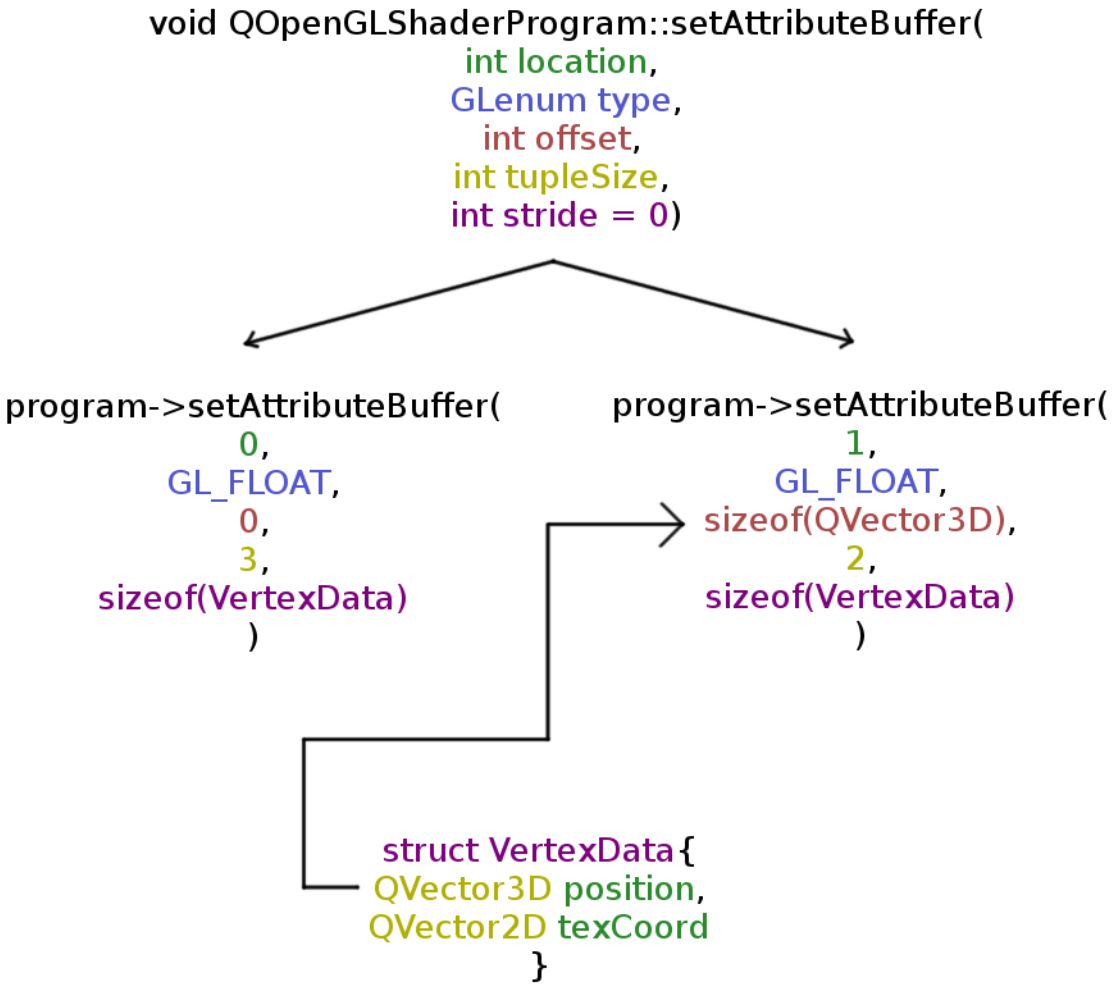

Abbildung 18: Veranschaulichung der Übergabe eines Interleaved Arrays

Da *mKinectVBO* ein Array an gemischten Informationen übergeben wurde (*Interleaved Array*,

vergleiche Listing 36 (Kapitel 4.4.1.2, Seite 63)), gestaltet sich die Extrahierung der verschiedenen Daten über zwei unterschiedliche *setAttributeBuffer()*-Aufrufe als etwas komplexer. Abbildung 18 zeigt, dass die Texturkoordinaten mit einem *offset* von *sizeof(QVector3D)* geladen werden müssen, da in dem Array immer alternierend Position und Texturkoordinate vorliegen. Ein neuer Block (*stride*) beginnt immer wieder bei *sizeof(VertexData)*, da ein Vertex aus einer Position (Typ QVector3D) und einer Texturkoordinate (Typ QVector2D) besteht.

Damit ist das Übergeben der Daten an die Shader abgeschlossen und mit *glDrawArrays(0, 4)* kann nun final gezeichnet werden. Die Funktion benötigt drei Parameter – den Modus (*GL_QUADS*), den Startwert (*0*) sowie die Anzahl der Vertices (*4*). *glDrawArrays()* zeichnet die Vertices in der Reihenfolge, in der sie übergeben wurden.

Eine wichtige Einstellung wurde allerdings bisher noch nicht erwähnt: Die Projektion. Um diese einzustellen, wird die Membervariable mProjectionMatrix mit einer orthogonale Matrix multipliziert, dessen Parameter nach den Erkenntnissen aus 3.2.2 (Aufbau der Projektion) gewählt wurden:

```
1     void MainWidget::resizeGL(int w, int h)
2     {
3         // Determine width and height of the widget
4         mWidth = w;
5         mHeight = h;
6         if(!mHeight)
7             mHeight = 1;

8         // Reset projection
9         mProjectionMatrix.setToIdentity();

10        // Set ortho projection
11        mProjectionMatrix.ortho(0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f);
12        // Move one unit to get behind the near plane
13        mProjectionMatrix.translate(0.0f, 0.0f, -1.0f);
14    }
```

Listing 40: Einstellung der orthogonalen Projektion

Der Render-Vorgang kann nun beginnen und Abbildung 19 zeigt das Ergebnis.

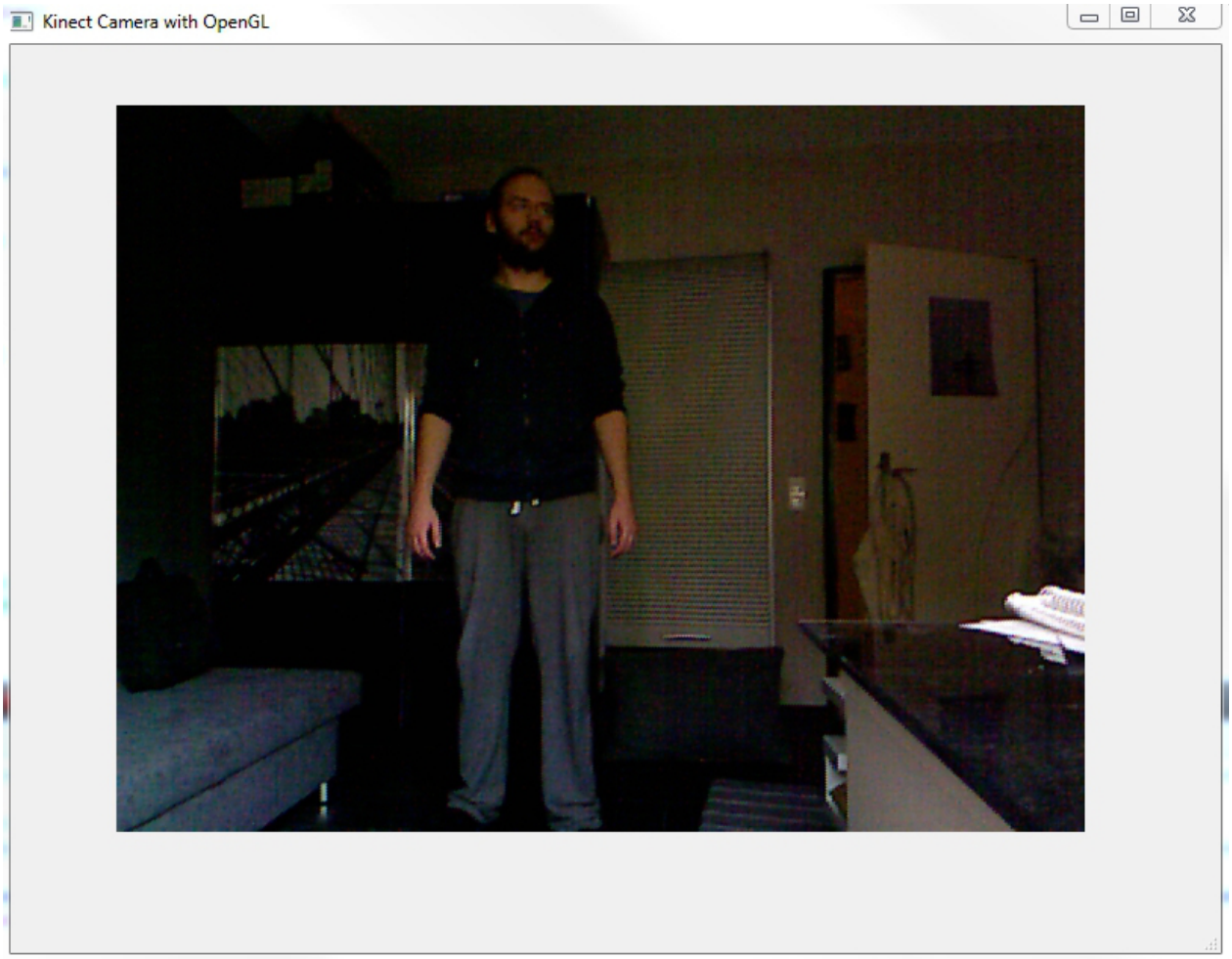

Abbildung 19: Color-Stream-OpenGL-Output

4.4.2 Skeleton-Stream in OpenGL

Bisher wurden in dieser Arbeit der Farb- und Tiefenstream der Kinect benutzt. In diesem Kapitel soll unter Verwendung von OpenGL ein dritter Stream-Typ zum Einsatz kommen, und zwar der Skelett-Stream.

Der Skelett-Stream kann bis zu sechs Personen gleichzeitig erfassen und ist eine Grundvoraussetzung für die finale Programmieraufgabe dieser Arbeit, welche das Wechseln von Kleidungstexturen durch eine Interaktion des Benutzers ist, da der Skelett-Stream erst eine Interaktion des Benutzers ermöglicht.

Genau wie der Farb- und Tiefenstream wird der Skelett-Stream erst initialisiert und es können im Weiteren neue Skeleton-Frames abgerufen werden.

4.4.2.1 Initialisierung des Skelett-Streams

Analog zu der Nutzung der bereits eingesetzten Farb- und Tiefen-Streams müssen zunächst einige Veränderungen an *KinectEngine::initKinect()* vorgenommen werden (ver

```
1     void KinectEngine::initKinect()
2     {
3         // ...
4
5         DWORD sensorFlags = NUI_INITIALIZE_FLAG_USES_COLOR |
                              NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX |
                              NUI_INITIALIZE_FLAG_USES_SKELETON;
6
7         // ...
8
9         HRESULT result;
10        result = mpSensor->NuiImageStreamOpen(
11            NUI_IMAGE_TYPE_DEPTH_AND_PLAYER_INDEX,
12            NUI_IMAGE_RESOLUTION_640x480,
13            0,
14            2,
15            NULL,
16            &mDepthStreamHandle);
17        if(FAILED(result))
18        {
19            qDebug() << "Depth stream could not be opened!";
20            return;
21        }

22        result = mpSensor->NuiSkeletonTrackingEnable(NULL, 0);
23        if(FAILED(result))
24        {
25            qDebug() << "Skeleton stream could not be opened!";
26            return;
27        }
28    }
```

Listing 41: Aktualisieren der Sensoren-Parameter für das Skelett-Tracking

Für den Tiefenstream muss bei Einsetzung des Skelett-Trackings beim Aufruf von

NuiImageStreamOpen() der Parameter NUI_IMAGE_TYPE von NUI_IMAGE_TYPE_DEPTH auf NUI_IMAGE_TYPE_DEPTH_AND_PLAYER_INDEX geändert werden, da der Spieler-Index relevant ist für das Erfassen eines Skeletts (siehe Listing 41, Zeile 11).

Des Weiteren bedarf es das Aufrufen einer bislang unbekannten Kinect-Funktion – *NuiSkeletonTrackingEnable()*. Mit Hilfe dieser Funktion wird das Skelett-Tracking gestartet.

Der nächste Schritt beinhaltet das Implementieren einer neu eingeführten Funktion *KinectEngine::processSkeleton()*, welche innerhalb von *KinectEngine::run()* aufgerufen wird.

4.4.2.2 Skelett-Stream-Schleife

Im Gegensatz zu Kapitel 4.2.1.2 und Kapitel 4.2.2.2 , wo für das Empfangen eines neuen Frame jeweils auf die Struktur NUI_IMAGE_FRAME zurückgegriffen wurde, führt Listing 42 mit der Struktur NUI_SKELETON_FRAME eine neue Struktur ein, die im weiteren Verlauf eingesetzt wird.

```
1     void KinectEngine::processSkeleton()
2     {
3         HRESULT result;
4         NUI_SKELETON_FRAME skeletonFrame = {0};
5         result = mpSensor->NuiSkeletonGetNextFrame(0, &skeletonFrame);
6
7         // continue ...
8     }
```

Listing 42: Abfragen eines neuen Skelett-Frames

Über die Funktion *NuiSkeletonGetNextFrame()* wird der nächste Skelett-Frame eingefordert und in *skeletonFrame* geschrieben[19]. Verlief diese Einforderung erfolgreich, so kann der Frame ausgewertet werden.

Der erste Schritt dieser Auswertung wird über die Funktion *NuiTransformSmooth()* ausgeübt

19 Dieser Funktion kann ein weiteres Argument übergeben werden und zwar die Millisekunden, die bis zu einem erneuten Aufruf gewartet werden soll. Dieser Wert existiert aus Performance-Gründen – das Skeleton-Tracking gehört zu den ressourcenintensiven Funktionen der Kinect SDK. Für diese Anwendung und der eingesetzten Hardware war eine Einstellung des Wertes ungleich 0 jedoch nicht nötig.

(siehe Listing 43), welche den Jitter der Skelett-Positionen zwischen den aufeinanderfolgenden Skelett-Frames reduziert. Dadurch wird eine geglättete Animation des Skeletts gestattet.

```
1     void KinectEngine::processSkeleton()
2     {
3         // ...
4
5         if(SUCCEEDED(result))
6         {
7             mpSensor->NuiTransformSmooth(&skeletonFrame, NULL);
8
9             // continue ...
10        }
11    }
```

Listing 43: Jitter-Reduzierung über NuiTransformSmooth()

Um das Skelett des aktuellen Benutzer zu extrahieren, besteht der übliche Weg darin, mit Hilfe einer äußeren Schleife alle gefunden Skelette zu durchlaufen und die Schleife frühzeitig zu verlassen, sobald die gewünschten Daten isoliert wurden. In einer inneren Schleife werden alle 20 Gelenke des Skeletts durchlaufen und ihre Positionen in einem zweidimensionalen Array gespeichert. Allerdings besitzen die Skelett-Gelenke keine direkten Positionsdaten. Um dennoch Zugriff auf deren Koordinaten zu erlangen, muss die Funktion *NuiTransformSkeletonToDepthImage()* aufgerufen werden.

Danach können mit der Funktion *NuiGetColorPixelCoordinatesFromDepthPixelAtResolution()* die gespeicherten Tiefenbild-Koordinaten der Gelenke in Farbbild-Koordinaten umgewandelt werden. Diese Umwandlung ist essentiell, da später das Skelett beziehungsweise die Kleidungstextur über das Farbbild gelegt werden soll. Würden beispielsweise die Tiefenbild-Koordinaten verwendet, so würde ein relativer Unterschied in der Position zwischen dem auf dem Farbbild sichtbaren Benutzer und den projizierten Kleidungstexturen entstehen, da der Farbsensor und der Tiefensensor unterschiedlich im Gerät positioniert sind (vergleiche Abbildung 1, Kapitel 2.1.1).

```
1     void KinectEngine::processSkeleton()
2     {
3         // ...

4         for(int i = 0; i < NUI_SKELETON_COUNT; ++i)
5         {
6             NUI_SKELETON_DATA& skeleton = skeletonFrame.SkeletonData[i];

7             if(skeleton.eTrackingState == NUI_SKELETON_TRACKED)
8             {
9                 LONG depthX, depthY, colorX, colorY;
10                USHORT depthValue;
11                const float kinectDepthOGLDepthRatio = 1.0f / 4000.0f;
12                for(int j = 0; j < NUI_SKELETON_POSITION_COUNT; ++j)
13                {
14                    NuiTransformSkeletonToDepthImage(
15                         skeleton.SkeletonPositions[j], &depthX, &depthY,
                              &depthValue, NUI_IMAGE_RESOLUTION_640x480);
16                    NuiImageGetColorPixelCoordinatesFromDepthPixelAtResolution(
17                          NUI_IMAGE_RESOLUTION_640x480,
                            NUI_IMAGE_RESOLUTION_640x480,
18                          nullptr, depthX, depthY, depthValue, &colorX, &colorY);
19                    mSkeletonJoints[j][0] = (GLfloat)colorX / (float)mWidth;
20                    mSkeletonJoints[j][1] =
21                              1.0f - (GLfloat)colorY / (float)mHeight;
22                    mSkeletonJoints[j][2] =
                          (GLfloat)NuiDepthPixelToDepth(depthValue)
23                              * kinectDepthOGLDepthRatio;
24                }
25                SkeletonVertexData skeletonVertexData
26                        = {&(mSkeletonJoints[0][0])};
27                emit sendSkeletonJoints(skeletonVertexData);
28                break;
29            }
30        }
31    }
```

Listing 44: Die äußere und innere Schleife von processSkeleton()

Listing 44 (Zeile 19 – 23) zeigt ferner, wie das unter Kapitel 3.2.1 (Umwandlung der Kinect-Koordinaten) besprochene Vorgehen der Koordinatenumwandlung zum Einsatz kommt. Durch die Division der Bilddimensionen werden in *mSkeletonJoints* Werte zwischen [0, 1] gespeichert.

Die Membervariable *mSkeletonJoints* ist vom Typ GLfloat[20][3]. Diese Dimensionen ergeben sich aus der Tatsache, dass der Skelett-Stream 20 (NUI_SKELETON_POSITION_COUNT ist eine Konstante, hinter der sich der Wert 20 verbirgt) Punkte des Körpers erfasst und jeder Punkt jeweils X-, Y- und Z-Koordinate besitzt.

```
1     struct SkeletonVertexData {
2         GLfloat* vertices;
3     }; Q_DECLARE_METATYPE(SkeletonVertexData);
```

Listing 45: Die Struktur SkeletonVertexData

Mit der Struktur *SkeletonVertexData* (Listing 44, Zeile 25) kommt darüber hinaus die zweite Struktur zum Einsatz, die über *structs.h* eingebunden werden kann (siehe Listing 45). Diese Struktur beinhaltet nur eine Eigenschaft und zwar einen Zeiger vom Typ GLfloat. Diese Struktur wird verwendet, um die Daten von *mSkeletonJoints* in einer Struktur zu verpacken und über das Signal *sendSkeletonJoints()* zu versenden. Der Grund, dass für diesen Zweck eine Struktur eingesetzt wird, ist ein Resultat aus dem Meta-Object-Compiler von Qt (*moc*). Diesbezüglich reicht es an dieser Stelle aus zu wissen: Die akzeptierten Argumente von Signal und Slot sind nicht beliebig – neben den Qt-eigenen Datentypen, wie QImage, die problemlos gesendet und empfangen werden können, müssen externe Datentypen im Meta-Object-Compiler angemeldet und registriert werden, damit sie zur Laufzeit erfolgreich gesendet werden können. Dabei gilt die Regel: Die zu registrierenden Metatypen müssen dabei Klassen oder Strukturen sein[20].

Diese Registrierung kann mit dem Array *mSkeletonJoints* nicht funktionieren, deswegen musste ein kleiner Umweg über die Struktur *SkeletonVertexData* gegangen werden.

Die Funktion *KinectEngine::processSkeleton()* ist hiermit abgeschlossen und es folgt nun die Implementierung des zugehörigen Slots *GLWidget::updateSkeletonJoints()*.

4.4.2.3 Empfangen und Rendern der Skelett-Daten

Listing 44 zeigt die Implementierung des Slots *updateSkeletonJoints()*. Es werden die Positionsdaten des Skeletts in den Buffer *mSkeletonVBO* geladen und *mSkeletonReady* auf *true* gesetzt, sodass innerhalb von *paintGL()* das Skelett gezeichnet werden kann.

20 Vgl. http://doc.qt.io/qt-4.8/qmetatype.html#qRegisterMetaType , letzter Aufruf: 25.10.15

```
void MainWidget::updateSkeletonJoints(SkeletonVertexData skeletonData)
{
    // Bind necessary vertex buffer object and allocate/write data
    mSkeletonVBO.bind();
    mSkeletonVBO.allocate(skeletonData.vertices,
                NUI_SKELETON_POSITION_COUNT * 3 * sizeof(GLfloat));
    mSkeletonVBO.release();
    // Drawing process can be started
    mSkeletonReady = true;
}
```

Listing 46: Impementierung des Slots updateSkeletonJoints()

```
1     void MainWidget::paintGL()
2     {
3         // ...
4
5         if(mTexturesReady)
6             mpGeometries->drawKinectPlane(&mShaderProgram);
7
8         glDisable(GL_TEXTURE_2D);
9         glLineWidth(5.0f);
10
11        if(mSkeletonReady)
12           mpGeometries->drawSkeleton(&mShaderProgram, &mSkeletonVBO);
13
14        glDisable(GL_LINE);

15       // ...
16    }
```

Listing 47: Erweiterung von paintGL() um die Skelett-OpenGL-Parameter

Innerhalb von *GLWidget::paintGL()* wird, nachdem die Farbbild-Textur gezeichnet wurde, der 2DTextur-Modus deaktiviert und dafür der Linien-Modus aktiviert (vergleiche Listing 47). Damit wurden alle nötigen Vorbereitungen getroffen und es kann über *GeometryEngine::drawSkeleton()* das Skelett mit Hilfe von Linien gezeichnet werden.

```
1     void GeometryEngine::drawSkeleton(QOpenGLShaderProgram *program,
                        QOpenGLBuffer *buffer)
2     {
3         // Bind buffers
4         buffer->bind();
5         mIndexBufSkeleton.bind();

6         // Locate vertex position data
7         program->enableAttributeArray(0);
8         program->setAttributeBuffer(0, GL_FLOAT, 0, 3, 0);

9         // Draw the skeleton figure with an ordered, indexed draw pattern.
10        glDrawElements(GL_LINES, 38, GL_UNSIGNED_SHORT, (void*)0);

11        // Disable attributes in shader and release buffers
12        program->disableAttributeArray(0);
13        buffer->release();
14        mIndexBufSkeleton.release();
15    }
```

Listing 48: Zeichnungsvorgang des Skeletts

Der in Listing 48 eingesetzte Indexbuffer *mIndexBufSkeleton* wurde bereits zum Initialisieren von *mpGeometries* über die Memberfunktion *GeometryEngine::initSkeleton()* mit Indizes geladen. Dieser Prozess, samt dem eingesetzten Index-Array, kann unter GeometryEngine.cpp eingesehen werden[21].

Bei dem Render-Prozess des Skeletts bedarf es nur der Aktivierung eines Shader-Attributs, da die Linien mit keiner Textur gefüllt werden sollen. Daher werden nur die Positionsdaten übergeben.

Der Befehl *glDrawElements()* ist im Gegensatz zum bisher eingesetzten *glDrawArrays()* ein Zeichnen-Kommando, welches die Nutzung eines Index-Buffers voraussetzt, da die einzelnen Vertices hierbei nicht in der übergebenen Reihenfolge gezeichnet werden, sondern anhand der Sortierung, die durch die im Index-Buffer enthaltenen Indizes vorgegeben wird. Im Fall von GL_LINES wird immer jeweils ein Vertex-Paar über eine Linie verbunden.

Das Skelett-Stream-Kapitel ist damit beendet und Abbildung 20 und 21 zeigen jeweils das Ergebnis; in einem Fall mit gezeichneter Farbbild-Textur und im anderen Fall vor schwarzem Hintergrund.

21 Das Array besitzt 38 Werte und ist daher suboptimal geeignet für ein Listing.

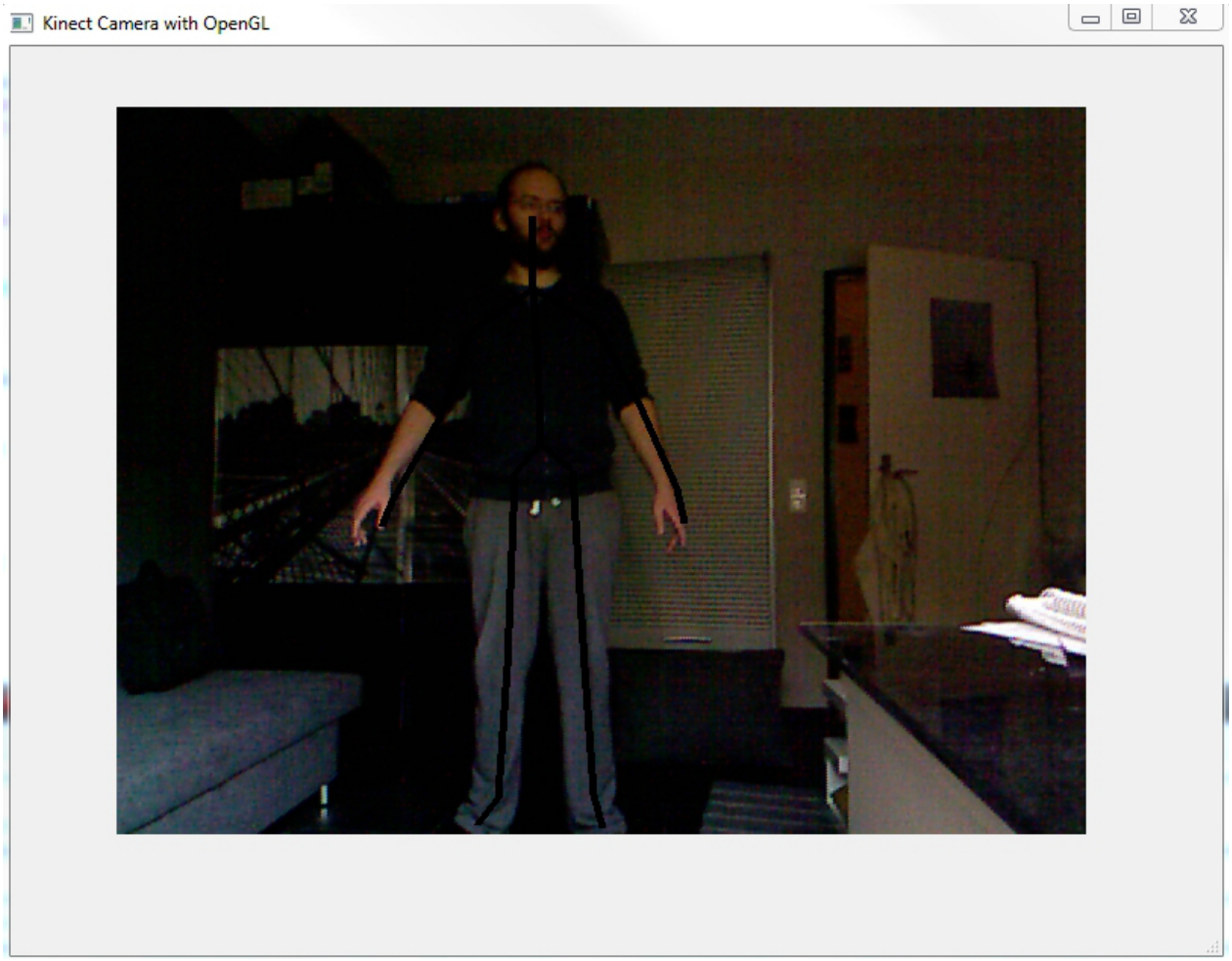

Abbildung 20: Skeleton-Stream-Output

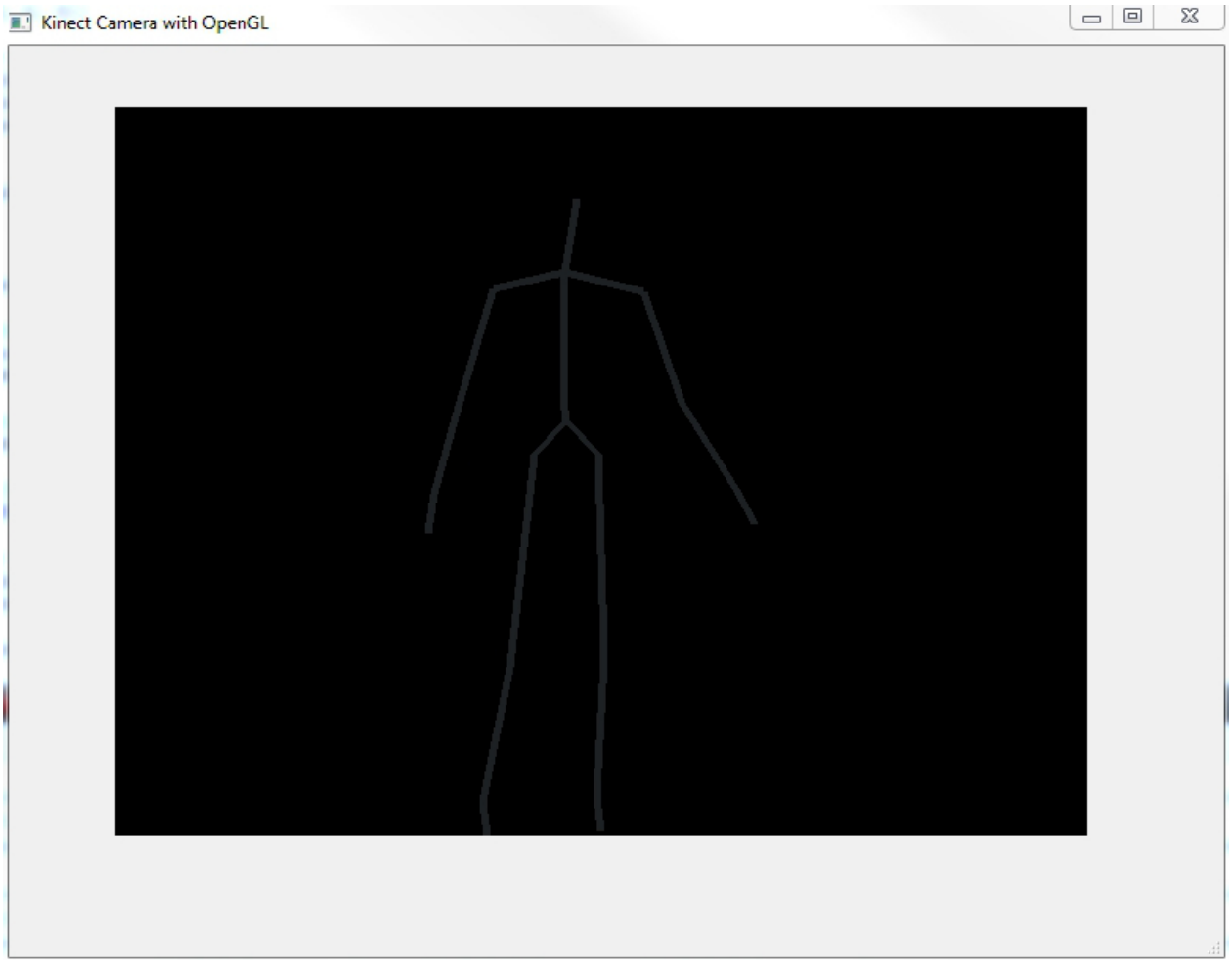

Abbildung 21: Skeleton-Stream-Output vor schwarzem Hintergrund

Der Einsatz des Skelett-Trackings wird im nächsten Kapitel fortgeführt. Das den Programmierteil der Arbeit abschließende Kapitel wird das Laden und Zeichnen von Kleidungstexturen behandeln.

4.4.3 Integration der Kleidungstexturen

Nachdem das vorige Kapitel erfolgreich den Einsatz des Skelett-Tracking der Kinect demonstriert hat, wird das Skelett-Tracking in diesem Kapitel dafür genutzt, um:

1. Den Torso über die zwei Schultergelenke und die zwei Hüftgelenke zu erfassen, damit eine Kleidungstextur auf das Viereck gezeichnet werden kann, das sich aus diesen vier Punkten ergibt.

2. Die zwei Hände zu erfassen, auf dass eine Bewegung der Hände beobachten werden kann.

Eine Änderung an dem Initialisierungsprozess muss also nicht vorgenommen werden. Da jedoch nur eine Teilmenge der erfassbaren Skelett-Gelenke benötigt wird, wird die Funktionsinhalt von *KinectEngine::processSkeleton()* in leicht abgewandelter Form in eine neue Funktion *KinectEngine:.processTorsoAndHands()* übertragen. Aufgrund der Tatsache, dass der prinzipielle Ablauf identisch ist, wird der Ablauf an dieser Stelle nicht erneut erläutert[22].

Dessen ungeachtet ist in *KinectEngine:.processTorsoAndHands()* eine für dieses Kapitel neu eingeführte Funktion *KinectEngine::evaluateHandMovement()* eingebettet, welche über die Membervariablen *mCurrentHandJoints* und *mLastHandJoints* über die Bewegung der Hände Buch führt (siehe Listing 50). Bei einem Überschreiten eines gewissen Schwellwertes wird das Signal *switchShirt()* an den Slot *GLWidget::updateShirtTexture()* gesendet (vergleiche Listing 49).

```
1     void KinectEngine::processTorsoAndHands()
2     {
3         // ...
4
5         // Calculate the distances of the hand positions
6         // (last frame, current frame).
7         evaluateHandMovement();
8         //
9         if(mHandMovementCounterLeft > 1.5)
10        {
11            emit switchShirt("next");
12            // qDebug() << "Swipe registered!";
13            mHandMovementCounterLeft  = 0;
14        }
15        else if(mHandMovementCounterRight > 1.5)
16        {
17            emit switchShirt("previous");
18            mHandMovementCounterRight = 0;
19        }

20        // ...
21    }
```

Listing 49: Einbettung des Gestentests evaluateHandMovement und Senden des Signals

22 Der kommentierte Funktionsrumpf befindet sich in KinectEngine.cpp

```
1     void KinectEngine::evaluateHandMovement()
2     {
3         // Calculate distances
4         qreal currentGapLeft  = qSqrt(qPow(mCurrentHandJoints[0][0]
5         - mLastHandJoints[0][0], 2.0f) + qPow(mCurrentHandJoints[0][1]
          - mLastHandJoints[0][1], 2.0f));
6         qreal currentGapRight = qSqrt(qPow(mCurrentHandJoints[1][0]
          - mLastHandJoints[1][0], 2.0f) + qPow(mCurrentHandJoints[1][1]
          - mLastHandJoints[1][1], 2.0f));

7         // Check if the left hand has made a quick movement or not.
8         if(currentGapLeft < 0.025)
9         {
10            mHandMovementCounterLeft = 0;
11        }
12        else if(currentGapLeft > 0.05)
13        {
14            mHandMovementCounterLeft++;
15        }

16        // Check if the right hand has made a quick movement or not.
17        if(currentGapRight < 0.025)
18        {
19            mHandMovementCounterRight = 0;
20        }
21        else if(currentGapRight > 0.05)
22        {
23            mHandMovementCounterRight++;
24        }

25        // Replace old hand positions with current hand positions.
26        for(int k = 0; k < 2; k++){
27            for(int l = 0; l < 3; l++){
28                mLastHandJoints[k][l] = mCurrentHandJoints[k][l];
29            }
30        }
31    }
```

Listing 50: Implementierung der Funktion evaluateHandMovement()

Damit der Slot *GLWidget::updateShirtTexture()* die Texturen wechseln kann, müssen die Texturen zuerst über die Funktion *GLWidget::initShirtTextures()* initialisiert werden:

```
void MainWidget::initShirtTextures()
{
    // Load image files into OpenGL textures
    for(int i = 0; i < 5; i++)
    {
        mpShirtTextures[i] = new QopenGLTexture(QImage(
            QString(":/shirt_%0.png").arg(i + 1)));
        mpShirtTextures[i]->setMinMagFAilters(QOpenGLTexture::Linear,
            QOpenGLTexture::Linear);
        mpShirtTextures[i]->setWrapMode(QOpenGLTexture::ClampToEdge);
    }
    // Make the first one the current shirt
    mpCurrentTexture = mpShirtTextures[0];
}
```

Listing 51: Initialisieren der Shirt-Texturen

Die Implementierung des Slots wird in Listing 52 dargelegt. Um die aktuelle Shirt-Textur (*mpCurrentTexture*) zu wechseln, wird lediglich der nächste beziehungsweise vorige Array-Wert ausgewählt.

```
1     void MainWidget::updateShirtTexture(const QString& switchTo)
2     {
3         if(switchTo == "next")
4         {
5             mTextureCounter++;
6             if(mTextureCounter > 4)
7                 mTextureCounter = 0;
8             mpCurrentTexture = mpShirtTextures[mTextureCounter];
9         }
10        else if(switchTo == "previous")
11        {
12            mTextureCounter--;
13            if(mTextureCounter < 0)
14                mTextureCounter = 4;
15            mpCurrentTexture = mpShirtTextures[mTextureCounter];
16        }
17    }
```

Listing 52: Wechseln des Shirts in updateShirtTexture()

Der letzte Schritt besteht in dem Aufrufen der Zeichnen-Funktion *GeometryEngine::drawShirtTexture()* innerhalb von *GLWidget::paintGL()*:

```
1     void MainWidget::paintGL()
2     {
3         // ...
4
5         if(mTexturesReady)
6             mpGeometries->drawKinectPlane(&mShaderProgram);
7
8         glEnable(GL_BLEND);
9         glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
10        mpCurrentTexture->bind();
11
12        glClear(GL_DEPTH_BUFFER_BIT);
13
14        if(mShirtReady)
15            mpGeometries->drawShirt(&mShaderProgram, &mShirtVBO);
16
17        glDisable(GL_BLEND);
18        mpCurrentTexture->release();
19
20        // ...
21    }
```

Listing 53: Integrieren der Shirt-Textur in den OpenGL-Kontext

Über die Funktion *glBlendFunc()* werden die Shirt-Textur *mpCurrentTexture* und die Farbbild-Textur überlagert, unter der Berücksichtigung des Alpha-Kanals.

Das Resultat ist in Abbildung 22 festgehalten.

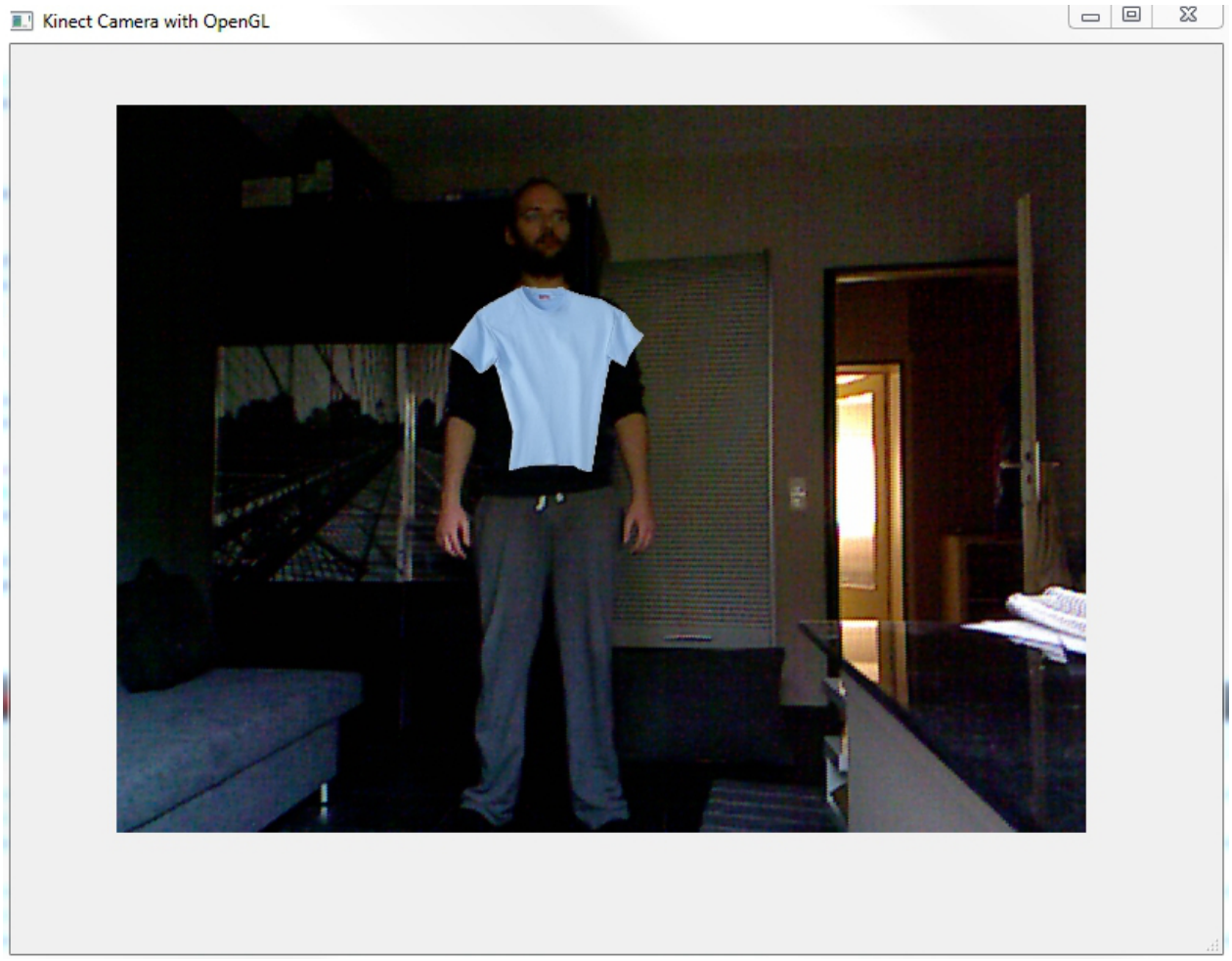

Abbildung 22: Zeichnen eines T-Shirts

5 Schluss

Der Schlussteil dieser Arbeit besteht in einer kurzen Analyse der Ergebnisse anhand der in der Einleitung festgelegten Zielsetzung. Der Schluss mündet daraufhin in einer persönlichen Reflexion.

5.1 Ergebnis

Um das Ergebnis dieser Arbeit zu messen, soll noch einmal die ursprünglichen Zielsetzungen rekapituliert werden:

- *„Erfassen (zunächst) einer Person, die sich vor dem Kameraobjekt befindet“*
  - Mit Hilfe des Skeleton-Tracking der Kinect konnte gezeigt werden, wie eine Person erfasst werden kann.
- *„Auslesen der Raumkoordinaten, um eine Projektion zu ermöglichen“*
  - Das Extrahieren und Umwandeln von Koordinaten war ein fundamentaler Bestandteil der Problemlösungen und konnte erfolgreich eingesetzt werden.
- *„Das Ermöglichen einer Interaktion zwischen Person und Programm“*
  - Der finale Teil der Programmierarbeit zeigt einen Weg auf, wie eine Gestensteuerung implementiert werden kann.
- *„Ausgabe einer Anzeige, auf der die Person zu sehen ist“*
  - Über die Klassen *VideoWidget* und *GLWidget* konnten Anzeigen ermöglicht werden, auf denen der Benutzer zu sehen ist.
- *„Projektion von Kleidungstexturen auf die Person“*
  - Zwar wurden die Kleidungstexturen auf fünf T-Shirt-Texturen beschränkt, aber die zugrundeliegende Technik konnte erfolgreich demonstriert werden.

Alles in Allem wurde mit einer Einleitung in die Thematik gefolgt von einer Analyse der verschiedenen Themengebiete über das Konkretisieren eines Konzepts bis hin zur Betrachtung der Programmierlösung letztendlich ein Ausgangspunkt für einen virtuellen Umkleideraum

geschaffen, der ein Fundament darstellt, das beliebig erweitert werden kann und gegebenenfalls von dem Desktop auf ein Kaufhaussystem portiert werden könnte.
Insbesondere durch die Verwendung einer modernen OpenGL-Programmierung durch Buffer und Shader ist ein Gebrauch von detaillierten, dreidimensionalen Kleidungs-Modellen denkbar.

5.2 Reflexion

Durch den gleichzeitigen Einsatz von drei mir bis dato unbekannten Programmierschnittstellen war der Einstieg in die Materie schwieriger als angenommen. Insbesondere das Verständnis und Zusammenspiel von Shadern und Buffern hat recht viel Zeit in Anspruch genommen, da ich bisher auf diesen Gebieten keinerlei Erfahrungen hatte.

So hat es Wochen gedauert, überhaupt ein lauffähiges Fundament zu legen und die parallele Einarbeitung in drei verschiedene APIs mit dem Hintergedanken, diese in Einklang zu bringen, stellt mich vor eine große Herausforderung. So mussten über die Wochen etliche Funktionen gestrichen werden, die ich ursprünglich als Ideen umsetzen wollte: Sprachsteuerung und 3D-Texturen sind Beispiele hierfür; die Bearbeitungsdauer ließ dann mit der Zeit dafür leider keinen Rahmen mehr.

Nichts desto trotz habe ich viel gelernt während dieser Zeit. Begriffe wie Shader sind nicht mehr nur Dinge, die man mal gehört hat, sondern ich habe nun eine klare Vorstellung davon (und größten Respekt vor Menschen, die das Schreiben von komplexen Shadern beherrschen) und könnte mir durchaus vorstellen, in Zukunft mehr in dieser Richtung auszuprobieren.

Ebenso habe ich durch den intensiven Umgang mit Qt ein C++-Framework gefunden, für das ich bereits jetzt viele weitere Ideen im Kopf habe, und auch die Kinect SDK ist mit dem in dieser Thesis behandelten Stoff noch lange nicht ausgereizt.

Abbildungsverzeichnis

Listingverzeichnis

Literaturverzeichnis

[1] Borenstein, Greg (2012): „Making things see“, Make

[2] Catuhe, David (2012): „Programming with the Kinect for Windows Software Development Kit“, Microsoft

[3] Cozzi, Patrick (2012): „OpenGL Insights“, CRC Press

[4] Jean, Jared (2013): „Kinect hacks“, Make

[5] OpenGL (2014): „OpenGL Book“, englisch, abgerufen am 21.10.2015
http://www.openglbook.com

[6] Qt (2015) , letzter Aufruf: 17.10.2015
http://doc.qt.io/qt-5/qopenglwidget.html#details

[7] http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod , letzter Aufruf: 08.10.2015

[8] Qt (2015), letzter Aufruf: 25.10.15
http://doc.qt.io/qt-5/qapplication.html#exec

[9] Microsoft (2013), „Natural User Interface“, letzter Aufruf: 25.10.2015
https://msdn.microsoft.com/en-us/library/hh855352.aspx

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelor-Thesis mit dem Titel:

Entwurf eines virtuellen Umkleideraums mittels Kinect und OpenGL

selbständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z. B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

(Unterschrift)