Konzeptionierung und Entwicklung eines Realtime-Plugin für Autodesk MotionBuilder

Studiengang Medieninformatik

Bachelorarbeit

vorgelegt von

Tim Pascal Lehr
geb. in Gießen

durchgeführt bei
metricminds GmbH & Co. KG, Frankfurt

Referent der Arbeit:	Prof. Dr. Cornelius Malerczyk
Korreferent der Arbeit:	M. Sc. Hans Christian Arlt
Betreuer bei metricminds:	Dipl.-Ing. Christoph Schulte

Friedberg, März 2016

Für meine Familie

Danksagung

Mein Dank gilt Prof. Dr. Cornelius Malerczyk und Hans Christian Arlt, die mich tatkräftig beim Verfassen dieser Arbeit unterstützt haben. Besonderer Dank gebührt auch Christoph Schulte, der als Betreuer eine sehr große Hilfe für mich darstellte und viel Zeit investierte um auf meine Fragen einzugehen, Korrektur zu lesen und Schwächen in der Arbeit aufzuzeigen. Großer Dank gilt außerdem Philip Weiss für die Möglichkeit diese Arbeit im Unternehmen metricminds durchzuführen. Ich danke Anas Khiami und Christian Dreher für ihre hilfreichen Ratschläge bei der Entwicklung des Prototypen, sowie Christian Schrod für seine Unterstützung bei der zeitlichen Planung der Arbeit.

Weiterhin gebührt großer Dank Laetitia Rezay, Teresa Eisinger und meiner Familie, die zahlreiche Stunden Korrektur gelesen und meine Motivation über den Verlauf dieser Arbeit kontinuierlich gestärkt haben. Abschließend bedanke ich mich bei allen metricminds Mitarbeitern für ihre Unterstützung, insbesondere bei den Teilnehmern der Prototypen-Evaluation.

Selbstständigkeitserklärung

Ich erkläre, dass ich die eingereichte Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Friedberg, März 2016

Tim Pascal Lehr

Sperrvermerk

Die vorgelegte Bachelorarbeit basiert auf internen, vertraulichen Daten und Informationen des Unternehmens metricminds GmbH & Co. KG. In den Quellcode der entwickelten Software dürfen Dritte, mit Ausnahme der Gutachter und befugten Mitgliedern des Prüfungsausschusses, ohne ausdrückliche Zustimmung des Unternehmens und das Verfassers keine Einsicht nehmen. Eine Veröffentlichung des Quellcodes ohne ausdrückliche Genehmigung des Verfassers ist vor dem 31. März 2018 nicht erlaubt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Diese Arbeit beschäftigt sich mit der Konzeption, Entwicklung und Evaluation eines Streaming-Plugins für die Realtime-Darstellung in Autodesk MotionBuilder. Die zur Entwicklung und Evaluation notwendige Infrastruktur wird vom Unternehmen *metricminds GmbH & Co. KG* in Frankfurt am Main zur Verfügung gestellt.

1.1 Motivation

Bereits seit Mitte der 1990er Jahre wird Motion Capture Technologie in der Videospielentwicklung eingesetzt. Mit der Einführung von *Motion Capture*-Technologien für Produktion von Spielfilm Animationen, stellten große Hollywood-Produktionen wie *The Lord of the Rings*[1] das enorme Potenzial der Technik auch für den Rest der Unterhaltungsindustrie unter Beweis. In den folgenden Jahren hat sich Motion Capture in allen Bereichen der Animationsbranche als Standard für authentisch menschliche Animationen etabliert. Heutzutage ist Motion Capture aus der Animationsproduktion nicht mehr wegzudenken.

Im Vergleich zu traditioneller Keyframe Animation bietet Motion Capture eine Menge Vorteile: Animationen sind nicht nur authentischer und schneller mit einem hohen Grad an Details produziert, sondern können auch deutlich komplexere Interaktionen zwischen Charakteren enthalten, ohne den Produktionsaufwand enorm zu steigern. Heutzutage gibt es kaum noch eine animationslastige Videospiel- oder Filmproduktion, die ohne Motion Capture auskommt. Besonders interessant für solche Projekte ist die Echtzeit-Darstellung der aufgenommenen Daten am Set, da sie direkt eine erste Qualitätskontrolle von Timing, Staging, Pacing und Acting ermöglicht. Speziell für Motion Capture optimierte Softwarelösungen wie Autodesk MotionBuilder[2] können dabei problemlos mehrere Charaktere mit Echtzeit-Daten animieren, sodass der Regisseur schon direkt am Drehort einen ersten Eindruck von der virtuellen Repräsentation der Darsteller erhält. Mittlerweile kommen auch vermehrt *virtuelle Kamerasysteme* zum Einsatz, die Filmkameras im virtuellen 3D-Raum repräsentieren und digitale Kameraführung aus der Post-Production in die Produktion verlagert.

[1] http://www.imdb.com/title/tt0120737/ (Zuletzt abgerufen: 14.02.2016)
[2] http://www.autodesk.com/products/motionbuilder/overview (Zuletzt abgerufen: 14.02.2016)

Seit einigen Jahrem findet der Begriff *Virtual Production* (siehe Kapitel 2.1) vermehrt Verwendung, welcher den digitalen Produktionsablauf moderner Filmproduktionen beschreibt und dabei von Technologien wie der virtuellen Kamera und der Pre-Visualisierung (siehe Abbildung 1.1) geprägt wurde. Im Gegensatz zu traditionellen Produktionsabläufen ist die Virtual Production deutlich flexibler in ihrem Ablauf, da viele Aufgaben die zuvor erst in der digitalen Post-Production ihrem Platz fanden, über den gesamten Produktionsablauf verteilt werden können (siehe Kapitel 2.1.1). Vorreiter der Virtual Production war James Camerons *Avatar*[3] aus dem Jahr 2009, welche als erste Großproduktion virtuelle Kamera-Rigs verwendete und die aufgezeichneten Motion Capture Daten in Echtzeit mithilfe der Autodesk MotionBuilder Realtime-Engine visualisierte. Ohne den Einsatz dieser Technologien, wäre die Produktion des Films unmöglich gewesen. Vor allem wirtschaftlich gesehen sind Virtual Production-Techniken sehr vorteilhaft, da sie mögliche Probleme in der Post-Production oftmals schon während den Dreharbeiten offenlegen.

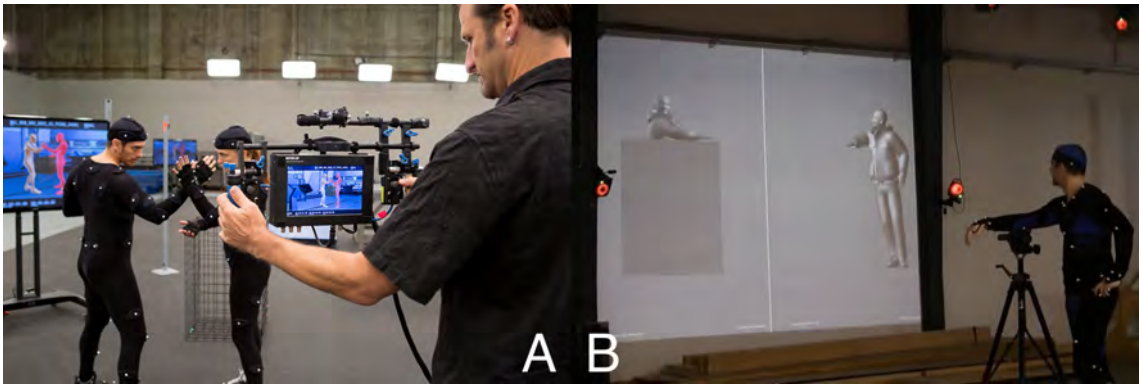


Abbildung 1.1: (A) Virtual Camera System mit Echtzeit-Darstellung im Einsatz auf der Warner Bros. Virtual Production Stage in Burbank. (B) MotionBuilder Realtime Demonstration bei metricminds in Frankfurt.

Quelle: (A) http://bit.ly/224c2Zg (WB Sound Burbank; Zuletzt abgerufen: 15.12.2015)
(B) http://bit.ly/1Yddhpr (Youtube (metricminds); Zuletzt abgerufen: 15.12.2015)

1.2 Problemstellung

Neben den vielen Vorteilen der Virtual Production Technologien, bringt ihr Einsatz jedoch auch eine Reihe von Nachteilen in der Produktion mit sich: Während traditionelle Keyframe Animation vergleichsweise geringe technische Voraussetzungen hat, sind zur Aufnahme von Motion Capture-Daten oftmals sehr teure und aufwändige Hardwarelösungen notwendig. Im Rahmen der Pre-Production muss die Virtual Production-Pipeline zusätzlich den speziellen Anforderungen der aktuellen Produktion angepasst werden. Mangels einer einheitlichen Schnittstelle, über die neue Geräte mit 3D-Softwarepaketen wie Autodesk MotionBuilder

[3] http://www.imdb.com/title/tt0499549/ (Zuletzt abgerufen: 14.02.2016)

kommunizieren können, erfordert die Anbindung und Anpassung von neuen Datenquellen wie z.B. Motion Capture-Handschuhen einen enormen Aufwand in der Softwareentwicklung.

Die nachträgliche Ergänzung von Motion Capture Daten um beispielsweise authentische Kamerabewegungen, benötigt ein Virtual Camera-System, das typischerweise ein laufendes Motion Capture-System voraussetzt. In der Post-Production kann dies zu Problemen führen, da die Arbeiten am Set zu diesem Zeitpunkt meistens schon abgeschlossen sind. Oftmals mangelt es zudem auch an der notwendigen Flexibilität um ein solches Problem effizient anzugehen, obwohl ein simples Kamerasystem mit Direktverbindung an die Post-Production Software, dieses Problem schnell lösen könnte. Ähnliches gilt für die Animation von Fingern, die im Regelfall einen sehr mühsamen Prozess in der Post-Production darstellt, sofern die Finger beim Dreh nicht aufgezeichnet wurden. Heutzutage gibt es zahlreiche Hardware-Lösungen, für die stationäre Aufzeichnung von Finger- und Handbewegungen. Diese könnten den beschriebenen Prozess beschleunigen, aber bieten jedoch oft keine Schnittstelle für 3D-Software wie MotionBuilder an.

Einige Hersteller, darunter auch Anbieter von Virtual Camera-Systemen wie beispielsweise Optitrack[4] [Sei15, S. 30], bieten proprietäre Plugins an, die ihre Hardware mit gängigen 3D Werkzeugen wie Maya oder MotionBuilder kompatibel macht. Problematisch ist jedoch, dass diese Plugins lediglich für die Nutzung von spezifischen Geräten konzipiert sind und dem Anwender keine Möglichkeit bieten, Eigenentwicklungen über die gebotene Schnittstelle anzuschließen. Autodesk liefert zusammen mit MotionBuilder mehrere Realtime-Schnittstellen aus, doch beschränken sich diese auf spezifische Eingabegeräte wie z.B. Videospiel-Controller oder Tastatur und Maus. Komplett außer Acht gelassen werden bisher Gerätekategorien wie Smartphones und Tablets, obwohl diese aufgrund ihrer vielseitig einsetzbaren Hardware, enormes Potenzial als Datenquelle bieten. Komponenten wie Touchscreen, Beschleunigungs- und Lagesensoren, können dazu genutzt werden um die Mensch-Maschine-Schnittstelle des Systems sinnvoll zu erweitern, insbesondere da die meisten Anwender heutzutage bereits mit den genannten Geräten vertraut sind und eine Steigerung der Produktionseffektivität somit sehr wahrscheinlich ist.

Zur Erweiterung der Mensch-Maschine-Schnittstelle bietet das auf C++ basierende Open Reality SDK[5] von MotionBuilder mehrere Programmierschnittstellen an, doch ist auch damit noch ein erheblicher Programmieraufwand für jedes Gerät notwenig, bis in Echtzeit Daten an MotionBuilder übertragen werden können. Kritisch ist vor allem die Kommunikation zwischen Datenquelle und der Realtime-Engine von MotionBuilder. Gerade im wahrscheinlichen Fall einer Übertragung der Daten über das lokale Netzwerk, setzt die Programmierung des Plugins fundiertes Wissen im dem Bereich der Netzwerktechnik und Socketprogrammierung voraus. Jede neue Schnittstelle muss weiterhin intensiv auf ihre Zuverlässigkeit getestet werden, was den Einsatz in der Produktion weiter hinauszögert.

[4] http://www.optitrack.com/products/insight-vcs/indepth.html (Zuletzt abgerufen: 14.02.2016)
[5] http://autode.sk/1STPMhQ (Zuletzt abgerufen: 24.02.2016)

1.3 Zielsetzung

Diese Arbeit beschäftigt sich mit der Suche nach einem möglichen Lösungsansatz für die in Kapitel 1.2 genannten Problematiken. Im Rahmen der Ausarbeitung wird ein Plugin Prototyp mit einem modularen Ansatz entwickelt, welcher es ermöglichen soll, aufgenommene Datensätze direkt am Set oder später in der Post-Production schnell und komfortabel zu erweitern. Anstatt für jede Datenquelle eine eigene Schnittstelle zu programmieren, bietet der Prototyp eine standardisierte Netzwerkschnittstelle. Diese kann eine variable Menge an Nutzdaten in Form von numerischen Werten (Gleitkommazahlen, Boolsche Variablen) empfangen und verarbeiten.

Als ein Anwendungsbeispiel für den Prototypen wird eine begleitende App zur Steuerung einer virtuellen Kamera über die Lagesensoren eines Apple iPads entwickelt. Sie soll verdeutlichen, welche neuen Möglichkeiten das Plugin auf dem Gebiet der Virtual Production eröffnet. So könnte die App beispielsweise genutzt werden um schnell authentische Kamerabewegungen in der Post-Production zu erzeugen, ohne das zusätzlich ein Motion Capture-System betrieben werden muss. Das zweite Anwendungsbeispiel, das im Rahmen dieser Arbeit entwickelt wird, verbindet den Leap Motion-Controller zur Aufzeichnung von Hand- und Fingerbewegungen über eine Mac App mit MotionBuilder. Die Anwendung könnte beispielsweise dazu genutzt werden, um simple Fingeranimationen mit authentischen Bewegungsdaten zu verbessern und die Post-Production so zu beschleunigen.

Der Erfolg des Prototypen wird abschließend anhand eines Evaluationsbogens, sowie einer Reihe Benchmark-Tests zur Überprüfung der Zuverlässigkeit der Software, bewertet. Die Testreihen überprüfen dabei kritische Punkte, wie etwa auftretende Latenzen und Fehler bei der Datenübertragung zwischen dem MotionBuilder Plugin und den Anwendungsbeispielen. Dabei werden unterschiedliche Testumgebungen einbezogen, sodass die Funktionalität des Prototypen möglichst breitflächig überprüft werden kann. Zusätzlich werden die notwendigen Rahmenbedingungen, zum zuverlässigen Betrieb des Plugins und der Anwendungsbeispiele, ermittelt.

1.4 Aufbau der Arbeit

Diese Arbeit ist in sieben Kapitel aufgegliedert: Kapitel 1, die Einleitung, beschäftigt sich mit den Hintergründen für die Wahl der Thematik, erklärt die Problemstellung und geht auf das gesteckte Ziel dieser Arbeit ein. Kapitel 2 „Stand der Technik“ gibt einen Einblick in aktuelle Entwicklungen auf dem Bereich der Virtual Production und ordnet die Arbeit thematisch ein. Kapitel 3 „Grundlagen“ bietet dem Leser das notwendige Basiswissen zum Verständnis, der in den Methodikkapiteln besprochenen Inhalte.

Die Methodik setzt sich aus den Kapiteln 4 „Konzeption“ und 5 „Implementierung“ zusammen. Sie beinhalten die Konzeption des Prototypen und die anschließende Umsetzung des beschriebenen Konzepts. In Kapitel 6 „Evaluation und Ergebnisse“, wird der Erfolg

des Prototypen anhand von Testreihen und einem Evaluationsbogen überprüft. Im letzten Kapitel wird die Arbeit noch einmal zusammengefasst und ein Ausblick auf mögliche Weiterentwicklungen des Prototypen gegeben.

1.5 Zusammenfassung

Diese Arbeit befasst sich mit der Entwicklung einer standardisierten Netzwerkschnittstelle in Form eines Plugins für die Realtime-Engine der Autodesk MotionBuilder Software. Ziel des Prototypen ist es, beliebige Datenquellen über ein vereinheitlichtes Netzwerkprotokoll in die Produktionspipeline einzubinden und die empfangenen Daten in Echtzeit zu verarbeiten. Die Logik zur Interpretation soll extern im grafischen Relation Constraint-Editor von MotionBuilder auch ohne Programmierkenntnisse aufgesetzt werden können. Damit soll der technische Aufwand, der bei der Einbindung neuer Eingabegeräte in die Pipeline entsteht, vermindert und die Flexibilität der Produktion erhöht werden. Das Plugin bewegt sich im wachsenden Bereich der Virtual Production, welche die Produktion von visuellen Effekten durch den Einsatz von Echtzeit- und Motion Capture-Technologien effizienter gestaltet.

Die Arbeit beinhaltet eine Einführung in aktuelle Entwicklungen in dem Bereich der Virtual Production und der untergeordneten Virtual Cinematography. Zu letzterer zählen Technologien wie z.B. die virtuelle Kamera, die in vereinfachter Form als Anwendungsbeispiel für die Netzwerkschnittstelle entwickelt wird. Aus dem Bereich des Motion Capture wird zudem ein intuitives Hilfsmittel zur Erstellung von Fingeranimationen, mithilfe der entwickelten Schnittstelle und dem stationären Leap Motion Finger-Tracking-Zubehör, umgesetzt. Die Arbeit bespricht alle Schritte der Entwicklung im Detail, angefangen bei der Konzeption der Software. Diese umfasst die grundlegenden Ideen zur Umsetzung der einzelnen Komponenten und beinhaltet das entwickelte Protokoll zur Kommunikation zwischen dem Plugin und den Datenquellen. Ein Großteil der Arbeit befasst sich weiterhin mit der Implementierung des entwickelten Konzepts und geht dabei im Detail auf die verwendeten Frameworks und Entwicklungsumgebungen ein. Anhand ausgiebig kommentierter Quellcode-Ausschnitte werden dem Leser die elementaren Abschnitte der Umsetzung näher gebracht.

Abschließend wird der Erfolg des umgesetzten Konzepts anhand von Benchmark-Tests und einem Evaluationsbogen überprüft. Die Ergebnisse der empirischen Messungen beweisen die technische Funktionalität des Plugins und der Anwendungsbeispiele. Zudem bestätigte die Auswertung des Evaluationsbogens, die Produktionstauglichkeit der entwickelten Softwarelösungen und damit den Erfolg des Prototypen.

Kapitel 2

Stand der Technik

In diesem Kapitel erhält der Leser einen Einblick in aktuelle Entwicklungen aus dem Bereich der Virtual Production, Echtzeit-Visualisierung und Motion Capture. Nach Abschluss dieses Kapitels, hat der Leser ein Verständnis dafür, was der aktuelle Stand der Technik auf diesen Gebieten ist und wie die Thematik der Arbeit sich dort einordnen lässt.

2.1 Virtual Production

Die ursprüngliche Definition des Begriffs *Virtual Production* wurde geprägt von dem Fortschritt der 3D-Pre-Visualisierung und neuen *Virtual Cinematography*-Technologien (siehe Kapitel 2.2), die es Regisseuren ermöglichten mit digitalen Inhalten (z.B. virtuelle Charaktere, Umgebungen) umzugehen, als seien sie ein Teil der realen Welt und des Live Action-Drehs. Techniken wie die Pre-Visualisierung von visuellen Effekten in Echtzeit, werden zur Eliminierung der Grenzen zwischen der analogen und der digitalen Welt genutzt. Diese Entwicklungen ermöglichen Regisseuren bereits am Set einen guten Gesamteindruck von komplexen Szenen mit visuellen Effekten zu erhalten. Mit intuitiven Werkzeugen wie der virtuellen Kamera, ist es dem Regisseur möglich, mit gewohnten Arbeitsschritten aus dem Live Action-Dreh, seine kreative Vision des Films auf die digitale Welt zu übertragen. Erstmalig breitflächigen Einsatz fanden diese revolutionären Technologien, in der Produktion von James Camerons *Avatar*[1] aus dem Jahr 2009. Die ersten Versuche mit Realtime-Technologie in der Spielfilmproduktion gab es jedoch bereits zehn Jahre zuvor in Stephen Sommers *The Mummy*[2].

In den letzten Jahren wurde die Definition der Virtual Production stetig erweitert. Die heutige Auffassung von Virtual Production schließt eine geteilte Asset-Pipeline über die gesamte Produktion ein. Dies bedeutet, dass Assets (digitale Produktionsinhalte), wie z.B. virtuelle Umgebungen und Charaktere, von Beginn an so entwickelt werden, dass sie idealerweise bis zum Ende der Produktion, von Nutzen sind. Dies spart nicht nur unnötige Iterationen, sondern hilft weiterhin die kreativen Gedanken und Visionen der Filmemacher, über den Verlauf

[1] http://www.imdb.com/title/tt0499549/ (Zuletzt abgerufen: 13.02.2016)
[2] http://www.ilm.com/vfx/the-mummy/ (Zuletzt abgerufen: 03.03.2016)

der Produktion zu erhalten. Die aktuelle Definition des *Virtual Production Committee* aus dem Jahre 2012 lautet daher wie folgt:

> „Virtual Production is a collaborative and interactive digital filmmaking process which begins with Virtual Design and digital asset development and continues in an interactive, nonlinear process throughout the production." [OZ15, S. 444]

Diese umfassende Definition der Virtual Production, wird durch die Weiterentwicklung der Technologien, in den nächsten Jahren vermutlich noch etwas konkretisiert. Im Rahmen dieser Arbeit werden zwei solcher Virtual Production-Werkzeuge, als Anwendungsbeispiele des Plugins umgesetzt. Um den Begriff der Virtual Production dennoch etwas mehr zu verdeutlichen, wird in den nachfolgenden Kapiteln, der Ablauf einer Virtual Production, mit dem traditionellen Produktionsablauf von visuellen Effekten verglichen. Weiterhin wird auf einzelne Technologien und deren Bedeutung für die Virtual Production eingegangen.

Abbildung 2.1: Zunehmend mehr Filmproduktionen setzen auf Virtual Production Sets mit viel Greenscreen, sodass die Pre-Visualisierung in Echtzeit eine zentrale Rolle spielt. Bild: Szene im Film *The Walk* mit dem finalen Effekt links und dem reinen Realfilm rechts.

Quelle: http://bit.ly/1QFf4KF (Atomic Fiction; Zuletzt abgerufen: 16.02.2016)

2.1.1 Vergleich: Traditionelle Produktion und Virtual Production

Mit der Einführung von Virtual Production-Techniken, änderte sich auch der Ablauf der traditionellen Filmproduktion. Dieses Kapitel bespricht die wichtigsten Kernpunkte der Virtual Production-Pipeline, sowie die Unterschiede in der Entwicklung von visuellen Effekten.

Sowohl die Virtual Production, als auch der traditionelle Produktionsansatz, beginnen mit dem Schreiben des *Drehbuch*. In der darauf folgenden Pre-Visualisierungsphase, kurz *Previs*, wird aus dem geschriebenen Wort eine visuelle Repräsentation der Szene erzeugt. Der

klassische Ansatz hierfür ist die Erstellung eines Storyboards oder simplen 2D-Animationen (*2D-Animatics*), welche einen ersten Einblick in den Aufbau der Szene ermöglichen. Hier beginnen auch schon die ersten Unterschiede zur Virtual Production: Anstelle eines Storyboards, setzt die moderne Pre-Visualisierung auf 3D-Animation. Dabei kommen schon erste Versionen der geplanten CG-Elemente zum Einsatz, die als 3D-Assets später in der Virtual Production Pipeline immer wieder aufgegriffen werden können. Ein weiterer Vorteil der 3D-Pre-Visualisierung ist die Möglichkeit, ohne großen Aufwand, komplexe Kamerafahrten und detailreiche Beleuchtungen zu visualisieren. 3D-Pre-Visualisierungen werden auch zunehmend dafür genutzt, komplexe Sequenzen oder gar ganze Filme, schon vor den Dreharbeiten, auf ihre Zuschauerwirkung zu testen und potenzielle Investoren von einem Projekt zu überzeugen (*Pitchvis*) [OZ15, S.46f].

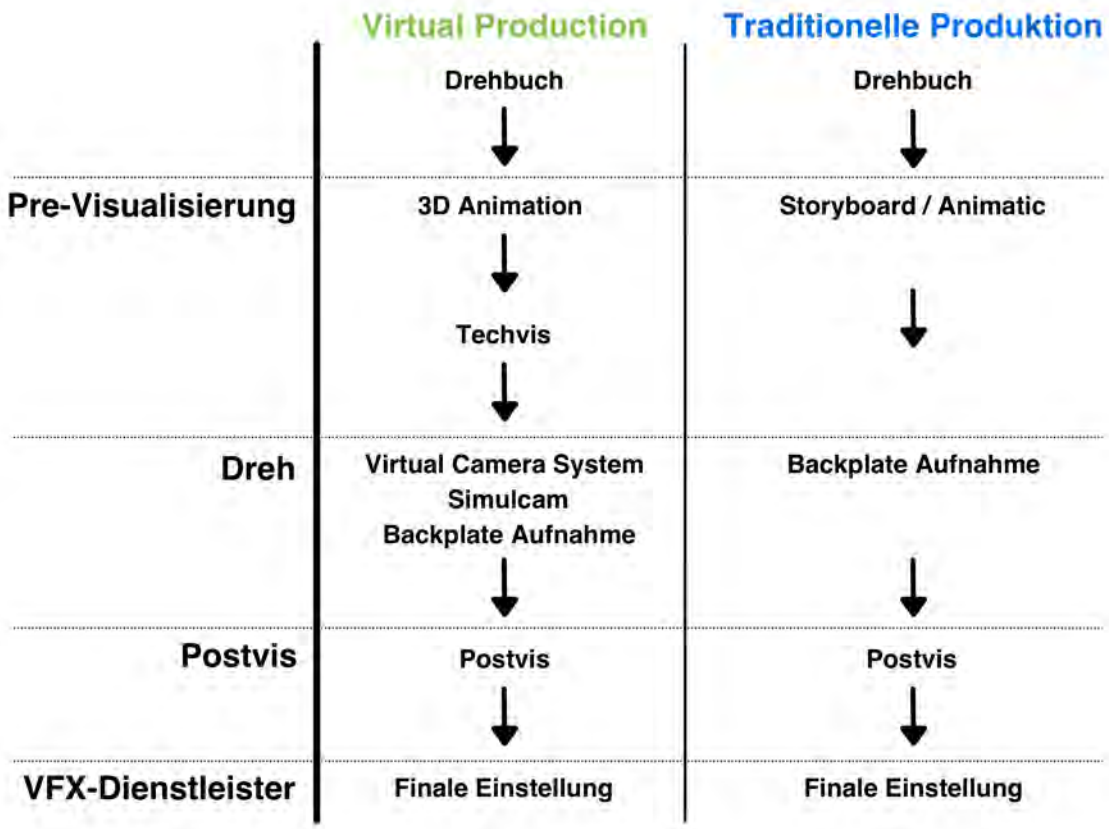


Abbildung 2.2: Vergleich des Produktionsablaufs von visuellen Effekten, in der traditionellen Produktion und der Virtual Production.

Ist die Pre-Visualisierung vom Regisseur abgesegnet, ist der nächste große Schritt in der traditionellen Produktion bereits der Dreh. Zwischen Pre-Visualisierung und Dreh findet allerdings noch einiges an Vorarbeit statt, in der unter anderem auch die technische Umsetzung des Drehs geplant wird. In der Virtual Production ist dieser Produktionsschritt die *Technical*

Previs, auch *Techvis* genannt. Sie beschäftigt sich mit der technisch akkuraten Planung des Drehs. Die Techvis betrachtet dabei insbesondere das Setdesign, die Beleuchtung, die Kameraführung und den Aufbau der Szene. In der Techvis wird nicht nur auf 3D Visualisierung zurückgegriffen, sondern auch auf traditionelle Planungsmittel wie Diagramme, um die technische Umsetzung in der realen Welt möglichst genau zu planen. Sie ist weniger ein Gedankenkonstrukt, als die Planung in eines traditionellen Drehs, und greift auf die detailreichen Informationen der Pre-Visualisierung zurück. Im Idealfall ist die Pre-Visualisierung bereits so realistisch, dass die Techvis schnell aus ihr entwickelt werden kann [OZ15, S. 47].

Während des Drehs bietet der traditionelle Produktionsansatz keinerlei Visualisierung der CG-Elemente, da diese erst in der Post-Production mit dem Live Action-Material kombiniert werden. Mit der Entwicklung von Techniken wie der Simulcam (siehe Kapitel 2.2.3), ist es in der Virtual Production hingegen möglich, mit CG-Umgebungen und Elementen zu arbeiten, als wären diese ein Teil des realen Sets. Die Simulcam übernimmt dabei bereits einen Teil der *Postvis*, dem nächsten Schritt in der Produktion, und kombiniert das Filmmaterial in Echtzeit mit den visuellen Effekten der Pre-Visualisierung.

Die Postvis existiert auch im traditionellen Produktionsablauf und hat die Aufgabe dem Regisseur ein Bild vom Zusammenspiel der verschiedenen Bildelemente und Effekte zu verschaffen. Aufgrund dieser Basis wird entschieden, ob die Szene in der bestehenden Form funktioniert und gegebenenfalls angepasst. Die Postvis ist dank detaillierter 3D-Pre-Visualisierungen, in der Virtual Production, schneller erstellt. Ein weiterer Vorteil: In der Postvis kann der Regisseur auf die bereits vorhandenen 3D-Assets der Pre-Visualisierung zurückgreifen, um die Postvis schneller zu verbessern und seine Vision besser zu kommunizieren. Ist die Postvis abgenommen, geht das Material in beiden Abläufen weiter an den VFX-Dienstleister, der die Post-Production übernimmt und auf Basis der Postvis, die finale Einstellung (*Shot*) erstellt.

Es lässt sich also festhalten, dass die modernen Methoden der Virtual Production, vor allem effizientere Workflows, für Filmproduktionen mit vielen visuellen Effekten bieten. Budgetplanungen sind dank aufschlussreicher Pre-Visualisierungen genauer und visuelle Effekte können mit weniger Iterationen realisiert werden. Weiterhin kann der Regisseur seine Vision akkurater umsetzen und hat mehr Flexibilität in seinen Entscheidungen.

2.1.2 Virtual Cinematography

Der Begriff *Virtual Cinematography* taucht im Zusammenhang mit Virtual Production, immer wieder auf. Das *VES Handbook* beschreibt die Anwendung kinematographischer Prinzipien (z.B. Kontinuität zwischen Szenen), auf eine computergenerierte Szene, als Virtual Cinematography [OZ15, S.445].

Virtuelle Sets bieten dem Regisseur viele Freiheiten bei der Umsetzung seiner Vision, die bei einem traditionellen Realfilm-Dreh nicht immer gegeben sind. Virtuelle Kameras sind nicht durch physikalische Gesetze in ihrer Positionierung- und Bewegung eingeschränkt und die Beleuchtung der computergenerierten Bildinhalte kann beliebig angepasst werden [OZ15,

S. 439ff]. Kameras in einer virtuellen 3D-Szene können ihre Position mit sofortiger Wirkung verändern, sodass sie in der Theorie sogar als Echtzeit-Schnittwerkzeug eingesetzt werden können [ER07, S.9].

Die Freiheiten der virtuellen Welt wirken sich jedoch oft zulasten der Ästhetik des Films aus, sofern sie nicht mit Vorsicht eingesetzt werden. Virtual Cinematography Technologien, wie z.B. Virtuelle Kamera-Systeme (siehe 2.2.3), helfen dem Regisseur im Umgang mit virtuellen Umgebungen und visuellen Effekten. Sie verwischen die Grenzen zwischen digitalen- und analogen Bildinhalten und ermöglichen die Arbeit mit visuellen Effekten, als wären diese ein Teil des realen Live-Action Sets. Als Anwendungsbeispiel des Plugin-Prototypen auf dem Gebiet der Virtual Cinematography, wird im Rahmen dieser Arbeit eine vereinfache Variante eines virtuellen Kamera-Systems, auf Basis einer iPad App entwickelt (siehe Kapitel 4.2). Die Anwendung ermöglicht die Animation einer virtuellen Kamera mit authentischen Rotationsdaten. Ohne Positionstracking ist die App jedoch weniger eine On-Set Lösung, als ein nützliches Werkzeug für authentische Arbeit in der Pre- und Post-Production.

2.1.3 Echtzeit 3D-Pre-Visualisierung

Einer der Hauptpfeiler der Virtual Production, ist die Echtzeit-Darstellung von visuellen Effekten, unabhängig von der Produktionsphase. Zu Beginn des Jahrtausends war dies aufgrund technischer Einschränkungen der Computertechnologie nur in einem stark eingeschränkten Rahmen möglich. Mit den technologischen Fortschritten des letzten Jahrzehnts, können jedoch sogar komplexe Effekte, in Echtzeit dargestellt werden.

Die Technologien, die dies ermöglichen, haben ihren Ursprung in der Videospielindustrie und der Entwicklung von sehr leistungsstarken Grafikprozessoren. Heutige Computersysteme, sowohl solche für Konsumenten, als auch für professionelle Anwender, sind in der Lage hochkomplexe Bildinhalte für Simulationen und Videospiele, in Echtzeit zu rendern (siehe Abbildung 2.3). Zeitgleich steigen die Bildraten, die 3D-Engines (Echtzeit-Computergrafik) produzieren können, stetig. Auf aktueller Hardware stellen 30 Bilder pro Sekunde, selbst mit aufwändigem Lighting und Shading, kein Problem mehr da. Damit bewegt sich die Technik bereits in einem Bereich, der über dem Kinostandard von 24 Bildern pro Sekunde liegt. Selbst hohe Bildraten von 48 Bilder pro Sekunde (*High Frame Rate, HFR*), die in der Produktion der *Hobbit*-Trilogie[3] von Peter Jackson genutzt wurden, stemmen aktuelle High-End Computersysteme mühelos.

Zwar ist es noch nicht möglich, komplett photorealistische Echtzeit-Effekte zu generieren, doch bietet der aktuelle Stand der Technik ausreichend Qualität, um der Filmproduktion von großem Nutzen zu sein. Viele Unternehmen haben dies bereits erkannt und bieten passende Softwarelösungen zur Echtzeit-Visualisierung von digitalen Inhalten an. Einige der aktuellen Entwicklungen und kommerziell erhältlichen Lösungen zur Pre-Visualisierung von visuellen Effekten in Echtzeit, werden in Kapitel 2.2.2 vorgestellt.

[3] http://www.imdb.com/title/tt1170358/ (Zuletzt abgerufen: 13.02.2016)

Abbildung 2.3: Nahezu fotorealistische Echtzeit-Grafik in modernen Videospielen, wie *Star Wars: Battlefront* (A/B) und *Rise of the Tomb Raider* (C), dank fortgeschrittener Hardware und Game-Engines.

Quelle: (A) http://bit.ly/1ok6I4C (Petri Levälahti; Zuletzt abgerufen: 15.02.2016)
(B) http://bit.ly/1KS78ZH (Petri Levälahti; Zuletzt abgerufen: 15.02.2016)
(C) http://bit.ly/1QhKKFj (Petri Levälahti; Zuletzt abgerufen: 15.02.2016)

2.2 Virtual Production Technologien

Dieses Kapitel geht näher auf einige der wichtigsten technischen Entwicklungen ein, die den Begriff der Virtual Production geprägt haben und auch noch in ihrer heutigen Form auszeichnen. Der Fokus des Kapitels liegt auf Technologien aus dem Bereich der Virtual Cinematography und Pre-Visualisierung, da die Thematik dieser Arbeit ebenfalls dort einzuordnen ist.

2.2.1 Realtime Performance Capture

Ein wichtiger Bestandteil aktueller Virtual Production-Entwicklungen, ist der enorme Fortschritt im Bereich des Motion Capture seit der Jahrtausendwende. Insbesondere *Performance Capture* (siehe Kapitel 3.1.3) hat sich in der Virtual Production etabliert, da sämtliche Aspekte des Schauspiels eines Darstellers eingefangen werden. Aktuelle Technologien ermöglichen die zuverlässige Aufzeichnung der Mimik mittels Videoanalyse. Zudem entwickeln sich immer mehr Echtzeit-Technologien zur Aufzeichnung detaillierter Fingeranimationen mittels kostengünstiger Hardware. Ein gutes Beispiel dafür ist der *Leap Motion-Controller*, des gleichnamigen Unternehmens (siehe Kapitel 3.4).

Stefan Seibert verwendete in seiner Bachelorthesis „Real-Time Set Editing in a Virtual Production Environment with an Innovative Interface“ [Sei15], die Finger-Tracking Fähigkeiten des Leap Motion-Controllers zur Umsetzung eines intuitiven Virtual Production-Werkzeugs. Die auf der *Unity* Game-Engine basierende Anwendung ermöglicht Anpassungen an einem virtuellen Set in Echtzeit. Dazu registriert der Leap Motion-Controller die Gestik des An-

wenders und gibt die gesammelten Tracking-Daten an Unity weiter, wo sie Werkzeuge zur Interaktion mit der virtuellen Umgebung steuern. Zur Darstellung der *Augmented Reality*-Werkzeuge und virtuellen Setelementen, verwendete Seibert ein *Oculus Rift*[4] HMD („*Head-mounted display*“, *Virtual Reality*-Brille). Der Unterschied zwischen *Augmented Reality* und *Virtual Reality* liegt darin, dass Ersteres die reale Welt interpretiert und diese um digitale Elemente erweitert. Virtual Reality-Anwendungen bestehen hingegen komplett aus virtuellen Inhalten.

Im Rahmen des *Leap Motion 3D Jam 2.0*[5] entwickelte Navid Nikbin das *Hand Capture*-Plugin[6] für MotionBuilder 2016 (siehe Abbildung 2.4). Die Erweiterung ermöglicht die Echtzeit-Übertragung von Hand- und Fingerbewegungen, mittels Leap Motion-Controller, an Autodesk MotionBuilder (siehe Kapitel 3.2). Das Plugin erkennt weiterhin mehrere Handgesten und kann die Tracking-Daten direkt auf ein standardisiertes Charakter-Rig (Steuerelemente zur Animation von Charakteren) übertragen. Ein offizielles Leap Motion-Plugin für MotionBuilder[7], wurde 2013 von Autodesk veröffentlicht. Schlechte Tracking-Ergebnisse führten zu geringem Interesse und dem Entwicklungsstopp des Plugins im Januar 2014. Das im Rahmen dieser Arbeit entwickelte Tracking-Beispiel verwendet eine aktuellere Version der Leap Motion-Software, die bessere Ergebnisse liefert.

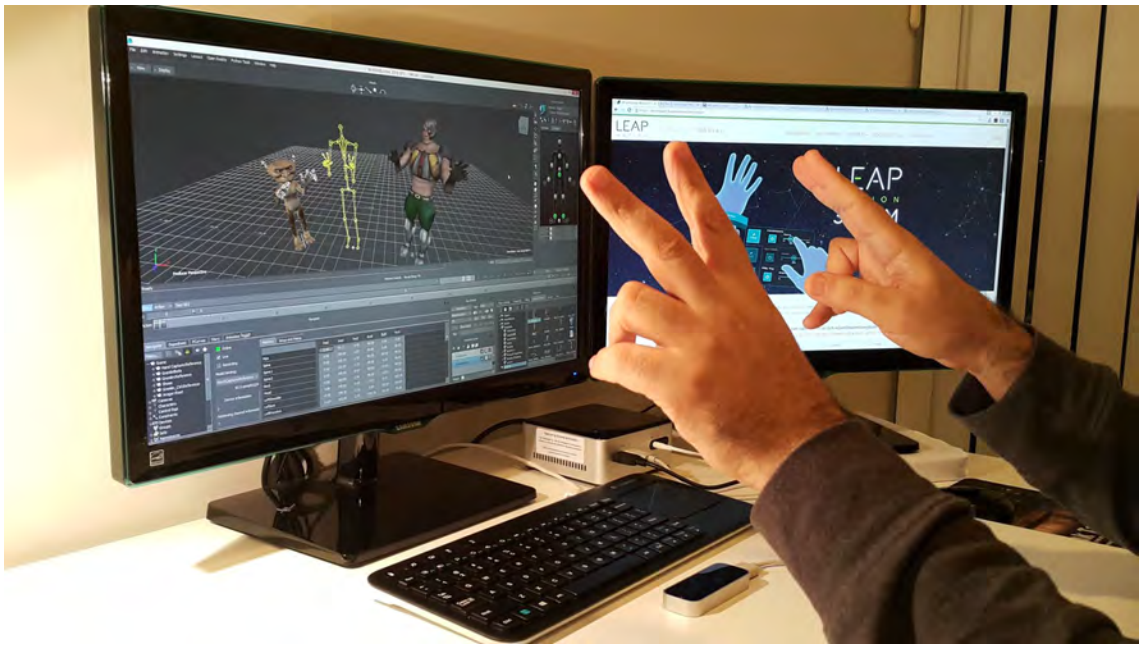


Abbildung 2.4: Leap Motion *Hand Capture*-Plugin für MotionBuilder im Einsatz.

Quelle: http://bit.ly/1QfrPLw (Navid Nikbin; Zuletzt abgerufen: 14.02.2016)

[4]https://www.oculus.com/en-us/dk2/ (Zuletzt abgerufen: 14.02.2016)
[5]http://bit.ly/1RCIwFQ (Zuletzt abgerufen: 14.02.2016)
[6]http://bit.ly/1QfrPLw (Zuletzt abgerufen: 14.02.2016)
[7]http://bit.ly/1R2Pq56 (Zuletzt abgerufen: 14.02.2016)

Für präzisere Echtzeit-Bewegungsaufzeichnung von Finger- und Handbewegungen mit kostengünstiger Hardware, kombinierten Benoît Penelle und Olivier Debeir in ihrem Paper „Multi-Sensor Data Fusion for Hand Tracking using Kinect and Leap Motion“, die Tracking-Daten mehrerer Sensorsysteme [PD14]. Zur Aufzeichnung feiner Hand- und Fingerbewegungen nutzte das Team die vergleichsweise präzisen Tracking-Daten des Leap Motion-Controllers, während die *Microsoft Kinect*[8] Kamera, stereoskopische Bilder des Anwenders liefert.

Andere Entwicklungen setzen auf optisches Tracking mittels Farbkodierung. Im Paper „Real-Time Hand-Tracking with a Color Glove“ [WP09], beschreiben Robert Y. Wang und Jovan Popovic die Aufzeichnung von Hand- und Fingerbewegungen mithilfe einer einzigen, stationären Kamera und einem mehrfarbigen Handschuh. Das entwickelte System ist sehr kostengünstig, leidet jedoch unter starken Ungenauigkeiten, besonders bei der Errechnung der Hand-Translation.

Stationäre Sensoren mit optischem Tracking, besitzen demnach auch Nachteile: Der begrenzte Sichtbereich der Sensoren, schränkt ihren Nutzen ein und setzt voraus, dass der Nutzer seine Hände immer in einer bestimmten Distanz zum Sensor bewegt. Eine Alternative dazu bieten Datenhandschuhe, welche die Rotationswinkel der einzelnen Fingergelenke, meistens mit Widerstand-basierten Sensoren messen. Erschwingliche Datenhandschuhe, wie im Paper „Wired gloves for every one“ [OCMM08] beschrieben, lassen sich leicht aus Drähten zusammenbauen und nutzen eine indirekte, optische Analyse. Sie bieten jedoch wenig Flexibilität am Set und schränken den Anwender aufgrund der Kabelgebundenheit in seiner Bewegungsfreiheit ein.

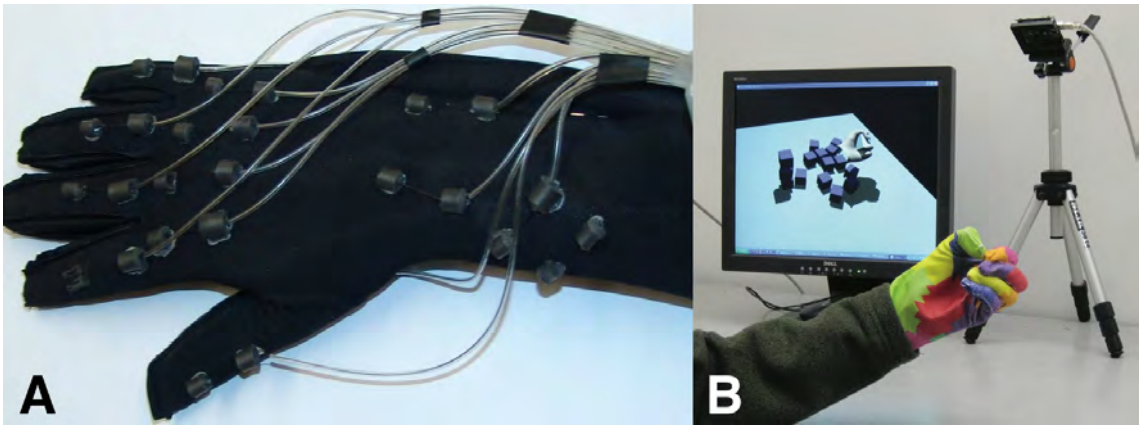


Abbildung 2.5: Kostengünstige Finger-Tracking Ansätze: (A) Mechanischer Handschuh mit indirekter, optischer Messung der Drahtlängen. [OCMM08] (B) Farb-Handschuh mit optischem Tracking. [WP09]

Quelle: (A) H. Ortega-Carrillo, E. Martínez-Mirón [OCMM08] (B) R. Y. Wang, J. Popovic [WP09]

[8]https://dev.windows.com/en-us/kinect (Zuletzt abgerufen: 14.02.2016)

Einen alternativen Ansatz bietet das Forschungsprojekt *Ubihand* [AM06] von Farooq Ahmad und Petr Musilek. Dabei handelt es sich um eine Kombination aus einer kabellosen Kamera, die der Anwender am Handgelenk trägt und einer Videoanalyse-Software, die Finger-Bewegungen aufzeichnet. Zur Bestimmung der Fingerhaltung analysiert die Software, die Kontouren der Fingerglieder im Videomaterial und errechnet daraus ein Tracking-Modell der Hand. Ubihand ist dabei in der Lage die Beugungswinkel der einzelnen Fingergelenke, über den relativen Abstand der Fingerglieder von der Kamera, zu ermitteln. Die Software erkennt weiterhin, ob die Finger gespreizt sind.

2.2.2 Echtzeit 3D-Visualisierungssoftware

Das derzeit meistbenutzte Werkzeug für die Echtzeit-Pre-Visualisierung von Produktionsinhalten ist *Autodesk MotionBuilder* (siehe Kapitel 3.2). Die Animationssoftware bietet eine sehr performante Engine, die hohe Polygonzahlen, diverse Physik-Simulationen und komplexe digitale Charakter in Echtzeit mit hohen Bildraten darstellen kann. MotionBuilder wird weiterhin von vielen Virtual Production Werkzeugen unterstützt und kann Motion Capture Daten von vielen Systemen, mittels Software-Plugin, in Echtzeit in die Szene integrieren.

MotionBuilder ist die Basis vieler Virtual Production Technologien, die auf die starken Echtzeit-Fähigkeiten der Software aufbauen. Eines der bekanntesten Beispiele der letzten Jahre ist die *Simulcam* (siehe 2.2.3), welche ursprünglich für die Produktion von *Avatar* entwickelt wurde und die Echtzeit-Komposition von Realfilm und visuellen Effekten ermöglicht. Ein anderes Anwendungsbeispiel für die Realtime-Funktionalität von MotionBuilder wird in der Veröffentlichung „Real - Time Motion Capture Technology on a Live Theatrical Performance with Computer Generated Scenery“ [AHA+10] aufgezeigt. Anthousis Andreadis und sein Team verwendeten die Autodesk Software, um ein digitales Theaterstück in Echtzeit mithilfe von Motion Capture zu realisieren.

Unter dem Namen *Shark 3D*[9] entwickelt die Spinor GmbH aus Deutschland, ein speziell auf Echtzeit-Visualisierung ausgelegtes Produkt. Die Software bietet eine leistungsstarke Realtime-Engine, die selbst komplexes Lighting in Echzeit rendern kann. Im Gegensatz zu MotionBuilder ist die Software jedoch nicht für den Einsatz mit Motion Capture Technologie ausgelegt. Stefan Seibert beschreibt die Shark 3D Engine in seinem Paper zur Entwicklung einer Echtzeit-Anwendung als mächtiges Werkzeug für Echtzeit-Renderings, bemängelt jedoch die kleine Nutzergemeinde und die eingeschränkte Unterstützung von Code-Erweiterungen [Sei15, S. 38-39]. Die Software wird derzeit eher in der Videospielindustrie und zur Erzeugung von Echtzeit-Effekten im Fernsehen, als im Bereich der Virtual Production verwendet.

Eine passendere Lösung für Virtual Production-Pipelines, ist die *Cinebox*[10] von Crytek. Die Software basiert auf der *CryEngine* des gleichen Unternehmens, welche als hochentwickelte Game-Engine in vielen AAA-Videospielen zum Einsatz kommt. Bei Cinebox handelt es sich um eine spezialisierte Version der CryEngine, mit Fokus auf den Einsatz im Bereich der

[9] http://www.spinor.com (Zuletzt abgerufen: 13.02.2016)
[10] http://www.cryengine.com (Zuletzt abgerufen: 13.02.2016)

Virtual Production und Pre-Visualisierung. Cinebox befindet sich allerdings noch mitten in der Entwicklung und ist nicht offiziell erhältlich. Eine Vorabversion wurde bereits für eine Reihe von Projekten eingesetzt, darunter auch zur Pre-Visualisierung des Kinofilms *Dawn of the Planet of the Apes*[11] von Matt Reeves.

Yafes Sahin und sein Team von der Filmakademie Baden Württemberg, verwendeten ebenfalls eine Vorabversion der Software, um den Kurzfilm *Dark Matter* zu produzieren [SSB14]. Für den Kurzfilm entwickelten die Studenten eine neue Virtual Production Pipeline, basierend auf Cinebox und Motion Capture Technologie. Die Crytek Software lieferte visuelle Effekte direkt am Set, die mit den Live-Action Aufnahmen einer Arri-Kamera in Echtzeit kombiniert wurden. Durch die realitätsnahe Echtzeit-Darstellung CryEngine, war es möglich, dass Lighting des realen- und digitalen Sets schon während des Drehs, in Einklang zu bringen. Cinebox wurde weiterhin in der Post-Production verwendet, um die finalen Effekte für den gesamten Kurzfilm in kürzester Zeit zu rendern.

Weitere Game-Engines, die vermehrt im Bereich der Virtual Production zur Pre-Visualisierung eingesetzt werden, sind die *Unreal Engine*[12] von Epic Games, sowie *Unity*[13] von Unity Technologies. Wie Michael Nitsche in seiner Publikation „Experiments in the use of game technology for pre-visualization“ [Nit08] beschreibt, wurde eine frühe Version der Unreal Engine bereits 2001, für die Pre-Visualisierung von Steven Spielbergs „A.I. Artificial Intelligence“ verwendet. Mit einer aktuelleren Version der Unreal Engine, realisierte Epic Games im Jahr 2015 die Echtzeit-Demo „A boy and his kite“, welche dynamisches Lighting unterstützt und sich qualitativ dem Fotorealismus nähert. Das Marketing-Projekt zeigt die filmische Funktionalität der Engine. [Mor15].

Abbildung 2.6: Echtzeit-Compositing von Realfilm und Pre-Visualisierung (Crytek Cinebox) am Set von *Dark Matter* [SSB14].

Quelle: http://bit.ly/1mDF6Wq (Filmakademie Baden-Württemberg GmbH; Zuletzt abgerufen: 15.02.2016)

[11] http://www.imdb.com/title/tt2103281/ (Zuletzt abgerufen: 13.02.2016)
[12] https://www.unrealengine.com (Zuletzt abgerufen: 13.02.2016)
[13] https://unity3d.com/ (Zuletzt abgerufen: 13.02.2016)

2.2.3 Virtuelle Kamera-Systeme

Die meist verbreitete Technologie aus dem Bereich der Virtual Cinematography, ist das *Virtual Camera System*, kurz *VCS* (dt. *Virtuelles Kamera-System*). Eine virtuelle Kamera besteht im Regelfall aus einem modifizierten Kamera-Rig, ausgestattet mit zusätzlichen Bedienelementen wie Joysticks, sowie Motion Capture Markern und einem Display. Über die Motion Capture Marker (siehe Kapitel 3.1.1) wird die Position und Rotation des Kamera-Rigs Weltkoordinatensystem bestimmt und mit den gesammelten Daten, eine Repräsentation der Kamera im virtuellen 3D-Raum erstellt. Anstelle von Motion Capture Markern können zur Positionsbestimmung auch Informationen von tragenden Kamerakränen, Inertialsensoren oder Dollys entnommen werden. Während sich die Kamera durch den Raum bewegt, wird auch deren virtuelle Repräsentation entsprechend aktualisiert und der Kameramann sieht das virtuelle Kamerabild in Echtzeit als würde er sich durch die CG-Umgebung bewegen.

Das Rig einer virtuellen Kamera bietet dem Kameramann typischerweise Zugriff auf alle wichtigen Parameter, die er von einer realen Kamera erwarten würde (Beispiel: Brennweite). Weiterhin lassen sich die Eigenschaften der Kamera beliebig skalieren, sodass der Kameramann mit seinen Bewegungen beispielsweise einen Helikopterflug oder Kamerakran simulieren kann. Virtuelle Kamerasysteme bieten der Crew am Set ein vertrautes Werkzeug zur Erzeugung von authentischen Aufnahmen, selbst wenn die gesamten Bildinhalte computergeneriert werden. Sie erfordern dennoch eine gewisse Einarbeitung, zur erfolgreichen Umsetzung des realen Filmwissen in der virtuellen Welt.

Abbildung 2.7 zeigt das Kamera-Rig des virtuellen Kamerasystems der Firma *metricminds*. Das im Unternehmen entwickelte System setzt sich im wesentlichen aus dem Rig, einem Tablet und einem modifizierten Game-Controller zusammen. Das Tracking der Kamera erfolgt über ein optisches Motion Capture System und passive Marker. Zur Echtzeit-Darstellung wird MotionBuilder in Kombination mit der Streaming-App *Splashtop*[14] verwendet.

Abbildung 2.7: Das virtuelle Kamera-Rig der Firma *metricminds*.

Quelle: Christian Krenzer (metricminds GmbH & Co. KG)

[14]http://www.splashtop.com/personal (Zuletzt abgerufen: 07.03.2016)

In den letzten Jahren hat sich eine ganze Reihe unterschiedlicher Ansätze zur Umsetzung virtueller Kameras entwickelt. Girish Balakrishnan entwickelte im Rahmen seiner Masterthesis "Virtual Cinematography: Beyond Big Studio Production" [BD13] das *MobileVCS*-System, für kleine Studios und Indie-Filmemacher. Für das System greift er auf günstige Hardware aus dem Consumer-Bereich zurück. Balakrishnan nutzt Sonys *Move.me*[15] Software in Kombination mit einer *Playstation 3* und dem *Playstation Move*-Videospielcontroller. Der Controller ist über eine Halterung mit einem Apple iPad verbunden und liefert sowohl Positions-, als auch Rotationsdaten. Die gesammelten Daten werden dann von der Move.me Software an die Unity-Engine (siehe Kapitel 2.2.2) gesendet, welche auf dem Tablet läuft und die Visualisierung des virtuellen Kamerabilds übernimmt (siehe Abbildung 2.8).

Während Balakrishnan für sein Kamerasystem die offizielle Sony Move.me Software verwendet, gibt es mittlerweile alternative Playstation Move Analysesoftware von Thomas Perl, die auch ohne ein Playstation 3 System lauffähig ist [Per12].

Abbildung 2.8: Das virtuelle Kamera-System *SmartVCS*, nutzt Playstation Hardware zur Realisierung eines präzisen Trackings im Weltkoordinatensystem. [BD13]

Quelle: http://bit.ly/1Qi070w (Girish Balakrishnan; Zuletzt abgerufen: 15.02.2016)

Gabriel Schmitz und Sebastian Kalkhoff entwickelten 2015 an der Fachhochschule Köln, das Open Source System *Project Wivica*[16]. Das kostengünstige Kamerarig des Systems besteht lediglich aus einem Videospielcontroller, einem Apple iPad und passiven Markern, die von einem optischen Motion Capture System aufgezeichnet werden. Über die Position der Motion Capture Marker, werden Position und Rotation des Kamera-Rigs im Weltkoordinatensystem errechnet und an Autodesk MotionBuilder übertragen. Der Controller dient zur Steuerung der MotionBuilder-Aufnahme und den Eigenschaften der virtuellen Kamera. Zur Visualisierung wird ein Videostream aus MotionBuilder über ein WiFi-Netzwerk an das iPad gesendet und dort angezeigt.

[15] http://de.playstation.com/moveme/ (Zuletzt abgerufen: 14.02.2016)
[16] http://www.project-wivica.com/ (Zuletzt abgerufen: 14.02.2016)

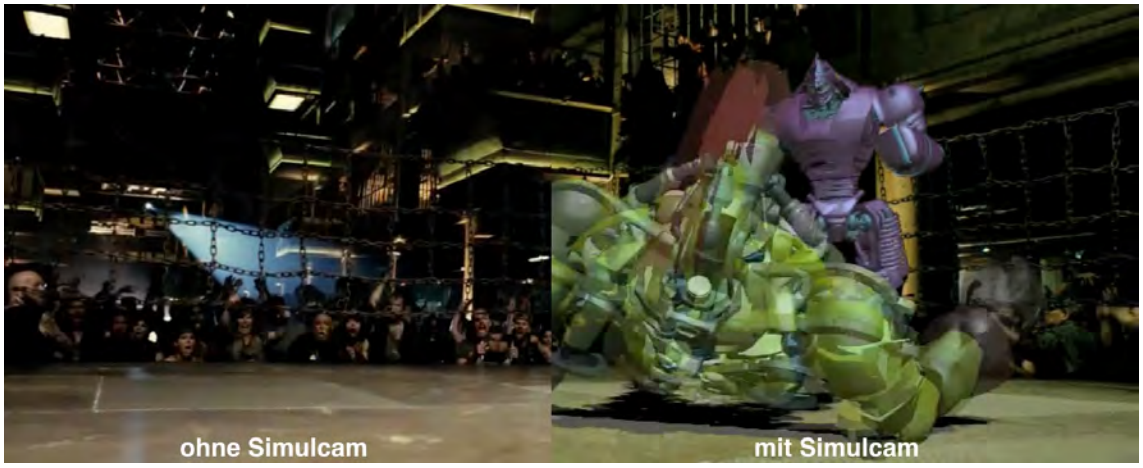


Abbildung 2.9: Einsatz der Simulcam-Technologie am Set von *Real Steel*. Die Previs-Animation wird in Echtzeit in das Bild der Live Action-Kamera integriert.

Quelle: http://bit.ly/1mXAKKY (Digital Domain; Video-Screenshot; Zuletzt abgerufen: 13.01.2016)

Eine der bekanntesten Umsetzungen der virtuellen Kamera, ist die sogenannte *Simulcam* (siehe Abbildung 2.9). Dabei handelt es sich um eine erweiterte Form der virtuellen Kamera, welche für die Produktion von James Camerons *Avatar*[17] entwickelt wurde. Die Simulcam kann nicht nur zur intuitiven Kameraführung in virtuellen Sets genutzt werden, sondern kombiniert die digitale Bildelemente außerdem mit parallel stattfindenden Live Action-Aufnahmen. Durch diese Vermischung der Welten in Echtzeit, kann der Kameramann beim Dreh die virtuellen Bildinhalte antizipieren und authentischere Aufnahmen mit besserem Timing und Framing erzeugen. In der Vergangenheit wurden Platzhalter als Hilfsmittel am Set verwendet. Diese vermitteln jedoch nie das gleiche Verständnis von der Szene, wie eine Echtzeit-Vorschau des visuellen Effekts in Aktion. In Shawn Levys *Real Steel*[18] wurde die Simulcam genutzt, um voranimierte Roboterkämpfe in einem realen Arena-Set zu filmen. Dazu wurde die Animation in Echtzeit auf der Simulcam abgespielt und die Kameramänner konnten so agieren, als würden die Roboter tatsächlich in der eigentlich leeren Arena kämpfen [OZ15, S.443].

Kommerzielle Systeme wie *Previzion* von Lightcraft[19] erweitern bestehende Kamerasysteme, um die Fähigkeiten einer virtuellen Kamera bzw. Simulcam. Mithilfe von spezieller Tracking-Hardware und einer Realtime-Rendering-Software ist das System in der Lage, eine Echtzeit-Komposition von Realfilm und visuellen Effekten durchzuführen. Für das Realtime-Tracking verwendet das Previzion System eine Mischung aus optischem Tracking und Inertialsensoren (siehe Kapitel 3.3). Ein weiteres System, dass primär auf optisches Tracking der Kamerabewegung setzt, ist *Ncam*[20] von Ncam Technologies. Ncam nutzt zwei zusätzliche

[17] http://www.imdb.com/title/tt0499549/ (Zuletzt abgerufen: 14.02.2016)
[18] http://www.imdb.com/title/tt0433035/ (Zuletzt abgerufen: 14.02.2016)
[19] http://www.lightcrafttech.com/overview/ (Zuletzt abgerufen: 14.02.2016)
[20] http://www.ncam-tech.com/ (Zuletzt abgerufen: 14.02.2016)

Kameras und markerlose Algorithmen (siehe Kapitel 3.1.3) zur Verfolgung der Kamerabewegung. Das System ermöglicht weiterhin eine Echtzeit-Komposition mit computergenerierten Bildelementen aus Autodesk MotionBuilder.

2.3 Fazit

Wie die vorherigen Kapitel aufzeigen, ist das Feld der *Virtual Production* Teil einer langjährigen Weiterentwicklung des Produktionsablaufs für visuelle Effekte (siehe Kapitel 2.2.2). Virtual Production macht sich moderne Computer-Hardware und Technologie der Videospielindustrie zu Nutze, um den Einsatz von Computergrafik, von der Post-Production auf den gesamten Produktionsablauf auszudehnen. Dies umfasst beispielsweise die Pre-Visualisierung von Szenen und Effekten mittels 3D-Animationen, sowie die Echtzeit-Darstellung von Motion Capture Aufnahmen am Set.

Autodesk MotionBuilder ist die derzeit meistbenutzte Softwarelösung, für die Darstellung von virtuellen Charakteren und Umgebungen in Echtzeit. Dank stetigem Fortschritt auf dem Gebiet des Motion- und Performance Capture, gibt es zunehmend mehr Möglichkeiten, lebensechte Bewegungen aus der analogen-, in die digitale Welt zu übertragen. Da aktuelle Virtual Production Pipelines sich in stetiger Entwicklung befinden und neue Technologien regelmäßig eingeführt werden, ist es von enormen Vorteil über eine standardisierte Netzwerkschnittstelle zur Einbindung neuer Datenquellen in die Echtzeit-Darstellung zu verfügen. Neue Virtual Production-Werkzeuge wie z.B. virtuelle Kamera-Systeme und Finger-Tracking-Lösungen, können so binnen kürzester Zeit getestet, evaluiert und in die Produktion eingebunden werden.

Als mögliches Anwendungsbeispiel des Plugins, wird in dieser Arbeit, der Leap Motion-Controller zur Steuerung eines Charakter-Rigs mit MotionBuilder verbunden. Der Prototyp unterscheidet sich hierbei von bestehenden Lösungen wie dem *Hand Capture*-Plugin (siehe Kapitel 2.2.1), da keine zusätzliche Software auf der MotionBuilder-Maschine installiert werden muss und der Leap Motion-Controller mit einem beliebigen Computer im Netzwerk verbunden sein kann. Andere Plugin-Lösungen erfordern, dass der Controller über USB mit dem Computer verbunden ist, auf dem die MotionBuilder Instanz läuft.

Das zweite Anwendungsbeispiel ist im Bereich der Virtual Cinematography angesiedelt und ermöglicht dem Anwender die Übertragung authentischer Rotationsdaten von den Inertialsensoren eines Apple iPads auf eine virtuellen Kamera. Im Vergleich zu anderen virtuellen Kamera-Systemen auf Tablet-Basis, wie etwa *Project Wivica* oder *MobileVCS* (siehe Kapitel 2.2.3), verzichtet der Prototyp auf jegliches Positions-Tracking und bietet im Gegenzug erhöhte Mobilität und und eine einfache Einrichtung. Das Positions-Tracking lässt sich über eine Erweiterung des Datensatzes hinzufügen, ist jedoch nicht Teil der umgesetzten Beispielanwendung. Diese lässt sich mit einem beliebigen iPad nutzen, dass sich im gleichen Netzwerk wie die MotionBuilder-Maschine befindet und eignet sich daher besonders als flexibles Werkzeug in der Pre- und Post-Production.

Kapitel 3

Grundlagen

Dieses Kapitel gibt einen Überblick über einige grundlegende Kenntnisse im Bereich der Computergrafik, Filmproduktion und Computertechnik, die für das Verständnis der in dieser Arbeit behandelten Thematik notwendig sind.

3.1 Motion Capture

Der Begriff des *Motion Capture* beschreibt die Aufzeichnung von Bewegungen. Dabei werden mithilfe spezieller Technologien Bewegungsabläufe von Menschen oder Objekten aufgezeichnet und von der analogen, in die digitale Welt überträgt. Heutzutage ist Motion Capture in der Film- und Spieleindustrie als Standard zur Erzeugung realitätsnaher Animation etabliert. Einige Motion Capture-Techniken werden seit den 1970er Jahren in der Medizin eingesetzt, um Bewegungsabläufe zu analysieren und bessere Diagnosen stellen zu können.

Bereits seit fast 100 Jahren wird eine vereinfachte Form des Motion Capture, dass sogenannte *Rotoscoping*, in der Animation angewandt. Dabei werden Elemente aus Live Action-Standbildern von einem Animator Bild für Bild nachgezeichnet, um die Bewegung auf einen gezeichneten Charakter zu übertragen (siehe Abbildung 3.1). Rotoscoping ist auch heute noch relevant, wird jedoch eher in der Produktion von visuellen Effekten angewandt, um Bildelemente zu entfernen oder zu ersetzen. Für die Produktion von authentischen Animationen kommen hingegen komplexe Motion Capture Systeme zum Einsatz, die optisch oder mechanisch funktionieren.

3.1.1 Optische Systeme

Heutige Motion Capture Systeme arbeiten meistens mit optischem Tracking von Markerobjekten. Dazu werden die Darsteller am ganzen Körper mit Markern bestückt um Starrkörper zu markieren, welche anschließend von mehreren Kameras erfasst werden. Ein kalibriertes Kamerasystem kennt das Verhältnis zwischen Kamerasensor, Linse und dem einfallenden Licht. Basierend darauf errechnet es den Einfallsvektor, über den das Licht der Marker auf den Sensor getroffen ist. Irgendwo entlang dieser Vektoren befinden sich die Marker im

Abbildung 3.1: Ein Beispiel für Rotoscoping als Animationsreferenz in Disneys *Peter Pan* (1953).

Quelle: http://bit.ly/1kF6FO9 (Rusty Blazenhoff; Zuletzt abgerufen: 04.01.2016)

Raum. Der Schnittpunkt aller Vektoren eines Markers, errechnet aus unterschiedlichen Kamerawinkeln, beschreibt die genaue Position des Markers im dreidimensionalen Raum. Dieser Prozess nennt sich *Rekonstruktion*, da die Position der Marker aus 2D-Bilddaten, im dreidimensionalen Raum rekonstruiert werden.

Damit die Marker von den Kameras auch als solche erkannt werden, müssen sie sich deutlich vom Rest der Aufnahme abheben. Bei einem System mit passiven Markern wird daher Infrarotlicht von den Kameras ausgestrahlt, welches von den reflektierenden Markern zurück in den Infrarot-empfindlichen Bildsensor der Kamera geworfen wird. Diese Methode ist relativ kostengünstig und bewährt, auch wenn sie störanfällig gegenüber anderen Lichtquellen und reflektierenden Oberflächen ist.

Aktive Marker enthalten LED-Technologie und strahlen Licht aus, anstatt dieses nur zu reflektieren. Nach dem Abstandsgesetz $I = \frac{1}{r^2}$ (I = Intensität, r = Radius) [GP03, S.126f] resultiert dies zwangsläufig in besserer Sichtbarkeit der Marker für die Kameras, da das Licht nicht reflektiert werden muss und durch die direkte Ausstrahlung vom Marker eine kürzere Strecke zurücklegt. Aktive Marker können daher die Größe des Motion Capture-Volumen deutlich erhöhen. Auch steigt der Signal-Rausch-Abstand der Aufnahme, sodass es sogar möglich ist, bei schlechten Sichtverhältnissen und Tageslicht zu drehen ohne stark fehlerhafte Daten zu erhalten. Aktive Marker bieten dank LED-Technologie außerdem die Möglichkeit der Farb- und Frequenzkodierung, sodass es für Software leichter ist, die einzelnen Marker zu identifizieren.

Daten, die mit optischen Systemen aufgenommen wurden, müssen im Anschluss an die Aufnahme noch von Rauschen und falsch identifizierten Markern bereinigt und gegebenenfalls um fehlende Markerdaten ergänzt werden. Auch die Zuweisung der einzelnen digitalen Marker zum entsprechenden Marker am Körper des Darstellers, muss meistens in der Post-Production manuell festgelegt werden. Zu den Vorteilen eines optischen Systems zählen seine Genauigkeit und die Bestimmung der Marker-Positionen im Weltkoordinatensystem. Jede Markerposition, und damit auch die Darsteller-Position, kennt das System als absolute Position im dreidimensionalen Raum. Aus den gesammelten Markerdaten werden im *Solving* die lokalen Rotationen der einzelnen Körperteile (Starrkörper) errechnet, sodass diese sich später auf einen digitalen Charakter bzw. dessen Skelett übertragen lassen.

Abbildung 3.2: Aktives *Vicon* Motion Capture System mit passiven Markern an Körper und Gesicht des Darstellers.

Quelle: http://bit.ly/1Wkz7mp (Vicon Motion Systems Ltd., Zuletzt abgerufen: 17.01.2016)

3.1.2 Inertiale Systeme

Neben den optischen Motion Capture Systemen, die auf Marker Technologie basieren, gibt es noch weitere Ansätze für die Aufzeichnung von Bewegungen. Die wohl vielversprechendste Alternative zum optischen Verfahren basiert auf *Inertialsensoren*, besser bekannt als Beschleunigungs- und Gyrosensoren (siehe Kapitel 3.3). Ein Ansatz ist es, den Darsteller mit einer Reihe von Inertialsensoren zu bestücken, über welche die lokale Rotationen der Körperteile aufgezeichnet und diese anschließend auf ein digitales Skelett übertragen werden. Diese Methode hat den Vorteil, dass keinerlei Kameras benötigt werden um die Bewegungen aufzuzeichnen. So kann das Motion Capture Volumen deutlich größer sein, als es mit optischen Systemen möglich wäre. Ein deutlicher Nachteil gegenüber der optischen Technik ist jedoch die fehlende Aufzeichnung der Position des Darstellers im Weltkoordinatensystem, was zu ungenaueren Ergebnissen führt. Der Grund: Aus den Daten der Inertialsensoren können lediglich lokale Rotationen errechnet werden.

Inertiale Systeme wie *Perception Neuron*[1] (siehe Abbildung 3.3) sind bereits Preis im dreistelligen Bereich erhältlich, womit sie für Privatpersonen und kleine Unternehmen finanzierbar sind. Optische Systeme sind aufgrund der teureren Hardware im Regelfall keine Option für diese Zielgruppe, sondern richten sich an größere Produktionen.

Abbildung 3.3: Die Auf Inertialsensoren basierende *Perception Neuron* Prototyp Hardware im Einsatz.

Quelle: http://bit.ly/1U4Di4l (Perception Neuron (Flickr), Zuletzt abgerufen: 17.01.2016)

3.1.3 Performance Capture

Während Motion Capture bereits großflächig eingesetzt wird, gewinnt auch die Aufzeichnung von Mimik (*Facial Motion Capture*) und Gestik in Form von Fingerbewegungen, immer mehr an Popularität. Mithilfe von Helmkameras werden direkt beim Motion Capture Dreh auch die Gesichtsbewegungen der Darsteller (siehe Abbildung 3.4), sowie feine Fingerbewegungen mit speziellen Handschuhen aufgezeichnet. Diese erweitere Form des Motion Capture wird als *Performance Capture* bezeichnet.

Beim Motion Capture des Gesichts gibt es mehrere Ansätze: Während die ersten Systeme noch mit aktiven oder passiven Markern im Gesicht der Darsteller arbeiteten, kommen

[1] https://neuronmocap.com (Zuletzt abgerufen: 05.01.2016)

zunehmend markerlose Systeme zum Einsatz. Diese analysieren das aufgenommene Videomaterial, suchen sich markante Punkte im Gesicht des Darstellers und verfolgen deren Bewegungsablauf. Keine der Techniken liefert bisher perfekte und fehlerfreie Ergebnisse, sodass erkannte Gesichtsposen immer manuell nachbearbeitet werden, um das gewünschte Ergebnis zu erzielen. Aus den verarbeiteten Daten wird dann wieder eine Deformation errechnet, die sich auf ein digitales Gesicht übertragen lässt. Die verbreitetste Softwarelösung für Facial Motion Capture ist derzeit *Faceware*[2] von Image Metrics.

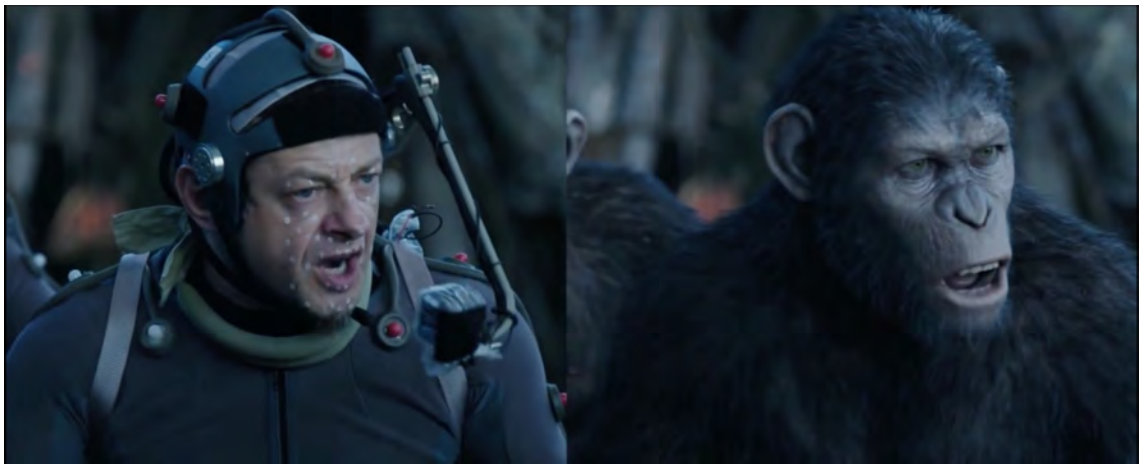


Abbildung 3.4: Performance Capture am Set von *Dawn of the Planet of the Apes* mit aktiven Markern am Körper der Darsteller und einer Helmkamera zur Aufzeichnung der Mimik.

Quelle: http://bit.ly/1RLAxFA (Wired / 20th Century Fox; Zuletzt abgerufen: 13.01.2016)

3.2 Autodesk MotionBuilder

MotionBuilder ist eine von Autodesk Inc. entwickelte 3D-Software, die sich im Gegensatz zu anderen Autodesk Softwarepaketen wie *Maya* oder *3ds Max* auf ein bestimmtes Anwendungsgebiet konzentriert, anstatt einen breit gefächerten Allround-Funktionsumfang zu bieten. Speziell für 3D-Charakter Animation entwickelt, bietet MotionBuilder ein sehr ausgereiftes Set an Animationswerkzeugen. Ursprünglich unter dem Namen *Filmbox* vermarktet, wurde es später von Autodesk übernommen, in MotionBuilder umbenannt und erfreut sich besonders in den letzten Jahren wachsender Popularität. Grund dafür ist die sehr gute Performance der Software, welche es ermöglicht, Animationen auf mehreren komplexen Charakter Rigs mit vielen Polygonen in Echtzeit wiederzugeben (siehe Abbildung 3.5).

Ein beliebtes Anwendungsgebiet für MotionBuilder ist die Bearbeitung von Motion Capture Daten (*Motion Editing*). MotionBuilder bietet eine Reihe nützlicher Werkzeuge für das

[2]http://facewaretech.com (Zuletzt abgerufen: 05.01.2016)

Motion Editing, wie z.B. das Story Tool, in dem aufgenommene Sequenzen zusammengesetzt und Übergänge geschaffen werden können. Das Ebenensystem bietet die Möglichkeit Motion Capture Daten komfortabel zu bearbeiten, ohne zwingend die Originalkurven verändern zu müssen.

Im Bereich der Virtual Production ist MotionBuilder ebenfalls sehr verbreitet. Besonders attraktiv ist die Software auf diesem Gebiet durch ihre Anbindung an andere Autodesk Softwarepakete und ihre einfache Erweiterbarkeit per Python oder C++ Programmierschnittstelle. Dank der sehr effizienten Engine mit Unterstützung von Echtzeit-Simulationen, wird die Software auch zunehmend im Bereich der On-Set Pre-Visualisierung und Virtual Cinematography (siehe Kapitel 2.1.2) eingesetzt. Virtuelle Kamera-Systeme basieren oft auf einer laufenden MotionBuilder Instanz im Hintergrund.

Abbildung 3.5: Eine komplexe Szene mit vielen Polygonen und Charakteren lässt sich mit MotionBuilder bei einer stabilen Bildrate in Echtzeit darstellen und bearbeiten.

Quelle: http://www.metricminds.com (bereitgestellt von metricminds GmbH & Co. KG)

3.3 Inertialsensoren im Apple iPad

Als ein Anwendungsbeispiel für den Schnittstellen-Prototyp, wird im Rahmen dieser Arbeit eine iOS App entwickelt, die Sensordaten aus den Hardwarekomponenten eines Apple iPad[3] ausliest und sie zur weiteren Verwendung an MotionBuilder überträgt. Explizit handelt es sich dabei um die Komponenten zur Messung der Drehrate und der Beschleunigung des Gerätes, aus denen die Lage des Gerätes im dreidimensionalen Raum bestimmt werden kann. Die aktuelle iPad Generation, dass *iPad Air 2*, verfügt laut den technischen Spezifikationen[4] über ein Drei-Achsen Gyroskop (Rotation), ein Magnetometer (digitaler Kompass),

[3] http://www.apple.com/ipad/ (Zuletzt abgerufen: 03.01.2016)
[4] http://www.apple.com/ipad-air-2/specs/ (Zuletzt abgerufen: 03.01.2016)

sowie drei-achsige Beschleunigungssensoren. Diese normalerweise recht großen Bauteile sind im iPad in Form von *MEMS*-Komponenten[5] (Mikrosystem, eng. *„microelectromechanical systems"*) integriert. Als Mikrosysteme sind sie nicht nur deutlich kleiner als herkömmliche mechanische Komponenten, sondern arbeiten auch energieeffizienter und sind vergleichsweise preiswert in der Herstellung. Ähnliche MEMS-Komponenten mit Gyroskop befinden sich unter anderem auch in anderer Unterhaltungselektronik, wie etwa der *Wii Remote Plus*[6] von Nintendo.

Im Testgerät, dem iPad Air 2, kommt der Beschleunigungssensor *BMA280*[7] von Bosch, sowie ein unbekannter Gyrometer Chip zum Einsatz. Der Hersteller verbaut in dem Gerät außerdem den eigenen *Apple M8*[8] Chip, der für die Auswertung der Sensoren zuständig ist und den Hauptprozessor so entlastet. Die für diese Thesis relevanten Sensoren, namentlich das Gyrometer und die Beschleunigungssensoren, lassen sich auch unter dem Begriff *Inertialsensoren* zusammenfassen und über das iOS SDK per Programmierschnittstelle komfortabel abfragen. Im folgenden Abschnitt möchte ich kurz die Funktionsweise dieser Sensoren und ihren Nutzen erläutern.

Abbildung 3.6: Mainboard eines *iPad Air* (Baujahr 2012) mit rot markiertem M7 Motion Coprozessor für die Verarbeitung der Bewegungsdaten. Der Aufbau ist ähnlich wie im *iPad Air 2*, dem verwendeten Testgerät.

Quelle: http://bit.ly/1KbLERS (IHS Technology; modifizierte Abbildung; Zuletzt abgerufen: 12.01.2016)

[5] https://en.wikipedia.org/wiki/Microelectromechanical_systems (Zuletzt abgerufen: 03.01.2016)
[6] http://bit.ly/1R5dzup (Zuletzt abgerufen: 03.01.2016)
[7] http://bit.ly/1ZIhP4B (Zuletzt abgerufen: 03.01.2016)
[8] https://en.wikipedia.org/wiki/Apple_motion_coprocessors (Zuletzt abgerufen: 03.01.2016)

3.3.1 Beschleunigungssensor

Die einfachste Möglichkeit externe Bewegungseinflüsse auf Geräte wie das iPad zu bestimmen, ist der Einsatz von drei Beschleunigungsmessern, von denen jeweils ein Sensor, eine der drei Raumachsen (x,y,z) abdeckt. Ein Beschleunigungssensor misst dabei die lineare Beschleunigung entlang einer Achse.

Simple Beschleunigungsmesser nutzen zur Messung eine Masse, die an einer Feder befestigt ist. Wird die Masse entlang der Feder beschleunigt, dehnt sich diese entsprechend der Kraft aus, die notwendig ist, um diese Beschleunigung zu erzeugen. Anhand der Federkonstante (die benötigte Kraft zur Ausdehnung um eine bestimmte Strecke) und der Dehnung der Feder, lässt sich die aufgewandte Kraft bestimmen. Da die Masse bekannt ist, lässt sich mit der Kraft, die Beschleunigung errechnen:

$$a = \frac{c \cdot s}{m} \tag{3.1}$$

c : Federkonstante
s : Ausdehnungsstrecke
m : Masse
a : Beschleunigung

Ist die Beschleunigung für alle drei Achsen ermittelt, lässt sich daraus auf Beschleunigungen aus beliebigen Richtungen schließen. Über das Verhältnis der drei Beschleunigungen, lässt sich unter Berücksichtigung der Erdbeschleunigung außerdem die Ausrichtung des Sensors bzw. Gerätes bestimmen.

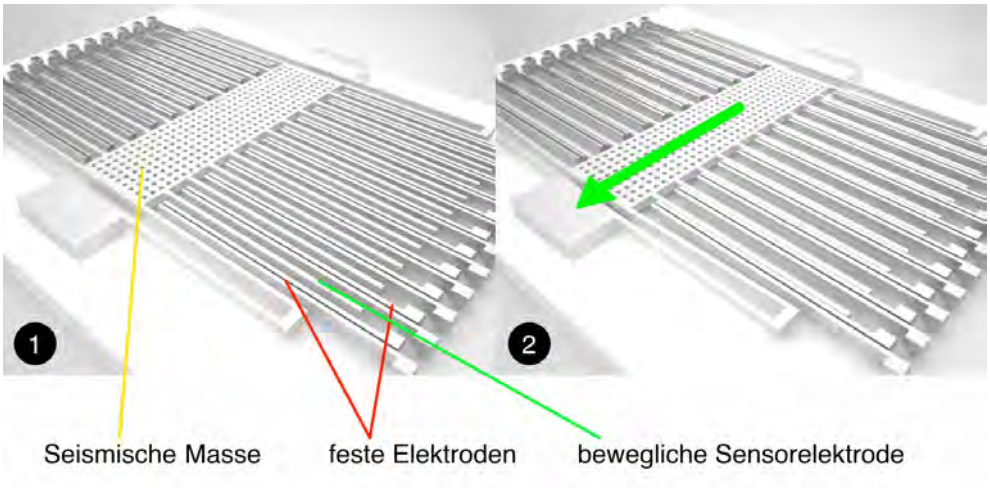


Abbildung 3.7: Funktionsweise eines MEMS-Beschleunigungssensors. (1) Sensor im Ruhezustand. Keine wirkende Beschleunigung auf die Masse. (2) Eine Beschleunigung wirkt auf die seismische Masse. Kapazitätsänderungen zwischen den festen Elektroden und der Sensorelektrode treten auf.

Quelle: http://bit.ly/1OqpEVo (Rudolf Herstek; modifizierte Screenshots; Zuletzt abgerufen: 13.01.2016)

Beschleunigungssensoren in kleinen Elektronikgeräten arbeiten nach dem gleichen Prinzip, setzen jedoch auf winzige Siliziumstrukturen, in denen eine schwingende Masse (meistens ein Kristall) mit seiner Sensorelektrode für Kapazitätsschwankungen zwischen fixierten Elektroden sorgt (siehe Abbildung 3.7). Diese entsprechen der Dehnung der Feder im vorherigen Beispiel und werden entsprechend interpretiert, um die Beschleunigung zu errechnen.

Sensoren dieser Art werden bereits seit vielen Jahren in Smartdevices wie dem iPad verbaut und dienen in erster Linie zur Anpassung des Bildschirminhalts an die Geräterotation. Viele Apps nutzen sie zudem als alternative Eingabemöglichkeit zum Touchscreen. Besonders in mobilen Videospielen findet diese Eingabemethode vermehrt Verwendung.

3.3.2 Gyrometer

Um die Lage des Geräts im dreidimensionalen Raum noch genauer zu bestimmen, wird seit einigen Jahren in Smartdevices zusätzlich ein sogenanntes *Gyrometer* verbaut. Hinter der Bezeichnung versteckt sich ein Drehratensensor (Gyroskop) zur Registrierung von Rotationsbewegungen. Dieser bestimmt keine Beschleunigungen, sondern Winkelgeschwindigkeiten am Sensor, also die Winkeländerung über eine Zeiteinheit. In Smartphones und Tablets kommt als Gyroskop ein *Vibrationskreisel*[9] (eng. *Vibrating structure gyroscope*) zum Einsatz.

Während große Gyroskope in Flugzeugen und der Raumfahrt mit Lasertechnik arbeiten, wird in MEMS-Gyroskopen die wirkende *Corioliskraft* F_c auf ein vibrierendes Element gemessen.[10]

$$F_c = 2mv \cdot \omega \cdot sin\theta \tag{3.2}$$

F_c : Corioliskraft
m : Masse
ω : Winkelgeschwindigkeit
v : Bewegungsrichtung
θ : Winkel zwischen Bewegungsrichtung v und Richtung der Rotationsachse des rotierenden Objektes

Die Genaugkeit der Messung hängt dabei immer von den verwendeten Materialien für den Vibrationskreisel ab. Oft kommt hier eine Kristallstruktur zum Einsatz, die bei Betrieb des Gyroskop eine gleichmäßige Schwingung hält. Wird das Gyroskop rotiert, erzeugt die wirkende Corioliskraft weitere Vibrationen im Kristall, die für eine Änderung im elektrischen Potenzial sorgt. Diese Änderung wird dann vom Sensor interpretiert und in eine Winkelgeschwindigkeit umgerechnet (siehe Abbildung 3.8).

[9] http://bit.ly/1TDT1XP (Zuletzt abgerufen: 06.01.2016)
[10] http://physik.wikia.com/wiki/Corioliskraft (Zuletzt abgerufen: 06.01.2016)

Ein Großteil der MEMS-Gyrometer wird heutzutage vom Unternehmen *InvenSense*[11] produziert, dem Marktführer in diesem Bereich. Die MEMS-Gyrometer der Firma kommen primär in Unterhaltungselektronik zum Einsatz.

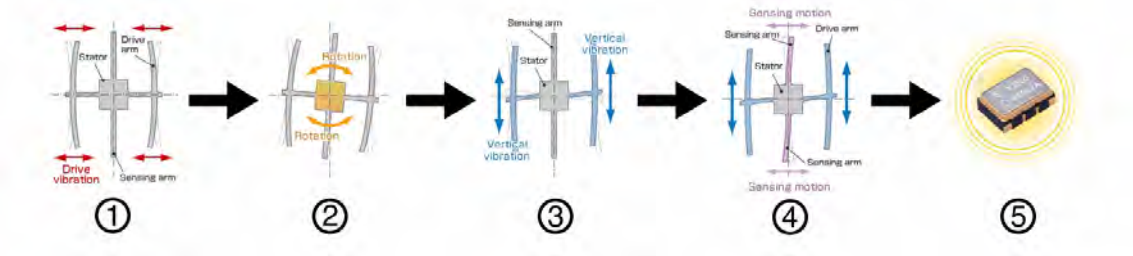


Abbildung 3.8: (1) Der Sensor erzeugt eine kontinuierliche horizontale Schwingung an den Treiberarmen. (2) Der Gyrosensor wird durch äußere Einwirkung rotiert. (3) Bei einer Drehung des Gyro erzeugt die Corioliskraft vertikale Schwingungen in den Treiberarmen. (4) Die stationären Sensorarme biegen sich unter dem Einfluss der schwingenden Treiberarme und beginnen zu schwingen. (5) Die Schwingung der Sensorarme erzeugt eine Potenzialdifferenz, von der die Winkelgeschwindigkeit abgeleitet wird.

Quelle: (1) - (5): http://bit.ly/1TDT1XP (Seiko Epson Corp., modifiziert; Zuletzt abgerufen: 06.01.2016)

3.4 Leap Motion

Als zweites Anwendungsbeispiel für den Plugin-Prototypen soll der *Leap Motion*-Controller[12] des gleichnamigen US-Unternehmens an die MotionBuilder Realtime Engine angebunden werden. Dabei handelt es sich um ein kleines Eingabegerät, dass über USB mit einem Computer verbunden und vor den Anwender gelegt wird. Das Gerät ist in der Lage, Hand- und Fingerpositionen im dreidimensionalen Raum mit hoher Geschwindigkeit zu bestimmen, ohne das der Nutzer dabei spezielle Sensoren oder Markierungen tragen muss.

Der Controller verfügt über zwei monochromatische Infrarot-Kameras, die das reflektierte Licht der drei verbauten Infrarot-Dioden aufzeichnen und die gesammelten Bilddaten an die Leap Motion-Software übertragen, welche auf dem verbundenen Computer laufen muss. Diese analysiert die gesammelten Daten und nutzt bisher noch unveröffentlichte Algorithmen, um aus den zwei 2D-Bildern der Kameras, 3D-Koordinaten für Hände und Finger zu errechnen. Die Software bietet eine Tracking-Genauigkeit im Millimeterbereich und unterstützt die parallele Aufzeichnung mehrerer Hände. Längliche Objekte wie z.B. Stifte werden als *Tool* registriert und aufgezeichnet. Die Kameras haben einen 150°-Blickwinkel und erkennen Objekte im Abstand von 25 bis 600 Millimetern zum Gerät.

[11] http://www.invensense.com (Zuletzt abgerufen: 06.01.2016)
[12] https://www.leapmotion.com (Zuletzt abgerufen: 17.01.2016)

Die gesammelten Daten stellt der Leap Motion-Controller dem Entwickler über das *Leap Motion SDK*[13] zur Verfügung. Das SDK bietet eine Vielzahl von Programmierschnittstellen an, über die sich Gesten, sowie Positions- und Bewegungsdaten der erkannten Objekte (Hände, Finger, Tools) abfragen lassen. Leap Motion unterstützt 14 Plattformen und Programmiersprachen (Stand: 17.01.2016), darunter auch C++ , Objective-C und Unity 3D.

Abbildung 3.9: (A) Visualisierung der Sicht des Leap Motion-Kameras auf die Hände des Anwenders. (B) Sichtfeld-Visualisierung des Leap Motion-Controllers.

Quelle: (A) http://bit.ly/1JSe6xp (Leap Motion Inc., Zuletzt abgerufen: 17.01.2016)
(B) http://bit.ly/1KoE8Sd (Leap Motion Inc., Zuletzt abgerufen: 17.01.2016)

3.5 TCP/IP

Für die Kommunikation der Datenquellen mit dem Plugin-Prototypen, wird ein eigenes Netzwerkprotokoll auf Basis von TCP/IP Verbindungen eingesetzt. Zum besseren Verständnis der Thematik, wird in diesem Abschnitt ein kleiner Einblick in die Netzwerkkommunikation mit TCP/IP gegeben.

Das *Transmission Control Protocol* (TCP) und das *Internet Protocol* (IP) sind eine zusammenarbeitende Protokollfamilie zum Datentransport in einem *dezentralen Netzwerk*. Unter letzterem versteht man ein Netzwerk, indem es keinen zentralen Knotenpunkt gibt, von dem alle Kommunikationen abhängig sind. Das bekannteste Beispiel hierfür ist das globale Internet, bei dem Datenpakete über unzählige Knotenpunkte an ihr Ziel geleitet werden. Sollte ein Knotenpunkt überlastet sein, kann das Datenpaket sein Ziel immer noch über eine alternative Verbindung erreichen. Das TCP Protokoll ist dabei in der *Transportschicht* des *OSI-Schichtenmodell* angesiedelt, während das IP Protokoll der *Vermittlungsschicht* zuzuordnen ist. Im vereinfachten *DDN-Modell* liegt das IP Protokoll auf der *Internetschicht*, während TCP auf der *Host-zu-Host Schicht* angesiedelt ist (siehe Tabelle 3.1).

[13]https://developer.leapmotion.com (Zuletzt abgerufen: 17.01.2016)

OSI-Modell	DDN-Modell
7. Anwendungsschicht	4. Anwendungsschicht
6. Darstellungsschicht	
5. Sitzungsschicht	
4. Transportschicht	3. Host-zu-Host Schicht
3. Vermittlungsschicht	2. Internetschicht
2. Sicherungsschicht	1. Netzzugangsschicht
1. Bitübertragungsschicht	

Tabelle 3.1: OSI-Modell und DDN-Modell der Netzwerkprotokolle. [Ker09, S. 176]

3.5.1 Transmission Control Protocol

TCP ist als Teil der Transportschicht für eine zuverlässige Übertragung der Daten zuständig. Als verbindungsorientiertes Protokoll stellt es sicher, dass die gesendeten Datenpakete stets vollständig und fehlerfrei beim Empfänger ankommen. Eine Alternative zu TCP ist das verbindungslose *User Datagram Protocol* (UDP), welches ebenfalls für den Transport von Daten zuständig ist. Im Gegensatz zu TCP besitzt das Protokoll jedoch keine Sicherheitsmechanismen, welche die korrekte Übertragung der Datenpakete sicherstellen.

Vor der Datenübertragung stellt TCP eine virtuelle Punkt-zu-Punkt Verbindung (*Point-to-Point Protocol*, kurz *PPP*) zur stabilen Kommunikation zwischen den Systemen her und versieht jedes Paket mit einer Sequenznummer. Bei der Ankunft am Empfänger, wird der Empfang bestätigt und über die Sequenznummer eine korrekte Reihenfolge der Pakete sichergestellt. Bleibt die Bestätigung seitens des Empfängers innerhalb der festgelegten Timeout-Zeit aus, sendet TCP das verlorene Paket einfach erneut. UDP hingegen sendet Datenpakete ohne Verbindungsaufbau und Kontrolle, was die Übertragung besonders schnell, aber auch sehr unzuverlässig macht. Eine erneute Übertragung des Pakets im Fall von Datenverlust ist nicht im Protokoll vorhergesehen.

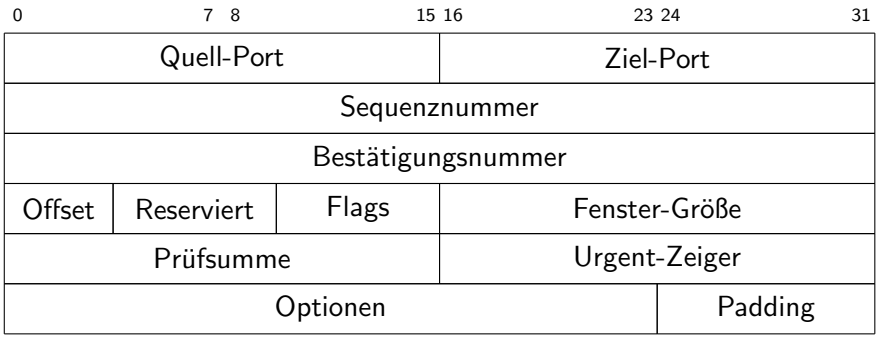


Abbildung 3.10: Aufbau eines TCP-Headers. [Ker09, S. 242]

Sowohl TCP als auch UDP stellen über die sogenannte *Portnummer* sicher, dass Datenpakete beim Empfängersystem dem korrekten Programm zugestellt werden. Eine Software kann sich dazu Ports im Betriebssystem reservieren, über die Datenverbindungen erfolgen. Portnummern ermöglichen es außerdem, von einem System zeitgleich mehrere Verbindungen zu einer IP-Adresse (siehe Kapitel 3.5.2) aufzubauen. Diese zusätzlichen Daten werden jedem TCP Paket in Form eines *Headers* (siehe Abbildung 3.10) hinzugefügt, der die Paketgröße um 20 Bytes erhöht. Hier punktet wieder das weniger komplexe UDP, welches lediglich acht Byte Headerdaten pro Paket benötigt.

3.5.2 Internet Protocol

Während TCP sich um die korrekte und fehlerfrei Übertragung der Daten kümmert, erfolgt die eigentliche Wegweisung (Routing) der Datenpakete über das Internet Protokoll in der darunter liegenden Vermittlungsschicht. Im Gegensatz zu UDP, dass seine Pakete an wahllos alle Teilnehmer im Netzwerk schickt, macht sich TCP das Internet Protokoll zu nutze, um die Datenpakete auch über mehrere *Subnetze* hinweg nur an das angepeilte Zielnetz zu senden, wo ein *Router* sich um die Zustellung an den korrekten Empfänger kümmert.

TCP Pakete werden über die sogenannte *IP-Adresse* ihren Empfängern zugeordnet. Diese sind einmalig im Netzwerk und werden meistens automatisch vom Router, allen Systemen im eigenen Netz anhand ihrer MAC-Adresse (physische Adresse der Netzwerkkarte eines Systems) zugewiesen. Eine IP-Adresse ist in mehrere Adressblöcke aufgeteilt, die den gesamten Adressraum in Subnetze aufteilen, sodass Programme Datenpakete auch an Systeme außerhalb ihres eigenen Netzwerks (Subnetz) adressieren können. Dies ist auch die Grundlage für die erfolgreiche Kommunikation im größten zusammenhängenden Netzwerk der Welt: dem Internet. Ein IP-Header (siehe Abbildung 3.11) ist mindestens 20 Byte groß.

Kommunikation in besonders großen Adressräumen wie dem Internet, wird durch *Domain Name Server* Dienste vereinfacht. Diese führen Buch über die Zuweisung von Domains (Beispiele: *example.com, beispiel.de*) zu IP-Adressen, sodass Datenpakete an `google.de` beispielsweise automatisch an den Google Server mit der IP-Adresse `195.13.231.157` weitergeleitet werden. (DNS Stand: 05.01.2016)

3.5.3 Sockets

Eine TCP/IP Verbindung basiert immer auf sogenannten *Sockets* als Grundlage für die Kommunikation zwischen zwei Computern. Ein Socket ist ein vom Betriebssystem bereitgestellter Kommunikationsendpunkt und kommt immer im Paar zum Einsatz. Ein Socket-Paar ermöglicht eine bidirektionale Kommunikation zwischen beiden Endpunkten und parallele Schreib- / Lesevorgänge auf beiden Seiten. Sockets werden von Applikationen beim Betriebssystem angefordert und belegen dort jeweils einen Port im Pool des Systems. Insgesamt 65535 Ports kann ein Betriebssystem zur Verfügung stellen, von denen jedoch viele standardmäßige für Dienste wie E-Mail und HTTP reserviert sind. Man unterscheidet zwischen zwei Typen von Sockets *Datagramm-Sockets* und *Stream-Sockets*. Bei Datagramm-

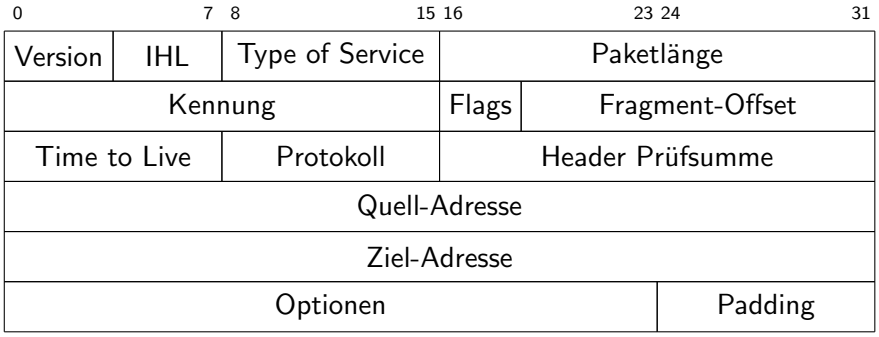


Abbildung 3.11: Aufbau eines IP-Headers. [Ker09, S. 226]

Sockets handelt es sich um verbindungslose Sockets, die für die Kommunikation per UDP verwendet werden. Stream-Sockets erfordern hingegen einen Verbindungsaufbau und für TCP-Kommunikation eingesetzt.

Zum Aufbau einer Stream-Socket-Verbindung wird das *Client-Server Modell* verwendet. Der Computer der als Server agiert, fordert einen freien Port beim System an und eröffnet einen *Listener-Socket*, der auf eingehende Verbindungen wartet. Auf der Seite des Clients wird ebenfalls ein Socket eröffnet und dieser mit der Adresse und dem Port des Servers verbunden. Der Server akzeptiert die Verbindung und benachrichtigt den Client. Die bidirektionale Verbindung ist nun hergestellt und die Computer können nun über das Socket-Paar kommunizieren. Nach Abschluss der Kommunikation werden die Sockets auf beiden Seiten terminiert und die belegten Ports damit wieder zur Verwendung freigegeben. Stream-Sockets bilden die Basis auf welcher der in dieser Arbeit besprochene Plugin-Prototyp mit externen Datenquellen über das Netzwerk kommuniziert.

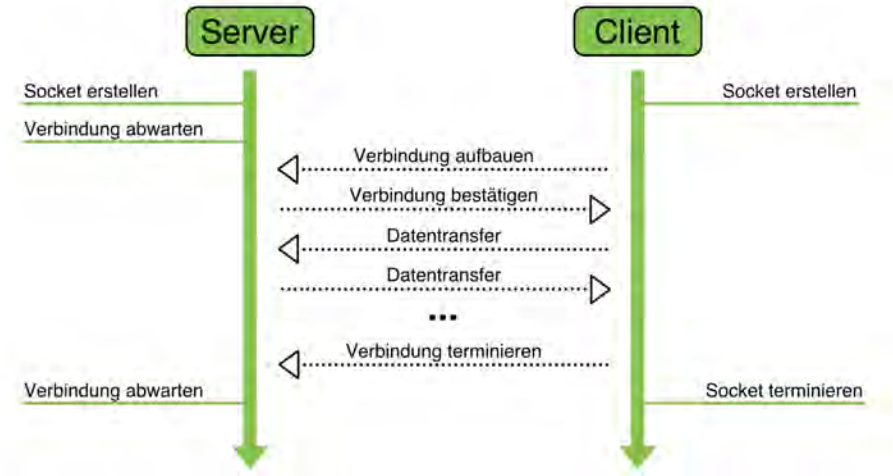


Abbildung 3.12: Grundlegender Kommunikationsaufbau über Stream-Sockets.

Kapitel 4

Konzeption

Dieses Kapitel beschäftigt sich mit der Konzeptionierung, der in dieser Arbeit besprochenen Softwarelösungen. Dies beinhaltet das MotionBuilder-Plugin, sowie beide Anwendungsbeispiele zur Überprüfung der Praktikabilität der Netzwerkschnittstelle. Es werden die Rahmenbedingungen für die Entwicklung gesetzt und die einzelnen Komponenten der Prototypen im Detail abgehandelt. Im Anschluss an dieses Kapitel hat der Leser das konzeptionelle Wissen erlernt um mit der Umsetzung der Software fortzufahren.

4.1 MotionBuilder Realtime Streaming Plugin

Dieses Kapitel befasst sich mit der Entwicklung des Realtime Streaming Plugins für Autodesk MotionBuilder. Im Zuge dessen wird das entworfene Kommunikationsprotokoll und die Strukturierung des Plugins besprochen. Weiterhin werden wichtige Entscheidungen der Konzeptionsphase begründet.

4.1.1 Zielsetzung

Die Aufgabe des Plugins ist die Einbindung neuer Datenquellen, an die Realtime-Engine von Autodesk MotionBuilder. Das Plugin soll universell nutzbar sein und keinerlei Änderungen am Quellcode zur Unterstützung neuer Eingabegeräte erfordern. Dazu muss es eine variable, aber dennoch strukturierte Datenmenge aus Gleitkommazahlen, über eine standardisierte Netzwerkschnittstelle, empfangen und verarbeiten können.

Es ist vorgesehen, dass die empfangenen Daten dem Anwender über ein *Relation Constraint* zugänglich gemacht werden. Dabei handelt es sich um ein kausales Netzwerk aus Szenenobjekten und Operatoren, deren Verbindungen über einen grafischen Editor verwaltet werden. Das Relation Constraint ist ein leistungsstarkes Werkzeug und lässt sich am ehesten mit dem *Node Editor* in Autodesk Maya vergleichen. Die einzelnen Datensätze werden in MotionBuilder mit einer Bezeichnung versehen, die eine korrekte Zuordnung der Nutzdaten ermöglicht.

Das Plugin hat dabei nicht die Aufgabe die eingehenden Daten zu interpretieren, sondern überlässt die Umsetzung der kompletten Logik dem Anwender. Diese wird im Relation Constraint wahlweise manuell oder skriptbasiert aufgesetzt. Eingehende Nutzdaten können so im Relation Constraint flexibel eingesetzt und die Logik zu deren Interpretation auch ohne Programmierkenntnisse angepasst werden.

Weiterhin ist ein Filtermechanismus für die Schnittstelle vorgesehen, der langsame Datenpakete aussortiert, um fehlerhafte Animationen zu vermeiden. Das Plugin muss die empfangenen Daten neben einer Echtzeit-Darstellung zudem in Keyframes schreiben können, sodass diese in der MotionBuilder-Szene verwendet werden können, unabhängig von dem Status der Datenverbindung. Damit das Plugin aktiv in der Produktion verwendet werden kann muss ein Minimum von 30 Samples pro Sekunde unabhängig von der Anwendung gewährleistet sein. Dies entspricht dem aktuellen Produktionsstandard der Animationsindustrie.

Das Plugin soll über einen simplen TCP-Netzwerksocket ansprechbar sein und mit Datenquellen über LAN oder Wireless LAN in einer lokalen Netzwerkumgebung kommunizieren können. Eine Verbindung über das Internet oder mehrere Netze ist nicht eingeplant. Wichtig ist außerdem, dass der Programmieraufwand zur Kommunikation mit dem Plugin auf Seiten der Datenquellen so gering wie möglich gehalten wird, damit neue Geräte zügig mit dem Plugin verbunden werden können. Weiterhin muss sichergestellt werden, dass die Software typische Anwenderfehler abfängt, eingehende Daten zuverlässig verarbeitet und auftretende Fehler für die Analyse protokolliert.

4.1.2 Vergleich: Python und C++ MotionBuilder API

Im Zuge der Planung ist zu entscheiden, mit welchen Mitteln das Plugin umgesetzt werden soll. Zur Integration in MotionBuilder bietet der Entwickler Autodesk hierzu zwei Programmierschnittstellen an, die entweder über die Skriptsprache *Python* oder die Compilersprache C++ angesprochen werden. In diesem Abschnitt werden die Vor- und Nachteile der beiden Schnittstellen verglichen und die Entscheidung getroffen, welche der beiden Lösungen für den Prototypen die geeignetere Wahl ist.

Das MotionBuilder *Software Development Kit* (SDK) erlaubt dem Nutzer die Funktionalität der Autodesk Software mithilfe von Plugins und Skripten erweitern oder für eigene Zwecke anzupassen. Dabei bietet die unter dem Namen *Open Reality SDK* (OR SDK) vertriebene Programmierschnittstelle (engl. *Application programming interface*, API) auf Basis von C++ Code einen anderen Funktionsumfang, als das auf Python basierende *Pyfbsdk*-Modul. Grund dafür ist der Aufbau des SDKs, welcher den Großteil der MotionBuilder Funktionalität über das OR SDK ansprechbar macht. Das Pyfbsdk-Modul ist hingegen dem OR SDK untergeordnet und legt die meisten C++ -Funktionen über analoge Funktionssignaturen für Python-Programmierer teilweise frei.

MotionBuilder Erweiterungen, die mit dem Open Reality SDK entwickelt werden, müssen mit der Entwicklungsumgebung *Microsoft Visual Studio*[1] als *Dynamically Linked Library* (DLL) kompiliert werden und in das entsprechende Verzeichnis gelegt werden. MotionBuilder lädt kompatible DLL-Dateien einmalig während des Startprozess und gibt dem Nutzer deren Funktionalität frei. Skripte auf Python-Basis besitzen hingegen die Funktionalität, zur Laufzeit auf Quellcode-Änderungen zu reagieren. Sie laufen über den MotionBuilder eigenen Python-Interpreter, und werden als Teil des MotionBuilder-Prozesses ausgeführt. Python ist eine sehr attraktive Option für viele Programmierer, da sie schnell erste Ergebnisse liefert und die Softwareentwicklung keine separate *IDE* (Entwicklungsumgebung) erfordert. MotionBuilder verfügt von Haus aus über einen integrierten Python-Editor. Die Skriptsprache wird außerdem von anderen Softwarepaketen wie Autodesk Maya unterstützt, was den Einstieg in die Entwicklung für diese Applikationen vereinfacht.

Bei der Entwicklung von Performance-kritischen Erweiterungen, ist das C++ -basierte Open Reality SDK, Python jedoch deutlich überlegen. Gerade Realtime-Anwendungen sollten immer auf Basis der OR SDK API entwickelt werden, da mit C++-Maschinencode eine bessere Performance bietet. Einige wichtige Funktionalitäten, darunter auch die Programmierung eines *Device* (siehe Kapitel 4.1.3), sind ausschließlich über das OR SDK zugänglich.

Da es sich bei dem geplanten Prototypen um eine Performance-kritische Anwendung für die Realtime-Engine von MotionBuilder handelt und die Umsetzung ein eigenes *Device* erfordert, wird letztendlich das Open Reality SDK für die Entwicklung gewählt. Das Pyfbsdk-Modul wird diesen Anforderungen, aufgrund der zuvor genannten Nachteile, nicht gerecht.

	Open Reality SDK (C++)	**Pyfbsdk-Modul (Python)**
Vorteile	- vollen Funktionsumfang des SDK - sehr gute Performance	- keine IDE notwendig - zugänglich und flexibel
Nachteile	- benötigt Microsoft Visual Studio - Code muss kompliliert werden	- eingeschränkte Funktionalität - schlechtere Performance

Tabelle 4.1: Vor- und Nachteile der verfügbaren MotionBuilder Programmierschnittstellen.

4.1.3 Plugin-Komponenten

Für die Planung eines MotionBuilder Plugins werden die einzelnen Funktionen, wie in der Softwareentwicklung üblich, in logische Komponenten aufgeteilt. Diese werden mit dem Open Reality SDK umgesetzt. Im Falle dieses Software-Prototypen ist die resultierende Struktur einfach und übersichtlich.

Der Großteil der Funktionalität, darunter die Netzwerkkommunikation, sowie die Speicherung und Verwaltung der Daten, werden von einem *Device* übernommen. Darunter versteht MotionBuilder eine Programmschnittstelle, die Daten senden und empfangen kann. Devi-

[1] https://www.visualstudio.com (Zuletzt abgerufen: 18.02.2016)

ces übernehmen die Kommunikation mit Eingabegeräten wie z.B. der *Microsoft Kinect*[2] oder Grafiktablets. Ein Device muss jedoch nicht zwangsläufig eine Hardware repräsentieren, sondern kann als Software-Schnittstelle auch für andere Anwendungszwecke verwendet werden. Somit wird ein Device entwickelt, das als standardisierte Schnittstelle für eingehende Daten genutzt werden kann. Der Empfang der Daten soll dabei über eine Socket-basierte Netzwerkschnittstelle erfolgen. Wie in Kapitel 4.1.1 erwähnt, findet die Interpretation der eingehenden Animationsdaten, extern, in einem *Relation Constraint* statt.

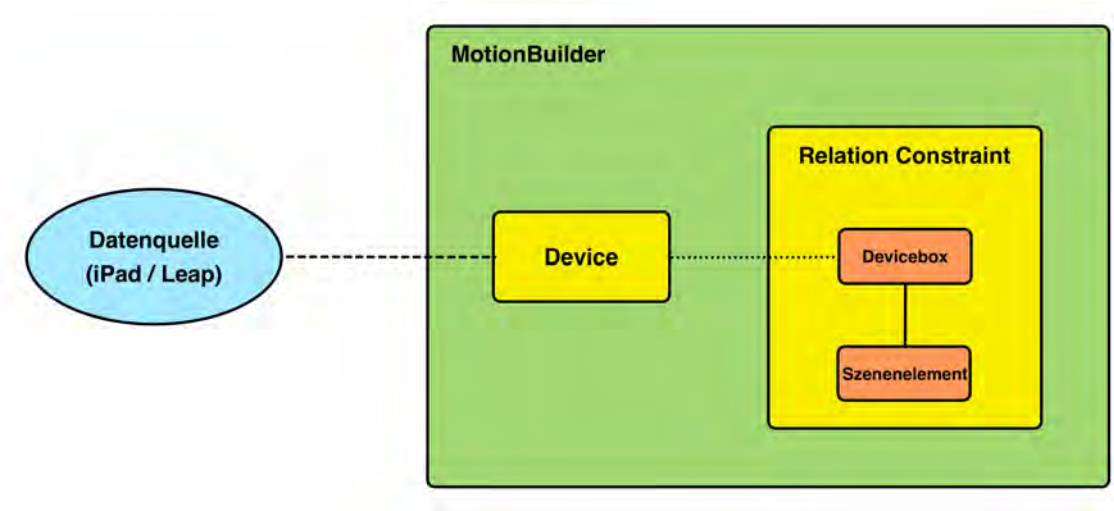


Abbildung 4.1: Struktur und Vernetzung des MotionBuilder Plugins mit den einzelnen Komponenten in Gelb.

4.1.4 Klassenstruktur

Basierend auf der geplanten Plugin-Struktur, besprochen in Kapitel 4.1.3, wird die Klassenstruktur der einzelnen Komponenten abgeleitet. Für die Entwicklung des Plugins wird das *OR Device Template* des Open Reality SDK als Vorlage gewählt. Dabei handelt es sich um Beispielcode, welcher MotionBuilder 2015 beiliegt und als Basis für die Entwicklung eigener Devices verwendet werden kann. Basierend auf der Struktur der Vorlage, wurde das vereinfachte Klassendiagramm in Abbildung 4.2 entwickelt.

Kern des Plugins ist diese *Device*-Klasse. Sie bildet die Schnittstelle zwischen den einzelnen Klassen des Plugins und kümmert sich um einige der wichtigsten Aufgaben: Die Verwaltung der eingehenden Daten findet komplett in der Device-Klasse statt. Dies beinhaltet die Evaluation der Daten für die Echtzeit-Darstellung, sowie die Speicherung der Daten in Form von Keyframes. Die ergänzte Device-Klasse umfasst zudem die Datenstruktur des Eingabegerätes und kümmert sich um die korrekte Einbindung dieser, in das FBX-Dateiformat. Zusätzlich steuert sie die Netzwerkkommunikation und versendet Anfragen, welche durch die *Hardware*-Klasse ausgeführt werden.

[2]https://dev.windows.com/en-us/kinect (Zuletzt abgerufen: 18.02.2016)

Die Hardware-Klasse ist die Schnittstelle zwischen der Device-Klasse und den Datenquellen, welche mit dem Plugin über eine Netzwerkverbindung kommunizieren. Die Klasse empfängt Befehle von der ergänzten Device-Klasse und kümmert sich komplett um den Aufbau der Verbindung, sowie der Kommunikation mit der Datenquelle. Eine ihrer Hauptaufgaben neben der Netzwerkkommunikation, ist die Verarbeitung der eingehenden Datenpakete. Die erweiterte Klasse analysiert den Paketinhalt, speichert die Nutzdaten und macht diese der Device-Klasse zur weiteren Verarbeitung zugänglich. Die angepasste Hardware-Klasse wendet außerdem Filtertechniken auf die eingehenden Pakete an, um verspätete Nutzdaten direkt zu ignorieren. Sollten Fehler oder ähnliche Ereignisse auftreten, die den Benutzer interessieren könnten, kümmert sich die ergänzte Hardware-Klasse um die Protokollierung dieser.

Abbildung 4.2: Vereinfachtes Klassendiagramm des MotionBuilder Plugins. Die zentrale Klasse ist das *Device*, welches über die *Hardware*-Klasse mit den Datenquellen kommuniziert. Die Oberfläche verwaltet die *Layout*-Klasse.

Die *Layout*-Klasse des Plugins übernimmt die Interaktion mit dem Anwender. Die Klasse beinhaltet den kompletten Code zur Erzeugung der Benutzeroberfläche und reagiert auf Eingaben des Benutzers. Änderungen an den Einstellungen des Device werden an die Device-Klasse weitergegeben. Die Layout-Klasse hält eine Referenz auf die Device-Klasse. Die Device-Klasse wiederum hält eine Referenz auf die Hardware-Klasse. So ist die indirekte Kommunikation zwischen den einzelnen Klassen möglich, wobei die *Layout*-Klasse an oberster Stelle in der Hierarchie steht.

4.1.5 Protokoll

Für die Kommunikation zwischen MotionBuilder Plugin und externen Datenquellen, muss ein Kommunikationsprotokoll ausgearbeitet werden (siehe Abbildung 4.3), dass es ermöglicht die übertragenen Nachrichten beidseitig einzuordnen und zu verarbeiten. Das Protokoll soll dabei möglichst einfach gehalten werden, damit die Implementierung bei der Inbetriebnahme einer neuen Datenquelle schnell umgesetzt werden kann.

Gewählt werden vier *Nachrichtenkennungen* auf Integer-Basis, die dem Empfänger signalisieren um welche Art von Paket es sich handelt. Die Kennung ist Teil jedes Pakets und wird vom Empfänger als erstes Element der Nachricht gelesen. Basierend auf der Kennung werden entsprechende Methoden aufgerufen, die zur Verarbeitung der restlichen Daten zuständig sind. Jedes Paket wird zudem mit einer *Nachrichtenummer* versehen, über die sich die Daten chronologisch einordnen lassen. Diese beiden Zahlen bilden den Nachrichtenkopf (Header).

Abschließend werden, sofern vorhanden, nachrichtenspezifische Daten an das Paket angehängt. Bei den Paketen zum Starten und Stoppen des Datenfluss entfallen diese, da diese Pakete lediglich als Signal interpretiert werden und schon anhand ihrer Kennung identifiziert werden. Die Pakete die zusätzliche Daten nutzen, sind der Strukturdialog (Strukturanfrage / -antwort), sowie das *Nutzdatenpaket* der Datenquelle.

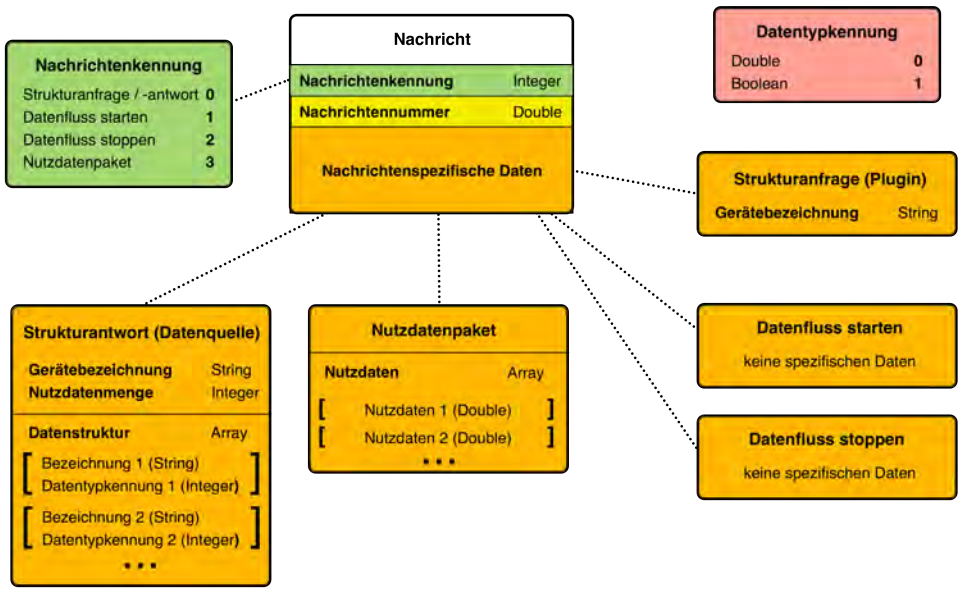


Abbildung 4.3: Aufbau der Datenpakete (Nachrichten) nach dem festgelegten Protokoll für die Kommunikation zwischen MotionBuilder-Plugin und Datenquelle.

Die Strukturanfrage wird vom Plugin mit der Bezeichnung der MotionBuilder-Instanz als nachrichtenspezifische Daten im Paket gesendet. Der Server (Datenquelle) antwortet mit einem Strukturpaket (Strukturantwort), dass die Datenstruktur der Nutzpakete in Form eines Arrays und die Anzahl der Daten als Integerzahl enthält. Dabei wird dem gemischten Array abwechselnd eine Datenbeschreibung (Zeichenkette bzw. *String*) und eine Datentypkennung (Integer) hinzugefügt. Die Datentypkennung ist für das Plugin besonders wichtig, da alle Daten an MotionBuilder in Form von Double-Werten übertragen werden, aber von der Software unterschiedlich interpretiert werden müssen. Ohne diese Zusatzinformation wäre die korrekte Zuordnung eines Datentyps nicht möglich.

Die Beschreibung wird vom Plugin-Prototypen benutzt um die einzelnen Datensätze, für den Anwender mit einer eindeutigen Bezeichnung (z.B. „Position Handfläche X“, „iPad Pitch-Winkel“) zu versehen. Dem Anwender sind dabei theoretisch keine Grenzen gesetzt, wie viele Daten dem Plugin übermittelt werden sollen. Es ist jedoch zu beachten, dass jeder übermittelte Wert die Datenverarbeitung beim Empfänger verlängert.

Nach erfolgreicher Übertragung der Datenstruktur und entsprechender Anfrage seitens des Plugins, übermittelt der Server den Datenfluss in Form von Nutzdatenpaketen. Ein Nutzdatenpaket enthält dabei genau ein *Sample* der Datenquelle. Das Nutzdaten-Array der Pakete muss den identischen Aufbau haben, wie zuvor in der Strukturanfrage spezifiziert.

4.1.6 Netzwerkkommunikation

Die Netzwerkkommunikation ist eine der wichtigsten Aufgaben des Plugins. Für den Aufbau der Netzwerkschnittstelle wurde das TCP/IP Protokoll (siehe Kapitel 3.5) gewählt. Datagramme (UDP) bieten geringe Übertragungszeiten und sind damit besonders gut für zeitkritische Anwendungen geeignet. Jedoch sind sie bei der Übertragung von großen Datenpaketen, ab circa 1500 Bytes, sehr unzuverlässig. Sowohl UDP-, als auch TCP-Pakete, werden bei der Übertragung automatisch fragmentiert, wenn sie eine vom Betriebssystem festgelegte Größe (*Maximum Transmition Unit*, MTU) überschreiten.

TCP wird UDP letztendlich vorgezogen, da es sicherstellt, dass alle Datenfragmente korrekt am Empfänger ankommen und das gesamte Datenpaket nicht beim Verlust eines einzigen Fragments verloren geht, wie es bei UDP-Paketen der Fall ist. Die höhere Latenz der TCP-Pakete, wird durch einen Filter in der Paketverarbeitung kompensiert. Die gesamte Kommunikation zwischen Plugin und Datenquelle, erfolgt über Stream-Sockets (TCP-Sockets). Diese werden so konfiguriert, dass sie die Ausführung der Software nicht blockieren (*Non-blocking*).

Die Socket-Verbindung wird von der *Device*-Komponente (siehe Kapitel 4.1.3) des Plugins aufgebaut und verwaltet. Eine Instanz des Device kann sich mit lediglich einer Datenquelle verbinden. Die Anzahl der möglichen Plugin-Devices ist dabei jedoch nicht begrenzt, sodass die maximale Anzahl parallel aktiver Netzwerkschnittstellen lediglich durch die Hardware des Systems und MotionBuilder selbst eingeschränkt wird. Jedes Device agiert als Client, während die zu verbindende Datenquelle den wartenden Server-Socket implementiert (siehe Abbildung 4.4).

Zum Aufbau der Verbindung öffnet das Device einen neuen Stream-Socket und sendet eine Verbindungsanfrage an die in den Einstellungen (Benutzeroberfläche) hinterlegte IP-Adresse. Der wartende Server-Socket der Datenquelle akzeptiert die Verbindungsanfrage und speichert den Socket des Device zur Kommunikation. Das Device sendet jetzt eine Strukturanfrage über das nun aktive Stream-Socket-Paar an die Datenquelle. Diese bereitet eine Antwort auf die Anfrage vor und sendet die eigene Nutzdatenstruktur über das Socket-Paar an das Device.

Das Device erkennt anhand des im Paket gesetzten Nachrichtentyps, worum es sich bei den Daten handelt und überprüft die Struktur anhand der Anzahl vorliegender Datensätze im Paket. Stimmt die Struktur mit der übermittelten Anzahl überein, werden auf Basis der Struktur-Datensätze neue Animationsknoten in MotionBuilder für Datensätze angelegt und diese mit Nullen initialisiert. Letzteres dient der Vermeidung von Programmabstürzen, durch ungültige Zugriffe auf leere Datensätze. Das Device ist nun bereit Daten von der Daten-

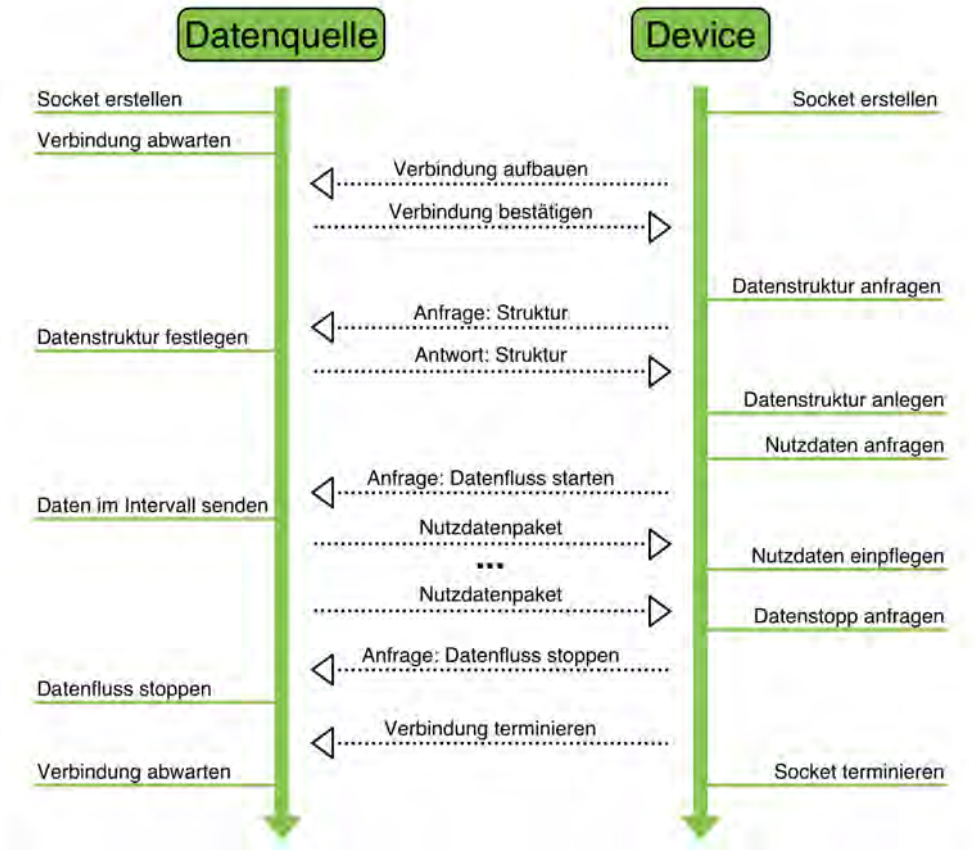


Abbildung 4.4: Ablauf der Netzwerkkommunikation zwischen Device und Datenquelle.

quelle zu empfangen und sendet eine entsprechende Anfrage. Ist die Quelle bereit für den Datenversand, beginnt diese damit in einem festgelegten Intervall (Samplerate) Nutzdaten über den Stream-Socket an das Device zu senden. Die Samplerate ist nicht Teil des Protokolls und muss sowohl in der Benutzeroberfläche des Device, als auch an der Datenquelle definiert werden. Ein Datenpaket der Quelle enthält genau einen Datensatz, der sich anhand der Nachrichtennummer chronologisch im Device einordnen lässt und den gleichen Aufbau einhält, wie zuvor in der Strukturantwort definiert. Bei jedem Lesevorgang, ausgelöst im Intervall der Samplerate, wird genau ein Datenpaket verarbeitet und der Rest des Puffers geleert. So wird ein Datenstau und damit Latenzen durch stark verspätete Pakete vermieden.

Um die Verarbeitung von fehlerhaften Nachrichten zu verhindern, wird jedes eingehende Datenpaket vom Device auf Inhalt geprüft. Ist die Datenstruktur fehlerhaft und unlesbar, wird das Paket ignoriert. Eine zusätzliche Filterfunktion entscheidet anhand der Nachrichtennummer und dem in den Device-Einstellungen definierten Grenzwert, ob das eingegangene Datenpaket noch gültig ist. Abhängig davon wird es entweder für die weitere Verwendung gespeichert oder direkt verworfen. Dieser Filter bietet zusätzliche Sicherheit vor Verarbeitungsfehlern, auch wenn die typische TCP-Konfiguration bereits sicherstellen sollte, dass

die Datenpakete am Socket, in der korrekten Reihenfolge vorliegen. Die Umsetzung der Filterfunktion ist daher besonders im Hinblick auf zukünftige Unterstützung von anderen Protokollen, wie z.B. UDP, empfehlenswert.

Initiiert der Anwender die Offline-Schaltung des Device in MotionBuilder, sendet dieses ein letztes Paket an die Datenquelle, dass den Stopp des Datenflusses signalisiert. Anschließend wird die Socket-Verbindung terminiert und die Datenquelle wartet auf neue, eingehende Verbindungen. Wird das Device nun erneut gestartet, läuft der Verbindungsaufbau erst einmal identisch ab. Nachdem die Datenquelle die Struktur an das Device gesendet hat, wird diese jedoch nicht erneut eingepflegt, sondern lediglich mit der bereits vorhandenen Struktur über die Anzahl der Datensätze verglichen. Stimmt die Anzahl der im Device vorhandenen Struktur mit der Anzahl der Daten in der Strukturnachricht überein, wird die Struktur als identisch angesehen und das Device fährt mit der Nutzdatenanfrage fort. Stimmt die Anzahl der Datensätze hingegen nicht, lehnt das Device die eingehende Struktur ab, gibt einen Fehler an den Anwender aus und wartet auf neue Nachrichten von der Datenquelle, die eine passende Datenstruktur liefern. Für den Betrieb des Device mit einer anderen Datenquelle, muss daher eine weitere Instanz des Device erzeugt und mit der neuen Datenquelle verbunden werden.

4.1.7 Plattformabhängigkeit

Da das Plugin mit unterschiedlichen Netzwerkumgebungen und Datenquellen funktionieren soll, gibt es Problematiken in der Netzwerkkommunikation, die adressiert werden müssen um typische Fehler zu vermeiden. Eine typische Fehlerquelle bei der Übertragung von Daten ist die unterschiedliche Datenstruktur verschiedener Betriebssysteme und Systemarchitekturen. Bei der Kommunikation zwischen unterschiedlichen Systemen, auch *Cross-Platform-Kommunikation* genannt, kann es zu Inkompatibilitäten und damit Problemen kommen.

Cross-Platform-Datenaustausch wird problematisch, sobald Zahlenwerte mit ins Spiel kommen. Während die Übertragung von Zeichenketten durch Enkodierungen wie *Unicode* (UTF-8, UTF-16) meist unproblematisch ist, sind Gleitkommazahlen und Integer-Werte weiterhin kritisch. Grund dafür sind die unterschiedlichen Methoden der Speicherreservierung von Compilern. So reservieren alte, 32-Bit basierte Systeme beispielsweise vier Bytes für ein `int` (Ganze Zahl) und auch heutige 64-Bit Systeme halten noch an dieser Konvention fest. Unterschiede gibt es jedoch bei anderen primitiven Datentypen wie `long`: Während Linux und Mac OS Systeme mit 64-Bit Architektur einem long-Wert jeweils 64 Bit Speicher reservieren, spricht ein 64-Bit Windows einem `long` lediglich 32 Bit Speicher zu. Durch die unterschiedliche Repräsentation des long auf binärer Ebene, lässt sich dieser auch nicht problemlos zwischen den Systemen übertragen.

Ein weiteres Problem ist die Byte-Reihenfolge (*Endianess*) von ganzzahligen Werten im Arbeitsspeicher. Ganzzahlige Werte wurden bei alten Rechnerarchitekturen mit dem höchstwertigen Bit an erster Stelle (ähnlich der Dezimalnotation) in den Speicher geschrieben. Da der Prozessor jedoch schneller mit den Rechenvorgängen beginnen kann, wenn das erste Bit

einer Zahl zeitgleich auch das niedrigwertigste Bit ist, wurde diese Reihenfolge umgedreht. Die Problematik dieser Entscheidung liegt auf der Hand: Alte Systeme interpretieren die Integer-Bits in einer anderen Reihenfolge als moderne Systeme. Als Resultat muss Software die Bitreihenfolge der kommunizierenden Architekturen stets berücksichtigen, falls Integer-Daten über das Netzwerk übertragen werden.

Um diese Probleme zu vermeiden, gibt es Bibliotheken, die sich auf die Enkodierung und Serialisierung von Daten spezialisiert haben. Sie kommen sowohl beim Empfänger, als auch beim Sender zum Einsatz und übernehmen die korrekte Enkodierung und Interpretation der Datenströme. Das nächste Kapitel vergleicht zwei der beliebtesten Formate und begründet die Wahl für die Entwicklung dieses Plugins.

4.1.8 Vergleich: JSON und MessagePack

Die Übertragung von Daten über eine Netzwerkverbindung ist nicht trivial. Einer der schwierigsten Aspekte ist die Enkodierung zur effizienten und korrekten Übertragung der Daten, insbesondere dann, wenn die vorliegenden Daten nicht in Form von Zeichenketten oder Bytes formatiert sind. Im Verlauf des letzten Jahrzehnts hat sich das *JSON*-Format[3] (JavaScript Object Notation) als beliebte Lösung für dieses Problem etabliert. Das Format ist unkompliziert aufgebaut und arbeitet mit einem String-basierten assoziativen Array (*Key-Value*) um Objekte für die Übertragung zu kodieren. Ein großer Vorteil des JSON-Formats ist dabei seine Lesbarkeit. Da sämtliche Objekte in Form von Zeichenketten abgelegt werden, lassen sich diese auch nach der Enkodierung noch leicht von Programmierern ohne zusätzliche Werkzeuge lesen. Der String-basierte Ansatz macht JSON zudem sehr robust gegenüber Problemen die auftreten, sobald eine Übertragung zwischen unterschiedlichen Systemarchitekturen erfolgt.

Da JSON sämtliche Zahlen in Form einer Zeichenkette enkodiert, fallen auch die in Kapitel 4.1.7 beschriebenen Probleme mit unterschiedlichen Wortlängen und Byte-Reihenfolgen weg. Zeitgleich wird bei der JSON-Kodierung jedoch auch deutlich mehr Speicher benötigt, da jede Stelle und Nachkommastelle einer Zahl als eigenes Zeichen kodiert wird. Beispiel: Die Zahl „2147483647“ benötigt als Integer-Wert lediglich vier Byte Speicherplatz. Speichert man sie jedoch als Zeichenkette sind es zehn Byte. Die benötigte Speichermenge im String-Format ist also mehr als doppelt so groß. Auch die Dekodierung aus dem Zeichenkettenformat hat ihre Nachteile, da die besonders bei Gleitkommazahlen vergleichsweise aufwändige Prozeduren notwendig sind um den Integer-String zurück in eine Zahl zu konvertieren. Diese Ineffizienz hinsichtlich Speicher und Geschwindigkeit, lässt das JSON-Format daher als Kandidat für den Plugin-Prototypen ausscheiden.

[3] http://www.json.org (Zuletzt abgerufen: 16.01.2016)

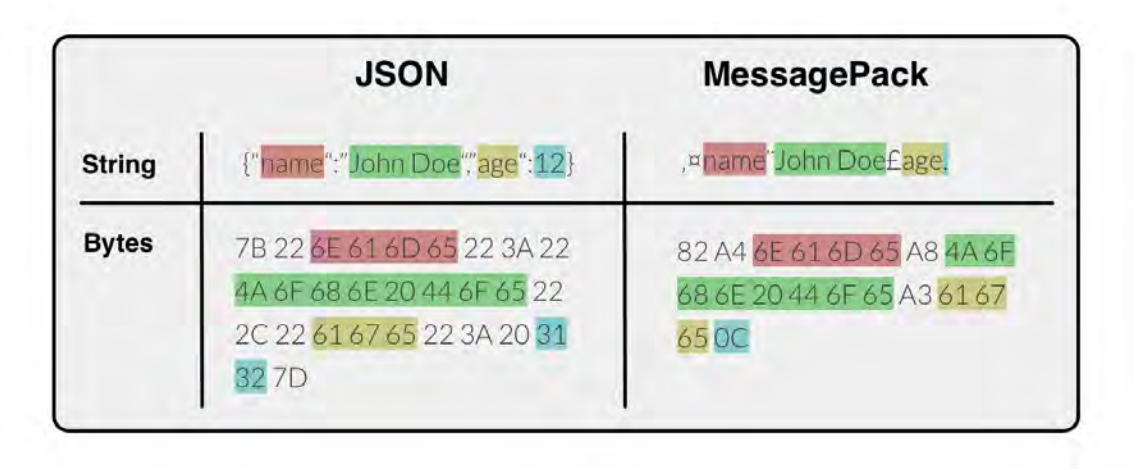


Abbildung 4.5: Vergleich zwischen einem mit JSON- und MessagePack enkodiertem Array. Das JSON-enkodierte Array ist 29 Bytes groß, wovon lediglich 17 Bytes Nutzdaten sind. Das MessagePack-enkodierte Array ist 20 Bytes groß, wovon 16 Bytes Nutzdaten enthalten.

Quelle: http://bit.ly/1Px6gp9 (Olaf Horstmann, modifizierte Abbildung, Zuletzt abgerufen: 25.01.2016)

Eine Alternative bietet das *MessagePack*-Format[4]. MessagePack verfolgt einen ähnlichen Ansatz wie JSON, setzt jedoch auf eine Serialisierung (binäre Repräsentation) der Objekte, anstatt diese in eine lesbare Zeichenkette zu konvertieren. Dies hat gleich mehrere Vorteile: Durch die binäre Darstellung der Objekte wird insbesondere für Zahlenwerte deutlich weniger Speicherplatz benötigt. Mit dem Verzicht auf eine lesbare Syntax spart sich MessagePack String-Trennzeichen, welche die JSON-Daten zusätzlich vergrößern. Noch wichtiger ist jedoch, dass die kleinere Datenmenge auch eine schnellere Dekodierung der Daten bedeutet. Bei einer Performance-kritischen Anwendung wie einem Realtime-Plugin ist dieser Punkt enorm wichtig, da das Zeitfenster für die Evaluation der Daten sehr gering ist und aufwändige Prozeduren zu Performanceproblemen führen können. Weiterhin kümmert sich MessagePack für den Programmierer um die zuvor beschriebenen Architektur-Problematiken und serialisiert Zahlenwerte in einem einheitlichen Format, sodass keine Probleme bei der Deserialisierung seitens des Empfängers auftreten.

Ebenso wie JSON, wird auch MessagePack dezentralisiert weiterentwickelt. Zwar gibt es eine allgemeine Spezifikation für den Aufbau des Formats, doch die einzelnen Implementierungen des Konzepts für unterschiedliche Programmiersprachen werden von unabhängigen Teams entwickelt. Über 50 Implementierungen des Formats (Stand: 16.01.2016) listet die Projekte Webseite[5], sodass es bereits für jede gängige Programmiersprache eine Implementierung geben sollte. Dies ist ein wichtiger Punkt, der für die Wahl des MessagePack Formats bei der Implementierung des Prototypen spricht. Das breite Spektrum an unterstützen Sprachen stellt sicher, dass der Aspekt der universellen Verwendbarkeit des Plugins nicht von mangeln-

[4]http://msgpack.org (Zuletzt abgerufen: 17.01.2016)
[5]http://msgpack.org/#languages (Zuletzt abgerufen: 17.01.2016)

der Programmiersprachen-Unterstützung untergraben wird. Das von Nicolas Fraser gepflegte Github-Projekt *Schemaless Benchmarks*[6] bietet ausführliche Performance-Vergleiche zwischen C / C++ Implementierungen von MessagePack und JSON. MessagePack-Varianten zeigten in den verschiedenen Tests fast durchgehend deutlich bessere Ergebnisse als vergleichbare JSON-Implementierungen (siehe Abbildung 4.6). Diese Ergebnisse und die zuvor aufgeführten Gründe sind entscheidend für die Nutzung von MessagePack als Serialisierungslösung für die Datenpakete.

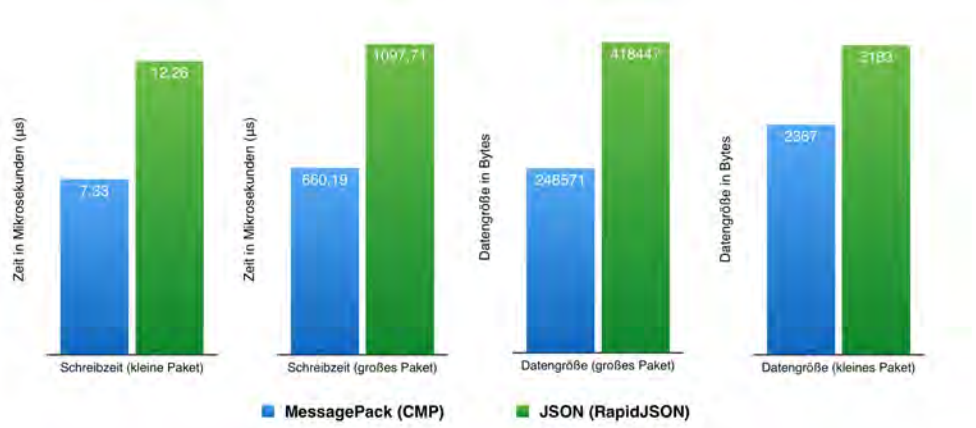


Abbildung 4.6: Benchmark Vergleich zwischen den den Implementierungen CMP (MessagePack) und RapidJSON (JSON), basierend auf Testergebnissen des *Schemaless Benchmarks* Projekts von Nicolas Fraser.

Quelle: http://bit.ly/1KjnEMT (Basiert auf Daten von Nicolas Fraser, Zuletzt abgerufen: 16.01.2016)

4.1.9 Grafische Benutzeroberfläche

Die Bedienung der Netzwerkschnittstelle ist in zwei Teile aufgeteilt: Die grafische Oberfläche des Device und das Beziehungsnetz des *Relation Constraint*, welche beide über den *Szenengraph* in MotionBuilder aufgerufen werden. Dieses Kapitel konzentriert sich auf die Konzeption der Benutzeroberfläche des Device. Das Relation Constraint stellt dem Anwender einen eigenen grafische Editor zur Verfügung, der bereits Teil der MotionBuilder-Software ist und im nächsten Kapitel genauer besprochen wird.

Für die Konfiguration der Netzwerkverbindung und den Device-spezifischen Einstellungen stellt das Device eine eigene Benutzeroberfläche zur Verfügung (siehe Abbildung 4.7). Diese wird angezeigt sobald der Anwender eine Instanz des Device im Szenengraph auswählt, die zuvor per „Drag'n'Drop" aus dem *Asset-Browser* erstellt hat. Die Oberfläche des Device gliedert sich in zwei Teile: Die linke Hälfte der Benutzeroberfläche enthält mehrere Bedienelemente zur Steuerung des Device und ist für alle Devices in MotionBuilder standardisiert. Drei Schalter kontrollieren den Online-, Echtzeit- und Aufnahmestatus des Device.

[6]https://github.com/ludocode/schemaless-benchmarks (Zuletzt abgerufen: 17.01.2016)

Über die Dropdown-Box *Model-Binding*, ist es möglich, die Animationsknoten eines Device, automatisch mit einem bestehenden Model oder Rig zu verknüpfen. Für Devices die diese Funktion nicht unterstützen, wie auch der hier besprochene Plugin-Prototyp, ist diese Option in der Benutzeroberfläche deaktiviert. Unten links befindet sich zudem eine Infobox, die das Plugin nutzen kann um den Benutzer über Statusänderungen und Fehler aufmerksam zu machen.

Die rechte Hälfte der Device-Benutzeroberfläche ist nicht standardisiert und basiert auf OR SDK Bedienelementen und dem Programmcode des Plugins. Devices nutzen diesen Teil der Benutzeroberfläche um dem Anwender Device-spezifische Bedienelemente und Einstellungsmöglichkeiten bereitzustellen. Die grafische Oberfläche ist einfach strukturiert um dem Anwender alle wichtigen Bedienelemente möglichst leicht zugänglich zu machen.

Da der Prototyp lediglich acht Bedienelemente zur Anpassung der Device-Einstellungen bereitstellt, wurde ein Layout basierend auf einem einzigen *View* (Ansicht) entworfen (siehe Abbildung 4.7). Eine Kartenreiter-Struktur mit mehreren Views, wie sie vom gewählten Device-Template (*OR Device Template*) genutzt wird, ist für diesen Anwendungsfall zu komplex und würde die Bedienung des Device erschweren. Um den Bedienelementen dennoch übersichtlich zu präsentieren, werden diese in mehrere logische Kategorien aufgeteilt und in Blöcken angeordnet. So werden beispielsweise alle Verbindungsparameter zu einem Block zusammengefasst, sodass diese sich als Einheit von den restlichen Bedienelementen abheben. Die zugehörigen Labels (Beschriftungen) werden einheitlich links platziert, um eine klare Trennung zwischen den Bedienelementen und deren Beschriftungen zu schaffen. Die Typen der Bedienelemente werden basierend auf üblichen Software-Konventionen gewählt: Für die Eingabe von frei wählbaren Werten (z.B. IP-Adresse) kommen flexible Textfelder zum Einsatz, während Dropdown-Listen mehrere Auswahlmöglichkeiten (z.B. Sample-Verfahren) kompakt präsentieren.

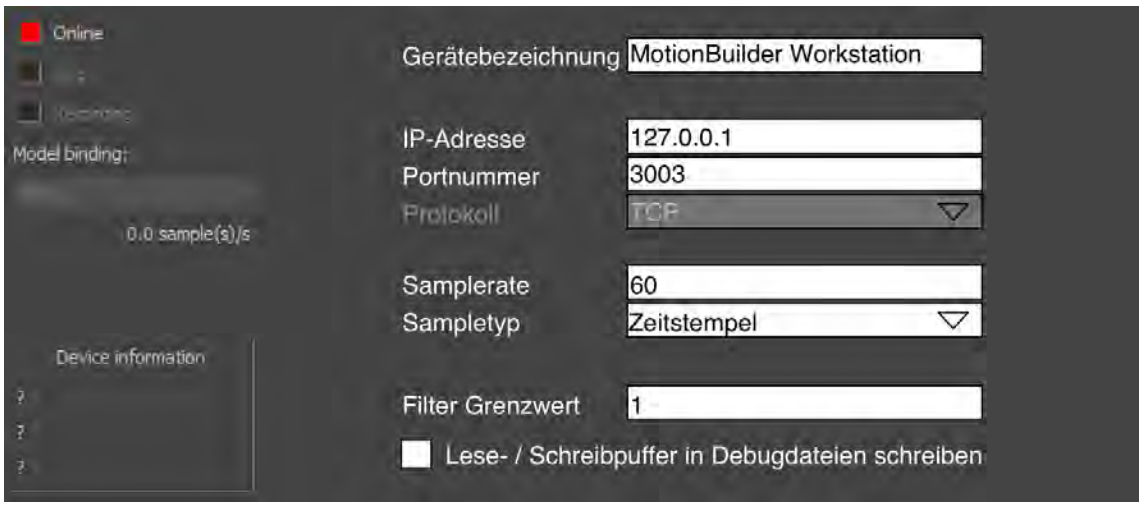


Abbildung 4.7: Entwurf der grafischen Benutzeroberfläche für das Plugin-Device.

Das erste Textfeld der Oberfläche enthält die Beschreibung für die Instanz des Device (z.B. „Mobu iPad Device“), die der Datenquelle zusammen mit der Strukturanfrage übermittelt wird und es ermöglicht, das Device eindeutig zu identifizieren, sollten mehrere Instanzen des Plugins parallel im Einsatz sein. Über drei weitere Bedienelemente kann der Anwender die Kommunikationseinstellungen anpassen. Dazu gehört die IP-Adresse, sowie die Portnummer der zu kontaktierenden Datenquelle, welche jeweils über ein Textfeld definiert werden können. Ein drittes Bedienelement soll zur späteren Weiterentwicklung des Prototypen umgesetzt werden und ermöglicht dem Nutzer die Wahl des Netzwerkprotokolls. Da im Rahmen dieser Arbeit jedoch lediglich ein Prototyp auf TCP-Basis entwickelt wird, bleibt dieses Bedienelement für den Benutzer vorerst deaktiviert.

Über ein weiteres Textfeld lässt sich die Samplerate des Device festlegen. Diese sollte mit der Samplerate der genutzten Datenquelle übereinstimmen, kann jedoch unabhängig davon festgelegt werden. Eine Dropdown-Liste ermöglicht zudem die Auswahl des Sample-Verfahrens in MotionBuilder. Das Sample-Verfahren bestimmt die Art und Weise, wie die einzelnen Datenpakete bei einer Aufnahme als Keyframe gespeichert werden (siehe Kapitel 5.1.7). Der Grenzwert des Nachrichtenfilters lässt sich ebenfalls über eine Textbox festlegen, deren Wert während einer laufenden Datenübertragung angepasst werden kann. Eine Checkbox ermöglicht dem Benutzer außerdem eine zusätzliche Debug-Funktion zu aktivieren, die sämtliche Inhalte des Lese- und Schreibpuffers der Socketverbindung in eine Datei protokolliert. Standardmäßig ist die Checkbox und damit auch die Funktion deaktiviert.

4.1.10 Relation Constraint

Eines der interessantesten Features von Autodesk MotionBuilder ist das *Relation Constraint*. Dabei handelt es sich um ein kausales Netzwerk aus Boxen (Knotenpunkte). Die Boxen in einem Relation Constraint können sowohl MotionBuilder Objekte, als auch mathematische Funktionen, Devices oder gar andere Relation Constraints in Form von *Makros*, repräsentieren. Die Boxen verfügen über Aus- und Eingänge, die genutzt werden können um Daten aus Objekten auszulesen oder einzuspeisen. Der Anwender kann wahlweise manuell oder per Programmcode, Beziehungen zwischen diesen Ein- und Ausgängen definieren, um Boxattribute abhängig von den ausgehenden Attributen einer anderen Box zu machen.

Boxen mit Ausgängen werden als *Sender* bezeichnet, Boxen mit Eingängen nennen sich *Receiver* (dt. „Empfänger“). Boxen die sowohl Ein- als Ausgang besitzen, werden als *Operator* bezeichnet. Während jedes MotionBuilder Objekt mit animierbaren Attributen sowohl als Sender, als auch Empfänger eingesetzt werden kann, handelt es sich bei Operatoren um Boxen die eingesetzt werden um mathematische Funktionen, Typ-Konvertierungen oder Vergleiche durchzuführen. Sie können im Netzwerk an eine beliebige Stelle zwischen Sender und Empfänger gesetzt werden.

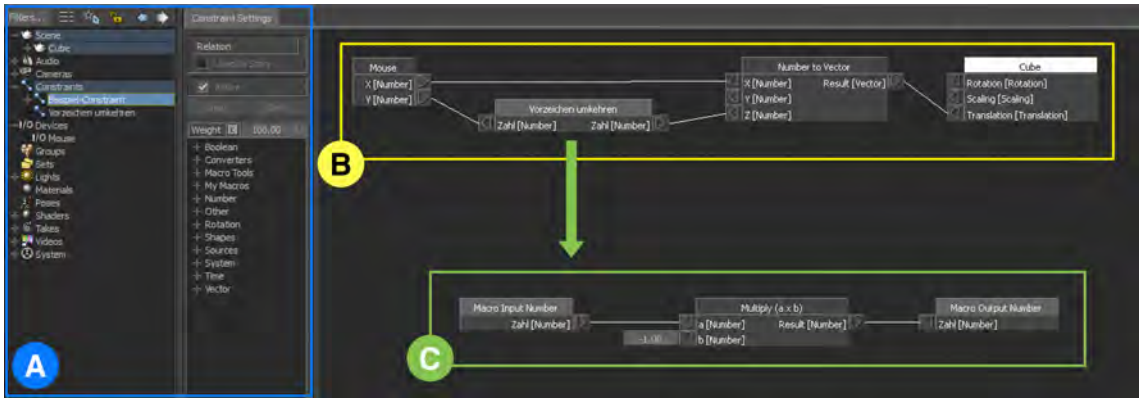


Abbildung 4.8: Screenshot des Relation Constraint-Editors mit dem Szenengraph (A, links) und den Constraint-Einstellungen (A, rechts). Rechts im Bild: Ein Beispiel-Constraint (B) mit einem Maus-Device als Sender, einem Cube als Empfänger, einem Operator und einem Makro (C) basierend auf einem zweiten Relation Constraint.

Der Einsatz von Operatoren macht das Relation Constraint zu einem leistungsstarken Werkzeug, da sie sehr schnell zu komplexen Strukturen zusammengesetzt werden können und so Programmlogik ersetzen, die sonst in Form eines Plugins implementiert werden müsste. Relation Constraints können zudem als *Makro* (Anweisungsfolge) in anderen Relation Constraint Instanzen agieren. Dazu werden dem Constraint spezielle Input- / Output-Boxen hinzugefügt, die sich in der Makro-Box des Constraints als entsprechende Ein- bzw. Ausgänge präsentieren.

Abbildung 4.8 zeigt den Relation Constraint-Editor mit einem geöffneten Beispiel zur Echtzeit-Steuerung einer Objekt-Translation per Mauscursor. Alle Inhalte der MotionBuilder-Szene, darunter auch Constraints und Devices, lassen sich über den *Szenengraph* aufrufen. Ist ein Constraint im Graph ausgewählt, werden daneben die verfügbaren Constraint-Einstellungen und Boxen angezeigt (siehe Abbildung 4.8-A). Rechts neben den Einstellungen stellt MotionBuilder das ausgewählte Beispiel-Constraint im Editor dar. Wie in Abbildung 4.8-B zu erkennen ist, nutzt das Beispiel-Constraint die Koordinaten des Maus-Device (Sender) um die Translation eines Würfels (Empfänger) anzutreiben. Das Vorzeichen der Y-Komponente des Device wird mithilfe eines zweiten Constraints, das als Makro fungiert, umgekehrt (siehe Abbildung 4.8-C). Ein „Number to Vector“-Operator setzt die einzelnen Zahlenwerte zu einem Vektor zusammen, der dem Cube übergeben wird.

Makros und Operatoren machen das Relation Constraint nicht nur zu einem geeigneten Werkzeug für den schnellen und interaktiven Prototypen-Bau, sondern auch zur idealen Ergänzung für die Schnittstellen-Funktionalität des Streaming-Plugins:

Mit der Erstellung einer Instanz des Plugin-Device durch den Anwender, legt der Prototyp ein neues Relation Constraint an und fügt das eigene Device als Sender hinzu. Bis zur Einpflege der Struktur der Datenquelle nach Verbindungsaufbau bleibt die Device-Box „leer“,

da noch keine Ausgänge angelegt wurden. Für jeden Datensatz, den die Quelle übermittelt, wird beim Strukturaufbau dann ein Animationsknoten angelegt, der im Relation Constraint einen neuen Ausgang darstellt. Eingehende Nutzdatenpakete werden dann vom Device direkt über die Ausgänge der Device-Box zugänglich gemacht und können mit Operatoren verarbeitet oder direkt mit einem anderen Objekt verknüpft werden. Dies ermöglicht die gesamte Dateninterpretation, in das Relation Constraint auszulagern, was die Datenquelle entlastet und dem Anwender zusätzliche Kontrolle, selbst im aktiven Betrieb der Schnittstelle, verleiht.

Dank der grafischen Repräsentation der Nutzdaten ist es auch Anwendern ohne jegliche Programmierkenntnisse möglich, mittels „Drag'n'Drop“, Programmlogik zu deren Verarbeitung und Interpretation aufzusetzen. Dieser Punkt ist enorm wichtig, da der typische MotionBuilder Anwender im Regelfall nur unzureichende Kenntnisse auf dem Gebiet der Programmierung besitzt.

4.2 Anwendungsbeispiel: iPad Kamerasteuerung

Als Demonstration für eine Virtual Cinematography Anwendung, die der Plugin-Prototyp mit nur wenig zusätzlichem Programmieraufwand ermöglicht, soll im Rahmen der Arbeit eine Tablet-App entwickelt werden, die als Datenquelle mit der Netzwerkschnittszelle des Plugins kommuniziert. Dieses Kapitel verdeutlicht die Konzeption der mobilen Anwendung und die Rahmenbedingungen der Entwicklung.

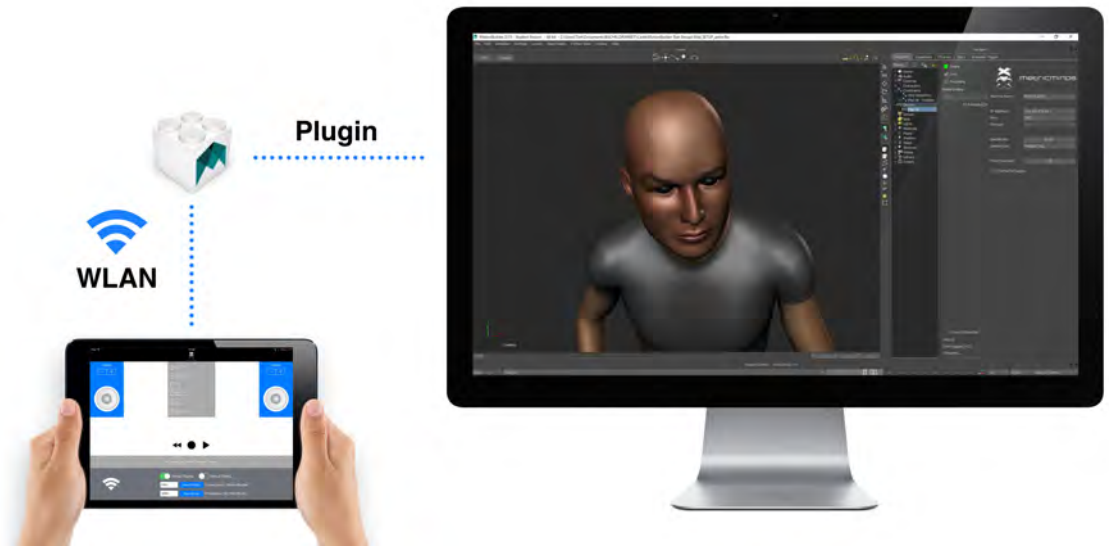


Abbildung 4.9: Konzept der Kamerasteuerung in MotionBuilder mithilfe der Inertialsensoren eines Apple iPad. Die Verbindung der Datenquelle (iPad) mit der Netzwerkschnittstelle des Plugins erfolgt über Wireless LAN.

Quelle: http://bit.ly/1X1VxIG (Fernando Cruz, modifiziert; Zuletzt abgerufen: 05.03.2016)
http://bit.ly/21MUnHH (Apple Inc., modifiziert; Zuletzt abgerufen: 05.03.2016)

4.2.1 Zielsetzung

Die iPad Anwendung soll eine natürliche Kamerasteuerung für MotionBuilder verwirklichen, indem sie die Daten der Inertialsensoren des Tablets (siehe Kapitel 3.3) nutzt, um ein Kamera-Objekt in der Szene mit authentischen Rotationsdaten zu versorgen (siehe Abbildung 4.9). Das Beispiel siedelt sich damit im Bereich der Virtual Cinematography an, die ein Teilgebiet der Virtual Production ist. Ein möglicher Anwendungsbereich der App sind Pre- und Post-Production. Sie kann verwendet werden um schnell natürliche Kamerabewegungen aufzuzeichnen ohne dafür ein virtuelles Kamera-System zu benötigen, dass im Regelfall ein aktives Motion Capture-System voraussetzt. Weiterhin sollen die übermittelten Daten mit wenig Arbeitsaufwand im Relation Constraint, auch andere Szenenobjekte antreiben, sodass die Anwendung vielfältig einsetzbar ist.

Zur Steuerung der Kamerarotation werden sowohl Beschleunigungs- als auch Gyrosensoren des Tablets angesteuert, um bestmögliche Ergebnisse zu erzielen. Die Umsetzung erfolgt mithilfe des iOS SDK und einem Apple iPad. Das SDK bietet gute Programmierschnittstellen zum Abruf aufbereiteter Sensorendaten an, sodass weniger Zeit mit dem Finetuning der Werte verbracht werden muss. Da über die verbauten Sensoren jedoch keine Positionsbestimmung im Raum möglich ist, soll die Steuerung der Translation mithilfe von zusätzlichen Touchscreen-Bedienelementen vorgenommen werden. Diese liefern zwar keine natürlich-authentischen Animationsdaten, bieten dem Anwender jedoch zumindest eine rudimentäre Steuerung der Translation.

Die gewonnenen Daten werden über eine WiFi-Verbindung, vorzugsweise über ein Ad-hoc-Netz (direkte Geräteverbindung ohne Router), an das MotionBuilder Plugin übertragen. Fehler und Latenzen müssen sich in einem Rahmen bewegen, der produktive Arbeit mit der App ermöglicht, ohne das viele Fehlerkorrekturen in MotionBuilder notwendig sind. Die Anwendung muss weiterhin eine einstellbare Samplerate für die Übertragung der Daten anbieten, sodass sie schnell an unterschiedliche Produktionsbedingungen anpassbar ist. Zur Fehlersuche im aktiven Betrieb der App ist ein Fehlerprotokoll zu implementieren. Die App muss auf der neusten iPad Generation (iPad Air 2) lauffähig sein und nach Möglichkeit auch ältere Modelle unterstützen.

4.2.2 Anwendungsstruktur

Die Anwendungsstruktur des iPad Anwendungsbeispiels ist unkompliziert gehalten und basiert auf dem Model-View-Controller Entwurfsmuster (siehe Abbildung 4.10). Dieses trennt Daten (*Model*), Benutzeroberfläche (*View*) und Programmlogik (*Controller*) voneinander. Die Anwendung besteht lediglich aus einem View, der die gesamte Benutzeroberfläche mit allen Bedienelementen beinhaltet. Die Programmlogik wird komplett in einer View-Controller Klasse implementiert, die auch die Daten (Model) hält. Der View-Controller hat Zugriff auf den View und implementiert auch sämtliche Logik hinter der Benutzeroberfläche. Zuzüglich zum View-Controller wird auf eine Reihe von Open Source Frameworks zurückgegriffen, welche die Logik für die Socketprogrammierung und weitere Programmteile liefern (siehe Kapitel 5.2.1).

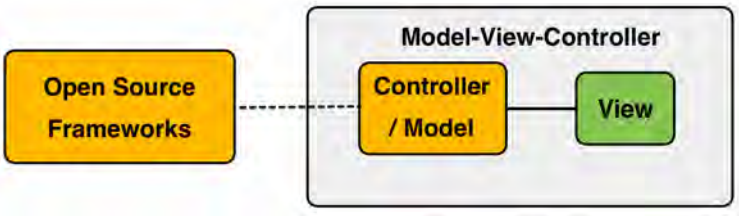


Abbildung 4.10: Struktur des iPad Anwendungsbeispiels.

4.2.3 Motion Tracking

Zur Echtzeit-Steuerung eines Szenenobjekts (Kamera) in MotionBuilder mit authentischen Bewegungsdaten, werden die Daten der Inertialsensoren des iPads abgefragt. Wie in Kapitel 3.3 beschrieben, verfügt das Gerät über drei Beschleunigungssensoren zur Messung der Beschleunigung entlang jeder Achse und ein Gyrometer, dass die Rotation des iPads um die eigenen Achsen aufzeichnet. Das iOS SDK bietet Entwicklern mehrere Programmierschnittstellen[7], die genutzt werden können um die Messdaten der Sensoren abzufragen.

Neben den rohen Messdaten der einzelnen Sensoren, bietet das SDK auch bereits verarbeitete Werte auf Abruf an. Diese werden dem Entwickler in Form der Gerätelage, der Geräterotation, der Wirkrichtung der Schwerkraft auf das Gerät und der externen Beschleunigung des Gerätes, zur Verfügung gestellt. Das iOS SDK nutzt nicht näher spezifizierte Algorithmen um diese Werte aus den rohen Sensordaten zu errechnen und löst damit einige Probleme für den Entwickler, die bei Verwendung der Rohdaten entstehen: Die unbearbeiteten Messwerte der Beschleunigungssensoren werden stets von der Erdbeschleunigung (Schwerkraft) beeinflusst und entsprechen daher niemals der tatsächlichen wirkenden Beschleunigung auf das Gerät. Das iOS SDK kombiniert die Messwerte der Beschleunigungssensoren daher mit der vom Gyrometer gemessenen Drehrate des Gerätes, um die Richtung der Erdbeschleunigung zu bestimmen und so genauere Beschleunigungswerte errechnen zu können.

Die verarbeiteten Bewegungsdaten bietet das iOS SDK in drei mathematischen Notationen an: Quaternionen, Rotationsmatrizen und Winkel im Bogenmaß. Letztere werden in Form der *Roll-Pitch-Yaw* Winkel (dt. *Roll-Nick-Gier*) notiert, die ursprünglich aus der Luftfahrt stammen und die Orientierung des Gerätes auf raumfester Basis beschreiben. Sie sind aufgrund ihrer einfachen Struktur am besten für das Anwendungsbeispiel geeignet.

Die Rotations- und Beschleunigungs-Achsen bei iPad / iPhone werden in der senkrechten Portrait-Orientierung des Geräts definiert. Basierend auf der Achsendefinition in Abbildung 4.11, ist die Rotation um die X-Achse als *Pitch*, die Rotation um die Y-Achse als *Roll* und die Rotation um die Z-Achse als *Yaw* festgelegt. Bei den Koordinaten handelt es sich um ein rechtshändiges System.

Zur Steuerung der Rotation der Kamera werden die verarbeiteten Roll-Pitch-Yaw Winkel über das SDK in einem festgelegten Zeitintervall von 10 Millisekunden abgefragt. Dies ist das kleinste Intervall, dass die Software dem Entwickler zur Verfügung stellt und bietet mit einer Frequenz von 100Hz ausreichend Abtastungen, um selbst sehr schnelle Bewegungen des iPads noch zu registrieren.

Die Roll-Pitch-Yaw Winkel werden dabei immer mit der Erdbeschleunigung bzw. Gravitation als Bezugspunkt berechnet. Der im SDK definierte Bezugsvektor des Gerätes beschreibt die Z-Achse in einer vertikalen Position entlang der Erdanziehung, mit den X- und Y-Achsen orthogonal zur Gravitation. Dies entspricht der horizontalen Lage des iPads auf einer ebenen

[7] http://apple.co/15CBRah (Zuletzt abgerufen: 23.01.2016)

Oberfläche mit der Rückseite unten aufliegend. Der Bezugsvektor (Gravitation) definiert sich laut SDK-Dokumentation[8] wie folgt:

$$\vec{g} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \tag{4.1}$$

Um die Verwendung der Messdaten zu erleichtern, bietet das iOS SDK die Möglichkeit den Bezugsvektor neu zu definieren. Dazu muss die Rotationsmatrix der verarbeiteten Messdaten in der neuen Bezugslage des Geräts gespeichert und bei jeder neuen Messabfrage invers mit der aktuellen Rotationsmatrix multipliziert werden. Sämtliche Winkelberechnungen arbeiten jetzt mit der korrigierten Rotationsmatrix und die neue Bezugslage wird bei der Berechnung der Roll-Pitch-Yaw Winkel berücksichtigt.

Die App ermöglicht es dem Anwender die Bezugslage jederzeit neu zu definieren, sodass sich die Software der Ausgangshaltung des Anwenders anpassen kann. Zur „Kalibrierung" der Bewegungssensoren sollte der Anwender das iPad vertikal im *Landscape*-Modus (Seitenlage) vor das Kamerabild am Monitor halten und den Bezugsvektor mit dem entsprechenden Bedienelement neu definieren. So wird garantiert, dass die Rotation der Kamera in Motion-Builder, die Bewegung des iPad möglichst genau wiederspiegelt.

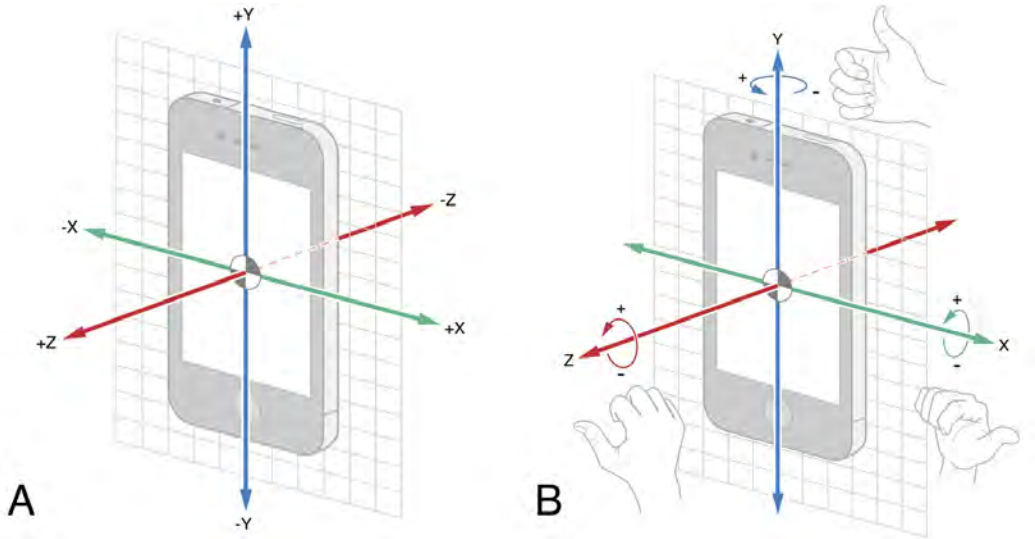


Abbildung 4.11: (A) Messung der linearen Beschleunigung entlang der drei Achsen des iOS-Geräts mit den Beschleunigungssensoren. (B) Messung der Rotation um die drei Achsen des iOS Geräts mit dem Gyrometer nach dem rechtshändigen System.

Quelle: (A/B) http://apple.co/15CBRah (Apple Inc., Zuletzt abgerufen: 23.01.2016)

[8] http://apple.co/1SDws85 (Zuletzt abgerufen: 23.01.2016)

4.2.4 Grafische Benutzeroberfläche

Für die Umsetzung der iPad App ist eine grafische Benutzeroberfläche notwendig, über die der Anwender Einstellungen zum Aufbau der Verbindung und Touch-basierte Eingaben vornehmen kann. Die Benutzeroberfläche ist in mehrere Abschnitte gegliedert, die eine klare Trennung zwischen Softwareinstellungen, Benutzerinformationen und Steuerungselementen schaffen (siehe Abbildung 4.12). Auf dem iPad ist lediglich eine Untersützung der horizontalen *Landscape*-Ausrichtung des Geräts geplant, da diese sich für die Anwendung des Geräts als virtuelle Kamera besser eignet. Gründe hierfür sind ein komfortablerer Halt des Tablets und eine bessere Erreichbarkeit der einzelnen Steuerelemente auf dem Touchscreen.

Die Sektion zur Konfiguration der Anwendung findet sich im unteren Bildschirmabschnitt. Dort kann der Anwender Netzwerkeinstellungen wie die Samplerate der Nutzdaten und den Port festlegen, auf dem das Gerät auf eingehende Verbindung der MotionBuilder Schnittstelle wartet. Der Serverstatus des Geräts soll über eine Schaltfläche kontrollierbar sein und Textfelder informieren den Nutzer über die aktuelle IP-Adresse des Tablets, sowie den Verbindungsstatus zum MotionBuilder Client. Eine weitere Schaltfläche ermöglicht die Neukalibrierung der Inertialsensoren um Ausgangsposition des Geräts für den Betrieb zu korrigieren. Wechselschalter ermöglichen das Aktivieren und Deaktivieren der Debug-Funktionen. Über dem Einstellungsbereich ist ein kleiner Abschnitt zur Anzeige von relevanten Informationen für den Benutzer reserviert. Dort werden unter anderem neue Verbindungen und Verbindungsänderungen angezeigt.

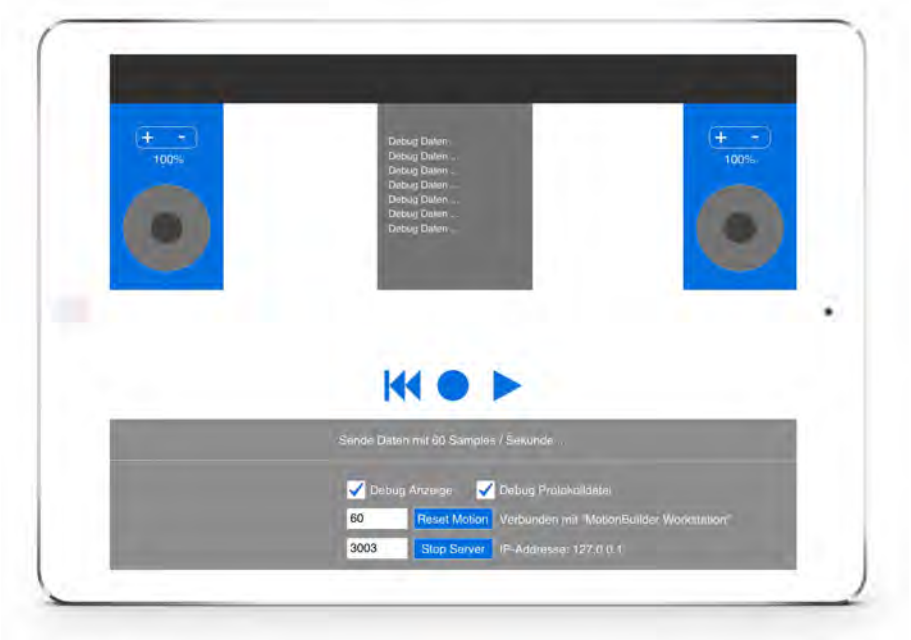


Abbildung 4.12: Entwurf der grafischen Oberfläche für das iPad Anwendungsbeispiel.

Quelle: http://bit.ly/1ZMM2Ob (Creative3x Ltd., iPad Mockup modifiziert, Zuletzt abgerufen: 25.01.2016)

Die oberen zwei Drittel des Bildschirms sind für zusätzliche Steuerelemente reserviert, die der Anwender nutzen kann um Eingaben an MotionBuilder zu senden. Auf Griffhöhe stehen dazu zwei digitale Analogsticks zur Verfügung, über die sich beispielsweise die Translation der Kamera im Raum steuern lässt. Zur präziseren Nutzung der Analogsticks sind über ihnen jeweils zwei Bedienelemente platziert, mit denen sich die Empfindlichkeit der Sticks regulieren lässt. In der Mitte der Benutzeroberfläche ermöglichen eine Reihe Schaltflächen, die Steuerung der Aufnahme und die Wiedergabe der Animation in MotionBuilder. Diese sind abseits der restlichen Steuerungselemente platziert, damit eine versehentliche Störung der Aufnahme während der Bedienung der Analogsticks ausgeschlossen wird.

Weiterhin kann auf Wunsch des Benutzers auch ein Debug-Textfeld eingeblendet werden, dass die wichtigsten Daten (z.B. Geräterotation, Analogstick-Position) zur Fehlersuche anzeigt. Aufgerufen über eine entsprechende Schaltfläche im Konfigurationsbereich der Oberfläche, wird es aus Platzgründen, zwischen den beiden Analogsticks, in der Mitte des Touchscreens angezeigt.

4.2.5 Datenübertragung

Damit Daten von der iPad Anwendung zu MotionBuilder übertragen werden können, muss zuvor eine Netzwerkverbindung zwischen den beiden aufgebaut werden. Der Verbindungsaufbau erfolgt wie in Kapitel 4.1.6 beschrieben. Die iPad Anwendung dient als Datenquelle und implementiert dementsprechend den Server-Socket für die Verbindung.

Ein Nutzdatenpaket beginnt mit denen vom Protokoll (siehe Kapitel 4.1.5) vorausgesetzten Daten, namentlich Nachrichtenkennung und Nachrichtennummer. Das darauf folgende Nutzdaten-Array, enthält die Rotationsdaten der Inertialsensoren in Form von drei Winkeln im Gradmaß: Roll, Pitch und Yaw. Zusätzlich zur Rotation werden die aktuellen Werte der beiden digitalen Analogsticks übermittelt. Diese werden in Form von zweidimensionalen Koordinaten gesendet, wobei die X-Komponente horizontale Bewegungen und die Y-Komponente vertikale Bewegungen des Sticks beschreiben. Um die Empfindlichkeit der Sticks zu regulieren, stehen zwei prozentuale Werte zwischen 1 und 100 zur Verfügung, die dem Datenpaket ebenfalls beigefügt werden.

Zur Fernsteuerung von MotionBuilder werden weiterhin boolsche Zahlenwerte (*0 / false* oder *1 / true*) für die *Play*-, *Record*- und *Rewind*-Buttons übertragen. Beim Betätigen eines Buttons wird der entsprechende Wert kurzzeitig auf *true* gesetzt um MotionBuilder die Benutzerinteraktion zu signalisieren. Insgesamt besteht das Datenpaket aus zwölf Datensätzen, von denen neun als Zahlenwerte und drei als boolsche Variablen in MotionBuilder interpretiert werden.

Abbildung 4.13: Struktur der Nutzdaten, die von der iPad Anwendung als Teil eines Nutzdatenpakets übermittelt wird.

4.2.6 Datenverarbeitung im Relation Constraint

Die eingehenden Nutzdaten der iPad App können in vielen Fällen einfach genutzt werden, ohne das vorab Verarbeitung der Daten von Nöten ist. Die drei übermittelten Boolean Werte für *Play*, *Record* und *Rewind*, können direkt mit dem entsprechendem Eingang in der Player-Control-Box verknüpft werden. Für die Roll-Pitch-Yaw Winkel muss lediglich der *Yaw*-Wert um den Faktor „-1,0“ multipliziert werden, damit das Vorzeichen des Winkels und damit auch die Rotationsrichtung mit der Benutzerbewegung übereinstimmt. Die Rotationswerte können anschließend auf einen Würfel übertragen werden, der als Elternelement für eine Kamera fungiert, die mittig zum Würfel ausgerichtet wird, um die optimale Übersetzung der Rotation auf die Kamera zu erhalten. Die Ausrichtung kann gegebenenfalls, auch im Betrieb der Anwendung, noch manuell korrigiert werden.

Die X- und Y-Werte der digitalen Analogsticks zur Steuerung der Translation müssen vor der Verwendung noch mit dem Wert für die Empfindlichkeit verrechnet werden. Dazu wird dieser prozentual mit den X- und Y-Werten multipliziert, um den maximalen Wertebereich anpassen zu können. Neue Eingabewerte der Sticks müssen zudem auf die vorherigen Translationswerte aufaddiert werden, sodass die Translation erhalten bleibt und nicht zurück in den Ursprung springt, sobald der Nutzer den Stick loslässt und dessen Werte zurückgesetzt werden. Die Translationswerte werden ebenfalls auf den zuvor angelegten Würfel übertragen, da dieser durch die Elternbeziehung zur Kamera auch deren Translation beeinflusst.

4.3 Anwendungsbeispiel: Leap Motion Tracking

Als Teil des zweiten Anwendungsbeispiels, soll der *Leap Motion*-Controller (siehe Kapitel 3.4) über den Plugin-Prototypen an MotionBuilder angeschlossen werden. Der Controller ermöglicht die Aufzeichnung von Hand- und Fingerbewegungen mithilfe stereoskopischer Infrarotkameras. Da es sich bei dem Gerät um ein reines USB-Zubehör ohne Netzwerkinterface handelt, muss eine Applikation basierend auf dem SDK des Herstellers geschrieben werden, welche die Daten vom Controller abfragt und an die Netzwerkschnittstelle des Plugins sendet.

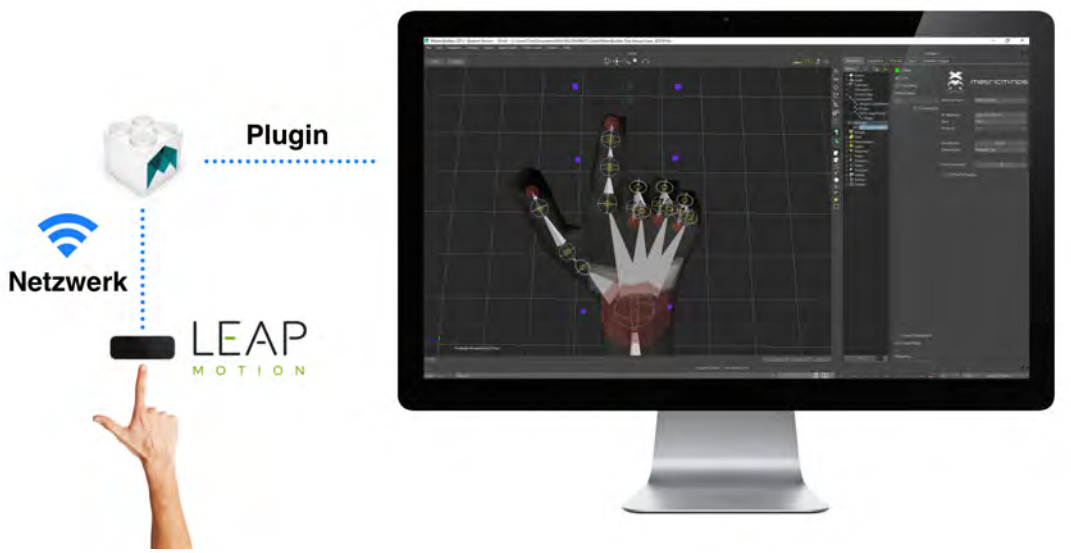


Abbildung 4.14: Konzept des Hand- und Finger-Trackings in MotionBuilder mit Daten des Leap Motion-Controllers. Die Verbindung der Datenquelle (Leap Motion Server) mit der Schnittstelle des Plugins erfolgt über eine lokale Netzwerkverbindung.

Quelle: http://bit.ly/1X1VxIG (Fernando Cruz, modifiziert; Zuletzt abgerufen: 05.03.2016)
http://bit.ly/21Qv9oD (Leap Motion Inc., modifiziert; Zuletzt abgerufen: 05.03.2016)
http://bit.ly/24Ih7HO (LiveLongAndTravel.com, modifiziert; Zuletzt abgerufen: 05.03.2016)

4.3.1 Zielsetzung

Zur Verbindung des Leap Motion-Controllers mit MotionBuilder wird eine Mac OS X App entwickelt, die das Leap Motion SDK ansteuert und die vom Controller gesammelten Daten abfragt. Die Daten werden dann per Netzwerk-Socket an das MotionBuilder Realtime-Plugin übertragen (siehe Abbildung 4.14). Um den Entwicklungsaufwand zu reduzieren und redundante Programmierarbeit zu vermeiden, wird als Basis für die Anwendung, auf den ebenfalls *Swift*-basierten Programmcode des iPad-Anwendungsbeispiels zurückgegriffen. Dies betrifft insbesondere jene Codeabschnitte, die für die Netzwerkkommunikation zuständig sind und einfach auf den Mac portiert werden.

Die Mac Anwendung muss die gegebenen Funktionen des Leap Motion SDK nutzen, um nutzbare Daten aus dem Gerät zu erhalten. Es wird die Position und Rotation beider Hände abgefragt und diese in Form von Koordinaten und Roll-Pitch-Yaw Winkeln an die Netzwerkschnittstelle des Plugins übertragen werden. Weiterhin werden die Richtungsvektoren sämtlicher Fingerknochen einer Hand über das SDK ermittelt und ebenfalls als Teil des Datenpakets an MotionBuilder gesendet. Zusätzlich zur Animation der Handrotation und -position, können diese Daten genutzt werden, um ein komplettes Hand-Rig in MotionBuilder anzutreiben. Dabei sollen die gesammelten Daten ausreichen, um die Fingergelenke des Rigs um eine Achse zu rotieren zu können. Die Interpration der Richtungsvektoren zur Bestimmung der Knochenrotation erfolgt erst im Relation Constraint (MotionBuilder).

Die App soll eine rudimentäre Benutzeroberfläche bieten, über die sich wichtige Parameter wie Netzwerkport und Samplerate einstellen lassen. Die Anwendung kann dazu genutzt werden, um simple Hand- / Fingeranimationen, wie z.B. das Bilden einer Faust, mit authentischen Bewegungsdaten zu erstellen und in Echtzeit aufzuzeichnen.

4.3.2 Motion Tracking

Zur Abfrage der Messdaten des Leap Motion-Controllers (siehe Kapitel 3.4), setzt der Hersteller des Gerätes die Verwendung des eigenen SDK voraus. Das SDK übernimmt die Auswertung und Verarbeitung der Rohdaten des Controllers. Dazu errechnet es aus den 2D-Bilddaten der Stereo-Kameras des Sensors über nicht näher spezifizierte Algorithmen ein 3D-Tracking-Modell und ignoriert automatisch störende Hintergrundobjekte wie Köpfe und Umgebungslicht im Kamerabild.

Der Programmierer kann für jedes Bild ein Tracking-Modell abrufen, welches die Messdaten aller Objekte und die erkannten Gesten enthält. Für die Umsetzung des Anwendungsbeispiels werden die Daten der Hände und Finger benötigt. Für die Positionsbestimmung der Hände, bietet das Leap Motion SDK die Distanz der Handflächen, gemessen in Millimetern, von der Position des Controllers an. Die lokale Position wird über dreidimensionale Koordinaten beschrieben, die ihren Ursprung im Leap Motion-Controller haben.

Zur Bestimmung der Handrotation, kann auf die Normalenvektoren der Handflächen und deren Richtungsvektoren zurückgegriffen werden. Der Normalenvektor spannt sich dabei orthogonal zur Handfläche auf (siehe Abbildung 4.15-B). Über diese beiden Vektoren lassen sich leicht die globalen Winkel für Roll, Pitch und Yaw bestimmen. Der Roll-Winkel wird über die Richtung des Normalenvektor errechnet, die sich ändert sobald die Hand seitlich gekippt wird. Für Pitch und Yaw können die Winkel über den Richtungsvektor der Hand bestimmt werden, der von der Handflächenposition in Richtung der Finger zeigt.

Etwas rechenaufwändiger als die Rotation der Hand, ist die Bestimmung der Rotation einzelner Fingerknochen. Hierzu bietet das SDK keine fertigen Funktionen an, doch lassen sich die Winkel zwischen den Fingerknochen, über die normalisierten Richtungsvektoren der Knochen bestimmen. Durch die eingeschränkte Beweglichkeit der Fingerglieder über eine Achse, ist le-

diglich eine zweidimensionale Änderung der Richtungsvektoren zwischen den Fingergliedern möglich. Man spricht von einem Freiheitsgrad (*Degree of Freedom*, DOF) pro Fingergelenk der Phalanx-Knochen, sowie zwei Freiheitsgraden für das Gelenk des Metacarpal-Knochen, welches sich um eine weitere Achse bewegen kann [SE03, S. 3]. Die zweite Achse, zum Spreizen der Finger, wird für dieses Anwendungsbeispiel jedoch ignoriert. Der Winkel zwischen zwei Vektoren [BS05, S. 194] errechnet sich wie folgt:

$$\cos\varphi = \frac{\vec{a} \cdot \vec{b}}{\sqrt{\vec{a^2} \cdot \vec{b^2}}} \tag{4.2}$$

φ : Winkel zwischen den normalisierten Richtungsvektoren
$\vec{a}$: Richtungsvektor 1
$\vec{b}$: Richtungsvektor 2

Die berechneten Winkel zwischen den Fingerknochen können später in MotionBuilder dazu genutzt werden, um ganze Finger-Rigs mit authentischen Rotationen zu animieren. Ein Nachteil des gewählten Ansatzes ist, dass nicht ermittelt wird, ob Finger gespreizt oder zusammengestaucht sind, was jedoch für einfache Fingeranimationen wie das Zeigen eines Fingers auch nicht zwingend notwendig ist.

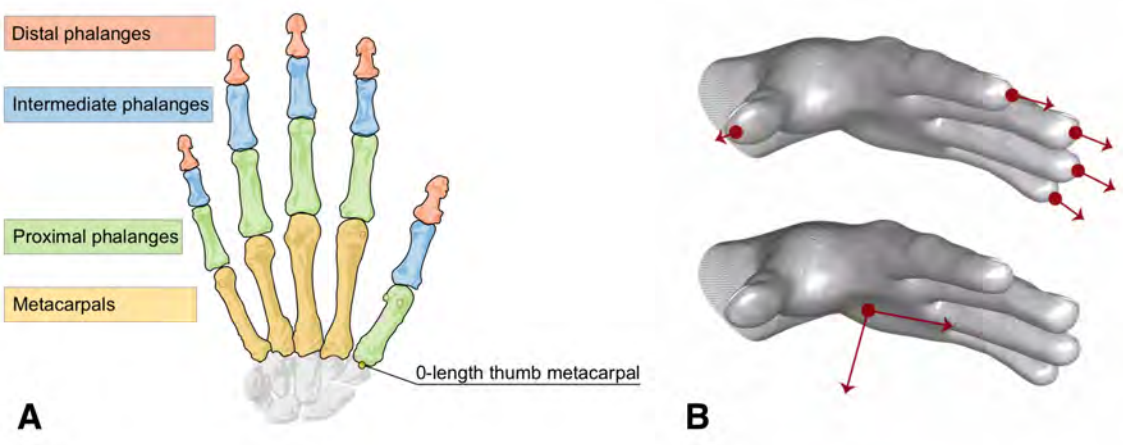


Abbildung 4.15: (A) Fingerknochen einer menschlichen Hand mit ihren medizinischen Bezeichnungen. (B) Oben: Visualisierung der Richtungsvektoren der letzten Fingerglieder. Unten: Visualisierung des Normalen- und Richtungsvektors der Handfläche.

Quelle: (A / B) http://bit.ly/1WXKCAm (Leap Motion Inc., Zuletzt abgerufen: 01.02.2016)

4.3.3 Datenübertragung

Der Verbindungsaufbau zwischen MotionBuilder und der Leap Motion Server Anwendung ist analog zum beschriebenen Verfahren in Kapitel 4.1.6. Dieses Anwendungsbeispiel sendet allerdings eine deutlich größere Menge an Nutzdaten, als die zuvor beschriebene iPad App zur Kamerasteuerung.

Die vom Leap Motion SDK gewonnenen Daten, werden in Form von Zahlenwerten (Double) an Device übertragen. Das Nutzdaten-Array enthält die Roll-Pitch-Yaw Winkel der beiden Handflächen im Gradmaß, sowie die Distanz der Handflächen vom Leap Motion-Controller, in Form von dreidimensionalen Koordinaten (X,Y,Z). Diese Daten können beispielsweise genutzt werden um zwei Hände mit Translation und Rotation zu animieren. Aufgrund der eingeschränkten Natur des Positions-Tracking der Handflächen, ist der Nutzen der Position für ein Handrig jedoch fraglich und eignet sich daher eher für die Animation anderer Objekte.

Zusätzlich werden die dreidimensionalen Richtungsvektoren für sämtliche Fingerknochen, der zuletzt erkannten Hand, übertragen. Die Vektoren werden zur Übertragung in ihre einzelnen Komponenten (X,Y,Z) aufgeteilt und müssen zur Verarbeitung im Relation Constraint wieder korrekt zusammengesetzt werden. Pro Finger sind dies zwölf Zahlenwerte, da jedes Leap Motion Finger-Datenset die Vektoren von vier Knochen enthält. Zusammen mit den Daten der Handflächen, werden in einem Nutzdaten-Array des Anwendungsbeispiels insgesamt 72 Zahlenwerte an das MotionBuilder Device übermittelt.

Nutzdaten-Struktur: Leap Motion Server

01	Left Palm Position X	Double	15	Thumb Bone 1 Z	Double
02	Left Palm Position Y	Double	16	Thumb Bone 2 X	Double
03	Left Palm Position Z	Double	17	Thumb Bone 2 Y	Double
04	Left Hand Roll	Double	18	Thumb Bone 2 Z	Double
05	Left Hand Pitch	Double	19	Thumb Bone 3 X	Double
06	Left Hand Yaw	Double	20	Thumb Bone 3 Y	Double
07	Right Palm Position X	Double	21	Thumb Bone 3 Z	Double
08	Right Palm Position Y	Double	22	Thumb Bone 4 X	Double
09	Right Palm Position Z	Double	23	Thumb Bone 4 Y	Double
10	Right Hand Roll	Double	24	Thumb Bone 4 Z	Double
11	Right Hand Pitch	Double	25	Index Bone 1 X	Double
12	Right Hand Yaw	Double	...	...	...
13	Thumb Bone 1 X	Double	71	Little Bone 4 Y	Double
14	Thumb Bone 1 Y	Double	72	Little Bone 4 Z	Double

Abbildung 4.16: Die Struktur des Daten-Arrays eines Nutzdatenpakets des Leap Motion Server Anwendungsbeispiel. Zur besseren Darstellung sind lediglich 27, der insgesamt 72 Datensätze aufgelistet. Die Struktur der ausgelassenen Fingerknochen Daten ist jedoch analog zu der vollständig aufgelisteten Struktur des Daumens (Nr. 13 - 24).

4.3.4 Datenverarbeitung im Relation Constraint

Bevor die Nutzdaten in MotionBuilder verwendet werden können, müssen diese erst mittels Relation Constraint wieder aufbereitet und verarbeitet werden. Für die Rotationsdaten der Hände, welche in der Roll-Pitch-Yaw Notation im Gradmaß übertragen werden, müssen die Werte angepasst werden, um die Rotationen korrekt wiederzugeben. Wichtig ist hierbei, dass die die Notation der Rotationen beachtet wird.

Die Rotationen der rechten Hand sind nach dem rechtshändigen System zu interpretieren, die Rotation der linken Hand entsprechend nach dem linkshändigen System. Dies resultiert darin, dass für die linke Hand das Vorzeichen für den *Roll*-Wert mit einer negativen Multiplikation um den Faktor „-1“ gewechselt werden muss. Für die rechte Hand muss das Vorzeichen des *Pitch*-Wert gewechselt werden. Die *Yaw*-Werte beider Hände benötigen ebenfalls einen Vorzeichenwechsel zur korrekten Darstellung.

Für die Verwendung der Hand-Position ist keine weitere Verarbeitung notwendig. Da sie jedoch relativ zum Controller in Millimeter gemessen wird, gilt es zu beachten, dass die Werte schnell extrem groß werden können. Die Rotation der einzelnen Fingerglieder erfordert hingegen eine komplexe Verarbeitung. Wie in Kapitel 4.3.2 beschrieben, werden jeweils die Winkel zwischen zwei Richtungsvektoren der einzelnen Fingerknochen errechnet. Dem Nutzer sollte außerdem die Möglichkeit gegeben werden, ein Additions-Offset und ein Multiplikator für jede Fingerglied-Rotation festzulegen, sodass etwaige Tracking-Fehler des Leap Motion SDK leicht mit diesen Parametern ausgeglichen werden können.

Kapitel 5

Implementierung

Dieses Kapitel der Arbeit beschäftigt sich mit der Umsetzung des in Kapitel 4 beschriebenen Konzepts. Es werden die Besonderheiten und Details geschrieben, auf die es bei der Implementierung des Konzepts zu beachten gilt. Das Kapitel enthält zur besseren Erläuterung zahlreiche Quellcode-Ausschnitte, die jedoch meistens zwecks Beispiel-Darstellung angepasst und gekürzt wurden.

5.1 MotionBuilder Realtime Plugin

5.1.1 Entwicklungsumgebung

Wie in Kapitel 4.1.2 ausführlich erläutert, ist für die Entwicklung des MotionBuilder Plugins die Verwendung des Open Reality SDK (C++) unabdingbar. Nur das OR SDK bietet die Mittel und Performance, die zur Umsetzung des Konzepts notwendig sind. Zur Kompilierung von Anwendungen, die das Open Reality SDK nutzen, sind bestimmte Softwarepakete und Einstellungen notwendig. Diese werden nachfolgend erläutert.

Motionbuilder Plugins, die mit dem OR SDK entwickelt wurden, bindet die Software in Form von DLL-Dateien zum Start der Anwendung ein. Damit diese *Dynamically Linked Libraries* von MotionBuilder in der Version 2015 akzeptiert werden, müssen sie mit *Microsoft Visual Studio 2012*[1] kompiliert werden (siehe Abbildung 5.1). Dabei handelt es sich um eine Entwicklungsumgebung der Firma Microsoft, die unter anderem auch die Entwicklung von C++, sowie klassischer C-Software unterstützt. Theoretisch ist es zwar möglich die Software auch mit anderen Versionen der Software zu kompilieren, doch wird dies nicht offiziell seitens Autodesk unterstützt und kann daher zu unerwarteten Fehlern führen.

Auch ist nicht garantiert, dass ein für MotionBuilder 2015 kompiliertes Plugin, auch mit anderen Versionen der Autodesk Software kompatibel ist. Für die Entwicklung von komplexen grafischen Oberflächen wird das *Qt*-Framework[2] in der Version 4.8.5 empfohlen. Im

[1] http://bit.ly/1ZPICdh (Zuletzt abgerufen: 26.01.2016)
[2] http://bit.ly/20qll3K (Zuletzt abgerufen: 26.01.2016)

Rahmen dieser Arbeit kommt das Framework jedoch nicht direkt zum Einsatz, da die Qt-basierten Bordmittel des OR SDK für die Entwicklung der grafischen Oberfläche des Plugins ausreichen.

Als Grundlage für die Entwicklung eines Open Reality Plugins empfiehlt sich die Verwendung eines der zahlreichen Beispielprojekte von Autodesk. Diese geben bereits die grundlegende Code-Struktur, sowie die notwendigen Visual Studio Projekt-Einstellungen vor. Die Beispielprojekte finden sich unter „OpenRealitySDK\samples“ im Installationspfad von MotionBuilder 2015. Nach Öffnen der Projektdatei sollten die Beispiele direkt kompilierbar und lauffähig sein. Es muss jedoch zusätzlich in den Projekteigenschaften unter „C/C++ “, „Code Generation“ der Wert „Runtime-Library“ auf „Multi-threaded Debug (/MTd)“ gesetzt werden. Dies stellt sicher, dass das kompilierte Plugin auch auf Systemem läuft, die keine Visual Studio Entwicklungsumgebung installiert haben.

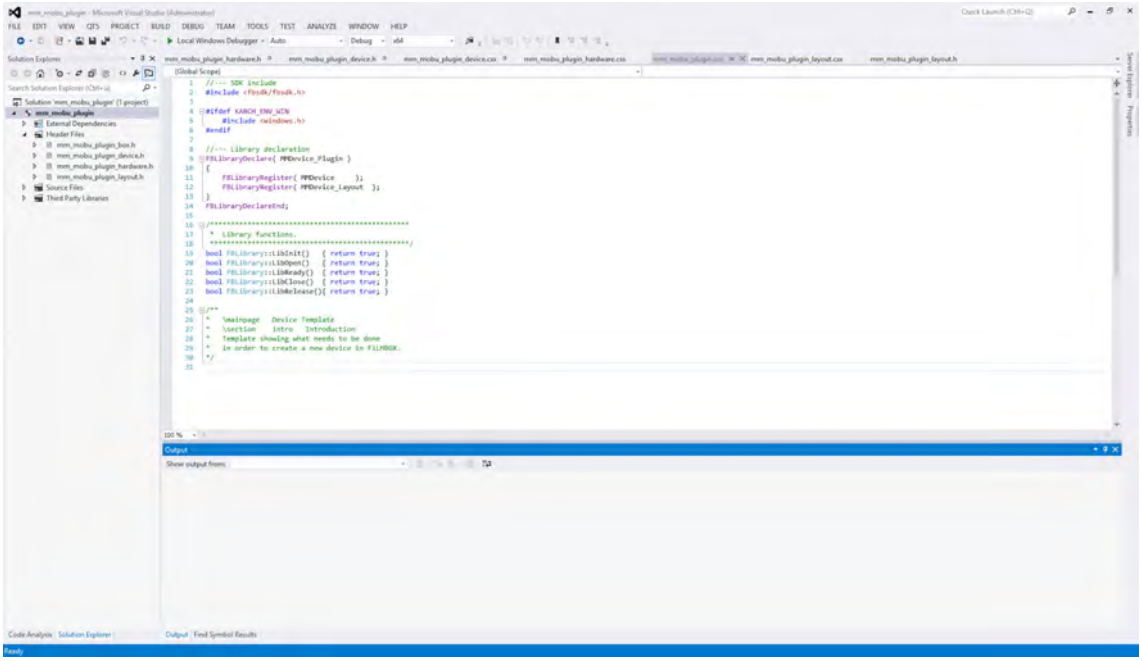


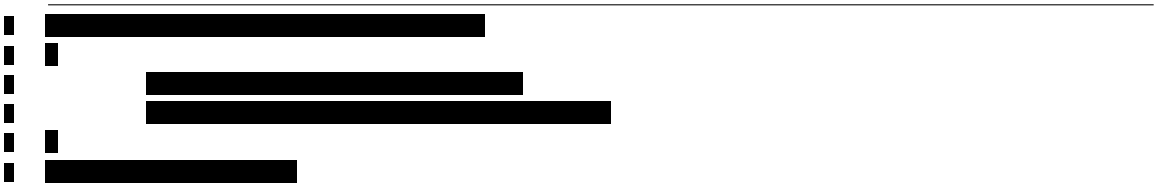
Abbildung 5.1: Open Reality SDK Projekt in Microsoft *Visual Studio 2012 Premium*.

Kompilierte Plugins müssen vom Anwender in den „bin\x64\plugins\“ Unterordner des MotionBuilder Installationsverzeichnis gelegt werden, von wo sie beim Start der Software geladen werden. Damit das Plugin nach dem Kompiliervorgang direkt in diesen Ordner gespeichert werden kann, muss Visual Studio im Administrator-Modus ausgeführt werden. Anderenfalls kann Visual Studio die Plugin-Datei nicht kopieren, da das Softwareverzeichnis unter aktuellen Windows-Versionen nur Administratoren Schreibzugriff bietet. Alternativ dazu kann das Plugin manuell kopiert werden, was jedoch ebenfalls Administrator Zugriff erfordert.

5.1.2 Open Reality SDK

Zur Entwicklung eines neuen *Device* mit dem Open Reality SDK gibt es mehrere Vorlagen im Lieferumfang von MotionBuilder, die sich an unterschiedliche Anforderungen richten. Wie in Kapitel 4.1.4 besprochen, wird für die Entwicklung des Prototypen das *OR Device Template* verwendet. Diese Vorlage enthält bereits das grundlegende Codeskelett für ein Device, dass über eine Netzwerkschnittstelle kommuniziert. Zur Verwendung der Vorlage wird der gesamte Projektordner aus dem Vorlagen-Ordner kopiert und die Projekteinstellungen der `ordevice_template.vcxproj` in Visual Studio angepasst (siehe vorheriges Kapitel).

Das Projekt beinhaltet insgesamt sieben Quellcode-Dateien, davon drei *Header*-Dateien (.h) für die Klassen des Projekts, sowie vier C++ -Quellcode Dateien (.cxx) mit dem Programmcode der Klassen und einer Bibliotheks-Deklaration für MotionBuilder (siehe Quellcode 1). Diese Deklaration befindet sich in der `or_device.cxx` Datei und ist essentiell für die interne Zuordnung des Plugins in MotionBuilder. Besonders wichtig ist die Wahl eines einzigartigen Bibliotheksnamen, sodass keine Konflikte mit den Plugins anderer Hersteller auftreten. Die Bibliothek registriert die einzelnen Komponenten des Plugins (Device, Grafische Oberfläche) als Teil von sich. Für die Entwicklung des Prototypen werden sämtliche Referenzen auf „`ORDevice_Template`“ in „`MMDevice`“ abgeändert. Weiterhin wird die Benennung der Quellcode-Dateien angepasst, sodass alle Dateien jetzt mit dem Prefix „`mm_mobu_plugin_`“ beginnen. So ist sichergestellt, dass die Namensgebung später keine Probleme verursacht.



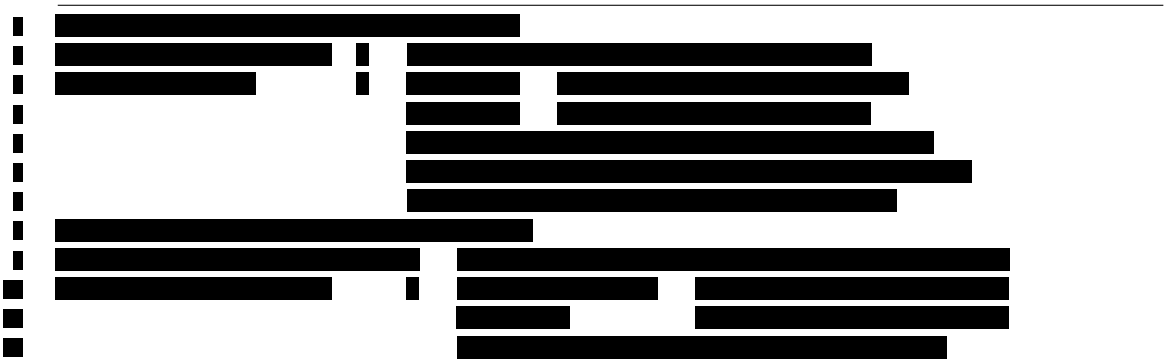
Quellcode 1: Deklaration der Bibliothek und Registrierung der einzelnen Komponenten.

Die wichtigste Klasse des Visual Studio Projekts ist `mm_mobu_plugin_device.cxx`, welche von `ordevice_template_device.cxx` abgeleitet wurde. Die Klasse enthält die wichtigsten Funktionen zur Erzeugung und internen Steuerung des Device, sowie die Verarbeitung der Nutzdaten. Diese Klasse hält außerdem eine Referenz auf `mm_mobu_plugin_hardware.cxx` (Vorlage: `ordevice_template_hardware.cxx`), welche die gesamte Netzwerkkommunikation und Paketverarbeitung implementiert. Der View der Anwendung ist in der Layout-Klasse `mm_mobu_plugin_layout.cxx` (Vorlage: `ordevice_template_layout.cxx`) definiert, die weiterhin Benutzereingaben über Events abfängt und Befehle an die Device-Klasse sendet.

Für die Entwicklung des Plugins wird ein Großteil des bestehenden Beispielcodes aus der Vorlage entfernt. Neben Beispielen zur Socket-Programmierung, enthält die Vorlage außerdem Code für die Einbindung eigener Datenquellen über serielle Schnittstellen, sowie ein Generator zur Erzeugung von Beispieldaten anhand mathematischer Funktionen. Da diese

Funktionalitäten jedoch nicht Teil des entwickelten Konzepts sind, werden sie komplett aus der Vorlage entfernt. Für die Layout-Klasse bedeutet dies besonders viele Änderungen, da die gesamte Benutzeroberfläche neu gestaltet werden muss. Vom verbleibenden Beispielcode wird fast ausschließlich die Struktur und Funktionsköpfe der Klassen übernommen, da die Vorlage zu spezialisiert zur Wiederverwertung der vorhandenen Logik ist.

Damit MotionBuilder die vom Plugin registrierten Klassen ihrer Funktion zuordneten kann, müssen die einzelnen Klassen in ihrem Quellcode außerdem eine weitere Registration hinzufügen (siehe Quellcode 2). Diese spezifiziert die Art der Implementierung (z.B. *Device* oder *Layout*) und liefert MotionBuilder zusätzliche Parameter zur Beschreibung der Klasse (z.B. Beschreibungstext, Icon).



Quellcode 2: Registrierung der Klassen-Implementierung im OR SDK.

5.1.3 MessagePack

Aus den in Kapitel 4.1.6 und Kapitel 4.1.8 genannten Gründen, wird die MessagePack Notation für die Enkodierung ein- und ausgehenden Datenpakete verwendet. Für C / C++ befinden sich gleich mehrere MessagePack-Bibliotheken in Entwicklung. Unter den kompaktesten Bibliotheken findet sich das Open Source Projekt *CMP*[3] von Charlie Gunyon.

Bei der Bibliothek handelt es sich um eine C-basierte Implementierung des MessagePack Formats, die jedoch auch mit C++-Code und -Compilern kompatibel ist. CMP ist darauf ausgelegt möglichst wenig Abhängigkeiten vorauszusetzen und erfordert im Gegensatz zu vielen anderen MessagePack-Bibliotheken, ebenfalls keine Anpassung der Projekteinstellungen. Die Bibliothek besteht lediglich aus einer Header- und einer Quellcode-Datei, sodass die Nutzung mit einer einzigen `import`-Referenz auf die Header-Datei möglich ist.

Leider setzen die Funktionen der Bibliothek eine Datei-basierte Datenstruktur zum Serialisieren und Deserialisieren voraus. Da das Device den Datenverkehr aber über Puffer im

[3] https://github.com/camgunz/cmp (Zuletzt abgerufen: 06.02.2016)

Arbeitsspeicher verwaltet, muss CMP um entsprechende Methoden zur Nutzung der Puffervariablen ergänzt werden. Das Open Source Projekt *cmp_mem_access*[4] von Patrick Spieler, erweitert CMP um die fehlende Puffer-Funktionalität. Um das Projekt zu verwenden, wird die „cmp_mem_access.h“ Quellcode-Datei importiert, welche die ergänzende Puffer-Funktionalität beinhaltet. Diese Helferklasse wird anstelle einer Datei, als Zugriffsziel, an CMP übergeben (siehe Quellcode 3).

CMP bietet eine Reihe unterschiedlicher Funktionen zum Lesen oder Schreiben von Daten in Bytecode an. Die Notation der resultierenden Daten entspricht dabei den aktuellen MessagePack Spezifikationen[5] (Stand: 06.02.2016) und ist mit anderen MessagePack-Bibliotheken kompatibel. Die notwendigen Methoden zur Steuerung der Leseposition im Puffer sind Teil von *cmp_mem_access*. Beispiele für die Datenverarbeitung mit CMP finden sich in Kapitel 5.1.5.

Quellcode 3: Einbindung und Initialisierung der CMP-Dateien in das Plugin.

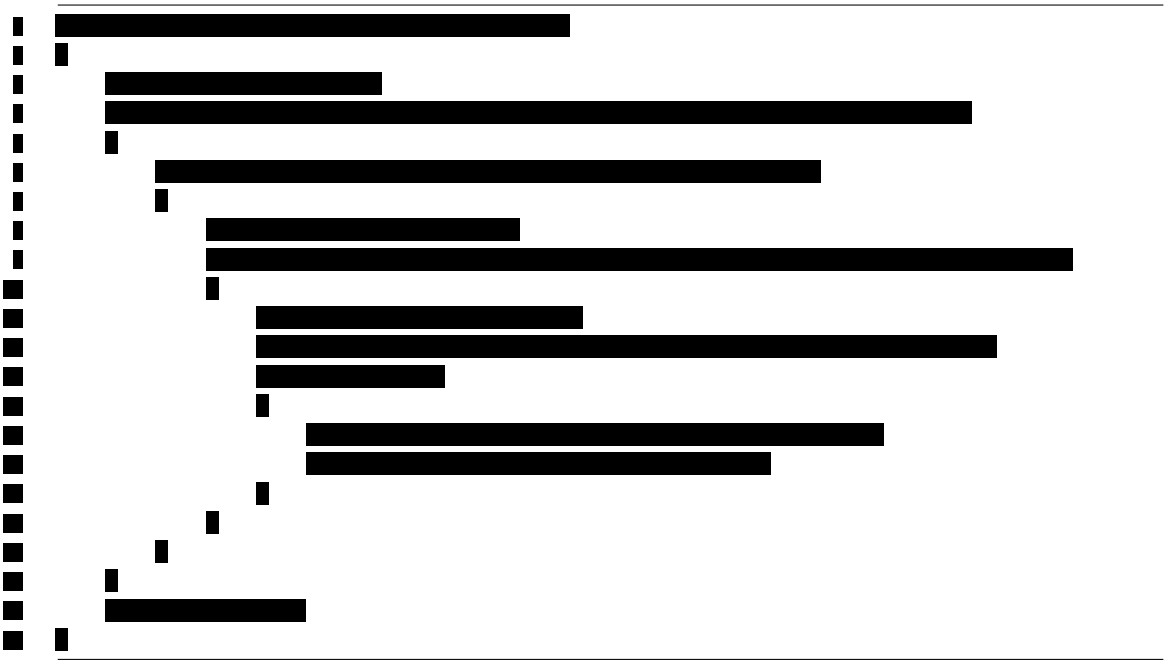
5.1.4 Socketkommunikation

Wie in Kapitel 4.1.6 im Detail beschrieben, implementiert das Device den Client-Socket der Verbindung zur Datenquelle. Die gesamte Netzwerkkommunikation wird dabei von der Device-Klasse gesteuert, die sich auch um die Verarbeitung der Daten kümmert. Der Verbindungsaufbau, die Kommunikation und die Verarbeitung der Pakete, wird hingegen an die Hardware-Klasse delegiert. Dazu hält die Device-Klasse eine Referenz auf die Hardware-Klasse und ruft bei Bedarf die entsprechenden Funktionen auf.

Der Verbindungsaufbau beim Start des Device erfolgt über die Funktion `StartDataClient()`, die von der `Open()`-Funktion der verwendeten OR SDK Vorlage abgeleitet wurde (siehe Quellcode 4). Die Funktion fragt das zu verwendende Protokoll ab und baut eine entsprechende Verbindung zur Datenquelle auf. Da *TCP* als einziges Netzwerkprotokoll in der Benutzeroberfläche fest eingestellt und implementiert ist, greift die Funktion stets auf den Code

[4] https://github.com/Stapelzeiger/cmp_mem_access (Zuletzt abgerufen: 06.01.2016)
[5] https://github.com/msgpack/msgpack/blob/master/spec.md (Zuletzt abgerufen: 06.02.2016)

für den Aufbau einer TCP-Verbindung zurück. Das OR SDK stellt die FBTCPIP-Klasse bereit, welche alle notwendigen Funktionen zur Kommunikation mittel TCP- oder UDP-Sockets enthält. Für den Aufbau eines TCP-Client Sockets werden Portnummer und IP-Adresse des Servers (Datenquelle) benötigt. Diese werden vom Benutzer in der grafischen Oberfläche eingeben (siehe Kapitel 5.1.9) und in der Hardware-Klasse gespeichert.

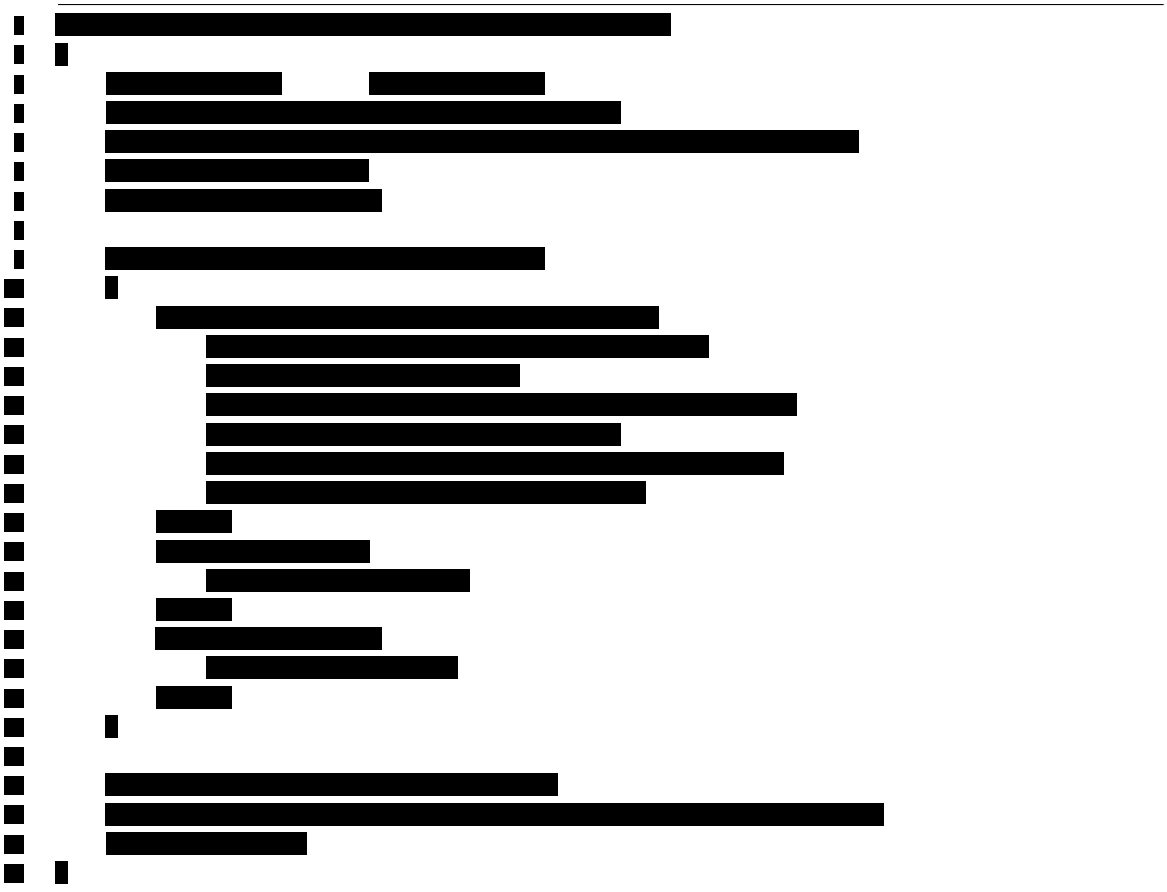


Quellcode 4: Erstellung eines TCP-Client-Sockets mit Verbindungsaufbau.

Bevor die Verbindung aufgebaut werden kann, muss ein TCP-Socket mit der `CreateSocket()`-Funktion der FBTCPIP-Instanz angelegt werden. Die eigene Portnummer wird dem Socket dabei vom System automatisch zugeteilt. Nach der Erstellung des Sockets wird mit der `connect()`-Funktion selbiger Klasse, eine Verbindungsanfrage an die hinterlegte IP-Adresse gesendet. Wird diese vom Server akzeptiert, steht die TCP-Verbindung. Die Device-Klasse kann nun damit beginnen, Anfragen über den neuen Socket in der Hardware-Klasse zu versenden und Antworten zu empfangen.

Zum Senden von Anfragen greift die Device-Klasse auf die `SendRequest()` Funktion der Hardware-Klasse zu (siehe Quellcode 5). Als Parameter wird die Nachrichtenkennung als Enum-Wert (siehe Kapitel 4.1.5) übergeben, die der Hardware-Klasse mitteilt, welche Anfrage zu senden ist. Die Funktion setzt dann die gewünschte Nachricht zusammen und sendet sie mit der `Write()`-Funktion der FBTCPIP-Instanz über die Socket-Verbindung an die Datenquelle.

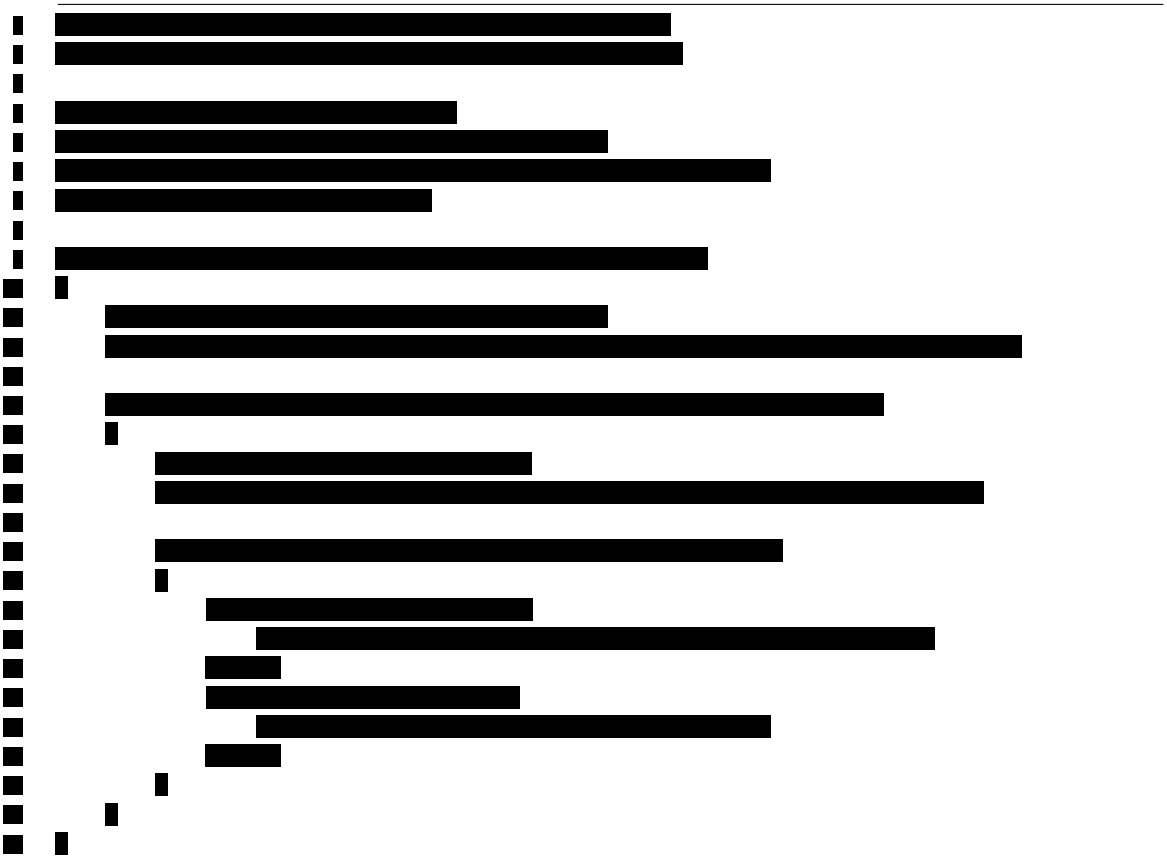
Eingehende Pakete werden mit der `Read()`-Funktion der TCP-Instanz in einen `Char`-Puffer mit der Größe 65535 geschrieben, was der maximalen TCP-Paketgröße von 64kB entspricht. Dieser wird anschließend in der Paketverarbeitung (siehe Kapitel 5.1.5) gelesen und interpretiert. Stoppt der Anwender dass Device, wird die Funktion `StopDataClient()` der Hardware-Klasse genutzt, um die Socketverbindung aufzulösen. Diese greift auf die `CloseSocket()`-Funktion der `FBTCPIP`-Instanz zurück, um die Verbindung korrekt zu beenden.



Quellcode 5: Senden einer Anfrage über den TCP-Socket an die Datenquelle.

5.1.5 Paketverarbeitung

Bei einer aktiven Socketverbindung, ruft die Device-Klasse zur festgelegten Samplerate (`SamplingPeriod`), die eigene `DeviceIONotify()`-Funktion auf, welche das Lesen und die Verarbeitung eingehender Daten mit `FetchData()` veranlasst. Diese Funktion liest die Daten wie im vorherigen Kapitel beschrieben mit `Read()` direkt vom Socket, deserialisiert die Pakete und verarbeitet die extrahierten Daten (siehe Quellcode 6).

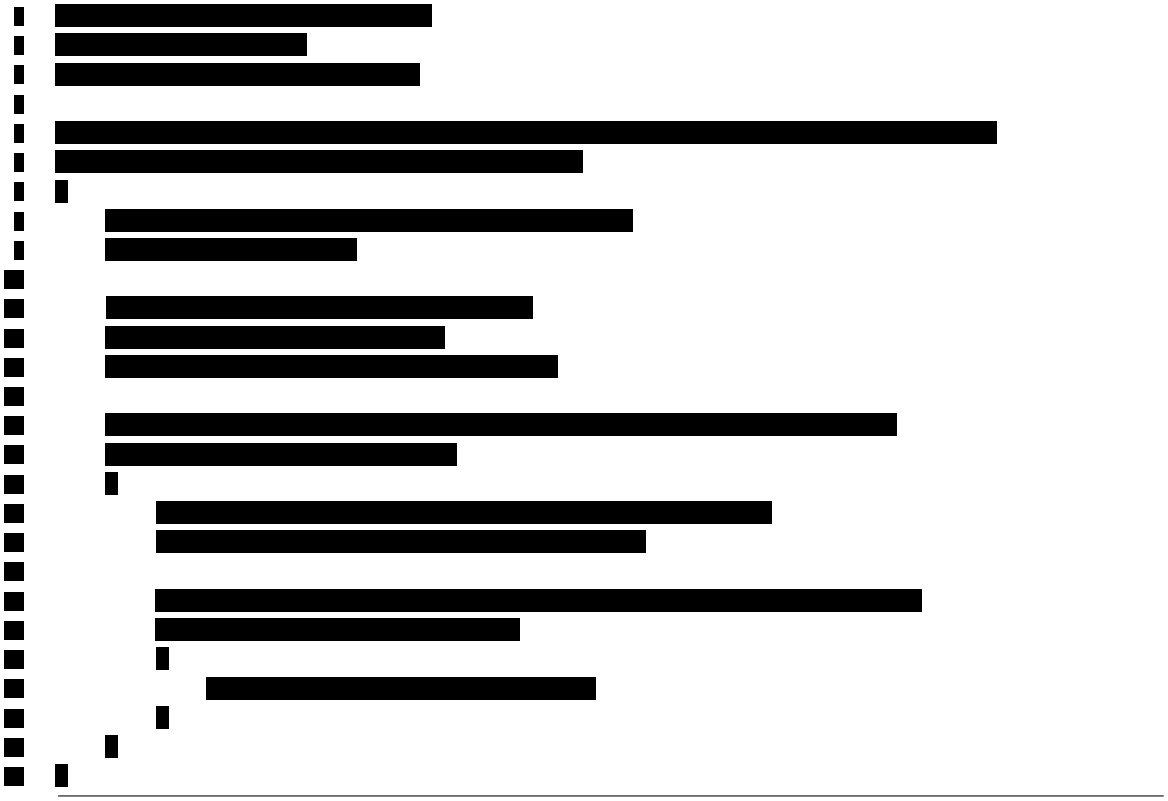


Quellcode 6: Vereinfachter Ausschnitt der Paketverarbeitung in der `FetchData()`-Funktion der Hardware-Klasse des Device.

Zuerst wird eine lokale Kopie des `Char`-Lesepuffers angelegt, welcher eingelesenen Bytecode der Socketverbindung enthält. Dies stellt sicher, dass der Puffer während der Verarbeitung konstant ist und nicht mit anderen Daten überschrieben wird. Als nächstes wird überprüft, ob Daten vom Socket gelesen werden konnten. Ist dies der Fall, wird mit der `cmp_read_array`-Funktion der CMP-Bibliothek, die Größe des übermittelten MessagePack Array bestimmt. Enthält dieses ein Element oder mehr, wird als Nächstes versucht die Integer-Nachrichtenkennung mit `cmp_read_int()` auszulesen. Diese steht nach Protokolldefinition (siehe Kapitel 4.1.5) immer an erster Stelle im Array und bestimmt das weitere Vorgehen der Paketverarbeitung.

Indiziert die Nachrichtenkennung eine Strukturantwort auf einer zuvor gesendete Anfrage, wird die `SetupDataStructure()`-Funktion der Hardware-Klasse aufgerufen, welche die Nachricht weiter verarbeitet. Die Verarbeitung der Strukturantwort und der Aufbau der Datenstruktur mitsamt Animationsknoten wird in Kapitel 5.1.6 näher behandelt. Die zweite Nachricht welche die Hardware-Klasse interpretieren kann, sind Nutzdatenpakete. Diese

werden nach Aufbau der Datenstruktur in einem festgelegten Intervall (Samplerate) von der Datenquelle gesendet und enthalten aktuelle Animationsdaten. Um einen Datenstau durch verspätete Nutzdatenpakete zu vermeiden, wird mit jedem `Read()`, der gesamte Inhalt des Puffers gelesen, aber nur das erste Nutzdatenpaket verarbeitet. Die restlichen Daten im Puffer, sofern vorhanden, werden verworfen.



Quellcode 7: Vereinfachter Quellcode-Ausschnitt zur Verarbeitung eines Nutzdatenpakets.

Bei eingehenden Nutzdatenpaketen, wird zuerst geprüft ob die boolsche Klassen-Variable `setupComplete` auf `true` gesetzt ist. Dies bestätigt den korrekten Aufbau der Datenstruktur und verhindert, dass Nutzdaten verarbeitet werden, bevor die notwendigen Animationsknoten vorhanden sind (siehe Kapitel Kapitel 5.1.6). Ist diese Bedingung erfüllt, wird das Nutzdatenpaket an die `UpdateNodeValues()` Funktion der Hardware-Klasse weitergereicht (siehe Quellcode 7). Diese prüft anhand der Nachrichtennummer ob der vom Benutzer gesetzte Grenzwert für das Alter eines Datenpakets überschritten wurde und verwirft dieses gegebenenfalls direkt. Befindet sich das Paket unterhalb des gesetzten Grenzwerts, wird als nächstes überprüft, ob die Anzahl der Datensätze im Nutzdaten-Array der vorgegebenen Anzahl in der Datenstruktur entspricht. Ist diese Bedingung ebenfalls erfüllt, wird das Array in einer Schleife durchlaufen und die `Double`-Werte mit der `cmp_read_double()`-Funktion der CMP-Bibliothek ausgelesen.

Die ausgelesenen double-Werte ersetzen die letzten Werte im nodeValues-Array der Hardware-Klasse, welches stets den aktuellsten Wert, für jeden Animationsknoten hält. Das Array ist eine Instanz der FBArrayTemplate-Klasse, welche ein Array mit dynamischer Größe, ähnlich des C++ Vector implementiert. Damit die Zuweisung der Nutzdaten korrekt ist, muss deren Reihenfolge mit der Reihenfolge der Datenstruktur übereinstimmen. Die eigentliche Verarbeitung der double-Werte für die Realtime-Darstellung und die Erzeugung der Keyframes bei einer Aufnahme, erfolgt in der Device-Klasse. Diese Vorgänge werden in Kapitel 5.1.7 behandelt.

5.1.6 Aufbau der Datenstruktur

Nach dem erfolgreichen Verbindungsaufbau zur Datenquelle (siehe Kapitel 5.1.4), sendet das Device eine Strukturanfrage an die Datenquelle. Diese enthält den in der Hardware-Klasse hinterlegten Namen der MotionBuilder Maschine (machineName), dient jedoch primär als Signal für die Datenquelle, die vorliegende Datenstruktur zu senden.

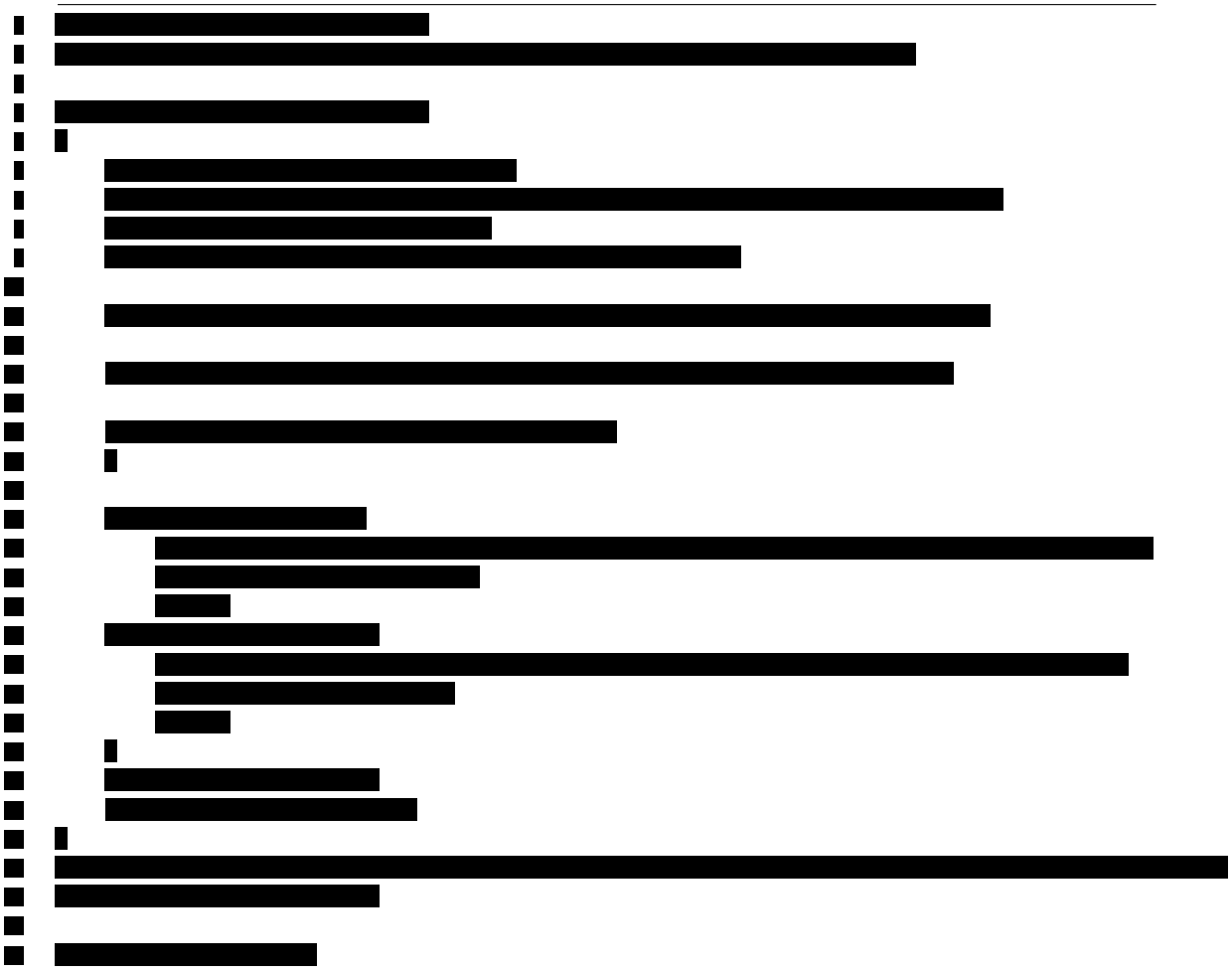
Die eingehende Strukturantwort wird, wie im vorherigen Kapitel erwähnt, an die Funktion SetupDataStructure() der Hardware-Klasse weitergegeben. Diese enthält den notwendigen Code zur Interpretation der Nachricht und dem Aufbau der Datenstruktur in MotionBuilder. Die Funktion hat über einen Parameter Zugriff auf das boxNodes-Array der Device-Klasse. Dabei handelt es sich um ein dynamisches Array (FBArrayTemplate), das die Animationsknoten (FBAnimationNode) des Device hält.

Im ersten Schritt liest die Funktion, die Nachrichtennummer und den mitgelieferten Namen der Datenquelle aus. Der Name liegt als Zeichenkette (UTF-8 Zeichensatz) vor und wird daher von CMP in mehreren Schritten ausgewertet: Zu Beginn wird die aktuelle Leseposition von CMP im Socketpuffer mit cmp_mem_access_get_pos abgefragt und in eine temporäre Variable geschrieben. Als nächstes wird die Länge der Zeichenkette mit cmp_read_str_size bestimmt und der Lesezeiger zurück auf die zuvor gespeicherte Leseposition gesetzt. CMP kann den String nun mit der Längenangabe über die Funktion cmp_read_str_size lesen. Dieses Vorgehen ist notwendig, um das Einlesen falscher Daten zu vermeiden.

Im nächsten Schritt wird die Anzahl der Datensätze ausgelesen, die dem eigentlichen Struktur-Array vorangestellt ist und von der Datenquelle als Integer-Wert serialisiert wurde. Der Wert wird dann mit der Größe des Array verglichen und bei Übereinstimmung wird mit der Auswertung fortgefahren. Sollte die Anzahl nicht übereinstimmen oder Fehler bei der Auswertung der Daten auftreten, wird der Vorgang abgebrochen und auf eine neue Strukturantwort mit korrektem Inhalt gewartet. Ist die Überprüfung hingegen erfolgreich, beginnt die Funktion mit der Auswertung des Struktur-Arrays.

Die Auswertung erfolgt in einer Schleife (siehe Quellcode 8). Für jeden Datensatz werden zwei Werte ausgelesen: Die Beschreibung des Werts in Form eines Strings, sowie die Datentypkennung in Form eines Integers. Anschließend wird basierend auf der Datentypkennung mit der AnimationNodeOutCreate()-Funktion der Device-Klasse, ein neuer Animations-

knoten für jeden Datensatz angelegt. Über ihre Parameter wird der Funktion mitgeteilt, um welchen Datentyp es sich handelt. Da alle Nutzdaten in MotionBuilder als `double`-Werte hinterlegt werden, wird anhand des Enum-Wert der Datentypkennung entschieden, ob es sich um einen echten `double`-Datensatz oder um einen `bool`-Datensatz handelt.



Quellcode 8: Vereinfachter Code-Ausschnitt zur Auswertung des Struktur-Arrays.

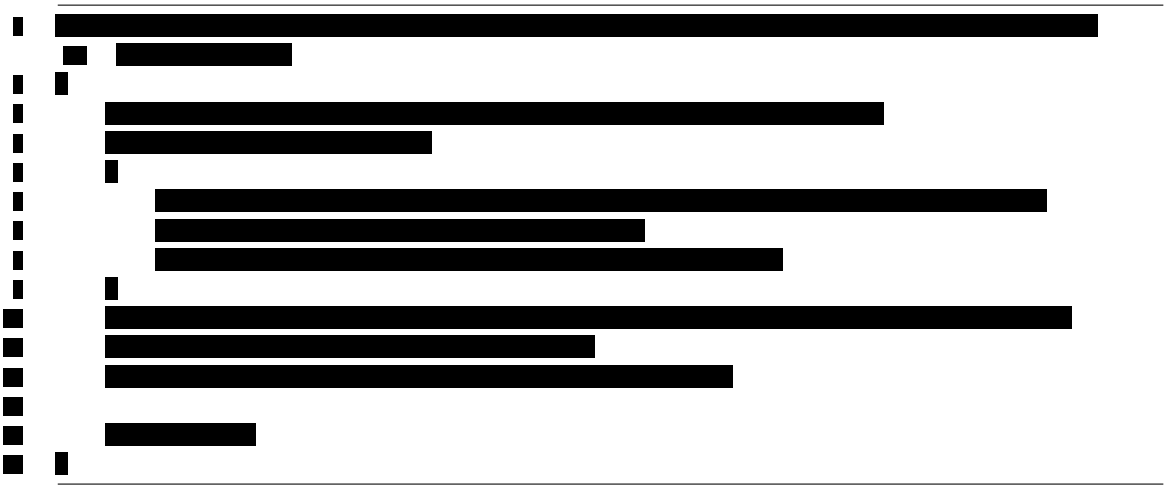
Die neu angelegten Animationsknoten werden zur späteren Verwendung dem `boxNodes`-Array der Device-Klasse hinzugefügt. Da das OR SDK leider keine Programmierschnittstelle anbietet, mit dem sich der Datentyp eines Animationsknoten bestimmen lässt, wird dieser zusätzlich in das dynamische Array `nodesTypes` geschrieben, welches ein Teil der Hardware-Klasse ist. So lässt sich auch noch zu einem späteren Zeitpunkt (siehe Kapitel 5.1.8), der Datentyp eines Animationsknoten bestimmen.

Nachdem für alle Datensätze im Struktur-Array ein Knoten angelegt wurde, wird im letzten Schritt auch für die Nachrichtennummer (`messageTag`) ein Animationsknoten angelegt und dem `boxNodes`-Array an letzter Stelle hinzugefügt. Wichtig ist, dass die Reihenfolge der Knoten im `boxNodes`-Array dabei nicht beeinflusst wird, da diese für die korrekte Zuweisung der Nutzdaten essentiell ist.

5.1.7 Nutzdatenverarbeitung

Die Verarbeitung der Animationsdaten findet in zwei unterschiedliche Vorgängen statt: Der erste Vorgang ist die Realtime-Evaluation der Werte zur Echtzeit-Darstellung in MotionBuilder. Der zweite Vorgang schreibt die Werte mit Keyframes (dt. *Schlüsselbilder*) in Animationskurven, sodass die aufgezeichneten Daten permanent gespeichert werden können.

Die Realtime-Evaluation der Daten findet in der `AnimationNodeNotify()`-Funktion der Device-Klasse statt und wird vom MotionBuilder *Animation-Thread* aufgerufen. Der Aufruf erfolgt dabei unabhängig von parallel laufenden Echtzeit-Aufgaben und erfolgt stets nach Bedarf, anstatt auf ein festgelegtes Intervall wie die Samplerate zurückzugreifen. Die Ausführung der Funktion ist besonders hoch priorisiert und sollte möglicht schnell abgeschlossen werden, da es sonst zu Problemen mit der Echtzeit-Darstellung kommen kann.

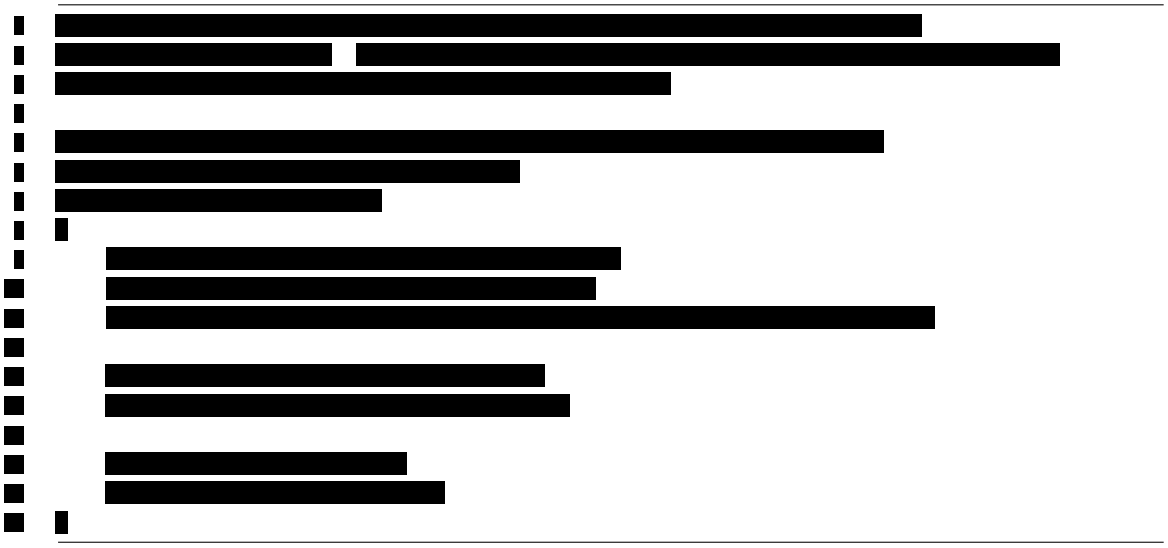


Quellcode 9: Evaluation der Nutzdaten für die Echtzeit-Darstellung der Animation.

Die Realtime-Evaluation nutzt die aktuellen Animationsdaten im `nodeValues`-Array der Hardware-Klasse, um die Echtzeit-Darstellung anzutreiben. Dazu werden die aktuellen `double`-Werte aller Animationsknoten in `boxNodes`, über die `Write()`-Funktion der Knoten, an die Realtime-Engine übergeben (siehe Quellcode 9). Diese verwendet die Werte zur Echtzeit-Darstellung der Animation. Damit die Realtime-Evaluation aufgerufen wird, muss der *Live*-Status des Device über dessen grafische Oberfläche zuvor gesetzt werden.

Soll die Aufzeichnung der Animationsdaten mithilfe von Keyframes erfolgen, muss der Anwender in der grafischen Oberfläche den *Recording*-Status setzen und die Aufnahme in der MotionBuilder Steuerung starten. Mit jedem Intervall-Aufruf der `DeviceIONotify()` werden nicht nur Daten vom Socket gelesen und verarbeitet, sondern im Fall von eingehenden Nutzdaten auch die Funktion `DeviceRecordFrame()` der Device-Klasse aufgerufen.

Die `DeviceRecordFrame()`-Funktion überprüft den Aufnahmestatus und übernimmt die Aufzeichnung der Daten durch das Setzen von neuen Keyframes in die Kurven der Animationsknoten. Dabei werden ähnlich wie bei der Realtime-Evaluation sämtliche Animationsknoten durchlaufen und mit den aktuellen Werten des `nodeValues`-Array versorgt. Statt `Write()` wird jedoch die `KeyAdd()`-Funktion der Knoten aufgerufen, welche je ein Keyframe, in den Datenbestand der Animationsknoten schreibt (siehe Quellcode 10). Die zeitliche Komponente des Keyframe ist dabei abhängig vom gewählten Sampling-Verfahren des Anwenders. Das Plugin verfügt insgesamt über drei verschiedene Sampling-Verfahren, die alle ihre Vor- und Nachteile haben.



Quellcode 10: Vereinfachter Beispielcode für die Aufzeichnung der Animationsdaten eines Frames, mithilfe des Nachrichtennummer-Sampling-Verfahrens.

Die genutzte OR SDK Vorlage enthält zwei Sample-Verfahren: Das Software-Zeitstempel Verfahren (`kFBSoftwareTimestamp`) setzt das Keyframe an die aktuelle Position des Players zum Zeitpunkt des Funktionsaufrufs. Dieses recht simple Verfahren hat den Nachteil, dass verspätete Pakete nicht immer zeitlich korrekt einsortiert werden und hohe Latenzen für verfälschte Animationen sorgen können.

Ist die Verbindung extrem zuverlässig und fehlerfrei, kann alternativ die Samplerate der Datenquelle als Anhaltspunkt genutzt werden (`kFBHardwareFrequency`). Hier werden die

Keyframes in dem von der Samplerate festgelegten Abstand gesetzt, was bei verlorenen, verspäteten oder unbrauchbaren Nutzdatenpaketen zwangsläufig zu Folgefehlern führt. Von der Verwendung dieses Verfahrens ist daher abzuraten.

Um die genannten Problematiken der Latenzen zu umgehen, wurde für die Umsetzung des Plugins ein drittes Verfahren entwickelt (`kFBHardwareTimestamp`, siehe Quellcode 10), dass standardmäßig verwendet wird und auf die Nachrichtennummer (`messageTag`) des letzten Datensatzes zurückgreift. Dabei wird zu Beginn der Aufnahme die Nachrichtennummer des ersten Keyframes als Referenz für die darauffolgenden Nutzdaten verwendet. Da jedes Paket die Nachrichtennummer um Eins inkrementiert und zudem genau ein Frame repräsentiert, lässt sie sich als zeitliche Zuordnung in MotionBuilder nutzen.

Im Falle eines schlechten Datensatzes entsteht an der entsprechenden Stelle in der Animationskurve eine Lücke, die vernachlässigbar ist. Verspätete Datensätze können durch ihre Nachrichtennummer trotzdem zeitlich korrekt als Keyframe gespeichert werden. Dies macht das Sampling-Verfahren deutlich robuster gegenüber potenzielle Fehlerquellen, als die zuvor besprochenen Verfahren der Beispielvorlage. Das Verfahren setzt jedoch auch voraus, dass die festgelegte Samplerate der Datenquelle mit dem Device übereinstimmt, da eine Zuordnung über die Nachrichtennummer sonst nicht möglich ist und in einer stark verfälschten Aufnahme resultiert.

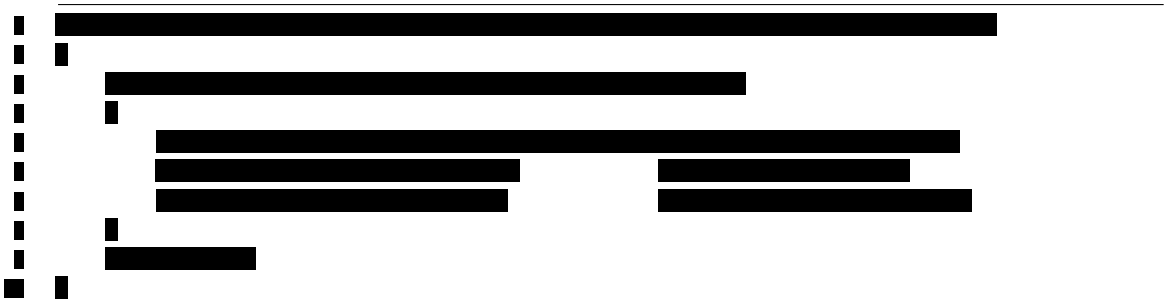
5.1.8 FBX Unterstützung

Aufgrund dynamischen Datenstruktur des Device, kann es zu Problemen beim Speichern und Laden von MotionBuilder-Szenen (FBX-Dateiformat) kommen. MotionBuilder speichert angelegte Keyframes automatisch, jedoch nicht die dazugehören Animationsknoten, welche im Regulärfall direkt im Code des Device definiert sind. Somit werden die Keyframes beim Öffnen einer FBX-Datei zwar mitgeladen, doch fehlen die zugehörigen Animationsknoten und ist Animation nicht darstellbar. Um dieses Problem zu umgehen, werden die beim Verbindungsaufbau angelegten Animationsknoten, explizit als *Attribute* in die FBX-Szene geschrieben bzw. geladen.

Die Implementierung der Logik erfolgt in den `FbxStore()` und `FbxRetrieve()` Funktionen der Device-Klasse. Speichert der Anwender die Szene, ruft MotionBuilder automatisch die `FbxStore()`-Funktion auf. Analog dazu wird die `FbxRetrieve()`-Funktion aufgerufen, sobald die Szene in MotionBuilder geladen wird. Beide Funktionen sind mit ihren Funktionsköpfen bereits im OR SDK registriert. Der Entwickler muss diese also lediglich implementieren um ihnen Funktionalität zu verleihen.

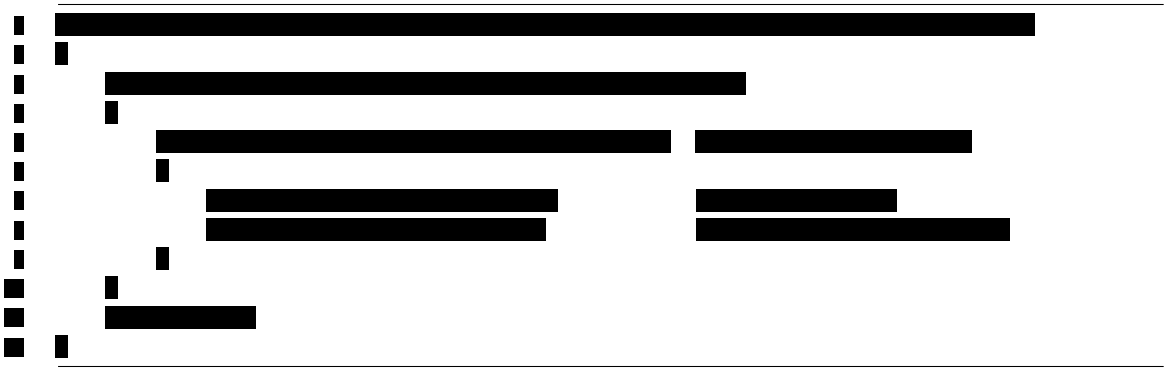
Zur Integration der Datenstruktur in das FBX-Format wird diese, ähnlich der Strukturantwort (siehe Kapitel 4.1.5), in Form einer Auflistung notiert. Die `FbxStore()`-Funktion (siehe Quellcode 11) schreibt dazu abwechselnd die Bezeichnung des Knotens als Zeichenkette und die Datentypkennung als Integer in die *Attributes*-Sektion des FBX-Formats. Dies erfolgt über die `FBFbxObject`-Instanz, welche als Parameter beider Funktionen vom OR SDK

übergeben wird. Über die Funktion `FieldWriteC()` des Objekts wird eine Zeichenkette in die Attribute geschrieben. `FieldWriteI()` schreibt hingegen einen Integer-Zahlenwert in die FBX-Szene. Um die Attribute zuordnen zu können, wird mit `FieldWriteBegin()` zuvor eine Bezeichnung als Zeichenkette gespeichert, die beim Auslesen der FBX-Datei als Einstiegspunkt für die Lesefunktion genutzt wird. Nachdem die gesamte Datenstruktur niedergeschrieben ist, wird der Attributsblock mit einem `FieldWriteEnd()` abgeschlossen.



Quellcode 11: Vereinfachtes Beispiel des Speichervorgangs eines FBX-Attributs.

Für den Lesevorgang beim Laden der FBX-Datei ist der Funktionsaufbau fast identisch (siehe Quellcode 12). Statt den verwendeten `Write`-Funktionen, werden die korrespondierenden `Read`-Funktionen der `FBFbxObject`-Instanz verwendet. Für das Auslesen einer Zeichenkette ist dies beispielsweise die `FieldReadC()`-Funktion. Neben der Datenstruktur wird weiterhin noch die Versionsnummer des Plugins in die FBX-Datei geschrieben, sodass veraltete Dateistrukturen abgefangen werden können. Zur Steuerung der Leseschleife für die Datenstruktur, muss zusätzlich die Anzahl der Daten als Zahlenwert gespeichert werden. Die Anzahl entspricht der im Protokoll definierten Datensatz-Anzahl der Strukturantwort.



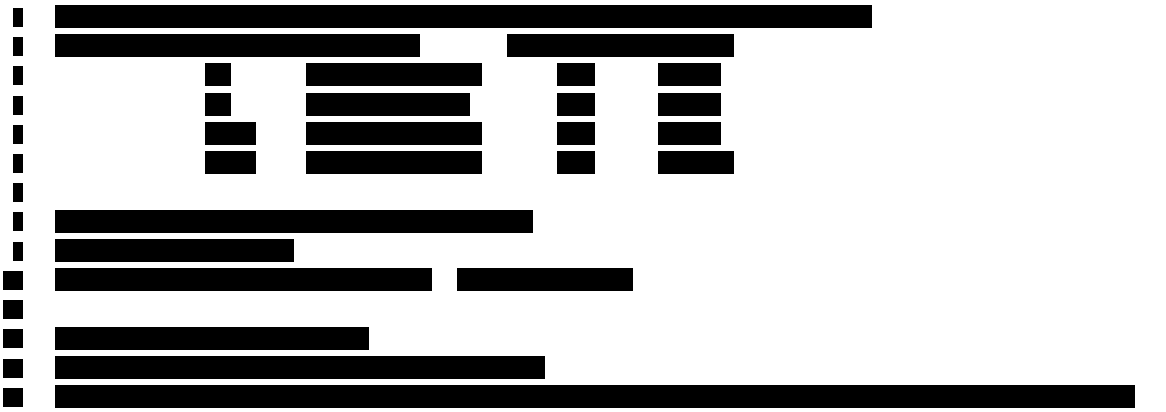
Quellcode 12: Vereinfachtes Beispiel des Ladevorgangs eines FBX-Attributs.

Konnte die Struktur erfolgreich aus der Datei gelesen werden, muss jeder Animationsknoten der Datenstruktur mit `AnimationNodeOutCreate()` wiederhergestellt werden. Wichtig ist außerdem, dass die Lese- und Schreibvorgänge stets während der Attribut-Phase der FBX-Verarbeitung stattfinden. So ist sichergestellt, dass die Attribute nur einmal geschrieben werden und auch wieder eingelesen werden, bevor die Animationsdaten der FBX-Datei entnommen werden. Dies ist notwendig, da die FBX-Funktionen, während der Verarbeitung einer Szenen-Datei mehrfach aufgerufen werden. Die aktuelle Phase lässt sich anhand des zweiten Funktionsparameters der `FbxStore()` und `FbxRetrieve()` Funktionen feststellen: `kFbxObjectStore`. Mit einer simplen `if`-Abfrage wird überprüft, ob der Wert `kAttributes` (Attribut-Phase) entspricht und die Logik zur Verarbeitung der Datenstruktur nur dann ausgeführt, wenn dies zutrifft.

Wurde eine korrekte Struktur für das Device eingelesen, muss abschließend die boolsche `strucLoadedFBX`-Variable der Hardware-Klasse auf *true* gesetzt werden. Dies verhindert, dass die Struktur der Datenquelle beim nächsten Verbindungsaufbau verarbeitet wird und veranlasst das Device stattdessen, die eingelesene FBX-Struktur zu verwenden.

5.1.9 Grafische Oberfläche

Die grafische Oberfläche (Layout-Klasse) der OR SDK Vorlage wird für die Umsetzung des Plugin-Prototypen komplett überarbeitet. Die Benutzeroberfläche des *OR Device Template* basiert auf einer Tab-Struktur, welche für die vergleichsweise kleine Oberfläche des Device zu aufwändig ist. Für das Device kommt stattdessen ein simpleres Layout mit nur einer Ansicht zum Einsatz, dass jedoch einige Bedienelemente der Vorlage übernimmt. Die Felder und Label für Samplerate, Sampling-Verfahren, IP-Adresse, Port und Protokoll werden übernommen und das Interface um Bedienelemente für die Gerätebezeichnung, den Filter-Grenzwert und das Debug-Protokoll ergänzt.



Quellcode 13: Beispielcode für die Erzeugung einer Benutzeroberfläche mit einem `FBEdit`-Textfeld und einer Callback-Funktion.

Der Aufbau des Interface erfolgt dabei komplett in der Layout-Klasse und ist in mehrere Schritte unterteilt (siehe Quellcode 13). Dabei gilt zu beachten, dass der Entwickler, wie in Kapitel 4.1.9) beschrieben, keinen Einfluss auf den linken Abschnitt des Oberfläche des Device hat. Dieser enthält die essentiellen Bedienelemente zur Steuerung eines Device und ist in Motionbuilder standardisiert. Der Aufbau des rechten Abschnitts ist hingegen komplett dem Entwickler überlassen. Im ersten Schritt (`UICreate()`) werden mit der `AddRegion()`-Funktion der `FBDeviceLayout`-Superklasse Regionen in der Oberfläche für die einzelnen Bedienelemente reserviert. Diese werden über ihre Namen, Dimensionen und Ursprung definiert. Anstatt absolute Angaben zu verwenden, können die Eigenschaften der Regionen alternativ auch in Relation zueinander (*Attachments*) gesetzt werden. Attachments vereinfachen die Gestaltung der Oberfläche enorm und werden für den Aufbau des Prototyp-Layouts verwendet.

Im nächsten Schritt werden den erzeugten Regionen, Bedienelemente zugewiesen. Dies erfolgt über die `SetControl()`-Funktion der Superklasse und setzt voraus, dass verwendete Bedienelemente zuvor in der Header-Datei der Layout-Klasse definiert wurden. Das OR SDK bietet eine Reihe unterschiedlicher Bedienelemente. Die Wichtigsten sind Label (`FBLabel`), Textfelder (`FBEdit`, `FBEditNumber`) und Dropdown-Listen (`FBList`). Nach der Zuweisung, nehmen die Bedienelemente die Eigenschaften (Dimensionen, Position) der Region an und werden in der Benutzeroberfläche dargestellt.

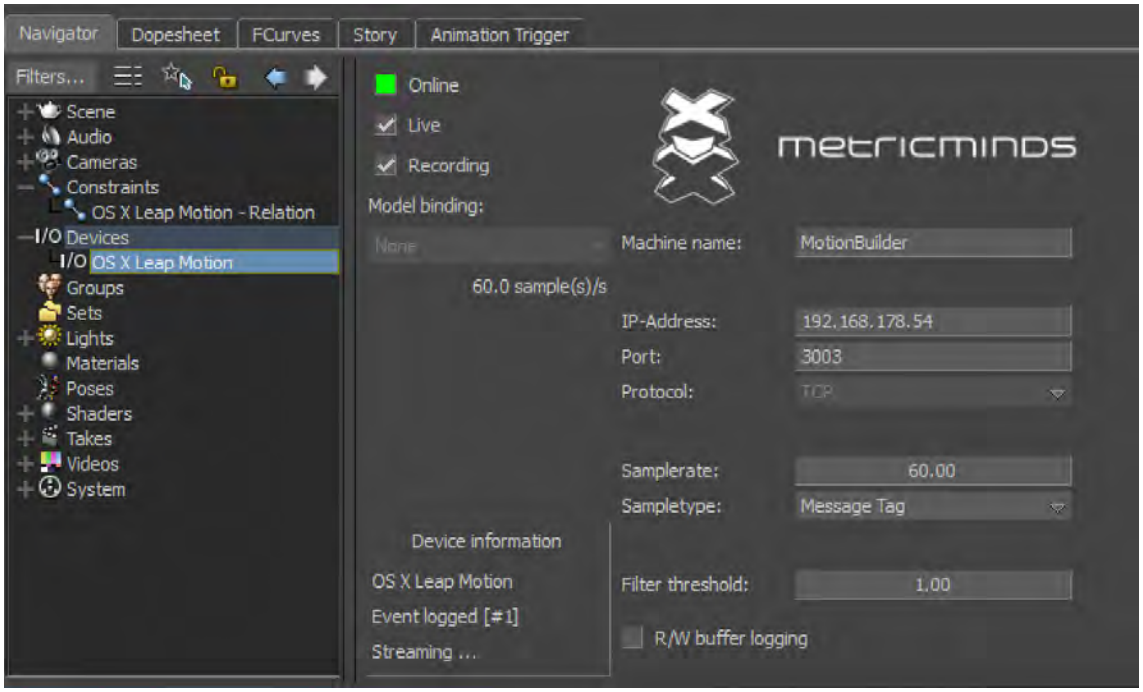


Abbildung 5.2: Screenshot der funktionsfähigen Benutzeroberfläche des Device (rechts), sowie des MotionBuilder-*Szenengraph* (links).

Im Anschluss an die Erzeugung der Benutzeroberfläche (siehe Abbildung 5.2), muss diese noch konfiguriert werden. Dies erfolgt in der `UIConfigure()`-Funktion der Layout-Klasse. Die Funktion setzt die notwendigen Attribute zur Darstellung der vorgesehen Inhalte und legt die sogenannten *Event-Callback*-Funktionen fest, welche im Falle von Benutzeränderungen aufgerufen werden. Dazu wird die `Add()`-Funktion auf das `onChange`-Atrribut des entsprechenden Bedienelements angewandt und die gewünschte Callback-Funktion für das *onChange*-Event als Parameter übergeben.

Die gesetzten Callback-Funktionen haben die Aufgabe, Eingaben durch den Benutzer an an die Device-Klasse weiterzuleiten und entsprechende Funktionsaufrufe zur Interpretation der Eingabe zu tätigen. Zusätzlich muss darauf geachtet werden, dass eingegebene Benutzerdaten stets in der Hardware- oder Device-Klasse gehalten werden. Die Layout-Instanz kann von MotionBuilder zur Laufzeit mehrfach aufgelöst und neu erstellt werden, sodass es zu Datenverlust innerhalb der Klasse kommt. Für die übernommenen Bedienelemente aus der OR SDK Vorlage, werden die vorgegebenen Callback-Funktionen mit marginalen Änderungen beibehalten. Für neu hinzugefügte Bedienelemente wird die Layout-Klasse um entsprechende Funktionen ergänzt.

5.2 Anwendungsbeispiel: iPad Kamerasteuerung

Die Umsetzung der ersten Datenquelle, einer iPad Anwendung zur Steuerung einer virtuellen Kamera, ist aufwändiger als die darauf folgende Umsetzung des Leap Motion-Tracking Servers. Allein die Entwicklung der App-Benutzeroberfläche erfordert einen enormen Aufwand, während ein Großteil des Quellcodes zur Kommunikation mit der Netzwerkschnittstelle für das zweite Anwendungsbeispiel übernommen werden kann.

Dieses Kapitel beschreibt die Implementierung der iPad Anwendung im Detail und behandelt wichtige Aspekte wie die Netzwerkkommunikation und Datenserialisierung. Am Ende des Kapitels hat der Leser das notwendige Wissen zur Umsetzung einer Protokoll-konformen Datenquelle für den Einsatz mit dem Plugin-Prototypen und kann ein Relation Constraint zur Interpretation eingehender Daten in MotionBuilder aufsetzen.

Abbildung 5.3: Die umgesetzte iPad Kamerasteuerung im Betrieb mit einer Testszene.

5.2.1 Entwicklungsumgebung

Zur Entwicklung des iPad Anwendungsbeispiels, wird die native Apple Entwicklungsumgebung *Xcode*[6] in der Version 7.2 verwendet. Xcode ist nur unter Mac OS X lauffähig und wird als Komplettpaket mit den aktuellsten SDKs für alle Apple Plattformen ausgeliefert. Die Software bietet ein alle wichtigen Tools für die Softwareentwicklung auf Apple Geräten und beinhaltet sogar Simulatoren für sämtliche iOS-Geräte.

[6]http://apple.co/1iJM6ju (Zuletzt abgerufen: 26.01.2016)

Projektname	Version	Link
MFLJoystick	commit e138fd1	github.com/MattFoley/MFLJoystick
GCDKit	1.1.5	github.com/JohnEstropia/GCDKit
MotionKit	0.8.1	github.com/MHaroonBaig/MotionKit
NetUtils for Swift	1.3.2	github.com/svdo/swift-netutils
SwiftSockets	0.20.6	github.com/AlwaysRightInstitute/SwiftSockets
Carthage	0.11.0	github.com/Carthage/Carthage

Tabelle 5.1: Verwendete Open Source Projekte für die Implementierung des iPad Anwendungsbeispiels. Die Links wurden zuletzt abgerufen am 29.01.2016.

Zur Umsetzung wird das iOS SDK in Version 9.2 und die Programmiersprache *Swift*[7] in Version 2.1 verwendet. Swift ist eine moderne Programmiersprache von Apple, die als Alternative zu *Objective-C* entwickelt wird. Swift bietet eine intuitivere Syntax und unterstützt aktuelle Programmiertechniken aus anderen Sprachen. Zur Einbindung von zusätzlichen Frameworks in das Projekt wird der *Carthage*[8]-Paketmanager für Xcode eingesetzt. Dabei handelt es sich um Open Source Projekt von Syo Ikeda, dass die Integration von Swift-Frameworks deutlich vereinfacht und Frameworks automatisch für den Entwickler aktualisiert.

Bei der Implementierung des Konzepts wird auf mehrere Open Source Frameworks zurückgegriffen, welche die Verwendung der Apple-Programmierschnittstellen vereinfachen oder neue Funktionalität einführen, die in Swift so noch nicht integriert wurde (siehe Tabelle 5.1).

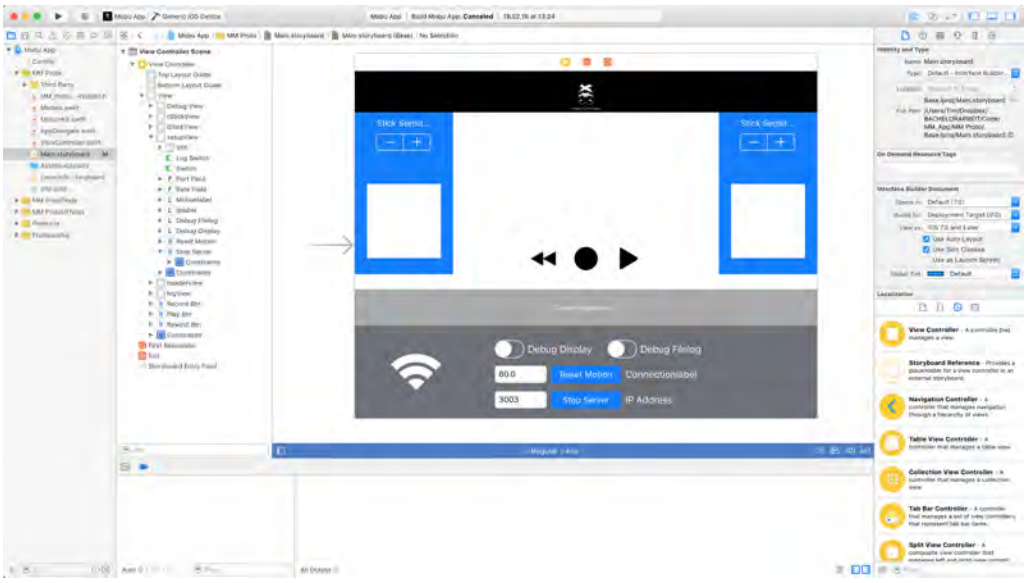


Abbildung 5.4: Apple Xcode 7.2 mit geöffnetem *Storyboard*-Editor.

[7]http://apple.co/1DgqEVo (Zuletzt abgerufen: 26.01.2016)
[8]https://github.com/Carthage/Carthage (Zuletzt abgerufen: 27.01.2016)

Als Basis für das Projekt wird in Xcode die „Single View Application“ Vorlage für iOS gewählt (siehe Abbildung 5.4). Diese bietet die optimale Grundlage für die Implementierung, da für die Umsetzung des Konzepts lediglich ein *View* (Benutzeroberfläche) benötigt wird, auf dem sämtliche Bedienelemente angeordnet sind. Enthalten ist ein sogenanntes *Storyboard*, für die Erzeugung der Benutzeroberfläche, sowie eine Swift-Datei für den Quellcode des *View-Controllers*. Letzterer implementiert sämtliche Programmlogik des Anwendungsbeispiels.

5.2.2 Motion Tracking

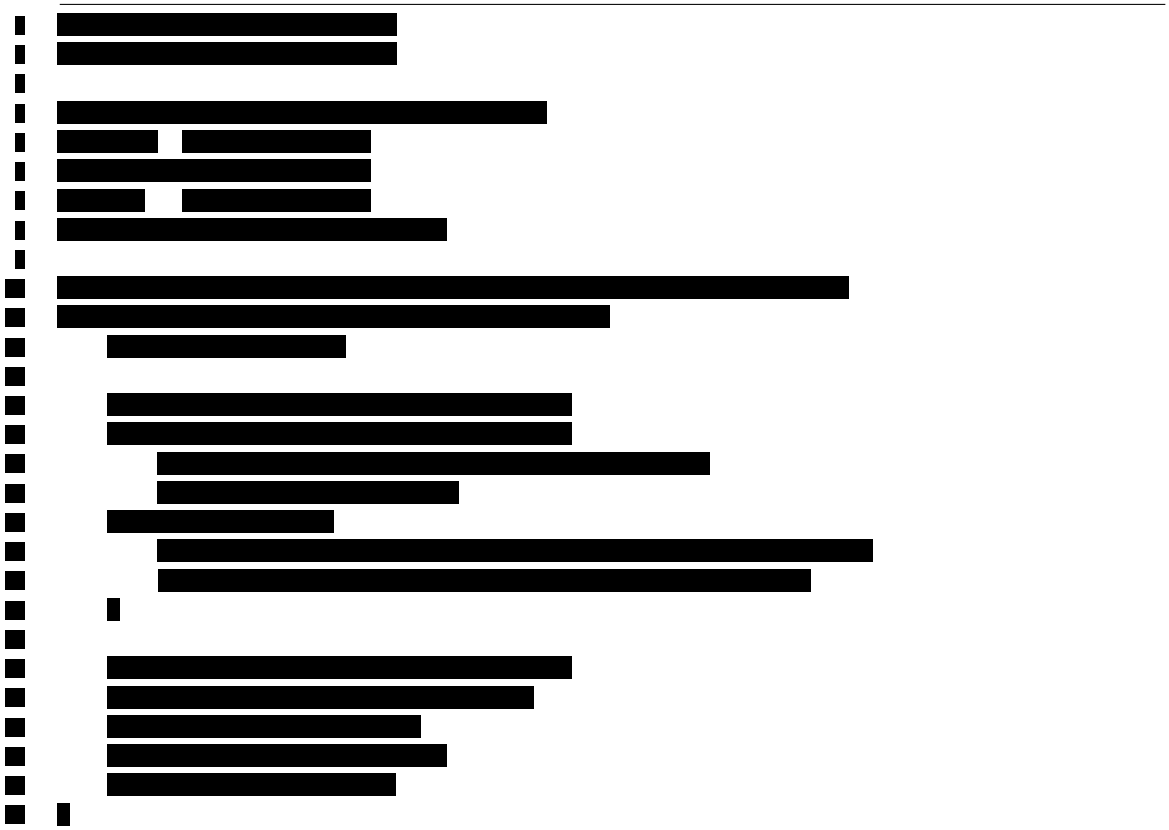
Bei der Implementierung der Bewegungsaufzeichnung über die Inertialsensoren des iPads, wird auf ein Open Source Projekt zurückgegriffen, das die Umsetzung des Programmabschnitts deutlich vereinfacht. Zur Abfrage der Sensordaten wird das GitHub Projekt *MotionKit*[9] verwendet. Es handelt sich dabei um eine Hüllklasse für das *CoreMotion Framework* von Apple, welches die Programmierschnittstellen für die Bewegungssensoren der iOS-Geräte beinhaltet. Das Projekt verbessert den Zugriff auf die Bewegungsdaten durch vereinfachte Funktionsaufrufe und kümmert sich um die Parallelisierung des Codes, sodass der Entwickler sich keine Gedanken um Threadverwaltung bei Abfrage von Bewegungsdaten machen muss.

Zur Einbindung von MotionKit wird die `MotionKit.swift` Datei aus dem Github Master-Branch (Hauptentwicklung) heruntergeladen und in das iOS Projekt mit XCode importiert. Da die aktuelle Version[10] (Stand: 26.01.2016) jedoch noch nicht Swift 2.0 kompatibel ist, ist eine Anpassung an die neue Swift Syntax notwendig. Hierzu kann der Xcode-interne Syntax Konvertierer genutzt werden, der über das *Edit*-Menü zur Verfügung steht und den Code fehlerfrei anpassen sollte. Anschließend wird eine MotionKit-Instanz im View-Controller initialisiert.

Der Ausschnitt in Quellcode 14 legt eine MotionKit-Instanz an und fragt die Gerätelage im dreidimensionalen Raum alle zehn Millisekunden ab. Die verwendete MotionKit-Funktion greift auf die entsprechende CoreMotion Programmierschnittstelle zu, welche ein `CMMotionDevice`-Objekt liefert, dass die aufbereiteten Sensordaten (siehe Kapitel 4.2.3) vom iOS SDK enthält. MotionKit bezieht ein `CMAttitude`-Objekt von `CMMotionDevice` und gibt dieses als Rückgabeparameter an den Entwickler weiter. Das `CMAttitude`-Objekt enthält alle aufbereiteten Lagedaten des iPad in Form von Quaternionen, Rotationsmatrizen und Winkeln im Bogenmaß.

Wichtig ist der Einbezug einer Referenzlage (Nulllage) des iPads bei der Auswertung der Daten. Diese wird einmalig bei der Aktivierung des Timers gesetzt, indem das `CMAttitude`-Objekt als Variable gespeichert wird. Bei folgenden Abrufen der Gerätelage wird dann die Rotationsmatrix des gespeicherten Referenzobjekts invers mit der aktuellen Matrix multipliziert. Dadurch errechnet sich die Gerätelage mit der gespeicherten Lage als Bezugslage.

[9] https://github.com/MHaroonBaig/MotionKit (Zuletzt abgerufen: 26.01.2016)
[10] http://bit.ly/1Qyx51A (Zuletzt abgerufen: 26.01.2016)

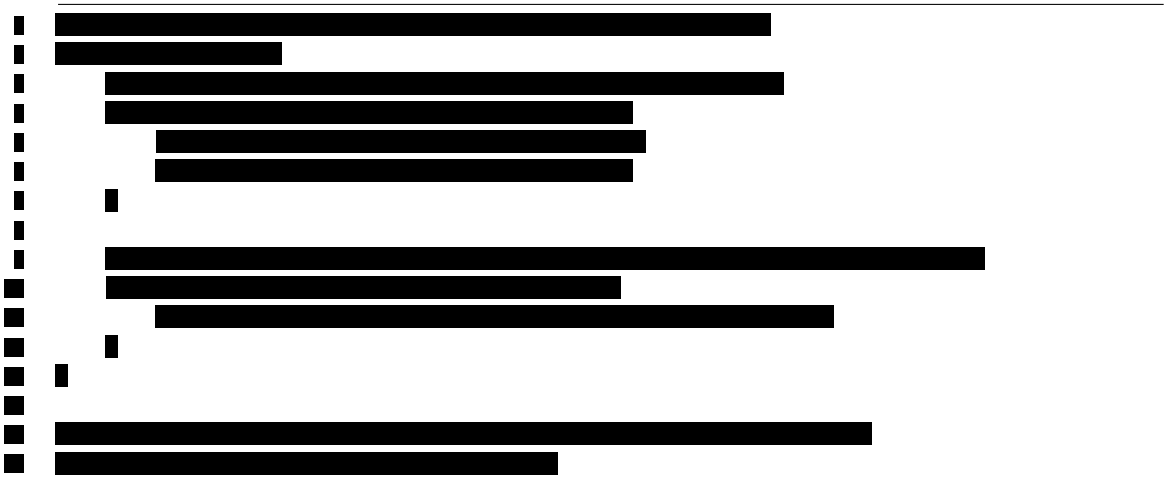


Quellcode 14: Bestimmung der iPad Roll-Pitch-Yaw Winkel mit MotionKit unter Berücksichtigung der Referenzlage.

Zur Konvertierung der Winkel aus dem Bogenmaß ins Gradmaß, ist eine einfache Umrechnung erforderlich. Der Wert im Bogenmaß wird um den Faktor 180 multipliziert und durch die *Pi*-Konstante geteilt. Um die Konvertierung zu erleichtern und die entstehenden Winkel auf eine sinnvolle Dezimalstelle runden zu können, wird eine *Extension* (Erweiterung) für den *Double*-Datentyp geschrieben. Diese erweitern alle Instanzen eines Objekttyps im gesamten Projekt um zusätzliche Funktionen, ohne dass der Originalcode der Objekt-Klasse verändert werden muss (siehe Quellcode 15).

5.2.3 Grafische Benutzeroberfläche

Die Implementierung der grafischen Benutzeroberfläche für die iPad App ist dank des *Storyboard*-Systems von Xcode sehr intuitiv und komfortabel. Der Storyboard-Editor ermöglicht die Erstellung einer grafischen Oberfläche mithilfe von „Drag & Drop"-Bedienung nach dem WYSIWYG-Prinzip (*„What you see is what you got"*). Software die mit dem iOS SDK entwickelt werden, unterliegt stets dem *Model-View-Controller* Entwurfsmuster.



Quellcode 15: Erweiterung des Double-Datentyps um zwei mathematische Funktionen.

Der Name Storyboard leitet sich von der gleichnamigen Filmtechnik ab, die eine Abfolge von Szenen verbildlicht. Im Xcode-Storyboard sind dies eine Abfolge von *Views* (Ansichten / Grafische Oberflächen), die mit animierbaren Übergängen (*Segues*) verknüpft sind.

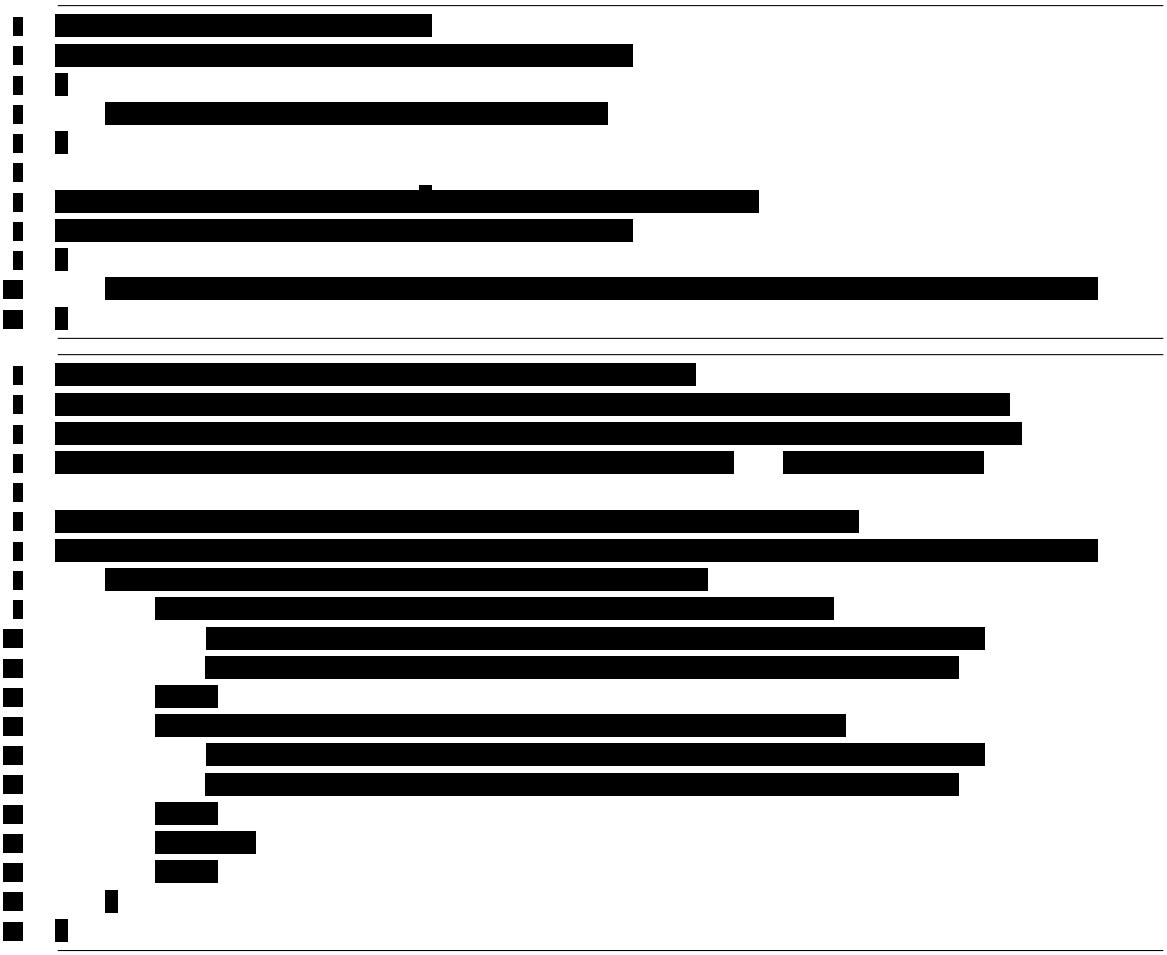
Für die iPad App wird lediglich ein View benötigt und alle notwendigen Bedienelemente, mit Ausnahme der virtuellen Analogsticks, sind bereits Teil des regulären iOS SDK. Für die Implementierung der Joysticks wird auf das Open Source Projekt *MFLJoystick*[11] von Matt Foley zurückgegriffen. Das Projekt bietet einen simplen virtuellen Joystick, der als Subklasse des `UIView` implementiert wurde. Zur Einbindung in das Projekt wird der aktuelle Quellcode von Github heruntergeladen und in das Xcode Projekt importiert. Wie schon zuvor bei MotionKit, benötigt der aktuelle Code von MFLJoystick[12] (Stand: 26.01.2016) eine Anpassung an die neue Swift 2.0 Syntax, die ebenfalls über den Swift-Konvertierer durchgeführt wird.

Nach dem erfolgreichen Hinzufügen des MFLJoystick Quellcode zum Projekt, muss für die beiden geplanten Joysticks je ein `UIView` im Storyboard erstellt werden. Im *Utility*-Fenster wird anschließend für beide Views „`MFLSwiftJoystickImplementation`“ als „*Benutzerdefinierte Klasse*“ festgelegt. Die beiden Views werden nun nach Programmstart als funktionsfähige Joysticks dargestellt, doch fehlt noch die notwendige Verbindung zum Quellcode des View-Controller, um die Positionen der Joysticks (X,Y) abzufragen. Dazu implementiert der View-Controller das `MFLSwiftJoystickDelegate`-Protokoll, welches die Delegation (*View-Controller*) bei Änderung der Joystick-Position benachrichtigt.

Da der Originalcode jedoch nicht für den parallelen Betrieb mehrerer Joysticks vorgesehen ist, wird das Protokoll um die Instanz des Joysticks ergänzt. Nach der Anpassung lässt sich

[11] https://github.com/MattFoley/MFLJoystick (Zuletzt abgerufen: 26.01.2016)
[12] http://bit.ly/1OWhqo3 (Zuletzt abgerufen: 26.01.2016)

bestimmen, welcher der Joysticks den View-Controller benachrichtigt hat und Daten liefert (siehe Quellcode 16). Abschließend werden die Views im Storyboard über das *Utility*-Fenster mit ihren `IBOutlet` Referenzen im Code verknüpft, sodass eine Verbindung zwischen View (Interface) und View-Controller (Code) hergestellt ist.

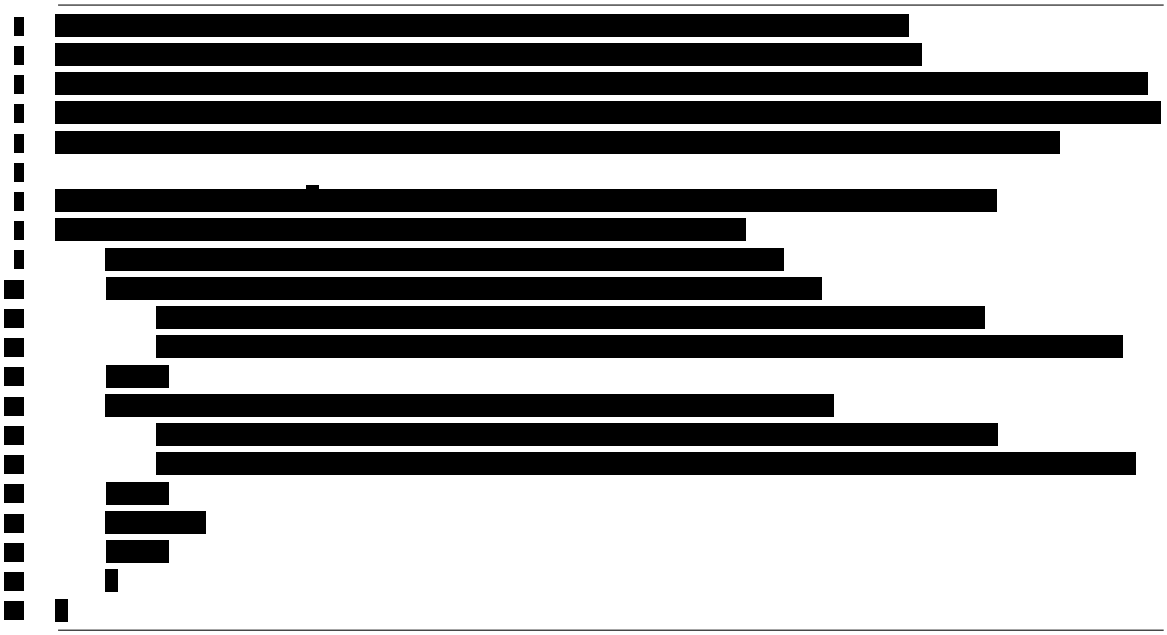


Quellcode 16: Oben: Erweiterung der MFLJoystick Protokollfunktion um die Joystick-Instanz als Parameter, sodass der Delegationsaufruf einer Instanz zugeordnet werden kann. Unten: Implementierung des anpassten Protokolls im View-Controller.

Für die restlichen Bedienelemente der Benutzeroberfläche können native Interface-Elemente des iOS Betriebsystems genutzt werden. Die Steuerelemente zur Empfindlichkeit der Joysticks werden mithilfe eines `UIStepper` umgesetzt, die ihre Zahlenwerte bei Benutzerinteraktion um einen bestimmten Intervall inkrementieren oder dekrementieren.

Der Nutzer soll über die Stepper zwischen Werten von 1 und 100 in Einser-Schritten wählen können. Um bei Änderungen den aktuellen Wert der Stepper an den View-Controller zu über-

mitteln, wird im Storyboard, dass *Value Changed*-Event des Steppers mit einer `IBAction` verknüpft. Dabei handelt es sich wie schon beim `IBOutlet` um eine Verbindung zwischen View-Controller Code und den Bedienelementen im Storyboard (View). Anstatt jedoch eine Objekt-Instanz im Storyboard einer Code Variable zuzuweisen (`IBOutlet`), ruft die `IBAction` eine Funktion auf, sobald das verknüpfte Interface-Event ausgelöst wird. Um den View-Controller Code kompakter zu halten, referenzieren beide Stepper die gleiche Funktion und werden über einen Vergleich ihrer Instanzen mit dem Event-Auslöser identifiziert (siehe Quellcode 17).



Quellcode 17: Implementierung einer `IBAction`-Funktion, welche auf Werteänderungen der Joystick-Stepper reagiert.

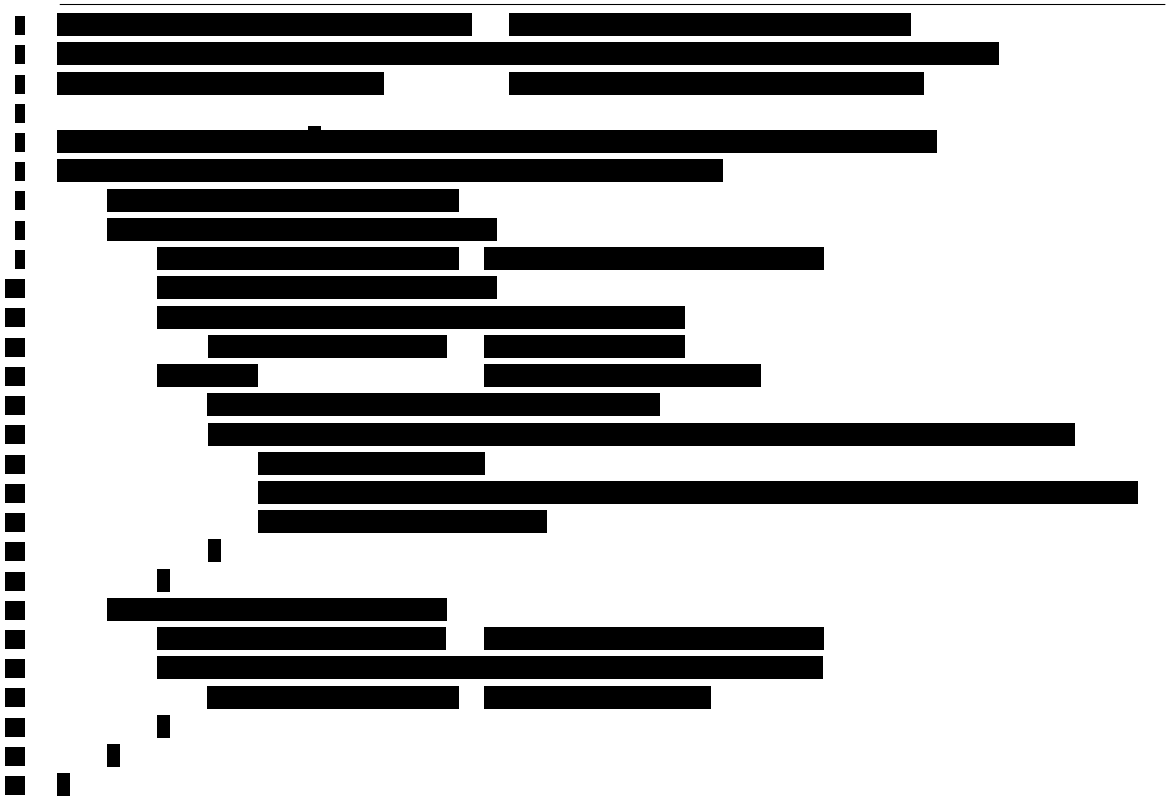
Für die Umsetzung der Debug-Anzeige zur Darstellung von aktuellen Variablenwerten wird ein `UILabel` (Label) mit einem `UIView` und einem `UISwitch` (Schalter) kombiniert. Der On- / Off-Status des Switch zur Ein- und Ausblendung der Debug-Anzeige genutzt. Bei Betätigung des Switchs (Event: *Value Changed*) wird die verknüpfte `IBAction` aufgerufen. Das Label wird zur besseren Darstellung als Unterelement des `UIView` angeordnet, sodass beide Elemente zeitgleich über das `hidden`-Attribut des View ausgeblendet werden können.

Für die Echtzeit-Aktualisierung des Labels kommt ein `GCDTimer` aus dem Open Source Projekt *GCDKit*[13] in der Version 1.1.5 zum Einsatz. GCDKit ermöglicht Entwicklern eine komfortablere Ansteuerung der *Grand Central Dispatch*[14] Programmierschnittstelle von Apple. Diese kümmert sich um eine effiziente Verteilung von Rechenaufgaben über die

[13] https://github.com/JohnEstropia/GCDKit (Zuletzt abgerufen: 27.01.2016)
[14] http://apple.co/20skZK8 (Zuletzt abgerufen: 27.01.2016

verfügbaren Prozessorkerne, sodass mehrere Aufgaben parallel abgearbeitet werden können. Aufgaben lassen sich Prioritäten zuweisen und das Grand Central Dispatch steuert die parallele Ausführung des Codes per *Multithreading*. GCDKit lässt sich über den *Carthage*[15]-Paketmanager für Xcode, als Swift-Framework in das Projekt einbinden.

Die `GCDTimer`-Klasse des GCDKit-Projekts ermöglicht eine parallelisierte, zeitgesteuerte Code-Ausführung in einem festgelegten Intervall. Für die Aktualisierung ist es notwendig den Timer der `GCDQueue.Main` Aufgaben-Warteschlange einzuordnen, da Änderungen an der Benutzeroberfläche immer im Hauptthread ausgeführt werden müssen. Anderenfalls zeigen diese keine Wirkung. Zur Einsparung von Systemressourcen wird der Timer lediglich alle 100ms ausgeführt. Dies ist zwar zu langsam für eine Echtzeit-Darstellung aller Änderungen, offenbart jedoch genug Informationen zur Fehlersuche im aktiven Betrieb der Anwendung.

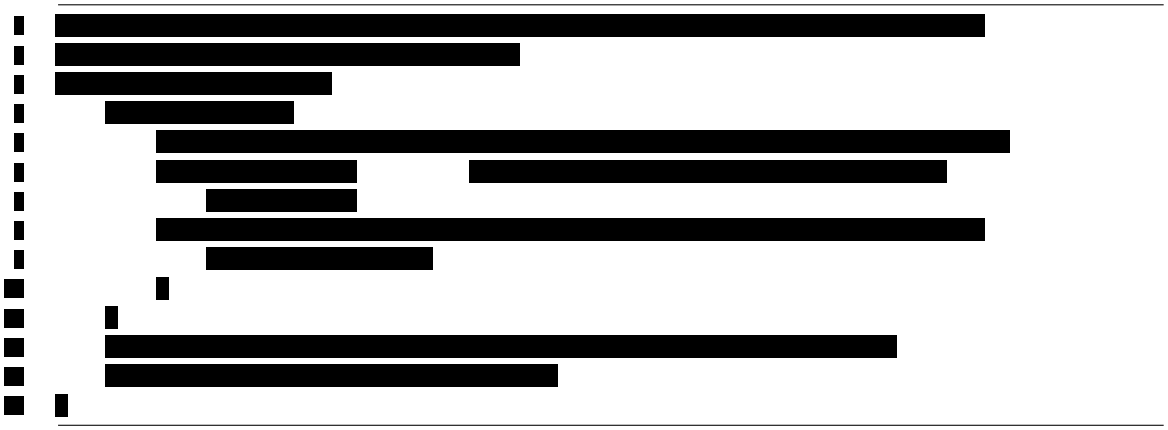


Quellcode 18: Implementierung der Debug-Anzeige mit *GCDKit* und einem Update-Intervall von 100 Millisekunden. Der Code wird von *Grand Central Dispatch* parallelisiert und läuft im Hauptthread.

[15]https://github.com/Carthage/Carthage (Zuletzt abgerufen: 27.01.2016)

Für tiefgreifendes Debugging steht eine Logging-Funktion zur Verfügung, die Protokoll über sämtlichen Datenverkehr führt. Dieser wird in eine Textdatei geschrieben, die der Anwender mit angeschlossenem Gerät über den *Dokumente*-Bereich der *Apple iTunes*[16] Software am Computer abrufen kann. Aktiviert wird die Logging-Funktion ebenfalls über ein `UISwitch`. Wichtig ist, dass in der `Info.plist` Datei des Xcode Projekts der Schlüssel „Application supports iTunes file sharing“ mit „YES“ eingetragen ist. Ist dies nicht der Fall, zeigt iTunes das Dokumentenverzeichnis der App nicht an.

Sämtliche Buttons der Benutzeroberfläche (MotionBuilder Steuerung, Serverstatus, Motion Kalibrierung) werden mit einfachen `UIButton`-Elementen implementiert, die beim *Touch Up Inside*-Event eine `IBAction`-Funktion im View-Controller aufrufen. Für die Eingabe von Samplerate und Portnummer werden dem Benutzer zwei `UITextField`-Elemente zur Verfügung gestellt, deren *Text*-Attribut in Zahlenwerte konvertiert wird und dem Rest des Programms über sogenannte *Computed Properties* (dt. „berrechnete Eigenschaften“) zur Verfügung gestellt. Bei letzterem handelt es sich um Variablen, die keinen Wert speichern, sondern stattdessen Getter- und Setter-Funktionen aufrufen. Sie eignen sich besonders, um den Quellcode einfach und leserlich zu halten (siehe Quellcode 19).



Quellcode 19: Beispiel für ein *Computed Property* in Swift am Beispiel der Portnummer.

Zur Anzeige der aktuellen IP-Adresse des iPads wird ein *UILabel* verwendet. Da Swift 2.1 noch keine native Programmierschnittstelle zum Abruf von Netzwerkinterface-Informationen bietet, wird auf das Open Source Projekt *NetUtils for Swift*[17] von Stefan van den Oord zurückgegriffen. Die verwendete Version 1.3.2 bietet komfortablen Zugriff auf alle Netzwerkinterfaces und deren Attribute (siehe Quellcode 20). Zum Abruf der Informationen greift das Framework auf Objectivce-C Code zurück, der die iOS-Programmierschnittstellen im Systemheader `ifaddrs.h` abfragt. NetUtils for Swift lässt sich mit Carthage in das Projekt einbinden.

[16] http://www.apple.com/de/itunes/ (Zuletzt abgerufen: 27.01.2016)
[17] https://github.com/svdo/swift-netutils (Zuletzt abgerufen: 27.01.2016)

Quellcode 20: Bestimmung der IP-Adresse mithilfe des *NetUtils for Swift* Framework.

Da für dieses Anwendungsbeispiel lediglich der Betrieb der Anwendung auf einem iPad (Tablet) vorgesehen ist, muss der kleine Bildschirm des iPhone bei der Entwicklung nicht berücksichtigt werden und die Bedienelemente im Storyboard können speziell für die Anforderungen des iPads angepasst und ausgerichtet werden. Die Positionierung der Bedienelemente wird im Storyboard mit *Constraints* verankert, die das Verhältnis zwischen den einzelnen Elementen des Views beschreiben. Die Anwendung unterstützt lediglich den Landscape-Modus (horizontale Haltung) des iPads (siehe Abbildung 5.5).

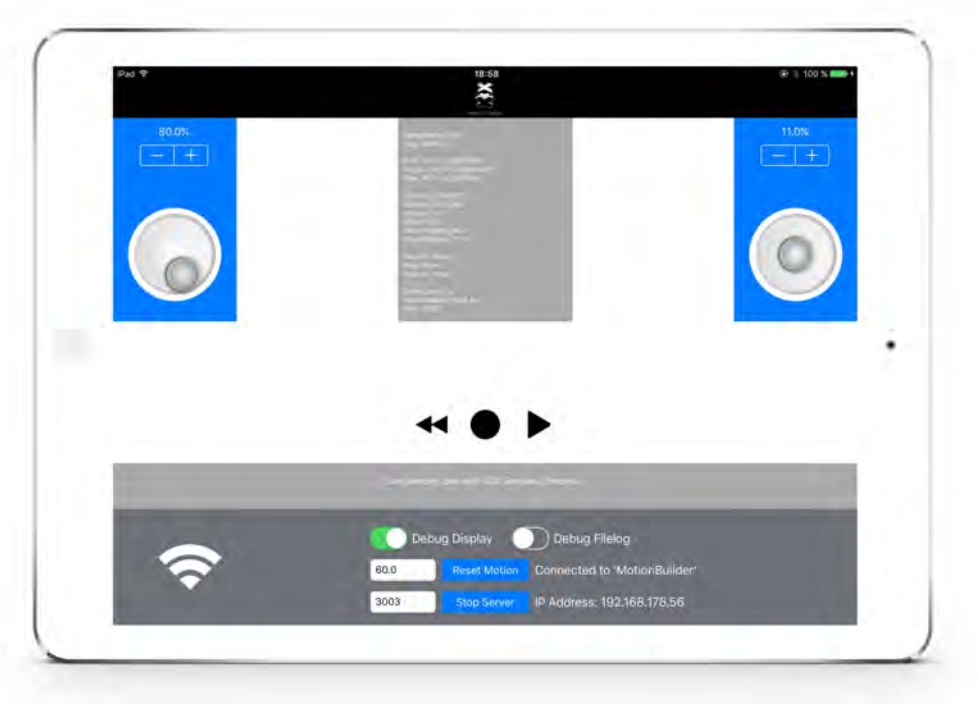


Abbildung 5.5: Screenshot der iPad-Benutzeroberfläche im Betrieb unter iOS 9.2.

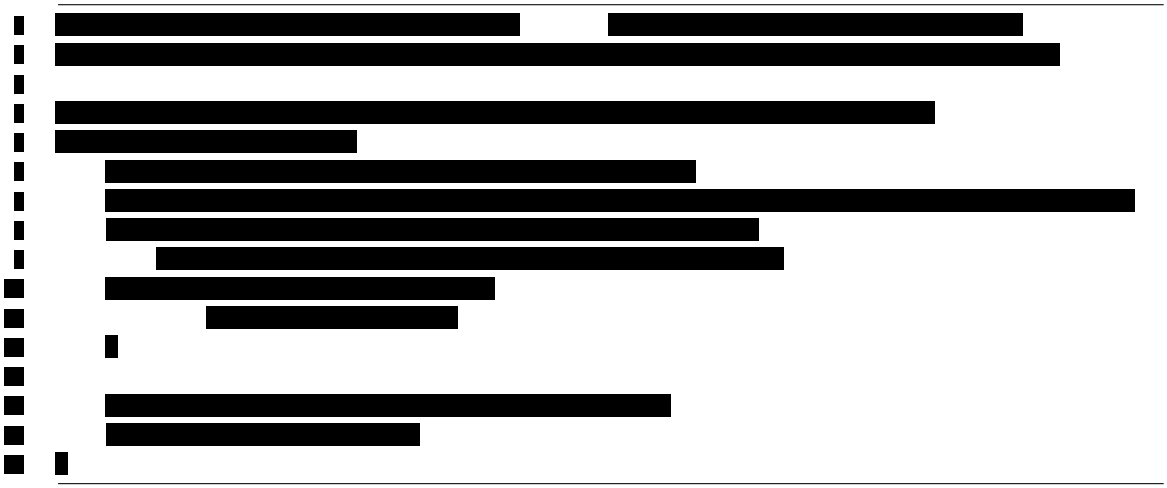
Quelle: http://bit.ly/1ZMM2Ob (Creative3x Ltd., iPad Mockup modifiziert, Zuletzt abgerufen: 25.01.2016)

5.2.4 Netzwerkkommunikation

Zur Netzwerkkommunikation mit Sockets auf der iOS-Plattform gibt es neben den reinen Apple Programmierschnittstellen eine ganze Reihe sehr ausgereifter Open Source Frameworks, wie z.B. *CocoaAsyncSocket*[18] von Robbie Hanson. Leider ist die Einbindung der aktuellen Version 7.4.2 (Stand: 27.01.2016) in ein Swift Projekt nicht zu empfehlen, da das Projekt noch komplett auf Objective-C basiert und es zu Inkompatiblitäten kommen kann.

Um die genannten Probleme zu vermeiden wird als Basis für die Socket-Programmierung das *SwiftSockets*[19] Open Source Projekt von Helge Heß verwendet. Das gesamte Framework basiert auf reinem Swift-Code, anstatt eine Objective-C basierte Brückenlösung zu bieten und kann mit Carthage eingebunden werden. Wie im Konzept spezifiziert (siehe Kapitel 4.1.6 und 4.2.5) implementiert die Anwendung als Datenquelle den Server-Socket der Verbindung. Dazu wird ein `PassiveSocketIPv4` an der spezifizierten Portnummer für den Server angefragt (siehe Quellcode 21).

Konnte der Port vom System für die Anwendung freigegeben werden, wird auf dem neu erstellten Socket die `listen()`-Funktion aufgerufen, welche den Server startet und auf eingehende Verbindungen wartet (siehe Quellcode 22). Um zu verhindern dass dies den Rest der Codeausführung blockiert, wird die Funktion mit GCDKit parallelisiert. Kommt eine Verbindungsanfrage von MotionBuilder wird diese akzeptiert und für die spätere Verwendung in einer Referenzvariable gespeichert. Die Implementierung sieht vor, dass maximal ein Client gleichzeitig verbunden ist.

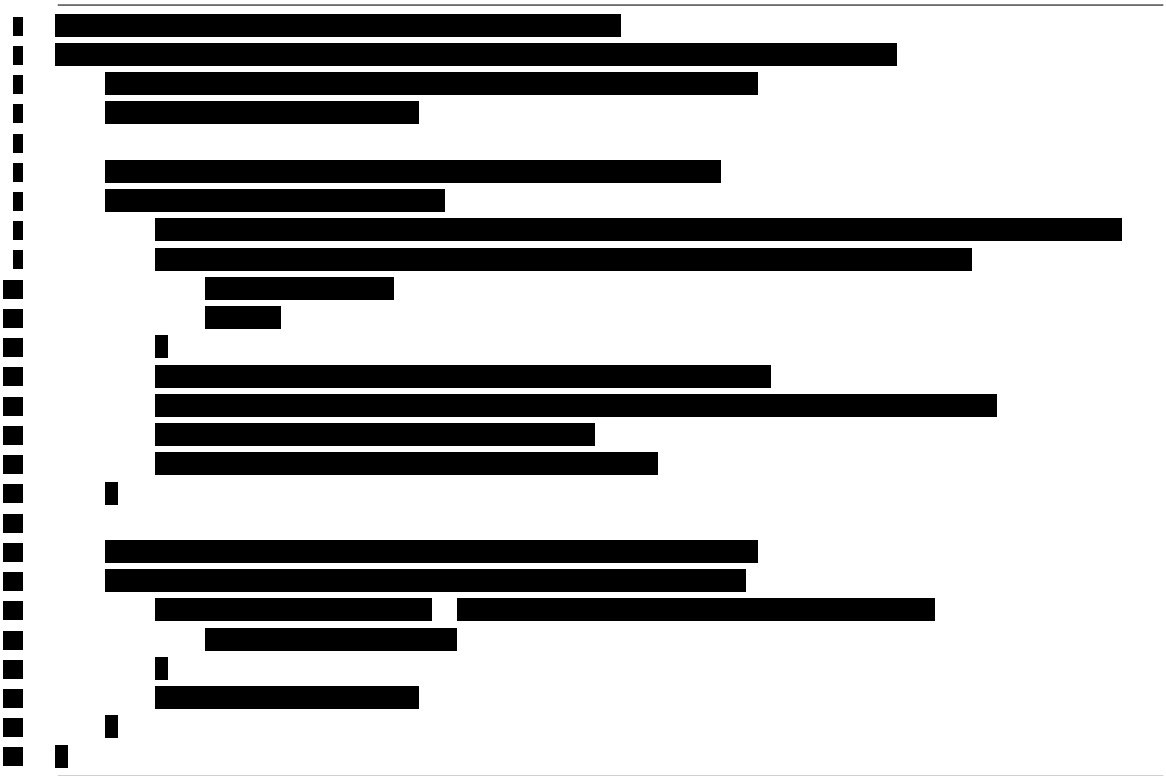


Quellcode 21: Erstellung eines Server-Socket (TCP) mit *SwiftSockets*.

[18] https://github.com/robbiehanson/CocoaAsyncSocket (Zuletzt abgerufen: 27.01.2016)
[19] https://github.com/AlwaysRightInstitute/SwiftSockets (Zuletzt abgerufen: 27.01.2016)

Bei eingehenden Daten auf dem Client-Socket wird die `onRead()`-Funktion aufgerufen, die den eingelesenen Datenpuffer mit dem *MessagePack.swift*-Framework[20] von Alexsander Akers verarbeitet (siehe Quellcode 22). Das Framework ist eine Implementierung der MessagePack Notation (siehe Kapitel 4.1.8) und basiert auf reinem Swift-Code. Der eingelesene Bytecode wird mit der `unpack()`-Funktion des Frameworks in ein `MessagePackValue` konvertiert, aus dem die einzelnen Bestandteile der Nachricht entnommen werden können. Der Inhalt der `onClose()`-Funktion des Server-Sockets spezifiziert das Vorgehen beim Schließen der Socketverbindung. Ist ein Schreib-Timer vorhanden, wird dieser pausiert und die Socketverbindung aus der Referenzvariable entfernt.

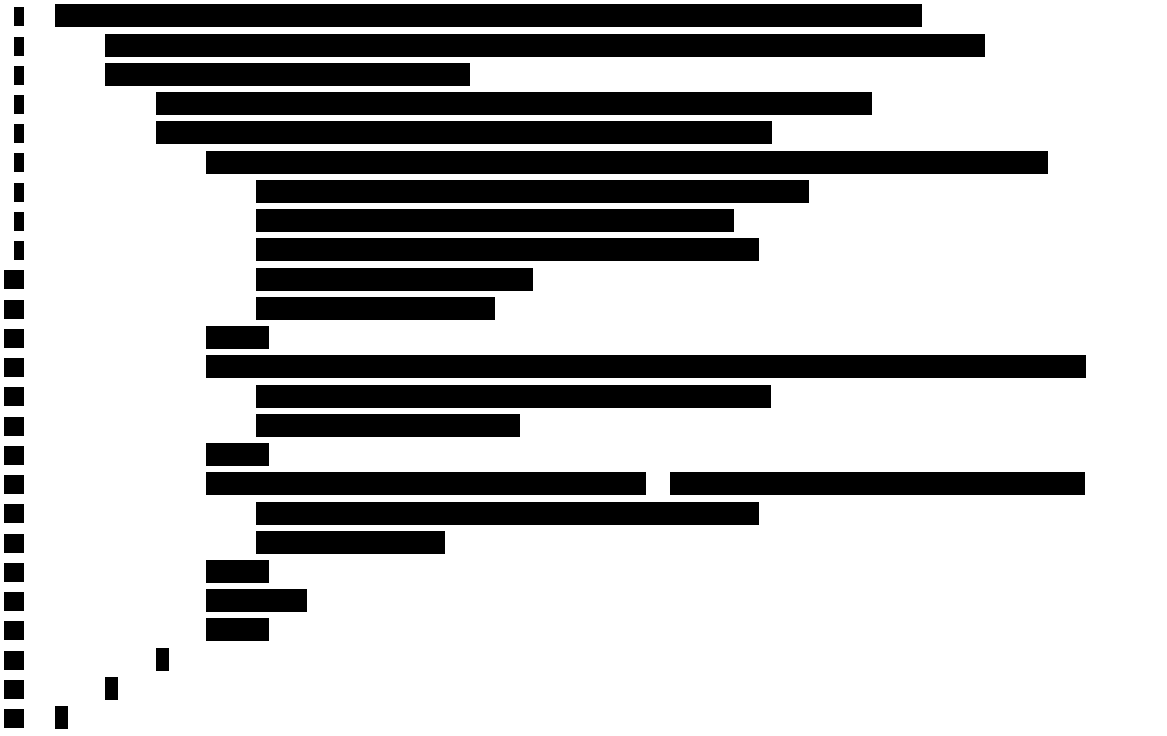
Zur Verarbeitung und Interpretation der eingelesenen Nachricht wird die `MessagePackValue`-Instanz an die View-Controller Funktion `processData()` weitergegeben (siehe Quellcode 23). Diese prüft zu Beginn, ob die vorliegende Instanz Daten enthält. Ist dies der Fall, liest die Funktion die Nachrichtenkennung aus, die nach Protokollspezifikation (siehe Kapitel 4.1.5) immer an erster Stelle steht. Anhand dieser wird bestimmt, wie das Datenpaket zu interpretieren ist und die entsprechende Funktion aufgerufen.



Quellcode 22: Der *SwiftSockets* Server wartet auf eingehende Verbindungen, akzeptiert diese und konvertiert eingehende Datenströme mit *MessagePack.swift* in ein lesbares Format.

[20] https://github.com/a2/MessagePack.swift (Zuletzt abgerufen: 27.01.2016)

Handelt es sich bei der eingelesenen Nachricht um eine *Strukturanfrage* an die App, so wird der mitgelieferte Name des Clients gespeichert und eine Strukturantwort gesendet (`sendStructure()`). Fragt der Client Nutzdatenpakete an, so wird ein Timer mit *GCD-Kit* angelegt, der Daten in der vom Benutzer festgelegten Samplerate an MotionBuilder sendet (`setupWriteTimer()`). Wird hingegen der Datenstopp angefordert, so wird der Timer pausiert und der Nutzdatenversand damit gestoppt.

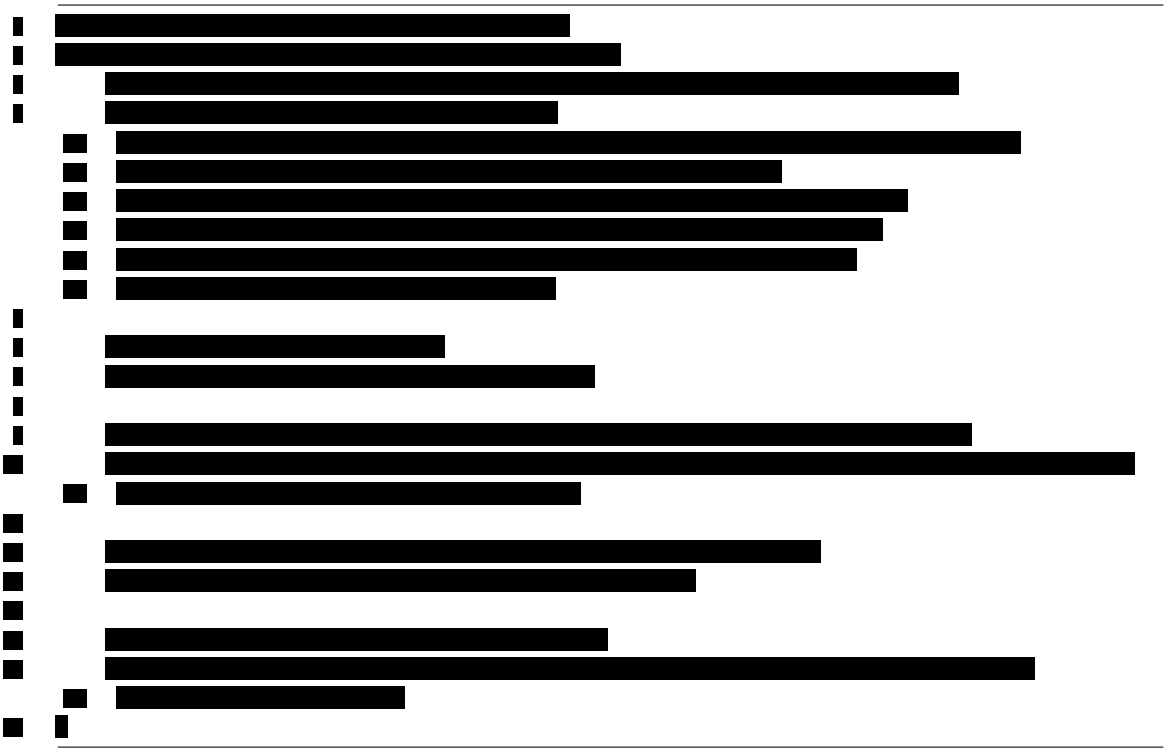


Quellcode 23: Auswertung der über den Client-Socket eingelesenen und deserialisierten MessagePack-Nachricht. Anhand der Nachrichtenkennung (Enum) wird entschieden, wie das Paket interpretiert werden muss.

Die Funktionen zum Senden der Strukturantwort und Nutzdatenpakete sind sich in ihrem Aufbau sehr ähnlich und unterscheiden sich vor allem durch den Dateninhalt. Beide Funktionen beginnen mit der Definition der Nutzdaten. Im Falle der Strukturantwort ist dies ein Daten-Array, dass abwechselnd je eine Bezeichnung (String) und eine Datentypkennung (Integer) enthält. Die Reihenfolge der Datensätze ist besonders wichtig, da diese auch beim Versand der Nutzdaten eingehalten werden muss. Sie wird dazu genutzt, um die einzelnen Nutzdaten in MotionBuilder dem passenden Datensatz zuzuordnen.

Laut Protokollspezifikation (siehe Kapitel 4.1.5) werden dem Array zudem noch mehrere Informationen vorangestellt. Neben der Nachrichtenkennung und Nachrichtennummer, ist dies noch der Name der Datenquelle (Beispiel: *iPad Air*), sowie die Menge der Da-

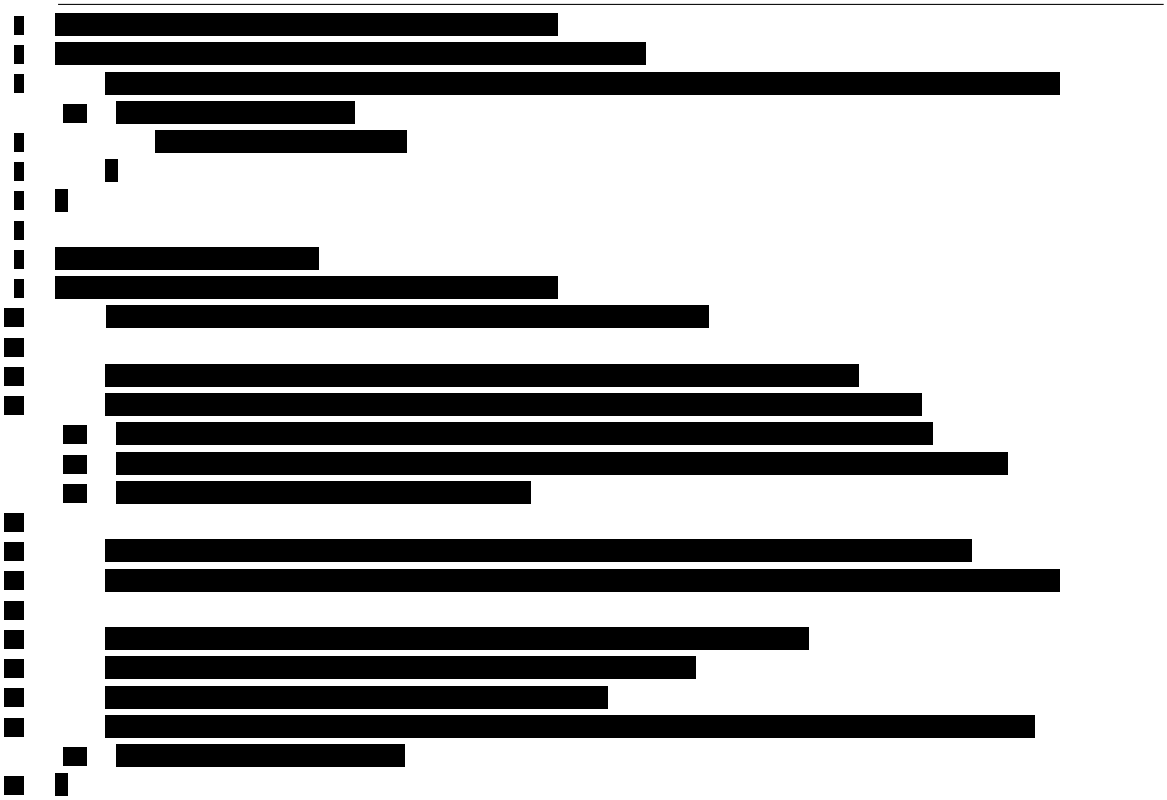
tensätze, die mit einem Nutzdatenpaket übertragen werden. Alle Daten werden in ein Array gepackt, dass mithilfe der `pack()`-Funktion des *MessagePack.swift* Frameworks in Bytecode serialisiert wird. Dieser wird über die asynchrone Schreibmethode des Socket-Frameworks (`asyncWrite()` an das verbundene MotionBuilder Plugin gesendet, ohne dass die Softtware während des Schreibvorgangs blockiert wird (siehe Quellcode 24).



Quellcode 24: Funktionen zum Senden der Strukturantwort. Die Datenmenge wurde im Beispiel für eine kompaktere Darstellung reduziert.

Nach der Anforderung des Datenfluss seitens des Clients wird ein `GCDTimer` aus dem *GCDKit*-Framework angelegt, der im vom Anwender spezifizierten Intervall (*Samplerate*), Nutzdatenpakete an den Client sendet. Zum Versand eines Datenpakets wird die `sendData()`-Funktion des View-Controllers aufgerufen. Diese legt ein Nutzdatenpaket nach Protokollspezifikationen an, dass mit *MessagePack.swift* in Bytecode serialisiert und per `asyncWrite()`-Funktion des Client-Sockets versendet wird (siehe Quellcode 25).

Auf jeden Schreibvorgang folgt zudem ein Aufruf der `writeStringLog()`-Funktion des View-Controllers, welche die geschriebenen Datenpakete in eine Textdatei protokolliert, sofern der Anwender diese Funktion in der Benutzeroberfläche aktiviert hat.



Quellcode 25: Funktionen zum Erzeugen eines Timers mit dem *GCDKit*-Framework und dem Senden von Nutzdatenpaketen per Socket.

5.2.5 Relation Constraint

Die Interpretation der eingehenden Nutzdaten in MotionBuilder ist bei diesem Anwendungsbeispiel mit wenig Aufwand verbunden. Die Umsetzung erfolgt mit zwei Relation Constraints, wovon eines bereits durch das Hinzufügen des Plugin-Device zur Szene erzeugt wird. Das automatisch angelegte Constraint beinhaltet unmittelbar nach der Erstellung lediglich die Box des Device.

Die Device-Box agiert als *Sender* und stellt dem Anwender sämtliche Nutzdaten über Ausgänge zu Verfügung. Die Box trägt den gleichen Namen wie das Device (Beispiel: *iPad Air*). Alle Ausgänge werden dabei von MotionBuilder alphabetisch sortiert, sodass die dargestellte Reihenfolge der Animationsknoten nicht immer exakt der spezifizierten Reihenfolge in der Strukturantwort entspricht. Zur Verknüpfung der Daten mit der Kamera wird der Szene ein *Cube* (Würfel) hinzugefügt, der Rotations- und Bewegungsdaten empfängt. Über ein *Parenting* (hierarchische Objekt-Beziehung) werden die Attribute eines beliebigen Szenenobjekts, wie z.B. einer Kamera, an die des Hilfswürfels gebunden (siehe Abbildung 5.6)
.

Die Rotation des Hilfswürfels setzt sich aus den Zahlenwerten (Gradmaß) für *Roll*, *Pitch* und *Yaw* zusammen (siehe Kapitel 4.2.3, Abbildung 4.11). *Roll* entspricht dabei der Rotation um die X-Achse, *Pitch* der Rotation um die Y-Achse und *Yaw* der Rotation um die Z-Achse des Hilfswürfels. Damit die Rotation um die Y-Achse sich wie erwartet verhält, muss der *Pitch*-Winkel noch über einen *Multiply*-Operator mit dem Wert „-1.0“ multipliziert werden. Dies ändert das Vorzeichen, sodass die Rotation den erwarteten Effekt zeigt. Wird dies nicht gemacht, wird die Rotation um die Y-Achse spiegelverkehrt dargestellt. Über einen *Number to Vector*-Operator werden schließlich alle drei Winkel zu einem Rotationsvektor konvertiert und mit dem Rotationseingang des Hilfswürfel verbunden.

Die boolschen Werte für *Play Button*, *Rewind Button* und *Record Button* können ohne weitere Verarbeitung mit dem *Transport Control*-Empfänger verbunden werden. Dieser findet sich in der Box-Liste links des grafischen Editors unter *System*. Der Empfänger ermöglicht die direkte Steuerung der Player-Steuerung von MotionBuilder und reagiert dabei auf Impulse. Ändert sich ein anliegender Bool-Wert an der Transport Control Box kurzzeitig von *false* auf *true* oder umgekehrt, löst der Empfänger das entsprechende Ereignis aus. Der *Play Button*-Ausgang am Device wird mit dem *Play*-Eingang am Transport Control verknüpft. Das gleiche Prinzip gilt für den *Record Button*-Wert. Der Ausgang *Rewind Button* am Device wird dem *Goto Beginning*-Eingang am Transport Control zugeordnet.

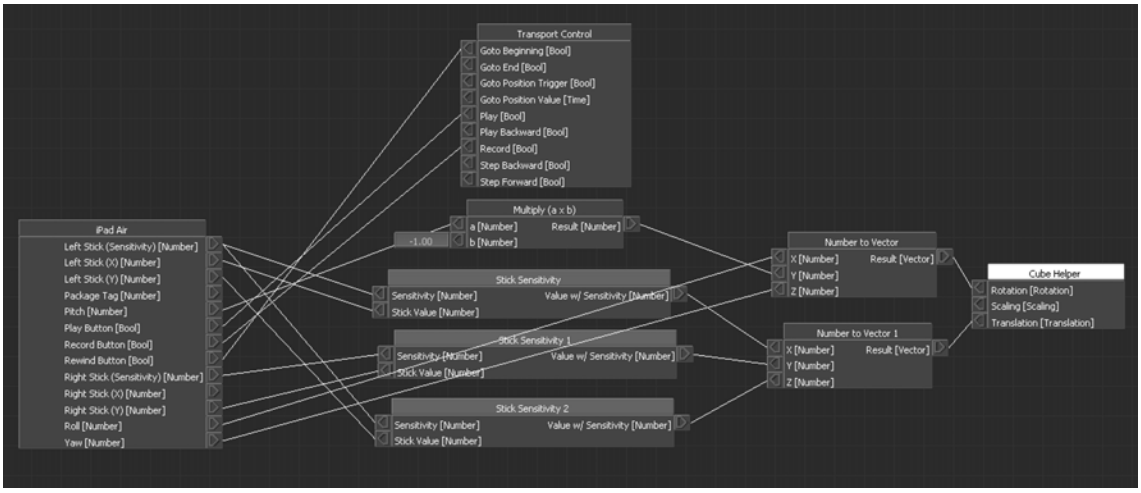


Abbildung 5.6: Der Aufbau des Relation Constraint zur Interpretation der eingehenden Nutzdaten der iPad-Anwendung.

Die komplexesten Verbindungen gehen auf die Joystick-Steuerung zurück. Bevor die einzelnen Werte der Joysticks mit der Translation des Würfels verknüpft werden können, mussen die Stick-Empfindlichkeit mit den X-Y-Werten skaliert werden. Dazu wird ein zweites Relation Constraint angelegt, welches als Makro fungiert (siehe Abbildung 5.7). Makros unterschieden sich nur unwesentlich von herkömmlichen Relation Constraints, besitzen jedoch als Sender und Empfänger spezielle *Input- und Output*-Boxen, die Ein- und Ausgänge in der Makro-Box repräsentieren. Sie gleichen damit logisch gesehen einem Programmkonstrukt.

Zur Verrechnung der Joystick-Empfindlichkeit werden zwei *Macro Input Number*-Boxen benötigt. Diese liefern später den Wert für die Empfindlichkeit (*Sensitivity*) und den Joystick-Wert (*Stick Value*). Der Wert für die Empfindlichkeit wird mit einem *Divide*-Operator durch „100.00“ geteilt, um einen Prozentsatz zu errechnen. Dieser wird im nächsten Schritt mit dem Joystick-Wert per *Multiply*-Operator verrechnet, sodass der Joystick-Wert entsprechend prozentual abgeschwächt wird.

Bevor der resultierende Joystick-Wert über eine *Macro Output Number* freigegeben wird, muss er noch mit dem vorherigen Wert verrechnet werden. Dies resultiert in einer kumulativen Änderung des Wertes. So ist sichergestellt, das die Positionsänderungen auch nach Loslassen der Joysticks erhalten bleibt, da die vom Joystick gelieferten Werte nicht als absolut interpretiert werden.

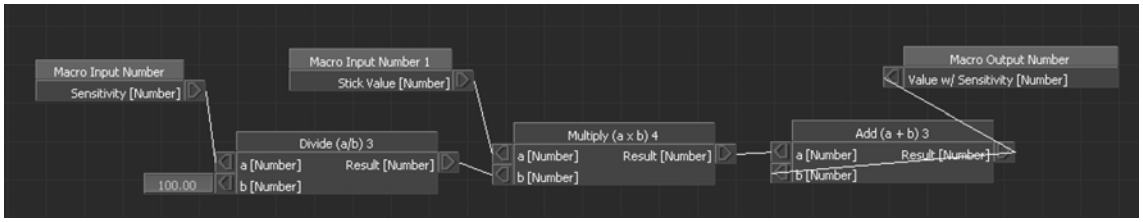


Abbildung 5.7: Makro (Relation Constraint) zur Verrechnung der Joystick-Empfindlichkeit.

Um das Makro einzusetzen, wird es aus der *My Macros*-Kategorie in der Boxliste in das Relation Constraint mit Device-Sender und Würfel gezogen. Es erscheint als neue Box mit den zuvor definierten Ein- und Ausgängen. Im letzten Schritt werden die zwei Joystick-Werte über das Makro mit einem *Number to Vector*-Operator verknüpft, der den resultierenden Vektor in die Translation des Hilfswürfels speist.

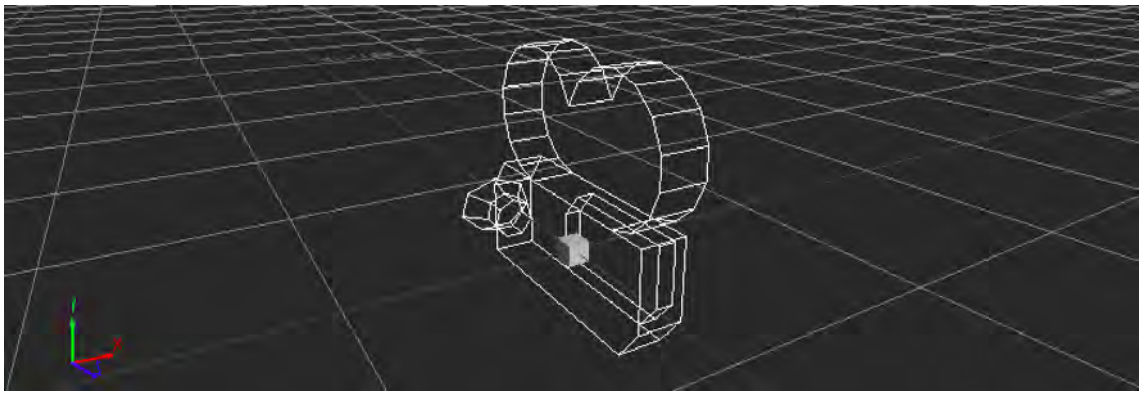


Abbildung 5.8: Positionierung des Hilfswürfel zur Kamerasteuerung.

Zur Steuerung des Würfels auf der horizontalen Ebene, die sich über die X- und Z-Raumachsen definiert, werden die X- / Y-Werte des linken Sticks verwendet. Vom rechten Stick wird lediglich der Y-Wert genutzt, welcher die vertikale Translation des Würfels entlang der Y-Achse

kontrolliert. Die Steuerung der Vertikalen ist dabei invertiert, ähnlich wie bei einer Flugzeugsteuerung. Auf Wunsch lässt sich diese jedoch durch das Einfügen eines *Multiply*-Operators mit dem Faktor „-1.0“ umkehren. In der hier beschriebenen Umsetzung (siehe Abbildung 5.6) wird darauf verzichtet.

Abschließend gilt es die Verbindung zwischen Hilfswürfel und Kamera herzustellen. Hierzu wird der Szene ein Kamera-Objekt hinzugefügt, diese mittig zum Würfel ausgerichtet und um 90 Grad um die Y-Achse rotiert. Der Hilfwürfel wird als *Parent* (Elternteil) der Kamera definiert, wodurch sich alle Transformationen, Rotationen und Skalierungen des Hilfswürfels, auch auf die Kamera auswirken. Hierzu wird der *Schematic View* der Szene (Hotkey: *Ctrl* + *W*) aufgerufen, welcher zwei Boxen für Kamera und Würfel enthält. Mit der „P“-Taste lässt sich der *Parenting Modus* aktivieren. Abschließend wird eine Linie mit der Maus vom Kamera-Objekt zum Hilfswürfel gezogen, um das Kamera-Objekt als Kindelement des Würfels zu definieren. Die Kamerasteuerung ist nun einsatzfähig.

5.3 Anwendungsbeispiel: Leap Motion Tracking

Die Implementierung des zweiten Anwendungsbeispiels, der Leap Motion Integration zum Tracking von Fingern (siehe Abbildung 5.9), ist weniger komplex als die Entwicklung der iPad Anwendung. Dies liegt daran, dass ein Großteil des iPad Quellcodes übernommen werden kann und die Umsetzung der Benutzeroberfläche mit weniger Aufwand verbunden ist. Das Kapitel konzentriert sich daher auf die Migration des Quellcodes von iOS zu OS X, sowie die Einbindung des Leap Motion SDK in die Anwendung. Abschließend wird die Interpretation der Tracking-Daten in MotionBuilder beschrieben.

Abbildung 5.9: Die umgesetzte Integration des Leap Motion-Controller im Betrieb mit aktiver Übertragung der Tracking-Daten auf das Aragor Hand-Rig.

5.3.1 Code Portierung von iOS zu OS X

Für die Entwicklung eines zweiten Anwendungsbeispiels wird der Code der iPad Anwendung auf die Mac OS X Plattform portiert. Hier wird explizit der Mac als Plattform gewählt, da sich beide Apple Systeme sehr ähnlich sind und die Swift-Programmiersprache unterstützen. Dies ermöglicht eine leichte Migration der Codebasis vom iPad auf den Mac und hilft dabei, mögliche Hardware-Einschränkungen des schwächeren Tablet-Computers aufzuspüren, da nun ein direkter Vergleich zwischen beiden Systemen möglich ist. Weiterhin lassen sich auf den leistungsstärkeren Mac-Systemen, umfangreichere Tests mit größeren Datenpaketen durchführen.

Obwohl die iOS-Plattform ursprünglich eine Abwandlung der OS X-Plattform war, haben sich die Plattformen in den letzten Jahren nicht nur in der grundlegenden Systemarchitektur voneinander entfernt. Beide Plattformen unterstützen die Sprachen Objective-C und Swift als Basis für native Softwareentwicklung und so ist zumindest ein Großteil des Codes ohne Probleme auf das jeweils andere System übertragbar. Während sich geschriebene Logik mit wenig Aufwand übertragen lassen, wird es schwierig, sobald die App intensiv systemspezifische Frameworks anspricht. Zwar sind viele der Open Source und Apple Frameworks für Multi-Plattform Anwendung ausgelegt, doch gibt es einige kritische Bibliotheken, die dies nicht sind.

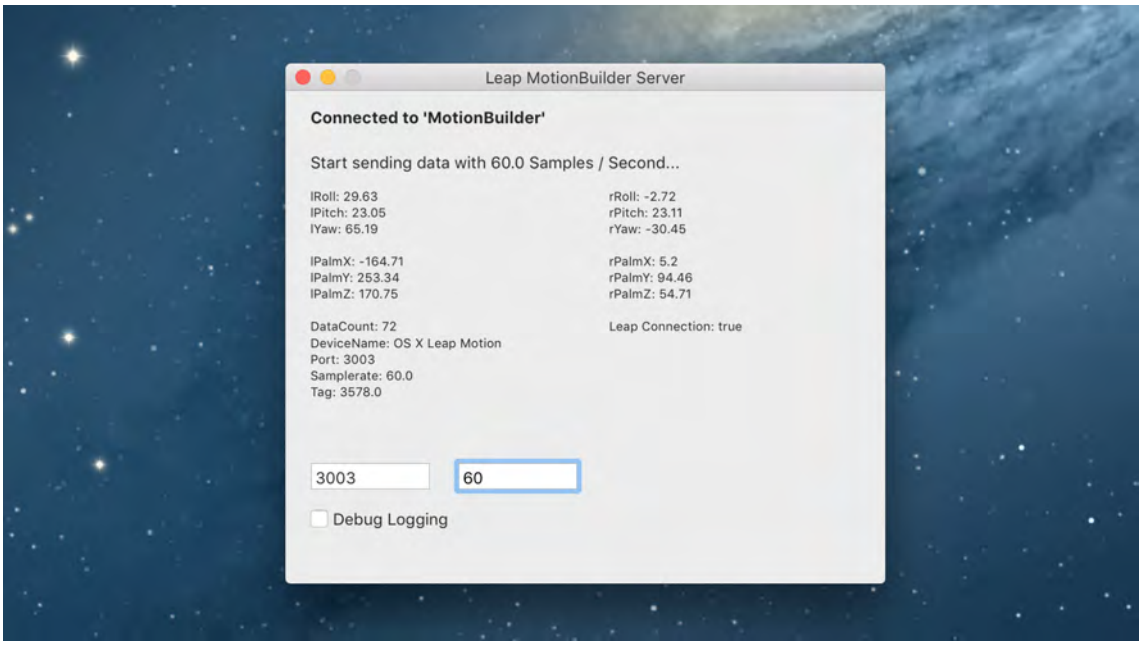


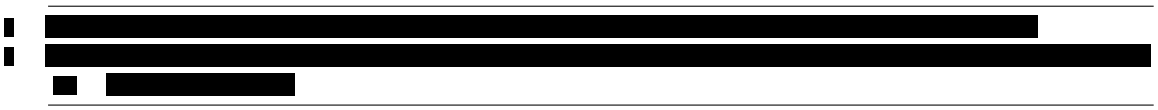
Abbildung 5.10: Benutzeroberfläche des implementierten Leap Motion Datenserver während des Betriebs. Die einzelnen Elemente wurden von der iPad Benutzeroberfläche adaptiert und an das Anwendungsbeispiel angepasst.

Die größte Hürde bei der Portierung des iPad Anwendungsbeispiels ist in diesem Fall die Umstellung von *UIKit* (iOS) auf *Cocoa* (OS X). Dabei handelt es sich um die zuständigen Frameworks für die Benutzeroberfläche des jeweiligen Systems. *UIKit* ist eine abgewandelte Variante von *Cocoa*, die aufgrund der Popularität mobiler Geräte jedoch mittlerweile eine deutlich größere Verbreitung gefunden hat. Glücklicherweise sind sich viele der Bedienelemente dennoch recht ähnlich, was den Aufwand der Portierung zumindest bei einer einfach gehaltenen Benutzeroberfläche geringfügig macht.

Für den Leap Motion-Datenserver werden die notwendigen Bedienelemente aus der iOS App durch ihre Desktop-Äquivalente ersetzt und der *UIKit*-Code entsprechend angepasst. Zu den übernommenen Elementen, gehören die Textfelder zur Eingabe von Port und Samplerate, die

Labels zur Ausgabe von Informationen für den Benutzer, sowie die Option zum Aktivieren der Protokollierung. *Cocoa* unterscheidet im Gegensatz zu *UIKit* im Programmcode nicht zwischen einem Label und einem Textfeld, sodass `UILabel`- und `UITextField`-Elemente auf dem Mac mit `NSTextField` umgesetzt werden. Auch die `UISwitch`-Komponente kennt Cocoa nicht. Das Bedienelement auf dem Mac welches dem Switch am ehesten entspricht, ist die Checkbox. Diese wird im Code in Form eines `NSButton` repräsentiert. Auch die Funktionen und Attribute der einzelnen Bedienelemente unterscheiden sich zwischen den beiden Systemen. Während der Inhalt eines `UITextField` mit dem `text`-Attribut abgerufen und gesetzt werden kann, wird dies bei einem `NSTextField` über `stringValue`-Atrribut erledigt.

Zusätzlich zu den Anpassungen an unterschiedliche Bibliotheken fallen außerdem noch plattformspezifische Aspekte an, die beachtet werden müssen. Die OS X-Plattform bietet seit Version 10.9 „Mavericks“ eine Systemfunktion namens *App Nap* an. Diese schaltet Apps die für den Benutzer nicht sichtbar sind (verdeckt oder ausgeblendet) in eine Art Schlafmodus, in dem sie keinerlei Prozessorleistung beanspruchen. Für manche Anwendungen kann diese Funktion jedoch problematisch sein, da das System unter Umständen die Ausführung von Aufgaben stoppt, sobald App Nap aktiv wird. Um dies zu vermeiden, muss die Anwendung ihre Aktivität beim System anmelden (siehe Quellcode 26).



Quellcode 26: Die Aktivität der Anwendung muss beim System mit Begründung gemeldet werden, um die *App Nap*-Funktion zu unterbinden.

Für Programmteile wie die Netzwerkkommunikation und das Serialisieren der Nutzdaten mit MessagePack, wird hingegen der Code des iPad Anwendungsbeispiel (siehe Kapitel 5.2) fast unverändert übernommen. Lediglich kleine Syntax-Änderungen sind an manchen Stellen notwendig um den Code auf dem Mac lauffähig zu machen, da die verwendeten Frameworks wie *SwiftSockets* und *GCDKit* auch für OS X kompiliert werden können.

5.3.2 Leap Motion SDK

Für die Integration des Leap Motion-Controllers in eine Mac Anwendung, bietet das SDK offizielle Unterstützung für die *Objective-C* Sprache. Das *Objective-C* Framework besteht dabei aus Hüllklassen für die Leap Motion C++-Bibliothek, welche den Kern des SDK bildet. Dieser lässt sich über einen sogenannten *Objective-C Bridging Header* auch mit Swift in Xcode verwenden. Eine Swift-Variante des SDK wird in der verwendeten Version 2.3 noch nicht geboten (Stand: 30.01.2016).

Zur Einbindung des Leap Motion SDK in ein neues Projekt wird der gesamte Objective-C „include“-Ordner dem Xcode Projekt hinzugefügt. Anschließend muss die kompilierte C++ -Bibliothek „`dylib.lib`“ den *Linked Frameworks and Binaries* im *General*-Reiter der

Projekteinstellungen hinzugefügt werden. Damit die Bibliothek während des Build-Vorgangs auch wirklich kopiert wird, muss zudem unter *Build Phases*, eine neue *Copy Files*-Phase angelegt werden. Dieser wird dann die Bibliotheksdatei hinzugefügt und der *Destination*-Parameter auf „Executables“ gesetzt, damit Xcode die Bibliothek an die korrekte Stelle im Programmpaket kopiert.

Zum Erzeugen des Objective-C Bridging Header, wird eine neue Objective-C Header Datei über das *Datei*-Menü erzeugt und dem Projekt hinzugefügt. In den Projekteinstellungen unter *Build Settings* wird anschließend für den Parameter *Objective-C Bridging Header*, der Pfad zur neu angelegten Header Datei angegeben. Während des Build-Vorgangs fragt der Swift Compiler den Inhalt dieser Datei ab und verwendet die dort aufgeführten Import-Befehle. Für das Leap Motion-SDK ist dies lediglich eine Zeile Code:

[redacted]

Um das Leap Motion SDK nach der Einbindung zu verwenden, muss als erstes eine Instanz des `LeapController` erzeugt werden. Dieser ist die zentrale Schnittstelle zwischen dem Programmcode und den SDK-Funktionen. Durch Implementierung des `LeapDelegte`-Protokolls und dessen Funktion `onFrame()` wird mit der Instanz des `LeapController` immer auf die aktuellsten Werte des SDK zurückgegriffen. Die Funktion `onFrame()` wird vom SDK ausgelöst, sobald neue Daten vom Controller bereit stehen. Über die `LeapController`-Instanz wird das aktuelle `LeapFrame` abgerufen, welches sämtliche Tracking-Daten für ein Frame (Standbild) enthält. Das Leap Motion-SDK stellt je nach Lichtverhältnissen zwischen 100 und 120 Frames pro Sekunde zur Verfügung.

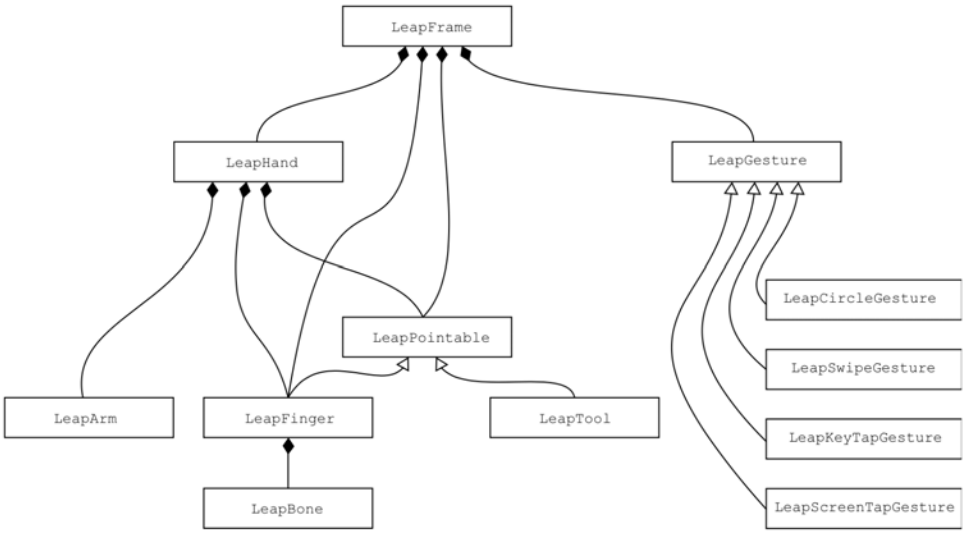
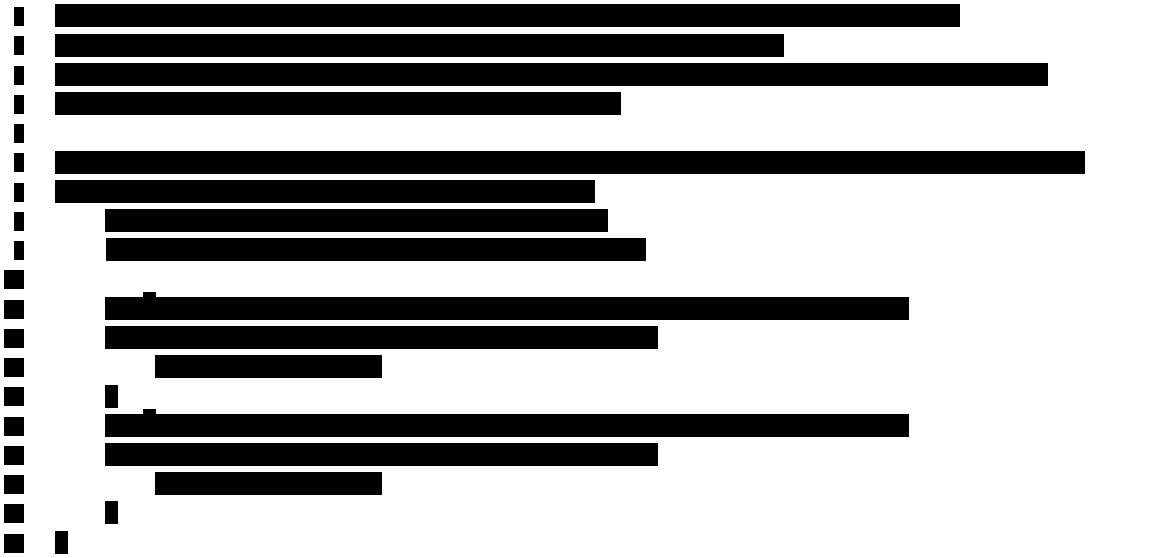


Abbildung 5.11: Die Hierarchie der Tracking-Daten im Leap Motion SDK.

Quelle: http://bit.ly/1Qzpe1G (Leap Motion Inc., Zuletzt abgerufen: 30.01.2016)

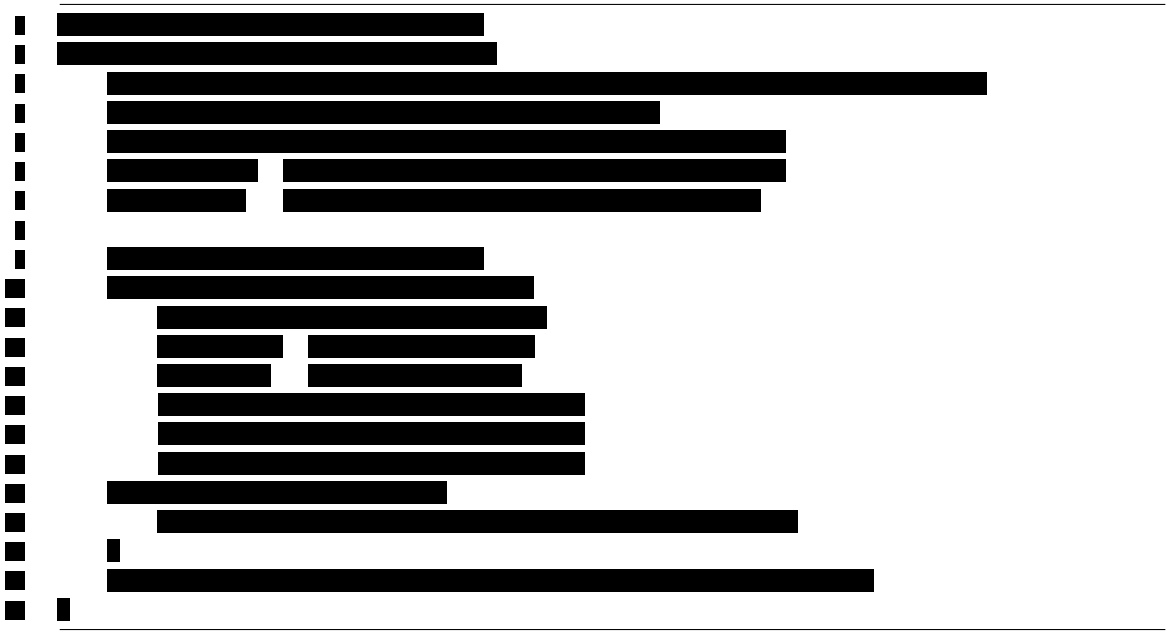
Ein `LeapFrame` kann über Attribute in seine Bestandteile zerlegt werden. Über das `hands`-Attribut des Frame wird ein Array aus `LeapHand`-Objekten abgerufen, welches die Tracking-Daten der erkannten Hände beinhaltet. Bis zu zwei Hände pro Frame werden vom Anwendungsbeispiel verarbeitet. Die `LeapHand`-Instanzen werden jeweils an die View-Controller Funktion `getHandData()` übergeben, welche die gewünschten Daten aus den Hand-Objekten extrahiert (siehe Quellcode 28). Für beide Hände sind dies die Position der Hand und die Rotation in der *Roll-Pitch-Yaw* Notation, wie in Kapitel 4.3.2 beschrieben.



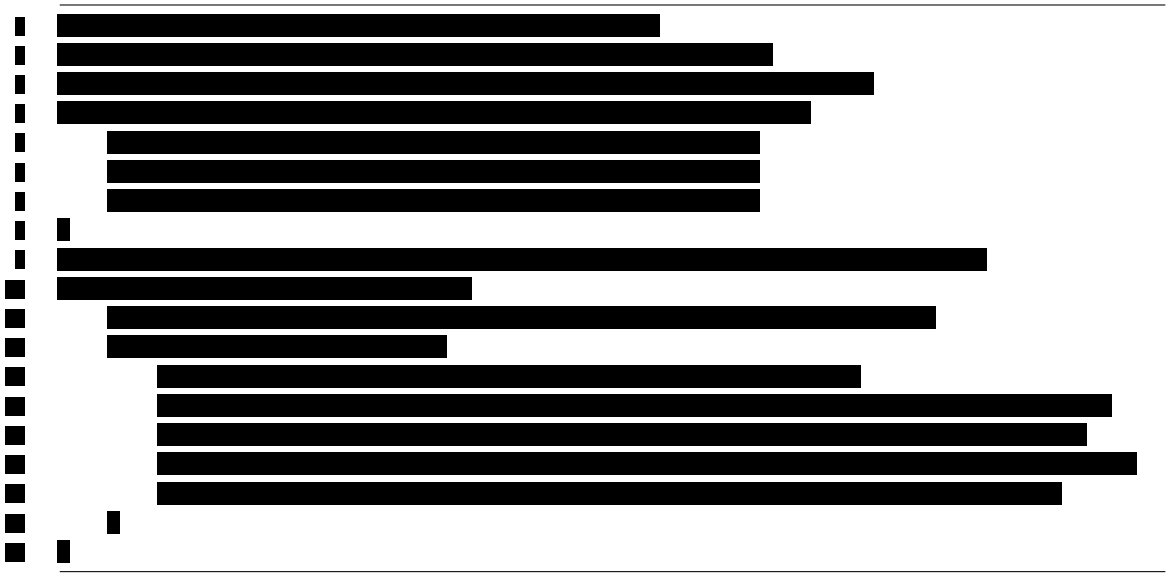
Quellcode 27: Implementierung des `LeapDelegate`-Protokolls zur Benachrichtigung bei neuen Tracking-Daten.

Für die zuletzt erkannte Hand werden außerdem die Daten der Finger bzw. Fingerknochen verwertet. Dazu wird über das `fingers`-Attribut der Hand, das `LeapFinger`-Array abgerufen. Dieses enthält alle fünf Finger der Hand und lässt sich wiederum für den Abruf der Daten aller vier Fingerknochen nutzen. Dazu wird das `bone`-Attribut jedes Fingers abgefragt, dass ein Array bestehend aus vier Knochen (`LeapBones`) liefert (siehe Kapitel 4.3.2, Abbildung 4.15-A)

Wie in Kapitel 4.3.2 beschrieben, werden die Richtungsvektoren aller Fingerglieder übertragen. Aus diesen lässt sich dann im Relation Constraint die Rotation der einzelnen Knochen errechnen. Zur Übertragung der Vektoren an MotionBuilder, müssen diese zuerst in ein mit dem Protokoll kompatibles Format gebracht werden. Hierzu werden die drei Komponenten jedes Vektors in einer Schleife mit der Hilfsfunktion `setBoneVector()` als `Double`-Werte in ein Array geschrieben, das dem Nutzdatenpaket später hinzugefügt wird. Ingesamt 72 Zahlenwerte werden mit jedem Nutzdatenpaket übertragen, wovon 60 nur für die Übertragung der 20 Knochenvektoren benötigt werden (siehe Quellcode 29).



Quellcode 28: Funktion zum Abrufen aller benötigten Tracking-Daten einer `LeapHand`.



Quellcode 29: Funktion zum Abruf der vier Fingerknochen aller fünf Finger einer Hand. Die gewonnenen Richtungsvektoren werden zur Übertragung in Doubles zerlegt.

5.3.3 Relation Constraint

Im Vergleich zum iPad Anwendungsbeispiel ist das Relation Constraint zur Verarbeitung der Leap Motion Daten deutlich umfangreicher. Insgesamt 72 Zahlenwerte werden vom Plugin empfangen, welche mit Ausnahme von Winkelkonvertierungen noch durch keinerlei Logik verarbeitetet wurden. Diese wird komplett im Relation Constraint des Device umgesetzt. Die Abbildungen in diesem Kapitel repräsentieren zur besseren Darstellung nur Teile der kompletten Box-Netze, die zum Verständnis der Umsetzung notwendig sind.

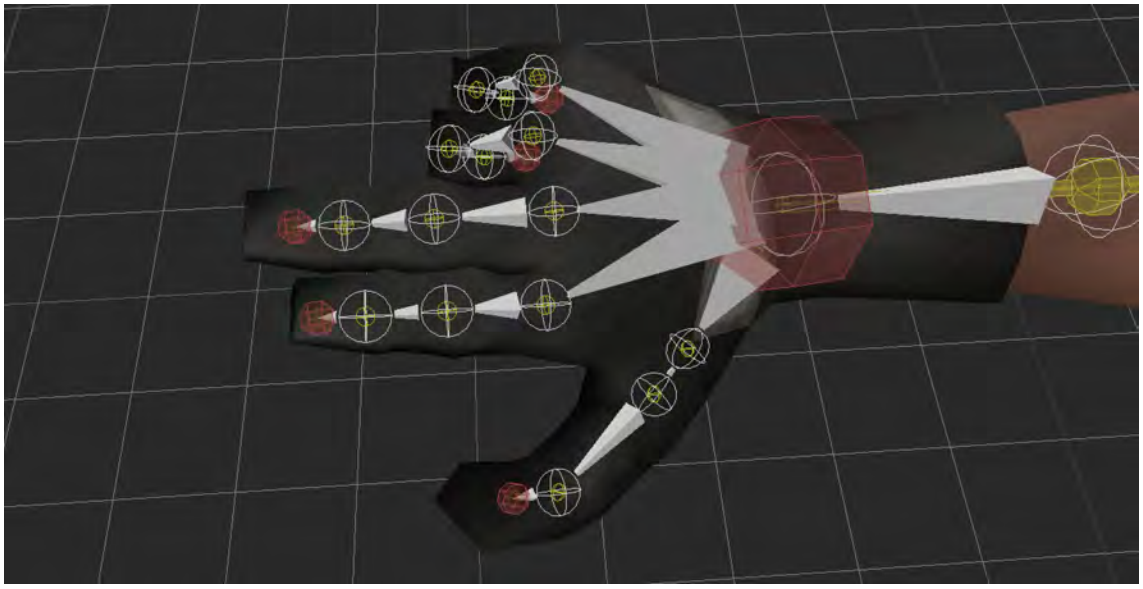


Abbildung 5.12: Der Aragor Template-Charakter mit generiertem Hand-Rig und übertragenen Leap Motion-Daten.

Um ein Charakter-Rig zur Demonstration zu haben, wird nach dem erfolgreichen Verbindungsaufbau zum Leap Motion Server als erstes ein „Aragor“ Modell aus dem *Template* Ordner über das *Assets*-Tab des Navigators der Szene hinzugefügt. Das Rig ist Teil jeder Autodesk MotionBuilder 2015 Installation und kann frei verwendet werden. Als nächstes muss ein *Control Rig* für das Modell über das *Character Controls*-Fenster angelegt werden. Ein Control Rig besteht aus Steuerelementen für animierbare Objekte, die zur Animation dieser Objekte genutzt werden.

Zur Steuerung der Handrotationen wird jeweils das *Right Hand*- und *Left Hand*-Control dem Relation Constraint hinzugefügt des Device hinzugefügt. Bevor die Rotationswerte der Hände mit den Controls verbunden werden können, müssen diese erst noch für die Verwendung aufbereitet werden. Über *Multiply*-Operatoren mit dem Faktor „-1“, werden die Vorzeichen der *Left Hand Roll* und *Left Hand Yaw* Winkel getauscht, sodass die Rotationen der linken Hand des Anwenders korrekt auf die virtuelle Hand übersetzt werden. Die Rotationen der linken Hand werden vom Leap Motion SDK nach dem linkshändigen System berechnet. Für die rechte Hand ist dies das rechtshändige System, sodass für die korrek-

te Darstellung der Rotationen der rechten Hand die Vorzeichen für *Right Hand Pitch* und *Right Hand Yaw* getauscht werden. Für beide Hände gilt: Roll treibt die Rotation um die X-Achse, Yaw die Rotation um die Y-Achse und Pitch die Rotation um die Z-Achse an. Die korrigierten Rotationswinkel werden dann über *Number to Vector*-Operatoren mit dem Rotationseingang des jeweiligen Control verbunden.

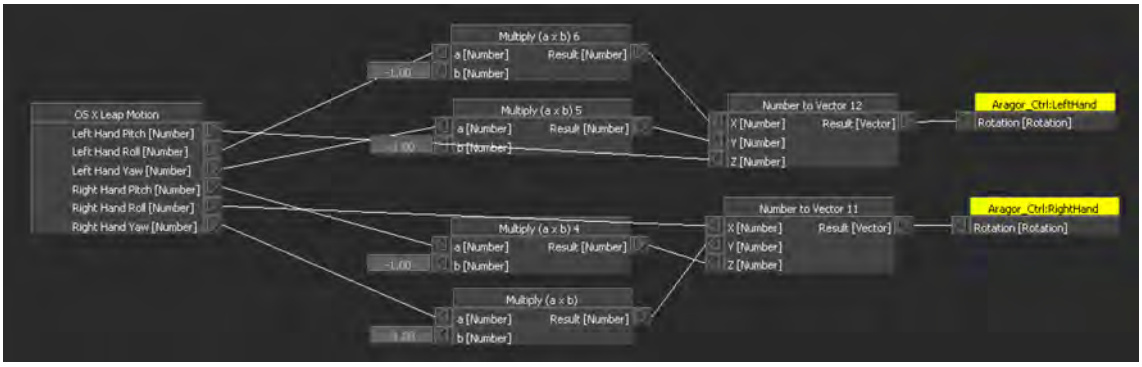


Abbildung 5.13: Abschnitt des Device Relation Constraint, welcher für die Verarbeitung und Interpretation der Handrotation zuständig ist.

Wie in Kapitel 4.3.2 beschrieben, wird neben der Rotation der Hand, auch die Position der Handfläche im Verhältnis zum Leap Motion-Controller aufgezeichnet. Diese wird in der Beispielanwendung an MotionBuilder übertragen, jedoch ist der Nutzen für die beschriebene Demonstration am *Aragor*-Rig extrem eingeschränkt. Durch die geringe Sichtweite des Leap Motion Sensors von etwa einem halben Meter, kann die Translation der Hand nur bedingt verwendet werden, sodass sie für dieses Beispiel ungenutzt bleibt.

Großen Nutzen für die Erstellung von authentischen Handanimationen, haben neben der Handrotation auch die Tracking-Daten der einzelnen Finger. Wie in Kapitel 4.3.3 beschrieben, werden sämtliche Richtungvektoren der Fingerknochen einer Hand zusammen mit den bereits genannten Daten gesendet. Pro Finger sind dies vier Richtungsvektoren, die zur Übertragung in je drei Komponenten aufgeteilt wurden. Die insgesamt 60 Zahlenwerte müssen nun im Relation Constraint wieder zu Vektoren zusammengesetzt und interpretiert werden. Das daraus resultierende Relation Constraint ist so komplex, dass übersichtshalber die Verwendung eines Makros unabdingbar ist (siehe Abbildung 5.14).

Zur Bestimmung der Fingerglieder-Rotation, wird wie in Kapitel 4.2.3 beschrieben, der Winkel zwischen jeweils zwei Knochen-Richtungsvektoren ermittelt. Dies entspricht der Rotation des Gelenks zwischen den beiden Knochen und kann z.B. zur Rotation eines Finger-Controls verwendet werden. Zur Berechnung des Winkels zwischen dem *Metacarpal*-Knochen (Knochen in der Handfläche) und dem ersten *Phalanx*-Knochen (Fingerglied), werden die Richtungsvektoren der beiden Knochen mithilfe von *Number to Vector*-Operatoren wieder zusammengesetzt. Anschließend wird der Winkel zwischen den Vektoren mit dem *Angle*-Operator errechnet, der beide Vektoren aufnimmt und einen Zahlenwert (Winkel im Gradmaß) ausgibt.

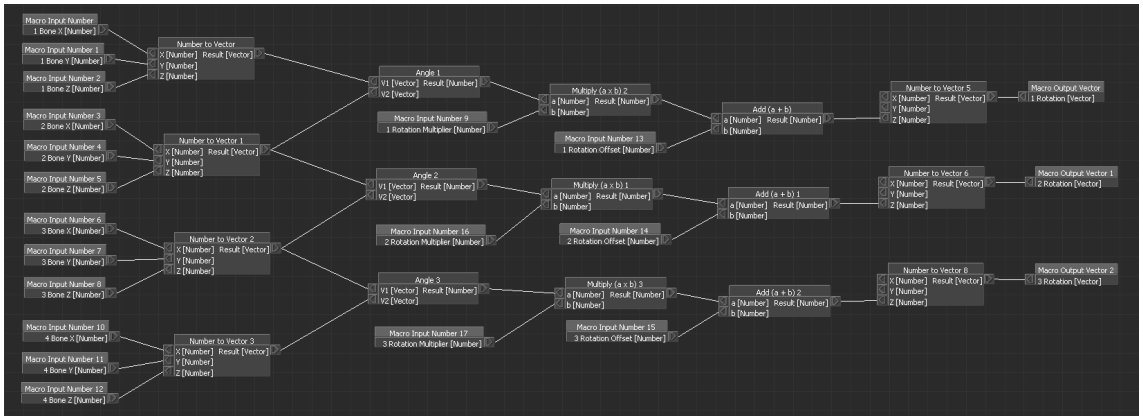


Abbildung 5.14: Aufbau des Makros zur Verarbeitung und Interpretation der zerlegten Richtungsvektoren der Knochen eines Fingers.

Im nächsten Schritt wird der Winkel, mit einem durch den Benutzer gesetzten Faktor multipliziert. Das Ergebnis aus dieser Rechnung wird wiederum in einen *Add*-Operator geleitet, der ein benutzerdefinierter *Offset* (Versatz) auf den Winkel addiert. Über den Multiplikationsfaktor und besagtem Offset, ist es dem Benutzer möglich, den errechneten Winkel besser zu kontrollieren und ggf. direkt im Relation Constraint, Fehler zu korrigieren. Abschließend wird der verarbeitete Winkel mit dem *Z*-Eingang eines *Number to Vector*-Operator verbunden, der den finalen Rotationsvektor für das Knochen-Control produziert. Dieser gesamte Vorgang wird noch zwei weitere Male für die beiden verbleibenden Fingerglieder mit den restlichen Richtungsvektoren wiederholt und alle errechneten Vektoren über *Macro Output Vector*-Boxen ausgegeben.

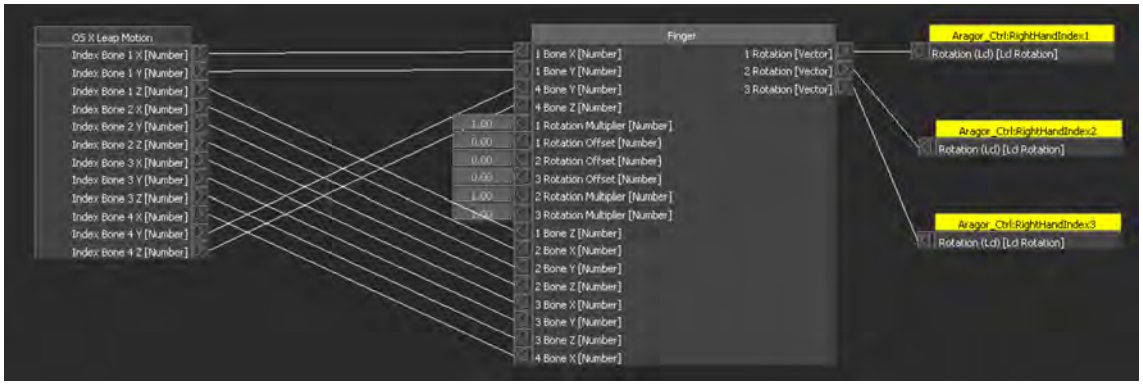


Abbildung 5.15: Aufbau des Constraint-Abschnitts zur Interpretation der Fingerdaten mittels Makro und die Übertragung dieser auf FK-Controls.

Das besprochene Makro wird im Relation Constraint gleich mehrfach eingesetzt, da jeder Finger die gleiche Prozedur durchlaufen muss, um die notwendigen Rotationsvektoren für das Rig zu erhalten. Der Einsatz des Finger-Makros ist in Abbildung 5.15 zu sehen. Das Makro ist der einzige Operator zwischen dem Sender (Device) und den Empfängern, die in diesem

Fall die drei *Forward Kinematics-Controls*, des jeweiligen Fingers darstellen. Dabei handelt es sich um Steuerelemente für die Rotation um einen Gelenkwinkel. Weiterhin ist wichtig, dass die FK-Control-Boxen, lokale Rotationen nutzen. Dies lässt sich über den Menüpunkt *Local Transformations* der Boxen festlegen und stellt sicher, dass die lokalen Rotationsvektoren der Fingergelenke, auch tatsächlich als solche von MotionBuilder interpretiert werden.

Kapitel 6

Evaluation und Ergebnisse

Im folgenden Kapitel wird der Erfolg des entwickelten Software-Prototypen überprüft. In einer Reihe von Tests wird die Qualität der Datenübertragung ermittelt, sowie etwaige Problemstellen und Einschränkungen der Software ausfindig gemacht. Weiterhin wird die Benutzererfahrung mit einer ausgewählten Testgruppe evaluiert und mögliche Verbesserungsvorschläge für zukünftige Versionen des Prototypen gesammelt.

6.1 Benchmarking

Um die Netzwerkkomponente des Plugins auf ihre Funktionalität und Zuverlässigkeit zu testen, wurde im Rahmen dieser Arbeit eine Reihe Benchmark-Tests durchgeführt, welche die messbaren Leistungen des Plugins anhand empirischer Testdaten aufzeigen. Die Durchführung der Benchmark-Tests ist in mehrere Kapitel aufgegliedert, beginnend mit der Beschreibung der verwendeten Testumgebung.

Kapitel 6.1.2 befasst sich mit der Überprüfung der Verbindungsqualität zwischen der Netzwerkschnittstelle und den entwickelten Anwendungsbeispielen. Hierzu werden in mehreren Testreihen die Verarbeitungsquote, die effektive Samplerate und die Latenz bestimmt. Die Verarbeitungsquote errechnet sich aus der Anzahl der gesendeten und tatsächlich verarbeiteten Datenpakete. Daraus lässt sich weiterhin direkt auf die effektive Samplerate schließen. Die Differenz dieser empirischen Parameter zu den optimalen Einsatzwerten zeigt wie effektiv und zuverlässig die umgesetzte Netzwerkschnittstelle arbeitet. Anhand der Stärke der Latenz lässt sich zudem ablesen, ob diese einen negativen Einfluss auf die Benutzerfreundlichkeit der Anwendungen hat.

In Kapitel 6.1.3 wird die minimale und maximale Datenrate ermittelt, mit denen die Anwendungsbeispiele in der Testumgebung zuverlässig arbeiten. Die Zuverlässigkeit wird abermals anhand der Verarbeitungsquote abgelesen und gibt Aufschluss darüber, welche Bedingungen ein System erfüllen muss um zufriedenstellende Ergebnisse mit dem Plugin-Prototypen zu erzielen.

6.1.1 Testumgebung

Um aussagekräftige Ergebnisse zu erhalten, wurden die Tests mit zwei unterschiedlichen Windows-Computersystemen an zwei Standorten mit ähnlicher Netzwerkumgebung durchgeführt. *System 1* verwendete das Windows 10 Betriebssystem von Microsoft, welches über die *Parallels Desktop 11 for Mac*[1] Software unter OS X 10.11 als virtuelle Maschine ausgeführt wurde. Der Computer verwendete den integrierten *AirPort Extreme* Netzwerkadapter, um eine Verbindung mit den WLAN-Testnetzwerken aufzubauen und erreichte damit im Durchschnitt eine Bandbreite von 65 Mbit pro Sekunde. *System 1* wurde in der Entwicklung des Plugin-Prototypen verwendet und diente weiterhin als primäres Testsystem.

System 2 arbeitete mit einem nativen Windows 8 Betriebssystem und wurde aufgrund der leistungsstarken Hardware als Vergleichssystem gewählt. Zur Kommunikation mit den Testnetzwerken kam ein *TP-Link Archer AC1200* Netzwerkadapter zum Einsatz, der über USB 2.0 mit dem Computer verbunden wurde. Der Adapter erreichte in der Testumgebung eine durchschnittliche WLAN-Bandbreite von 117Mbit pro Sekunde. Beide Testsysteme arbeiteten mit dem Build „02/26/2014“ von *MotionBuilder 2015*.

System	Betriebssystem	Technische Spezifikationen
System 1	Windows 10 (Virtuelle Maschine)	CPU: Intel i7 2,6Ghz Quad-Core RAM: 8 GB DDR3 Wifi: Apple AirPort Extreme (802.11 a/b/g/n)
System 2	Windows 8	CPU: 2x Intel Xeon 2,3Ghz Six-Core RAM: 36GB DIMM Wifi: TP-Link Archer AC1200 (802.11 ac/a/b/g/n)

Tabelle 6.1: Verwendete Computersysteme bei der Test-Durchführung.

Bei den getesteten Netzwerkumgebungen handelte es sich jeweils um ein WLAN-Netzwerk nach dem *802.11n* Standard für Drahtlosnetzwerke [Ker09, S.200]. In beiden Fällen kam ein *FRITZ!Box 7390* Router der Firma AVM zum Einsatz, der im 2,4Ghz und 5Ghz Frequenzband sendet. Die getesteten Systeme stellten automatisch eine stärkere Verbindung mit dem 2,4Ghz-Netzwerk her, weshalb dieses für sämtliche Tests genutzt wurde, trotz der geringeren Bandbreite von maximal 130Mbit pro Sekunde. Beide Netzwerke wurden parallel von anderen Geräten aktiv genutzt. Die Tests wurden somit unter realistischen Bedingungen durchgeführt.

Netzwerk	Router-Typ	Netzwerkspezifikationen
Netzwerk 1	AVM FRITZ!Box 7390	802.11n / 2,4Ghz + 5Ghz / Kanal 13
Netzwerk 2	AVM FRITZ!Box 7390	802.11n / 2,4Ghz + 5Ghz / Kanal 01

Tabelle 6.2: Verwendete Netzwerkumgebungen bei der Test-Durchführung.

[1] http://www.parallels.com/products/desktop/ (Zuletzt abgerufen: 20.02.2016)

Die App zur Kamerasteuerung wurde auf einem *iPad Air* und einem *iPad Air 2* auf ihre Lauffähigkeit getestet. Für die Durchführung der Benchmark-Tests wurde jedoch lediglich das iPad Air 2 mit iOS 9.2 verwendet, da dieses keinerlei zusätzliche Software installiert hatte. Die Beeinflussung der Testergebnisse durch andere Software wurde damit auf ein Minimum reduziert.

6.1.2 Verbindungsqualität

Da es sich bei dem entwickelten MotionBuilder Plugin um eine Echtzeit-Anwendung handelt, ist die Qualität der Verbindung ein kritischer Punkt zur Überprüfung des Erfolgs des Prototypen. Für den produktiven Einsatz muss der Prototyp ein Minimum von 30 Samples pro Sekunde empfangen und korrekt verarbeiten. Dieses Bildraten-Minimum ergibt sich aus dem aktuellen Branchenstandard für Animationen, der für Videospiele bei 30 Bildern pro Sekunde und für Spielfilme bei 24 Bildern pro Sekunde liegt. Unterschreitet die effektive Samplerate diesen Grenzwert entstehen mangels Nutzdaten Lücken in der Animation. Dies erhöht den Arbeitsaufwand der mit der Aufnahme einer Animation einhergeht, da die entstandenen Lücken manuell gefüllt werden müssen.

Um die Qualität der Datenübertragung zu messen, wurde eine Reihe von Tests durchgeführt, welche die Verbindungsqualität des Plugins zu den Anwendungsbeispielen (iPad, Leap Motion), anhand der zeitnah übermittelten Datenpakete überprüften. Der Begriff *zeitnah* ist dabei abhängig von der angepeilten Samplerate der Datenquelle.

Je höher die Samplerate, desto öfter liest das Plugin den Puffer der TCP-Socketverbindung und wertet pro Lesevorgang genau ein Paket aus (siehe Kapitel 5.1.5). Verspätete Pakete, die sich auf dem Puffer des Sockets stauen, werden zwar vom Socket gelesen, jedoch nicht weiter verarbeitet. Dies führt zwangsläufig zu Datenverlust, vermindert jedoch zeitgleich die zeitliche Differenz zwischen dem Versand der Daten von der Quelle und der Echtzeit-Visualisierung in MotionBuilder. Nachfolgend wird die Quote der verarbeiteten Datenpakete als *Verarbeitungsquote* bezeichnet. Eine Verarbeitung aller verspäteten Nutzdatenpakete würde zwangsläufig in einer zu hohen Verzögerung resultieren und den Nutzen der Daten für eine Echtzeit-Darstellung enorm einschränken. Die Netzwerkschnittstelle ist als ineffizient zu betrachten, wenn mehr als 20 Prozent der gesendeten Daten nicht verarbeitet werden. Für zukünftige Arbeiten wird als alternative Problemlösung eine Hybrid-Kommunikation, basierend auf je einer TCP- und UDP-Verbindung vorgeschlagen (siehe Kapitel 7).

Das Balkendiagramm in Abbildung 6.1 zeigt den prozentualen Anteil der verarbeiteten Nutzdatenpakete (Samples), über den Verlauf von zwölf Testverbindungen mit der iPad App zur Kamerasteuerung. Alle Testverbindungen erfolgten dabei zu *System 1* und dauerten je fünf Minuten (18000 Samples). Mit 60 Samples pro Sekunde entspricht die getestete Samplerate 200 Prozent des aktuellen Branchenstandard von 30 Samples pro Sekunde und bietet damit auch bei schwankender Verbindungsqualität ausreichend Puffer. Zur Errechnung der Verarbeitungsquote, wurde die Anzahl der versendeten Nutzdatenpakete seitens der Datenquelle, mit der Anzahl der verarbeiteten Pakete in MotionBuilder verglichen. Die Ergebnisse

zeigen, dass die Verarbeitungsquote geringfügig schlechter ist, wenn MotionBuilder durch die Echtzeit-Darstellung oder Aufnahme der Daten zusätzliche Rechenaufgaben erhält.

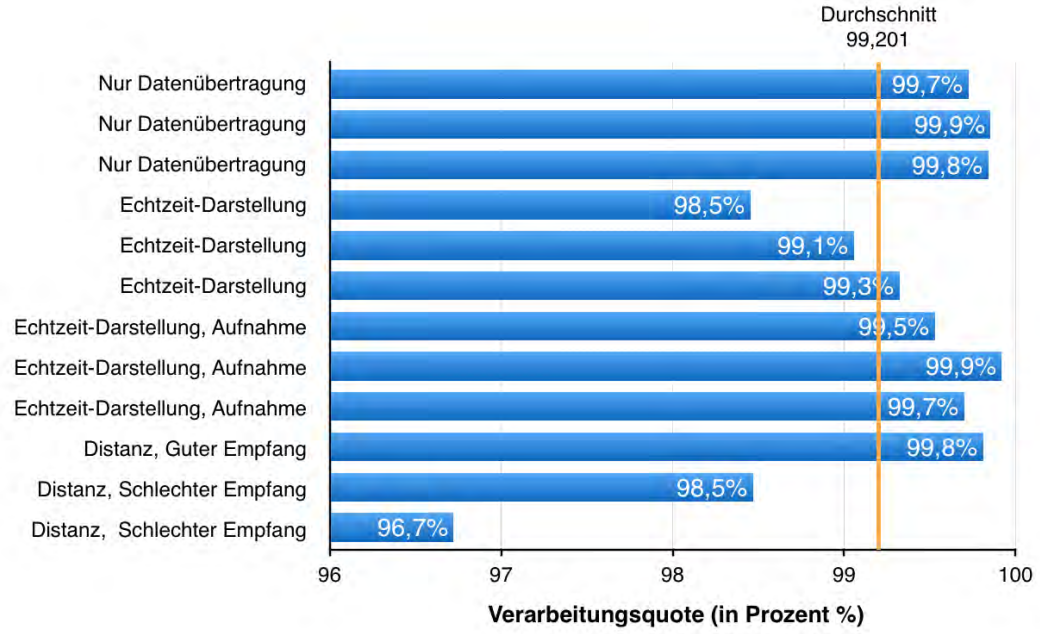


Abbildung 6.1: Verarbeitungsquoten der Nutzdatenpakete des iPad-Anwendungsbeispiels. Gemessen über zwölf Testverbindungen mit je fünf Minuten Testzeit und 60 Samples pro Sekunde.

Die schlechteste Verarbeitungsquote (96,7 Prozent) wurde in etwa 25 Meter Distanz vom Router, mit schwacher Signalstärke auf dem iPad verzeichnet. Dies entspricht einer effektiven Samplerate von circa 58 Samples pro Sekunde und liegt damit nur 2 Samples unter den angepeilten 60 Samples pro Sekunde. Für eine Echtzeit-Anwendung ist diese Differenz zu vernachlässigen. Ist eine stabile Samplerate jedoch unabdingbar, wäre die Verwendung einer höheren Samplerate eine mögliche Lösung, sodass trotz Datenverlust ausreichend Samples pro Sekunde von MotionBuilder verarbeitet werden. Im Durchschnitt erreichte die iPad-Anwendung eine Verarbeitungsquote von 99,2 Prozent, was einer effektiven Samplerate von 59,5 Samples pro Sekunde entspricht und damit fast verlustfrei arbeitet.

Das gleiche Testverfahren wurde auch angewandt um die Verarbeitungsquote der Nutzdaten des Leap Motion-Anwendungsbeispiels zu errechnen (siehe 6.2). Insgesamt zehn Testverbindungen mit je drei Minuten Testzeit, aktiven Finger-Bewegungen durch den Tester und einer Datenrate von 60 Samples pro Sekunde wurden durchgeführt. Dabei diente *System 1* mit laufender Server-Anwendung und MotionBuilder, sowohl als Sender, als auch als Empfänger. Eine Verfälschung der Ergebnisse findet nicht statt, da auch hier sämtliche Daten zuerst über den Router geleitet werden, bevor sie am Socket des MotionBuilder-Plugins ankommen.

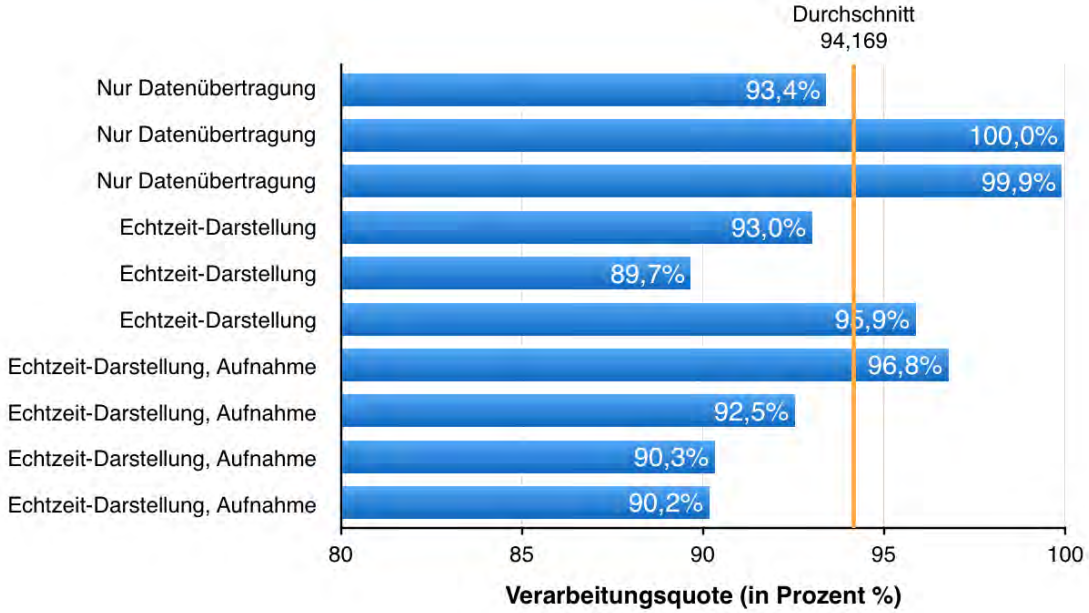


Abbildung 6.2: Verarbeitungsquoten der Nutzdatenpakete des Leap Motion Tracking-Servers. Gemessen über zehn Testverbindungen, mit je drei Minuten Testzeit und 60 Samples pro Sekunde.

Die durchgeführten Testverbindungen ergaben eine durchschnittliche Verarbeitungsquote von 94,2 Prozent, was einer effektiven Samplerate von 56,5 Samples pro Sekunde entspricht. Weiterhin lässt sich beobachten, dass die Verarbeitungsquote über den Verlauf der Tests sehr stark schwankt. Zwei der Verbindungen konnten fast keine verspäteten Datenpakete verzeichnen, während ein Testdurchlauf mehr als 10 Prozent der gesendeten Daten nicht verarbeitete. Damit ist der Datenverlust des Leap Motion-Server im Vergleich zur iPad-Anwendung deutlich größer, bewegt sich jedoch noch unterhalb der gesetzten Grenze von 20 Prozent und ist somit noch als produktionstauglich einzustufen. Ein möglicher Grund für die verzögerte Übertragung der Datenpakete ist die sechsfache Größe der Nutzdatenpakete im Vergleich zum ersten Anwendungsbeispiel. Dies macht sie entsprechend anfälliger für Verzögerungen und kurzzeitige Störungen in der Netzwerkkommunikation.

Das Balkendiagramm in Abbildung 6.3 zeigt interessanterweise, dass sich die Verarbeitungsquoten mit zunehmender Laufzeit der Datenübertragung verändern. In einstündigen Verbindungstests erreichte der Leap Motion-Server sehr gute Verarbeitungsquoten und sogar fast die festgelegte Samplerate von 60 Samples pro Sekunde. Das iPad-Anwendungsbeispiel konnte eine effektive Samplerate von 57,4 Samples pro Sekunde verzeichnen. In einem weiteren Test mit beiden Verbindungen parallel aktiv, verbesserte sich die Verarbeitungsquote noch einmal minimal. Dies ist wahrscheinlich auf Schwankungen in der Auslastung des Drahtlosnetzwerks zurückzuführen, welches parallel von anderen Geräten für die Kommunikation mit dem Internet verwendet wurde. Die Auswertung der Tests zeigt, dass beide Anwendungsbeispiele auch über lange Testzeiträume zuverlässige Ergebnisse erzeugen können und sich

unter diesem Aspekt für den Einsatz in der Produktion qualifizieren. Die bessere Leistung des Leap Motion-Server lässt sich vermutlich auf die leistungsstärkere Desktop-Hardware von *System 1* zurückführen, da das verwendete iPad Air 2 über deutlich weniger Arbeitsspeicher und Prozessorleistung verfügt.

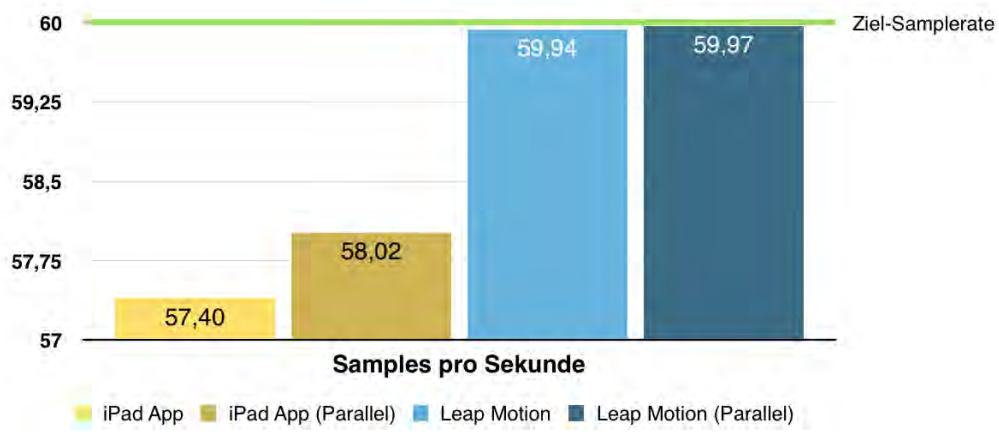


Abbildung 6.3: Langzeittest der Verarbeitungsquote, hier dargestellt als die effektive Samplerate in MotionBuilder. Die Messungen erfolgten über den Zeitraum von je einer Stunde mit gesendeten 60 Samples pro Sekunde.

Zur Messung der Latenz zwischen dem Sendevorgang der Nutzdaten und der Darstellung in der MotionBuilder Realtime-Engine, wurde ein optisches Verfahren genutzt. Mithilfe eines einzelnen Fotos wurden die angezeigten Nachrichtennummern in der grafischen Oberfläche der Datenquelle und MotionBuilder verglichen. Mithilfe der Differenz aus den zeitgleich aufgenommenen Nachrichtennummern lässt sich die Verzögerung zwischen Datenquelle und Realtime-Engine bestimmen. Das verwendete Verfahren ist jedoch nie fehlerfrei und lediglich eine Schätzung, da sich beide grafischen Oberflächen in unterschiedlichen Intervallen aktualisieren und somit nicht sichergestellt ist, dass die angezeigten Nachrichtennummern den jeweils aktuellsten Stand repräsentieren.

In Testreihen zu Eingabe-Latenzen in Videospielen[2] wurden bereits Latenzen von 200 Millisekunden als störend empfunden. Wie die Graphen in Abbildung 6.4 und Abbildung 6.5 zeigen, bewegen sich die geschätzten Latenzen der beiden getesteten Anwendungsbeispiele im Bereich zwischen 30 und 220 Millisekunden.

Die iPad-Anwendung verursachte mit durchschnittlich 91 Millisekunden eine geringere Latenz, als der Leap Motion-Server. Dessen größere Nutzdatenpakete wurden im Durchschnitt innerhalb von 125 Millisekunden übertragen und verarbeitet. Damit sind die Latenzen im Betrieb beider Anwendungen für den Anwender spürbar. Sie bewegen sich jedoch zeitgleich in einem Rahmen, in dem sich der Anwender auf die Latenzen einstellen und mit den Anwen-

[2]http://bit.ly/1R1l1Yo (Zuletzt abgerufen: 06.03.2016)

dungen produktiv arbeiten kann. Diese akzeptablen Verzögerungszeiten werden trotz Nutzung einer TCP-Verbindung, durch die zuvor beschriebene Vernachlässigung von verspäteten Nutzdaten ermöglicht. Die Einschätzung der Störung durch hohe Latenzen ist jedoch nicht allgemeingültig und muss daher im Rahmen der Evaluation einer neuen Datenquelle erneut überprüft werden.

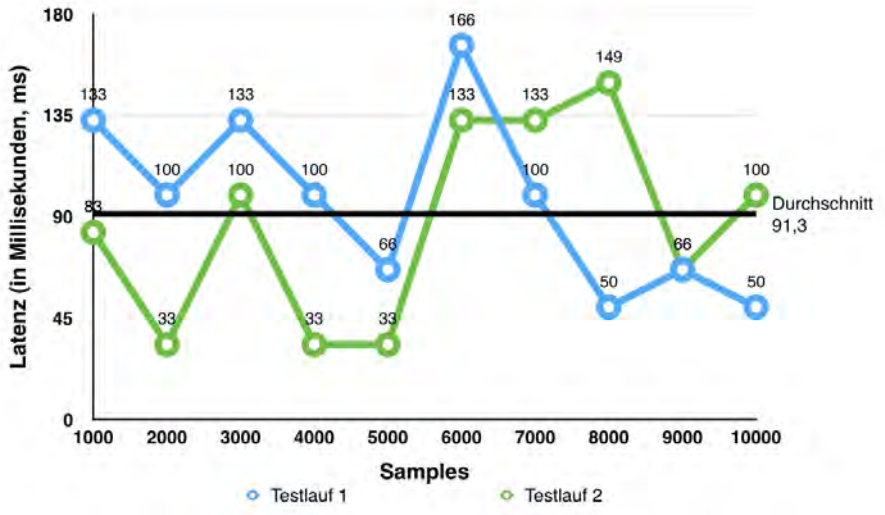


Abbildung 6.4: Schätzung der Latenzen zwischen der iPad-App und dem Plugin-Prototypen, gemessen in zwei Testläufen mit je 10000 Samples, bei 60 Samples pro Sekunde.

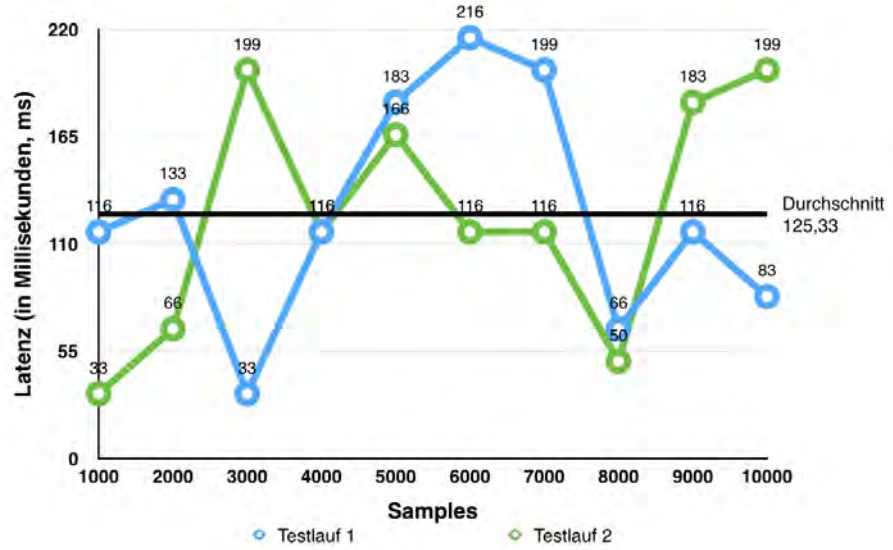


Abbildung 6.5: Schätzung der Latenzen zwischen dem Leap Motion-Server und dem Plugin-Prototypen, gemessen in zwei Testläufen mit je 10000 Samples, bei 60 Samples pro Sekunde.

6.1.3 Datenrate

Um die Einschränkungen der Netzwerkkompatibilität des Plugin-Prototypen zu bestimmen, wurde eine Reihe von Tests durchgeführt, die Erkenntnisse zur Datenrate der Software im Netzwerk liefern. Als Datenrate wird die Datenmenge definiert, die maximal über eine Netzverbindung in einem festgelegten Zeitfenster gesendet werden kann. Zwecks der Evaluation wird der Begriff auf die Menge der Daten, die das Plugin mit einer akzeptablen Quote verarbeiten kann, ausgedehnt. Die Untergrenze wird mit einer Verarbeitungsquote von 80 Prozent definiert.

Zur Messung der notwendigen Netzwerk-Bandbreite für eine stabile Kommunikation zwischen Datenquelle und Plugin wurde eine Testreihe mit der iPad-App zur Kamerasteuerung durchgeführt. Das Anwendungsbeispiel wurde gewählt, da es einer geringeren Bandbreite bedarf als der Leap Motion-Server, welcher deutlich mehr Datensätze in einem einzelnen Nutzdatenpaket überträgt. Zur Simulation unterschiedlicher Netzwerkumgebungen wurde der *Network Link Conditioner*[3] des iOS-Betriebssystems verwendet. Dieser ist Teil der iOS-Entwicklerwerkzeuge und ermöglicht unter anderem, das künstliche Einschränken der Bandbreite des iPad-Netzwerkadapters.

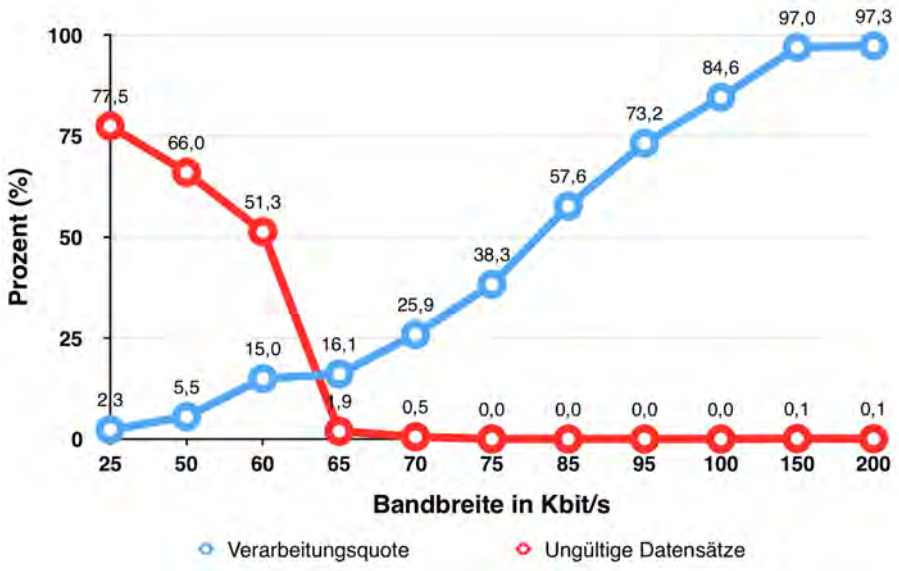


Abbildung 6.6: Messung der Verarbeitungsquote und des prozentualen Anteil der ungültigen Datensätze bei sehr eingeschränkter Netzwerkbandbreite, mithilfe des iPad-Anwendungsbeispiels (60 Samples pro Sekunde) und dem iOS-*Network Link Conditioner*.

[3] http://apple.co/1oT02ba (Zuletzt abgerufen: 21.02.2016)

In der Testreihe, zu sehen in Abbildung 6.6, wurden Bandbreiten zwischen 25Kbit und 200 Kbit pro Sekunde untersucht. Die iPad-App sendete dazu 60 Samples pro Sekunde an das MotionBuilder-Plugin, über die in der Bandbreite eingeschränkte Socket-Verbindung. Dies entspricht einer Datenmenge von circa 56Kbit pro Sekunde. Um diese Datenmenge mit einer ausreichenden Verarbeitungsquote von über 80 Prozent zu übertragen, ist eine Bandbreite von mindestens 100Kbit pro Sekunde erforderlich.

Ab 150Kbit pro Sekunde ist nur noch ein sehr geringfügiger Verlust an Nutzdaten zu verzeichnen, da ausreichend Bandbreite zur Verfügung steht und die Nutzdatenpakete ohne große Verzögerung eintreffen und verarbeitet werden. Unterhalb von 65Kbit pro Sekunde, reicht die Bandbreite nicht für eine zeitnahe Übertragung aus und ein Großteil der Datenpakete wird nur in Bruchstücken vom Netzwerksocket gelesen. Diese Nutzdatenpakete sind unbrauchbar und werden ebenso verworfen wie zu langsame Nutzdatenpakete.

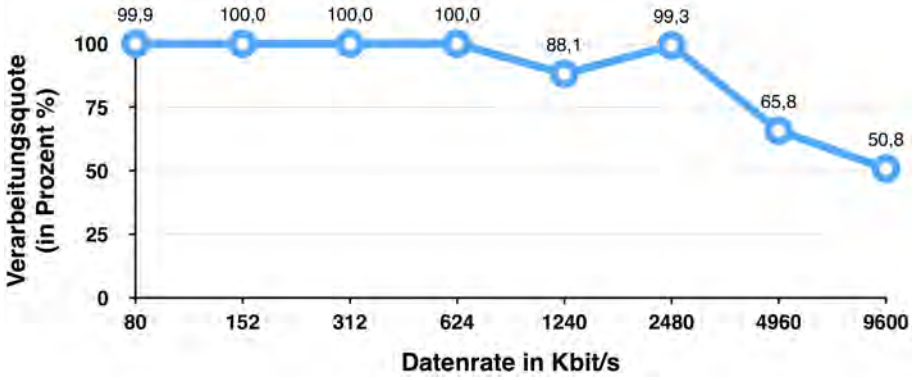


Abbildung 6.7: Die ermittelten Verarbeitungsquoten von *System 1*, im Verhältnis zur Datenrate in *Netzwerk 2*.

In einer weiteren Testreihe wurde die maximale Datenrate ermittelt. Dazu wurde der Leap Motion-Server in *Netzwerk 2* mit unterschiedlichen Sampleraten betrieben und so die Obergrenze der Bandbreite bestimmt. Im Fall von *System 1* lassen sich Sampleraten von bis zu 480 Samples pro Sekunde nutzen, ohne einen nennenswerten Datenverlust durch Verzögerung zu riskieren (siehe Abbildung 6.7). Dies entspricht einer effektiven Datenrate von circa 2480Kbit pro Sekunde.

Im Fall von *System 2* ist die maximale Datenrate, mit gerade einmal 256Kbit pro Sekunde bzw. 50 Samples pro Sekunde, deutlich geringer (siehe Abbildung 6.8). Bereits bei 280Kbit pro Sekunde verarbeitete MotionBuilder nur noch 37,1 Prozent der eingegangenen Nutzdatenpakete. Nach näherer Untersuchung konnte das Problem auf die TCP-Implementierung des verwendeten Windows 8 Betriebssystems eingegrenzt werden.

Die Socketverbindung scheint ab einer Datenrate von 256Kbit pro Sekunde Pakete zusammenzufassen, was ihre Ankunft am Socket minimal verzögert und einen Datenstau verursacht. Der Plugin-Prototyp interpretiert diesen Stau wie eine starke Verzögerung und

verarbeitet lediglich das erste Paket, dass aus dem Puffer gelesen wird. Dies hat zur Folge, dass die Verarbeitungsquote, und damit auch die effektive Samplerate, einbricht. Es lässt sich also festhalten, dass der aktuelle Prototyp nicht mit jedem Windows-System gleichermaßen effektiv arbeitet. Gegebenenfalls ist eine Anpassung der Netzwerkkonfiguration zur zuverlässigen Übertragung von großen Datenmengen notwendig.

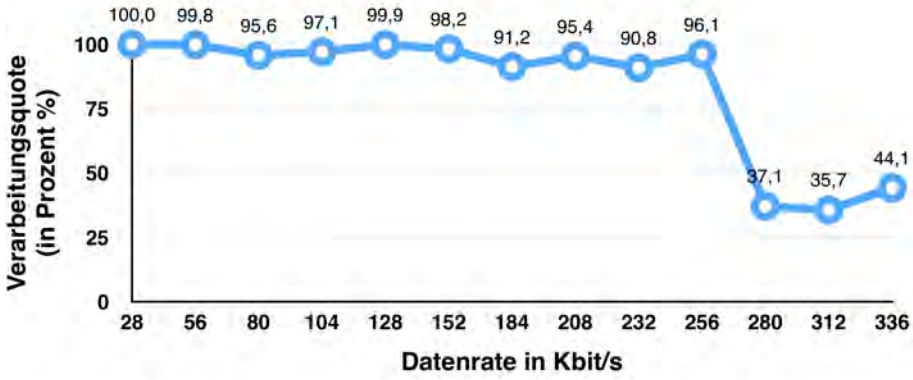


Abbildung 6.8: Die ermittelten Verarbeitungsquoten von *System 2*, im Verhältnis zur Datenrate in *Netzwerk 2*.

6.2 Evaluationsbogen

Zur Bewertung der nicht messbaren Aspekte, wie z.B. der Benutzerfreundlichkeit und Intuitivität der Prototypen, wurden im Rahmen der Evaluation eine Gruppe aus insgesamt elf Probanden mit dem System vertraut gemacht. Diese führten jeweils mehrere Feldtests durch und bewerteten ihre Erfahrung in einem vorgefertigten Evaluationsbogen. Bei den Testpersonen handelte es sich um Mitarbeiter der *metricminds GmbH & Co. KG*. Die Testumgebung bestand aus *System 1* mit einer Drahtlos-Verbindung zu *Netzwerk 1*, sowie einem *iPad Air 2* für die Ausführung der App, zur Kamerasteuerung.

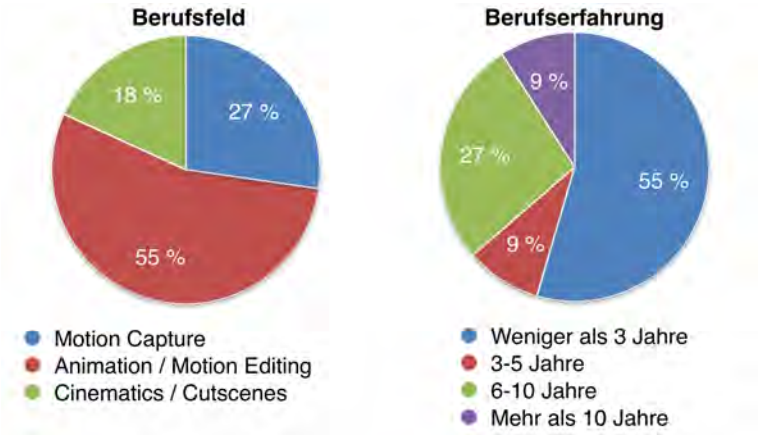


Abbildung 6.9: Berufsfelder und -erfahrung der Testpersonen.

Für die Tests wurden Mitarbeiter aus unterschiedlichen Berufsfeldern, mit unterschiedlicher Berufserfahrung in der 3D-Branche gewählt. Fast zwei Drittel der Testpersonen arbeiteten erst fünf Jahre oder weniger in ihrem Berufsfeld. Alle Probanden hatten jedoch ausreichend Vorerfahrung mit Autodesk MotionBuilder und konnten die Verwendung des Prototypen in den Kontext ihres regulären Arbeitsablaufs setzen.

Mit 55 Prozent der Befragten, arbeitet der Großteil der Probanden im Bereich Animation und Motion Editing. Sie sind die wahrscheinlichsten Anwender für das umgesetzte Leap Motion-Anwendungsbeispiel, da die Animation von Fingern bei *metricminds* derzeit in der Post-Production angesiedelt ist. 18 Prozent der befragten Mitarbeiter sind im Feld der Cinematics und Cutscenes tätig. Für sie ist insbesondere das iPad-Anwendungsbeispiel zur Steuerung einer virtuellen Kamera von Interesse, da die virtuelle Kameraführung ein wichtiger Aspekt bei der Erstellung dieser Inhalte ist.

6.2.1 MotionBuilder Device

Im Rahmen der Evaluation wurden die Probanden mit der Erzeugung einer Device-Instanz des Plugins, sowie der Herstellung einer Verbindung zur gewünschten Datenquelle vertraut gemacht. Im Evaluationsbogen bewerteten über 90 Prozent der Befragten die Einrichtung des Device-Prototypen in MotionBuilder mit der Note *Gut* oder *Sehr Gut*. Lediglich eine

Testperson vergab die Note *Befriedigend*. Die Benutzerfreundlichkeit der grafischen Oberfläche bewerteten ca. 18 Prozent mit der Note *Sehr Gut* und 64 Prozent mit der *Gut*. Je neun Prozent vergaben *Befriedigend* und *Ausreichend*. Unter den genannten Verbesserungsvorschlägen fand sich unter anderem der Wunsch nach erläuternden Hilfetexten und einer automatischen Auflistung aller verfügbaren Datenquellen im Netzwerk. Der Wunsch nach „einfacherem Plug & Play“ zeigt, dass die Schritte zum Aufbau einer Datenverbindung, in zukünftigen Versionen des Prototypen, noch optimiert werden können.

Auf die Frage, welche Vorteile sie in der Nutzung des Device für ihren Workflow sehen, wurden als Beispiele „schnelleres Prototyping“ und die nachträgliche Ergänzung der Szene um „natürliche Animationen“ genannt. Das positive Fazit der Testpersonen bestätigt den Erfolg des Prototypen hinsichtlich der Benutzerfreundlichkeit und Zugänglichkeit.

	Sehr gut (1)		Gut (2)		Befriedigend (3)		Ausreichend (4)		Mangelhaft (5)		Ungenügend (6)		Durchschnitt
	Σ	%	Σ	%	Σ	%	Σ	%	Σ	%	Σ	%	Ø
Einrichtung des Device	5	45,45	5	45,45	1	9,09	0	0,00	0	0	0	0	1,64
Benutzerfreundlichkeit der Benutzeroberfläche	2	18,18	7	63,64	1	9,09	1	9,09	0	0	0	0	2,10

Abbildung 6.10: Evaluationsergebnisse des Device-Prototypen.

6.2.2 iPad Kamerasteuerung

Zur Messung des Erfolgs des iPad Anwendungsbeispiels, wurde den Testern eine Motion-Builder Testszene, mit dem in Kapitel 5.3.3 besprochenen Relation Constraint-Aufbau, zur Verfügung gestellt. Zusätzlich wurde der Szene eine Beispielanimation hinzugefügt, damit die Testpersonen ein Objekt vor Augen hatten, dass es zu filmen galt. Dabei sollten alle wichtigen Funktionen der App ausprobiert werden: Rotation und Translation der Kamera, sowie die Steuerung der Aufnahme über Touch-Eingaben.

	Sehr gut (1)		Gut (2)		Befriedigend (3)		Ausreichend (4)		Mangelhaft (5)		Ungenügend (6)		Durchschnitt
	Σ	%	Σ	%	Σ	%	Σ	%	Σ	%	Σ	%	Ø
Bedienung der App	1	9,09	8	72,73	2	18,18	0	0,00	0	0	0	0	2,09
Intuitivität der Kamerasteuerung	3	27,27	3	27,27	4	36,36	1	9,09	0	0	0	0	2,27
Steuerung der Rotation	3	27,27	7	63,64	1	9,09	0	0,00	0	0	0	0	1,82
Steuerung der Translation	2	18,18	2	18,18	5	45,45	2	18,18	0	0	0	0	2,64
Steuerung der Aufnahme	4	36,36	5	45,45	1	9,09	1	9,09	0	0	0	0	1,91
Latenz im Betrieb	3	27,27	5	45,45	3	27,27	0	0,00	0	0	0	0	2,00
Stabilität der Verbindung	4	36,36	7	63,64	0	0,00	0	0,00	0	0	0	0	1,64
Produktionstauglichkeit	1	9,09	7	63,64	2	18,18	1	9,09	0	0	0	0	2,27

Abbildung 6.11: Evaluationsergebnisse des iPad-Anwendungsbeispiels.

Mit der Durchschnittsnote *Gut* (2,09), zeigten sich die Tester zufrieden mit der generellen Bedienung der iPad Anwendung. Über die Hälfte der Tester (55 Prozent) empfanden die Steuerung der virtuellen Kamera mit der entwickelten iPad-App als intuitiv (*Sehr gut, Gut*), während 45 Prozent eine neutrale Stimme (*Befriedigend, Ausreichend*) abgaben. Mit einer durchschnittlichen Bewertung von 1,82 (*Gut*), bekam insbesondere die Präzision der Rotationssteuerung mittels Inertialsensoren, positives Feedback. Die Steuerung der Translation, über die virtuellen Joysticks, bekam hingegen gemischte Rückmeldungen mit 18 Prozent der

Bewertungen nur *Ausreichend*, sowie einer Durchschnittswertung von 2,64 (*Befriedigend*). Ein Tester wünschte sich eine anpassbare bzw. organischere Platzierung der Joysticks, um eine komfortablere Bedienung zu ermöglichen. Weiterhin wurde die zentrale Position der Buttons zur Aufnahmesteuerung bemängelt, da der Anwender eine Hand vom Gerät nehmen muss, um die Schaltflächen betätigen zu können. Abgesehen von diesem Verbesserungsvorschlag, bewerteten die Tester die Aufnahmesteuerung im Durchschnitt mit *Gut* (1,91).

Die Latenz im Betrieb des iPad-Anwendungsbeispiels wurde schon im vorherigen Kapitel besprochen und im Durchschnitt auf ca. 90 Millisekunden geschätzt. Die Auswertung des Evaluationsbogens ergänzt diese Einschätzung um den subjektiven Eindruck der Tester. Diese bewerteten die spürbare Latenz im Durchschnitt mit *Gut* (2,00) und bemerkten keine störende Verzögerung in der Datenübertragung. Gleiches gilt auch für die Stabilität der Verbindung zum MotionBuilder-Device, welche von den Testpersonen die Durchschnittsnote 1,64 (*Gut*) erhielt. Insgesamt hielten etwa 73 Prozent der Befragten das Anwendungsbeispiel für Produktionstauglich, während 27 Prozent sich neutral demgegenüber äußerten.

Die Frage nach Zusatzfunktionalität, beantworteten mehrere Tester mit dem Wunsch nach der Darstellung des Kamerabilds auf dem Touchscreen des iPad. Laut Probanden ist der Blickwechsel zwischen iPad und MotionBuilder nicht nur verwirrend, eine Darstellung des Kamerabilds auf dem iPad würde auch die Bedienung der virtuellen Joysticks erleichtern. Weiterhin besteht der Wunsch nach Touchscreen-Bedienelementen für den Kamerazoom und den Wechsel der MotionBuilder-Aufnahme. Mehrere Tester konnten sich vorstellen, die iPad Anwendung in ihren Arbeitsablauf aufzunehmen, um Kamerafahren authentischer zu gestalten und z.B. „Camerashake“ zu erzeugen.

6.2.3 Leap Motion Tracking

Zur Messung des Erfolgs des zweiten Anwendungsbeispiels, dem Leap Motion-Server, wurde den Testpersonen auch hier eine Testszene in MotionBuilder zur Verfügung gestellt. Der Aufbau dieser entsprach dem in Kapitel 6.2.3 beschriebenen Szenenaufbau zum Finger-Tracking mithilfe des Leap Motion-Controller.

Die Testpersonen sollten unterschiedliche Fingerposen ausprobieren und die Reaktionszeit des Systems testen, um sich ein Bild von der Produktionstauglichkeit des Anwendungsbeispiels machen zu können. Im Gegensatz zur Evaluation des iPad-Anwendungsbeispiels, wurde die grafische Oberfläche der Server Anwendung keiner Bewertung unterzogen, da diese keinerlei Relevanz im aktiven Betrieb des Trackings hat.

	Sehr gut (1)		Gut (2)		Befriedigend (3)		Ausreichend (4)		Mangelhaft (5)		Ungenügend (6)		Durchschnitt
	Σ	%	Σ	%	Σ	%	Σ	%	Σ	%	Σ	%	Ø
Finger-Tracking Präzision	0	0,00	4	36,36	4	36,36	3	27,27	0	0	0	0	2,91
Hand-Tracking Präzision	1	9,09	3	27,27	5	45,45	2	18,18	0	0	0	0	2,73
Stabilität der Verbindung	1	9,09	6	54,55	3	27,27	1	9,09	0	0	0	0	2,36
Latenz im Betrieb	2	18,18	4	36,36	4	36,36	1	9,09	0	0	0	0	2,36
Produktionstauglichkeit	2	18,18	3	27,27	6	54,55	0	0,00	0	0	0	0	2,36

Abbildung 6.12: Evaluationsergebnisse des Leap Motion Tracking-Anwendungsbeispiels.

Die ersten Fragen des Evaluationsbogens bezogen sich auf die Präzision der übermittelten Leap Motion-Tracking Daten und deren Übersetzung auf ein Hand-Rig. Die Testpersonen bewerteten die Präzision des Finger-Tracking Prototypen im Durchschnitt mit *Befriedigend* (2,91), wobei circa 27 Prozent der Befragten nur ein *Ausreichend* vergaben. Sechs Personen nannten als Verbesserungswunsch, eine genauere Übersetzung der Leap Motion-Daten auf die digitale Hand. Auch fehlte einem Tester die Kontrolle über die Spreizwinkel der Finger, welche bisher nicht Teil des Prototypen sind. Das Tracking des Handgelenks schnitt mit einer durchschnittlichen Note von 2,73 (*Befriedigend*) etwas besser ab.

Die Stabilität der Datenverbindung zu MotionBuilder bewerteten die Tester durchschnittlich mit *Gut* (2,36), was jedoch deutlich unter der Durchschnittsnote von 1,63 liegt, welche die Stabilität der iPad-Verbindung erhielt. Mit einer Durchschnittsnote von 2,36, wurde auch die spürbare Latenz im Betrieb etwas schlechter eingestuft als es bei der iPad-Anwendung (2,00) der Fall war. Diese Ergebnisse korrelieren mit den Testergebnissen, der in Kapitel 6.1.2 durchgeführten Benchmark-Tests, für kurze Datenverbindungen. Es lässt sich also festhalten, dass die Latenz und Stabilität der Verbindung im Betrieb, abhängig von der zu transportierenden Datenmenge ist.

Trotz der durchwachsenen Meinung bezüglich der Genauigkeit des Finger-Trackings, gaben 45 Prozent der Befragten positives Feedback zur Produktionstauglichkeit des Anwendungsbeispiels. Die Durschnittsnote lag hierfür bei 2,63 (*Befriedigend*). Zwei der Tester wünschten sich das Speichern einzelner Handposen als Zusatzfunktionalität, da die manuelle Erstellung dieser in der Produktion sehr zeitaufwändig ist. Weiterhin wurde der Wunsch nach einer einfachen Sperrung der Finger geäußert, sodass lediglich einzelne Finger aufgenommen werden können. Mehrere Tester konnten sich vorstellen, das Anwendungsbeispiel in ihren Arbeitsablauf aufzunehmen, um schnell, natürliche Hand- und Fingerposen zu erzeugen.

6.3 Ergebnisse

Basierend auf den Benchmark-Testergebnissen, sowie der ausgewerteten Evaluation, lässt sich ein abschließendes Fazit zum Erfolg der Software-Prototypen ziehen. Wie in Kapitel 1.3 ausführlich beschrieben, war es das Ziel der Arbeit, eine standardisierte Netzwerkschnittstelle für die MotionBuilder Realtime-Engine zu entwickeln, die es ermöglicht, neue Eingabegeräte mit wenig Arbeitsaufwand in die Produktion einzubinden. Diese soll den technischen Aufwand moderner Virtual Production-Pipelines verringern und die Flexibilität in allen Phasen der Produktion erhöhen. Um die Möglichkeiten des Plugins unter Beweis zu stellen, wurden weiterhin zwei Anwendungsbeispiele konzipiert und entwickelt, die zeitaufwändige Prozesse in der Produktion von Animationen, vereinfachen sollen.

In Kapitel 6.1 wurde mittels umfassender Benchmark-Tests belegt, dass der entwickelte Plugin-Prototyp funktionstüchtig ist und die technischen Anforderungen an die Netzwerkschnittstelle (siehe Kapitel 4.1.1) erfüllt. Zu diesen gehört unter anderem der stabile Betrieb des Plugins in einer lokalen Netzwerkumgebung, geringe Latenzen, sowie die Verarbeitung einer variablen Menge numerischer Daten. Letzteres ermöglicht die Verwendung der Schnittstelle mit unterschiedlichen Datenquellen. Die erfolgreiche Verbindung zweier unterschiedlicher Eingabegeräte (iPad, Leap Motion-Controller) über das Plugin und die Umsetzung der kompletten Logik im Relation Constraint, beweisen die universelle Einsetzbarkeit der Netzwerkschnittstelle. Die Testergebnisse enthüllen jedoch auch, dass der entwickelte Prototyp noch nicht auf allen getesteten Systemkonfigurationen die gleiche Datenrate bietet.

Die Auswertung des Evaluationsbogens zeigt, dass der Prototyp aus Sicht der typischen Anwendergruppe, die gesetzten Ziele erfüllt. Die entwickelte Netzwerkschnittstelle ermöglichte es, zwei zuvor problematische Arbeitsabläufe, mit nur wenig Programmieraufwand, um hilfreiche Eingabemethoden zu ergänzen. Die Auslagerung der Daten-Interpretation in ein Relation Constraint war dabei ausschlaggebend für die schnelle Entwicklung funktionsfähiger Prototypen. Die umgesetzten Anwendungsbeispiele wurden dabei von den Testpersonen als produktionstauglich eingestuft.

Die Testpersonen zeigten sich zufrieden mit der iPad Kamerasteuerung und konnten sich vorstellen, die Anwendung zur Erzeugung natürlicher Kamerabewegungen in ihren Pre- und Post-Production Workflow aufzunehmen. Auch das umgesetzte Leap Motion Hand- und Finger-Tracking in MotionBuilder sehen die Tester als intuitives Werkzeug zur schnelleren Erstellung von natürlichen Hand- und Fingerposen. Für die Aufnahme kompletter Fingeranimationen, waren die Resultate laut der Testgruppe, jedoch noch nicht präzise genug. Die spürbaren Latenzen sind dank selektiver Datenverarbeitung geringfügig genug, sodass die Anwendungsbeispiele für den Echtzeit-Betrieb geeignet sind.

Basierend auf der durchgeführten Evaluation lässt sich abschließend festhalten, dass die entwickelte Software die empirisch geforderten Daten bereitstellt und die Produktionstauglichkeit der Prototypen in ersten Tests nachwies. Das Plugin beschleunigt die Einbindung neuer Datenquellen in die Virtual Production-Pipeline durch Verringerung des technischen

Aufwands seitens MotionBuilder und beweist sich in den durchgeführten Tests als zuverlässige Schnittstelle zur Erweiterung der Realtime-Engine in der Animationssoftware. Die Auslagerung der Programmlogik zur Interpretation eingehender Daten mittels Relation Constraints ermöglicht es Anwendern, ohne Programmierkenntnisse Anpassungen an der Logik im Betrieb der Software vorzunehmen. Das Plugin erweitert die Virtual Production-Pipeline um ein standardisiertes Werkzeug, das in allen Phasen der Produktion Einsatz findet und sowohl die Effizienz als auch die Effektivität steigert.

Kapitel 7

Zusammenfassung und Ausblick

Mit dem Wandel des traditionellen Produktionsablaufs von visuellen Effekten hin zu den modernen Methoden der Virtual Production, ändern sich auch die Anforderungen an die Technik. Die Virtual Production Pipeline greift auf Technologien wie Motion Capture und Echtzeit-Visualisierung von 3D-Animationen zurück, um die Produktion von visuellen Effekten intuitiver und effizienter zu gestalten. Virtual Cinematography-Technologien wie z.B. virtuelle Kamera-Systeme, sind ein Teil der Virtual Production und ermöglichen den natürlichen Umgang mit digitalen Effekten, als wären sie ein Teil der realen Welt. Gerade die Pre-Visualisierung profitiert von der heutigen Technik und setzt zunehmend auf 3D-Animationen anstatt auf klassische Methoden wie das Storyboard. Autodesk MotionBuilder ermöglicht den Einsatz dieser Techniken, da es komplexe virtuelle Szenen in Echtzeit darstellen kann.

Während proprietäre Plugins von unterschiedlichen Motion Capture-System Herstellern für MotionBuilder existieren, gibt es darüber hinaus keine einheitliche Schnittstelle zur Einbindung neuer Eingabegeräte. Die Verbindung einer solchen Datenquelle mit MotionBuilder bedeutet somit, dass die Programmierung eines neuen Plugins notwendig ist. Dieser Programmieraufwand sollte so gering wie möglich gehalten werden, damit mit jeder neuen Datenquelle effizienter und standardisiert umgegangen werden kann. Diese Arbeit befasst sich deshalb mit der Entwicklung einer standardisierten Netzwerkschnittstelle, die ein einheitliches Kommunikationsprotokoll definiert. Dies erlaubt die Kommunikation beliebiger Datenquellen mit der MotionBuilder Realtime-Engine. Die komplette Logik zur Interpretation der von dem Plugin bereitgestellten Daten innerhalb von MotionBuilder wird in einem Relation Constraint umgesetzt. Dabei handelt es sich um ein kausales Netzwerk aus Szenenobjekten und Operatoren, die mittels Ein- und Ausgängen, in Beziehungen zueinander gesetzt werden. Die grafische Repräsentation der Netzwerke ermöglicht ein schnelles Aufsetzen und Anpassen der Logik, sogar im Echtzeit-Betrieb. Diese standardisierte Herangehensweise an die Schnittstellen-Programmierung und die Bereitstellung der Daten, verringert den technischen Aufwand und erhöht die Flexibilität in der Produktion.

Zur Umsetzung der Schnittstelle wurde das C++-basierte Open Reality SDK genutzt, welches MotionBuilder 2015 beiliegt und die Entwickung von DLL-basierten Erweiterungen

ermöglicht. Das Open Reality SDK bietet dabei vollen Zugriff auf alle MotionBuilder-Programmierschnittstellen und zeitgleich eine wesentlich bessere Performance als das Python-SDK. Das Plugin ist als *Device* realisiert. Devices repräsentieren Schnittstellen zur Ein- und Ausgabe von Animationsdaten in MotionBuilder.

Die Kommunikation zwischen Device und Datenquelle erfolgt über eine Socketverbindung auf TCP-Basis nach einem standardisierten Kommunikationsprotokoll. Dieses erfordert die Übermittlung der Datenstruktur von der Quelle an das Device, damit dieses entsprechende Animationsknoten in MotionBuilder erzeugen und bereitstellen kann. Anschließend wird ein Nutzdaten-Fluss in Echtzeit vom Device bei der Datenquelle angefordert und die eingehenden Daten in die angelegten Animationsknoten zeitdiskret geschrieben. Weiterhin können diese Daten als Keyframes gespeichert werden. Um störende Latenzen zu vermeiden, werden stark verspätete Datenpakete nicht verarbeitet, sondern direkt verworfen. Die Datenstruktur wird außerdem in der MotionBuilder-Szene gespeichert um sie für die erneute Verwendung bereitzustellen.

Zur Demonstration der Fähigkeiten der Netzwerkschnittstelle wurden im Rahmen dieser Arbeit zwei Anwendungsbeispiele als Datenquellen für die Verwendung im Bereich der Pre- und Post-Production entwickelt:

Das erste Anwendungsbeispiel ist dem Gebiet der Virtual Cinematography zuzuordnen und erzeugt natürliche Kamerarotationen mithilfe einer iOS Anwendung für das Apple iPad. Die iOS App nutzt die Daten der im iPad integrierten Inertialsensoren, um die lokale Rotation des Gerätes zu bestimmen und diese in Form von Roll-Neig-Gier Winkeln an die entwickelte MotionBuilder Netzwerkschnittstelle zu übermitteln. In MotionBuilder werden diese Winkel genutzt, um die Rotation einer virtuellen Kamera zu steuern. Mangels Positions-Tracking wird dem Anwender über virtuelle Joysticks eine rudimentäre Steuerung der Kameratranslation zur Verfügung gestellt. Zusätzlich bieten mehrere Touchscreen-Bedienelemente Kontrolle über die Aufnahmesteuerung der MotionBuilder Szene. Mit dieser Anwendung werden reale Kamerabewegungen in die virtuelle Welt überführt und aufgezeichnet.

Das zweite Anwendungsbeispiel nutzt den Leap Motion-Controller um ein Hand-Rig zu animieren. Die dafür entwickelte Server-Anwendung für Mac OS X bezieht die notwendigen Hand- und Finger-Tracking Daten vom Leap Motion SDK, das der Hersteller des USB-Zubehörs für mehrere Plattformen bereitstellt. Das Anwendungsbeispiel überträgt sowohl Rotation als auch Translation für die Handflächen des Anwenders, konzentriert sich jedoch auf die Animation der Finger. Diese Daten werden über das Plugin an MotionBuilder übermittelt. In einem Relation Constraint, werden aus den Richtungsvektoren der einzelnen Fingerknochen, die Beugungswinkel der Gelenke errechnet und diese zusammen mit den Hand-Tracking-Daten auf ein Hand-Rig übertragen. Die manuelle Erstellung von Finger-Animationen in der Post-Production ist zeitaufwändig, jedoch oft unvermeidlich, da noch viele Motion Capture-Studios auf die Aufzeichnung von Fingerbewegungen am Set verzichten. Das Anwendungsbeispiel hat daher seinen größten Nutzen als Werkzeug zur Erstellung von Fingerposen und groben Animationen in der Post-Production.

Mit einer abschließenden Evaluation der entwickelten Prototypen wurde überprüft, ob das Plugin die gesteckten Anforderungen erfüllt und die umgesetzten Anwendungsbeispiele bereits produktiv genutzt werden können. Dazu wurde eine Reihe Benchmark-Tests durchgeführt, welche die technischen Aspekte der Software überprüfen. Weiterhin wurde ein Evaluationsbogen erstellt, den mehrere Mitarbeiter der Firma *metricminds*, nach einem Test der Prototypen beantworteten. Die Auswertung der Testergebnisse hat gezeigt, dass die Performance des Plugins sich in einem produktionstauglichen Rahmen bewegt. Dank selektiver Verarbeitung der eingehenden Datenpakete liegen die Latenzen trotz TCP-Verbindung im spürbaren, aber nicht störenden Bereich. In den Testreihen waren vereinzelt Datenraten von mehr als 2400 Kbit pro Sekunde möglich, ohne dabei nennenswerte Datenverluste oder störende Latenzen zu riskieren. Die maximale Datenrate ist jedoch abhängig vom eingesetzten System und schwankt je nach TCP-Konfiguration stark.

Zur Steigerung der maximalen Datenrate wäre für zukünftige Arbeiten ein alternativer Paketfilter denkbar, der die auf Fremdsystemen verzögerte TCP-Paketzustellung abfängt. Eine weitere Möglichkeit besteht in der Umsetzung einer UDP-unterstützten Hybrid-Kommunikation, die als Alternative zur reinen TCP-Verbindung auf problematischen Systemkonfigurationen verwendet werden könnte. Wichtige Steuerdaten würden dabei weiterhin über die verlustfreie TCP-Verbindung gesendet werden, während die Übertragung der Nutzdaten über eine schnellere UDP-Verbindung erfolgt. Alternativ zur Netzwerkkommunikation wäre auch die Einbindung der Datenquellen mittels serieller Schnittstelle eine Option. Über diese könnten beispielsweise kleine Computersysteme wie *Raspberry Pi*[1] oder Mikrocontroller-Platinen wie *Arduino*[2] direkt mit MotionBuilder kommunizieren, ohne dass eine Netzwerkverbindung notwendig ist. Für zukünftige Anwendungen ist es weiterhin sinnvoll, die Netzwerkschnittstelle des Plugins um Ausgabe-Funktionalität zu erweitern. So könnte das Plugin z.B. Animationsdaten in Echtzeit an andere Software übertragen. Das Plugin wäre somit nicht nur als Empfänger, sondern auch als Datenquelle einsetzbar.

Die Auswertung des Evaluationsbogen zeigt, dass die Tester zwar zufrieden mit der Bedienung der Schnittstelle sind, jedoch noch Verbesserungspotenzial beim Aufbau der Datenverbindung sehen. So würde beispielsweise eine automatische Auflistung aller verfügbaren Datenquellen im Netzwerk den Verbindungsaufbau für die Anwender erleichtern. Sowohl die iPad Kamerasteuerung als auch die Leap Motion-Tracking Applikation erhielten überwiegend positive Resonanz von den Testern. Für zukünftige Arbeiten sind dennoch einige Anpassungen an der Software zu empfehlen:

Die meistgewünschte Zusatzfunktion für die iPad Kamerasteuerung ist eine Anzeige des virtuellen Kamerabilds auf dem Touchscreen des Tablets. Dies würde die Blickwechsel zwischen iPad und MotionBuilder unnötig machen und zeitgleich die Bedienung der virtuellen Joysticks erleichtern. Ein bemängelter Aspekt des Leap Motion-Trackings ist die Genauigkeit der Finger-Animation. Diese könnte durch den Zusammenschluss mehrerer Leap

[1] https://www.raspberrypi.org/ (Zuletzt abgerufen: 29.02.2016)
[2] https://www.arduino.cc/ (Zuletzt abgerufen: 29.02.2016)

Motion-Controller verbessert werden, welche die Hand aus unterschiedlichen Blickwinkeln erfassen. Seitens des Herstellers wird lediglich ein Controller pro System unterstützt, weshalb die Tracking-Daten von mehreren Quellen im Relation Constraint zusammengesetzt werden müssten.

Andere Anwendungen des Plugins wären beispielsweise der Empfang und die Übertragung von Facial Motion Capture-Daten in Echtzeit auf einen virtuellen Charakter in MotionBuilder. Weiterhin wäre das Tracking von Körperteilen mittels Inertialsensoren ohne Drittherstellersoftware in Echtzeit umsetzbar. Ebenfalls denkbar wäre eine Anwendung auf Basis mehrerer Datenquellen, ähnlich dem Forschungsprojekt von Penelle und Debeir [PD14]. Diese kombinieren die Daten des Leap Motion-Controllers und Microsoft Kinect, um verbesserte Hand-Tracking-Ergebnisse zu erhalten. Zusammenfassend lässt sich feststellen, dass das Plugin die Entwicklung vieler Anwendungen ermöglicht, die Virtual Production-Pipelines in unterschiedlichen Phasen der Produktion effektiv bereichern und effizienter gestalten.

Glossar

C++	Erweiterte, sehr effiziente Form der Programmiersprache *C*, die sowohl zur System- als auch zur Anwendungsprogrammierung verwendet werden kann.
Dynamically Linked Library (DLL)	Dynamische Programmbibliothek für Windows. Eine DLL-Datei kann von mehreren Anwendungen zeitgleich genutzt werden.
Entwicklungsumgebung (IDE)	Anwendungspaket zur Softwareentwicklung. Beispiele: *Microsoft Visual Studio, Apple Xcode*.
FBX-Dateiformat	Proprietäres Dateiformat der Firma *Autodesk*, welches ursprünglich für *MotionBuilder* entwickelt wurde und den Datenaustausch zwischen unterschiedlichen 3D-Anwendungen ermöglicht.
Gyrometer (Gyroskop)	Drehratensensor zur Registrierung von Rotationsbewegungen. Siehe Kapitel 3.3.2.
Inertialsensoren	Gruppenbezeichnung für *Beschleunigungs-* und *Gyrosensoren*. Siehe Kapitel 3.3.
Keyframe Animation	Traditionelle 2D / 3D Animationstechnik, die auf Schlüsselbildern und Zwischenbildern basiert.
Leap Motion	Hard- und Softwarelösung für optisches Tracking von Fingern und Händen. Erfordert einen Leap Motion-Controller. Siehe Kapitel 3.4.
MessagePack	Software-Standard zur plattformunabhängigen Übertragung von Daten. MessagePack benutzt zur Daten-Kodierung Bytecode. Siehe Kapitel 4.1.8.

MotionBuilder	ehem. *Filmbox*. 3D-Animationssoftware der Firma *Autodesk* mit hohem Verbreitungsgrad auf dem Gebiet des Motion Capture und Motion Editing. Siehe Kapitel 3.2.
Motion Capture	Verfahren zur Aufzeichnung analoger Bewegungsabläufe von Menschen und Objekten mit anschließender Übersetzung dieser, in die digitale Welt. Siehe Kapitel 3.1.
Motion Tracking	Bezeichnet unterschiedliche Verfahren zur Aufzeichnung analoger Bewegungen aus der realen Welt.
Optische Marker	Passive oder aktive Marker die Licht reflektieren bzw. ausstrahlen und so von optischen Motion Capture-Systemen erfasst werden können. Siehe Kapitel 3.1.1.
OSI-Modell	engl. *Open Systems Interconnection Model*. Verbreitetes Referenzmodell für Netzwerkprotokolle. Es handelt sich dabei um ein Modell mit sieben Schichten, dass die Entwicklung von Protokollen unterstützen soll.
Performance Capture	Erweiterte Form des Motion Capture, die auch Mimik (Gesichtsbewegungen) und feine Fingerbewegungen beinhaltet. Siehe Kapitel 3.1.3.
Plugin	Optionales Software-Modul zur Erweiterung oder Anpassung bestehender Anwendungen.
Post-Production	Nachbereitung des Filmmaterials. Visuelle Effekte werden hinzugefügt und das fertige Material für die Distribution vorbereitet.
Pre-Production	Planung und Vorbereitung der Produktion. Beinhaltet unter anderem auch die Pre-Visualisierung.
Pre-Visualisierung	Visualisierung beschriebener Szenen in der Filmproduktion. Sie kommuniziert die Vision des Regisseurs besser als eine Beschreibung in reiner Textform. Die Pre-Visualisierung kann in Form eines simplen Storyboards umgesetzt werden. Mittlerweile wird vermehrt Computergrafik zur Pre-Visualisierung von Szenen mit vielen visuellen Effekten verwendet.

Relation Constraint	Numerisches / kausales Netzwerk zwischen mehreren MotionBuilder-Objekten, um sie in eine direkte Verbindung zu setzen.
Socket	Vom Betriebssystem bereitgestellter Endpunkt einer Netzwerkverbindung (TCP / UDP). Siehe Kapitel 3.5.3.
Software Development Kit (SDK)	Anwendungspaket bestehend aus Werkzeugen zur Entwicklung neuer Software für bestimmte Hard- oder Software-Plattformen. Beispiel: *iOS SDK*.
Swift	Objektorientierte Programmiersprache von Apple. Wird hauptsächlich zur Entwicklung von iOS- und Mac-Anwendungen verwendet.
Transmission Control Protocol (TCP)	Verbindungsorientiertes Protokoll zur Netzwerkkommunikation. Siehe Kapitel 3.5.1.
User Datagram Protocol (UDP)	Verbindungsloses Protokoll zur Netzwerkkommunikation. Siehe Kapitel 3.5.1.
Virtual Camera System (VCS)	Ein tragbares Kamera-Rig, ausgestattet mit einem Bildschirm, das eine virtuelle Kamera im dreidimensionalen Raum repräsentiert. Siehe Kapitel 2.2.3.
VFX (Visual Effects)	dt. *Visuelle Effekte*. Computergenerierte Bildelemente in der Filmproduktion.
Virtual Cinematography	Beschreibt die Anwendung kinematographischer Prinzipien auf eine virtuelle Szene. Siehe Kapitel 2.1.2.
Virtual Production	Beschreibt den gesamten digitalen Workflow einer modernen Filmproduktion. Dieser umfasst meistens eine digitale Pre-Visualisierung in der Pre-Production, sowie eine Echtzeit-Visualisierung der digitalen Inhalte am Set. Siehe Kapitel 2.1.
Virtual Production-Pipeline	Hard- und Software-Werkzeuge, die in den unterschiedlichen Phasen der Virtual Production verwendet werden. Siehe Kapitel 2.2. Beispiele: *Autodesk MotionBuilder, Virtual Camera System*.

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[AHA+10] Andreadis, Anthousis ; Hemery, Alexander ; Antonakakis, Andronikos ; Gourdoglou, Gabriel ; Maur, Pavlos ; Christopoulos, Dimitrios ; Karigiannis, John N.: Real-Time Motion Capture Technology on a Live Theatrical Performance with Computer Generated Scenery. In: *2010 14th Panhellenic Conference on Informatics (PCI)*, IEEE, 2010, S. 148–152

[AM06] Ahmad, Farooq ; Musilek, Petr: UbiHand. In: *ACM SIGGRAPH 2006 Research posters*. New York, New York, USA : ACM Press, 2006, S. 159

[BD13] Balakrishnan, Girish ; Diefenbach, Paul: *Virtual Cinematography: Beyond Big Studio Production*. New York, New York, USA, Diplomarbeit, 2013

[BS05] Bronstein, I.N. (Hrsg.) ; Semendjaew, K.A. (Hrsg.): *Taschenbuch der Mathematik für Ingenieure und Studenten*. 6. Auflage. Wissenschaftlicher Verlag Harri Deutsch GmbH, 2005. – ISBN 3817120060

[ER07] Elson, D K. ; Riedl, M O.: A lightweight intelligent virtual cinematography system for machinima production. In: *AIIDE* (2007)

[GP03] Gascha, Heinz ; Pflanz, Stefan: *Physik verständlich*. Weltbild, 2003. – ISBN 9783817460649

[Ker09] Kersken, Sascha: *IT Handbuch für Fachinformatiker*. Galileo Press, 2009. – ISBN 9783836214209

[Mor15] Moran, Gavin: Pushing photorealism in a boy and his kite, ACM, Juli 2015

[Nit08] Nitsche, Michael: *Experiments in the use of game technology for previsualization*. New York, New York, USA : ACM, 2008

[OCMM08] Ortega-Carrillo, Hernando ; Martínez-Mirón, Erika: Wired gloves for every one. In: *the 2008 ACM symposium*. New York, New York, USA : ACM, Oktober 2008, S. 305–306

[OZ15] Okun, Jeffery A. (Hrsg.) ; Zwerman, Susan (Hrsg.): *The VES Handbook of Visual Effects: Industry Standard VFX Practices and Procedures*. Second Edition. Focal Press, 2015. – ISBN 9780240825182

[PD14] Penelle, Benoît ; Debeir, Olivier: *Multi-sensor data fusion for hand tracking using Kinect and Leap Motion*. New York, New York, USA : ACM, 2014

[Per12] Perl, Thomas: *Cross-Platform Tracking of a 6DoF Motion Controller*. Fakultät für Informatik der Technischen Universität Wien, TU Wien, Diplomarbeit, 2012

[SE03] Singh, Karan ; ElKoura, George: *Handrix: Animating the Human Hand*. Toronto, Canada : Department of Computer Science, University of Toronto, 2003

[Sei15] Seibert, Stefan: *Real-Time Set Editing in a Virtual Production Environment with an Innovative Interface*. Stuttgart Media University, Diplomarbeit, Februar 2015

[SSB14] Sahin, Yafes ; Spielmann, Simon ; Backhaus, Martin: Dark matter. In: *ACM SIGGRAPH 2014 Talks*. New York, New York, USA : ACM Press, 2014

[WP09] Wang, Robert Y. ; Popović, Jovan: Real-time hand-tracking with a color glove. In: *ACM Transactions on Graphics (TOG)* 28 (2009), Juli, Nr. 3, S. 63