FH JOANNEUM – University of Applied Sciences

Physiogame: Leap Motion zur Bewegungstherapie
Modulare Anwendungsentwicklung in JavaScript

Diplomarbeit

zur Erlangung des akademischen Grades eines

„Diplomingenieurs für technisch-wissenschaftliche Berufe“

eingereicht am Master-Studiengang Informationsmanagement

Verfasser:

Mario Ranftl, BSc.

Betreuer:

Prof. DI Dr. Alexander Nischelwitzer

Graz, 19. Dezember 2013

FH | JOANNEUM
University of Applied Sciences

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Graz, 19. Dezember 2013

Mario Ranftl, BSc.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AMD	Asynchronous Module Definition
API	Application Programming Interface
CSS	Cascading Style Sheet
CSV	Comma-separated Values
DOM	Domain Object Model
FPS	Frames Per Second
HCI	Human Computer Interaction
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
JS	JavaScript
JSON	JavaScript Object Notation
MIT	Masachusetts Institute of Technology
NPM	Node.js Package Manager
SDK	Software Development Kit
TDD	Test Driven Development
USB	Universal Serial Bus
VR	Virtual Reality
WebGL	Web Graphics Library

Danksagung

Ich möchte mich bei meiner Familie – Otto, Roswitha, Kerstin und Simon – für die Unterstützung und Ermöglichung meines Studiums bedanken. Ohne euch wäre alles niemals möglich gewesen. Ihr werdet immer ein großer Teil meines Lebens sein.

Ein besonderes „Danke!“ geht an Nicole Eibel und Christian Anton Fuchs. Ohne eure seelische Unterstützung wäre diese Arbeit nicht fertig, ohne eure Korrekturen wäre diese Arbeit nicht lesbar und ohne euer Feedback wäre Physiogame voller Bugs gewesen. Des Weiteren bedanke ich mich bei den Korrekturlesern Georg Felgitsch, BSc. und Armin Eibl.

Zusätzlich möchte ich mich bei meinen Betreuer Prof. DI Dr. Alexander Nischelwitzer für das Vertrauen, das Feedback und die Möglichkeit der Kooperation mit den Studiengängen Ergo- und Physiotherapie der FH JOANNEUM bedanken. Darüber hinaus danke ich auch DI (FH) Sandra Schadenbauer für die Koordination der Meetings und Mithilfe bei den Usability-Tests.

Schlussendlich bedanke ich mich bei den externen Lehrenden Julia Unger, MSc, Romana Toriser, Barbara Gödl-Purrer, MSc und Mag. Dr. rer. nat. Petra Gröbl für die Mithilfe bei der Konzeption von Physiogame und das Feedback während der Entwicklungsarbeiten. Abschließend geht noch ein Danke an die Kinder und an das Personal – insbesondere Ergotherapeutin Melanie Tax-Haarkamm und Physiotherapeutin Claudia Dockner – der „Kids-Chance“-Abteilung der Maria Theresia Klinik in Bad Radkersburg für die Ermöglichung der Usability-Tests.

Kurzfassung

Leap Motion ist eine neue natürliche Benutzerschnittstelle und ermöglicht eine präzise Bewegungserkennung von Händen und Fingern. Ein medizinisches Einsatzgebiet für Motion Tracking Technologien ist der Bereich der virtuellen Rehabilitation. Dieser Bereich ist mit Leap Motion jedoch noch unerschlossen.

Im Rahmen dieser Diplomarbeit wird die Applikation Physiogame entwickelt, die auf ergo- und physiotherapeutischen Anforderungen basiert und in der Bewegungstherapie eingesetzt werden soll. Die Umsetzung erfolgt dabei mit JavaScript (JS). Zur Implementierung von Physiogame müssen daher die Konventionen, Patterns und Fallen in JS erkannt und konzeptionelle Entscheidungen zu Modularisierung, Automatisierung, zum Testen des Quellcodes und zum Management der externen Abhängigkeiten getroffen werden.

Diese Diplomarbeit verfolgt somit multiple Ziele und besteht aus mehreren Teilen. Zu Beginn werden Motion Tracking Technologien und das Produkt Leap Motion analysiert. Anschließend werden die Anforderungen an eine Applikation zur virtuellen Rehabilitation bzw. zur Bewegungstherapie spezifiziert. Nachfolgend wird der professionelle Umgang mit JS und die Konzeption von modularen JS-Applikationen vermittelt, um schlussendlich Physiogame zu implementieren und die Einsatzfähigkeit der Applikation im Rahmen eines Usability-Tests mit Patienten und Experten zu überprüfen.

Ausgehend von den Resultaten des Usability-Tests und der Einschätzung von Experten, dürfte sich Physiogame tatsächlich zur Bewegungstherapie eignen. Mit Physiogame konnte eine interaktiv konfigurierbare Applikation entwickelt werden, die sowohl im Browser als auch als Desktop-Applikation ausgeführt werden kann.

Abstract

Leap Motion is a new natural user interface that allows to capture the movement of hands and fingers in a highly precise manner. This product could as well be applicable in the field of medicine for virtual rehabilitation systems, because such systems often exploit motion tracking technologies. In spite of that, Leap Motion is currently unused within this field.

Within this thesis the application Physiogame for the use in exercise therapy is developed. This application uses Leap Motion and addresses requirements in the fields of physiotherapy and occupational therapy. Moreover, Physiogame is implemented in JavaScript (JS) and it is therefore essential to identify the pitfalls, code conventions and common design patterns of this programming language. Likewise, concepts to achieve modularity, automation, a testing environment and dependency management with JS have to be analyzed.

This thesis pursues multiple objectives and consists of several parts. First of all, motion tracking technologies and the product Leap Motion itself are analyzed. In the next step, the requirements for developing an application for virtual rehabilitation are specified. Then, the thesis discusses how to design modular JS-applications and points out how Physiogame is implemented. Finally, the usability of Physiogame is evaluated with patients and experts and the results are reviewed.

Based on the results of the usability test and expert assessments, it could be confirmed that Physiogame is in fact suitable for exercise therapy. In addition to that, the application is interactively configurable and can be executed as standalone desktop application or in browsers.

1 Einleitung

Forschungen im Gebiet der „Human Computer Interaction“ (HCI) haben in den letzten Jahren zu zahlreichen neuen Produkten und Trends geführt. Dies betrifft vor allem den Bereich der Motion Tracking Technologien, beispielsweise das Produkt Microsoft Kinect, mit den viele neue innovative Anwendungsbereiche erschlossen werden konnten (vgl. [Wang und Popović 2009], S. 1).

Im Feld der HCI werden diese neuartigen Benutzerschnittstellen als „natural user interfaces“ klassifiziert. Das Hauptziel dieser natürlichen Benutzerschnittstellen ist die Interaktion zwischen Mensch und Maschine zu revolutionieren: Sie soll schneller erlernbar und vor allem intuitiver werden (vgl. [Villaroman u. a. 2011], S. 227). Die Hand als Interaktionsmedium ist durch die Relevanz der Gebärdensprache und der Natürlichkeit der Bewegungen dabei besonders geeignet (vgl. [Chen 2008], S. 1).

Leap Motion ist eine neue Benutzerschnittstelle, die sich dem präzisen Fingertracking annehmen soll. Das Gerät ist etwas größer als ein typischer USB-Stick und erkennt Hände, Finger, Stifte und deren Gesten innerhalb eines ca. 50x50x50 cm großen Interaktionsraumes (vgl. [Pierce 2012] und [Heise 2012c]). Die möglichen Anwendungsgebiete dieses Produktes sind aufgrund der Neuartigkeit gegenwärtig noch unerschlossen, die Überprüfung der medizinischen Einsatzfähigkeit stellt folglich ein relevantes Forschungsgebiet dar.

Im Rahmen dieser Diplomarbeit wird eine Applikation namens Physiogame konzipiert um die Einsatzfähigkeit von Leap Motion im Bereich der Physio- und Ergotherapie zu analysieren. Es soll eine Applikation zur virtuellen Rehabilitation geschaffen werden, durch die die Beweglichkeit der Hände bzw. der Finger trainiert werden kann. Ein Hauptziel ist dabei, dass die Applikation sowohl im

Browser als auch als Desktop-Applikation lauffähig ist. Da Adobe Flash mittlerweile nicht mehr als die zukunftweisende Technologie im Browser-Bereich zu betrachten ist, erfolgt die Umsetzung von Physiogame ausschließlich mit Web-Technologien.

Eine Auseinandersetzung mit JavaScript (JS) – der Programmiersprache des Webs – um die angesprochene Applikation zu implementieren, ist folglich unausweichlich. Neben der eigentlichen Überprüfung der Einsatzfähigkeit von Leap Motion in der Physio- und Ergotherapie via Physiogame, beleuchtet diese Diplomarbeit daher auch die Vorgangsweise und die Schwierigkeiten bei der Entwicklung von JS-Applikationen. Durch diese Untersuchung soll somit weiters ein Ausgangspunkt für die Konzeption und Implementierung von zukünftigen Projekten mit JS geschaffen werden.

In den nachfolgenden Abschnitten werden die Ziele dieser Diplomarbeit definiert und sowohl die Struktur als auch die Methodik festgelegt.

1.1 Zielsetzung

In dieser Diplomarbeit soll sowohl das Produkt Leap Motion – durch Entwicklung der Applikation Physiogame – auf die Einsatzfähigkeit im Bereich der Ergo- und Physiotherapie untersucht als auch der Umgang mit der Programmiersprache JS (bzw. Web-Technologien im Allgemeinen) zur Konzeption und Implementierung von interaktiven Applikationen analysiert werden. Daraus ergeben sich folgende Zieldefinitionen für diese Arbeit:

- Definition des Forschungsgebietes Motion Tracking, Klassifikation und Vorstellung von bisherigen Produkten und Technologien,
- Vorstellung der natürlichen Benutzerschnittstelle Leap Motion und Analyse der Interaktionsmöglichkeiten,
- Definition von Anforderungen zur Entwicklung der Applikation Physiogame auf Basis von Kriterien für „Virtual Rehabilitation“-Systeme und von

Experten-Meinungen von Lehrenden der Studiengänge Physiotherapie und Ergotherapie der FH JOANNEUM,

- Analyse der Eigenheiten und Einsatzgebiete von JS und Vorstellung einer Vorgangsweise zur Konzeption von JS-Projekten,
- Dokumentation der Implementierung und des konzeptionellen Aufbaus von Physiogame und Veranschaulichung der Funktionalität,
- Testen der Usability von Physiogame sowohl mit Experten aus dem ergo- und physiotherapeutischen Bereich als auch mit tatsächlichen Patienten und schlussendliche Diskussion der Resultate.

Wie durch die hohe Anzahl an Zielen aus der vorherigen Auflistung ersichtlich wird, müssen zur Entwicklung der Applikation Physiogame und Untersuchung der Einsatzfähigkeit von Leap Motion verschiedene Forschungsgebiete angesprochen werden. Mangels der Kenntnisse des Autors im ergo- und physiotherapeutischen Bereich, wurden die Anforderungen an die Applikation Physiogame im Rahmen einer Kooperation mit Lehrenden aus den Studiengängen Physiotherapie und Ergotherapie an der FH JOANNEUM erarbeitet. Es ist somit nicht das Ziel dieser Diplomarbeit, den medizinischen Hintergrund der umgesetzten Benutzer-Interaktionen innerhalb der Applikation Physiogame zu beleuchten, sondern ausschließlich die Usability und damit die tatsächliche Einsatzmöglichkeit von Physiogame (und somit Leap Motion) mit Experten und Probanden zu testen und diese Ergebnisse zu analysieren.

1.2 Aufbau der Arbeit

Diese Diplomarbeit besteht – inklusive dieses Kapitels – aus insgesamt zehn Kapiteln, die aufeinander aufbauen und daher auch in dieser Reihenfolge gelesen werden sollten.

Im Kapitel „Motion Tracking und natürliche Benutzerschnittstellen“ auf Seite 21 wird zu Beginn der Forschungshintergrund zu Motion Tracking Technologien

beleuchtet, damit verbundene Begriffe abgegrenzt und die bisherigen technologischen Ansätze klassifiziert. Auf Basis dieses Wissens wird Leap Motion im Kapitel „Leap Motion“ auf Seite 36 vorgestellt, die Input-Daten analysiert und ein Testlauf in Form eines Prototypen unternommen. Die Anforderungen an die zu entwickelnde Applikation Physiogame werden anschließend im Kapitel „Anforderungen an die Applikation Physiogame“ auf Seite 50 definiert.

Physiogame wird ausschließlich mit Web-Technologien erstellt und verwendet die Programmiersprache JS. Die Eigenheiten, Einsatzgebiete und die Signifikanz von JS werden daher im Kapitel „Die mannigfaltigen Gesichter von JavaScript“ auf Seite 61 vorab beleuchtet, bevor im darauffolgenden Kapitel „Konzeption von JavaScript-Applikationen“ auf Seite 80 eine Vorgehensweise erläutert wird, um Applikationen mit JS zu konzipieren. Im Kapitel „Implementierung der Applikation Physiogame“ auf Seite 121 werden danach Architektur und Details zur Implementierung von Physiogame erörtert.

Die tatsächliche Funktionalität von Physiogame wird im Kapitel „Vorstellung der Applikation Physiogame“ auf Seite 136 vorgestellt. Der Ablauf des Usability-Tests zur Applikation wird danach im Kapitel „Durchführung eines Usability-Tests“ auf Seite 156 definiert und die Ergebnisse analysiert und kritisch diskutiert.

Schlussendlich werden im Kapitel „Zusammenfassung und Ausblick“ auf Seite 171 die Erkenntnisse aus dieser Diplomarbeit nochmals zusammengefasst und kritisch betrachtet. Den Abschluss bildet ein Ausblick auf zukünftige Entwicklungen im Forschungsgebiet und der Applikation Physiogame.

1.3 Methodik

Die Definition und Klassifikation von Motion Tracking Technologien ist für das Verständnis der Funktionsweise von Leap Motion relevant, da hierdurch sowohl das Potential als auch die Einschränkungen vermittelt werden können. Dieser Teil erfolgt daher bereits in den nachfolgenden beiden Kapiteln.

Die Applikation Physiogame muss zahlreiche Anforderungen erfüllen, damit sie für den Einsatz in der Bewegungstherapie geeignet ist. Im vierten Kapitel wird daher das Gebiet der virtuellen Rehabilitation analysiert und Kriterien ausgearbeitet. Die Anforderungen an Physiogame werden dann im Anschluss vorgestellt. Es handelt sich dabei tatsächlich um die End-Anforderungen an die Applikation, die sich aus der Kooperation mit Lehrenden der Studiengänge Physiotherapie und Ergotherapie der FH JOANNEUM nach Meetings und Feedback-Runden ergeben haben.

Es existieren natürlich auch technische Anforderungen für Physiogame (Stichwort Web-Applikation) und um diesen zu genügen ist eine Auseinandersetzung mit JS unausweichlich. JS besitzt zahlreiche Eigenheiten, wodurch bisherige Vorgehensweisen aus anderen Programmiersprachen nicht mehr anwendbar sind. Die Sprache ist aber äußerst vielfältig einsetzbar und sehr populär. Allerdings ist die Konzeption von Applikationen mit JS keinesfalls als trivial anzusehen und erfordert sowohl Kenntnis der Fallen, Konventionen und Patterns der Programmiersprache als auch das Treffen einiger Entscheidungen um Modularisierung, Automatisierung und Testen zu ermöglichen. Der professionelle Umgang mit JS wird daher mit zahlreichen Code-Beispielen im fünften und sechsten Kapitel vermittelt.

Die Umsetzung von Physiogame konnte mit diesen Kenntnissen danach erfolgen und der tatsächlich umgesetzte Spielablauf (aus den Anforderungen) wird beschrieben. Dies erfolgt im siebten und achten Kapitel. Schlussendlich wird ein Usability-Test durchgeführt um die Einsatzfähigkeit der Applikation (und somit von Leap Motion) zu überprüfen. Die Durchführung erfolgt dabei sowohl mit Ergo- und Physiotherapeuten als auch mit tatsächlichen Patienten und wird in direkt in den Therapieräumen des Kindertherapiezentrums Kids-Chance der Maria Theresia Klinik in Bad Radkersburg abgehalten. Die Ergebnisse dieses Usability-Tests – erhoben aus Fragebögen, durch Beobachtung und aus den automatisierten Statistiken von Physiogame – werden im neunten Kapitel zu guter Letzt analysiert und kritisch betrachtet.

2 Motion Tracking und natürliche Benutzerschnittstellen

Das Verlangen, die Interaktion zwischen Mensch und Computer zu verbessern geht weit zurück. Innerhalb dieses Kapitels wird eine Einführung in das Gebiet der natürlichen Benutzerschnittstellen und Motion Tracking gegeben. Es ist hierzu dienlich, diese Begriffe und das Gebiet um Motion Tracking genauer abzugrenzen. Dies erfolgt im ersten Abschnitt „Begriffsdefinition und Abgrenzung“ auf dieser Seite. In weiterer Folge werden im Abschnitt „Historie von HCI und natürlichen Benutzerschnittstellen“ auf Seite 23 die historischen Vertreter der natürlichen Benutzerschnittstellen vorgestellt.

Die Funktionsweise von Motion Tracking Technologien fällt unterschiedlich aus. Im Abschnitt „Klassifikation von Motion Tracking Technologien“ auf Seite 25 wird daher ein Klassifizierungsschema definiert, das die Einteilung dieser Technologien erlaubt. Zum Abschluss dieses Kapitels wird eine Zusammenfassung gegeben.

2.1 Begriffsdefinition und Abgrenzung

In den folgenden Unterabschnitten werden die Begriffe Motion Tracking, Motion Capture und Motion Sensing definiert und voneinander abgegrenzt. Diese Begriffe werden im Rahmen dieses Kapitels sehr häufig verwendet, daher ist es unumgänglich sie vorab in hinreichender Weise zu beleuchten. Aufbauend wird zudem definiert, was die Motivation hinter natürlichen Benutzerschnittstellen ist und wie diese mit Motion Tracking zusammenhängen.

2.1.1 Motion Tracking und Motion Capture

Die Begriffe Motion Tracking und Motion Capture werden häufig sowohl in wissenschaftlicher Literatur als auch umgangssprachlich synonym zueinander verwendet. Es macht an dieser Stelle jedoch Sinn sich auf einen Begriff für den Rest der Arbeit zu beschränken.

Motion Capture wird gemeinhin mit der „Ganz-Körper"-Aufzeichnung von Bewegungen assoziiert um nachträglich Computeranimationen daraus zu erstellen. Der Begriff hat somit eine längere Tradition in der Entertainment-Branche, beispielsweise im Rahmen einer Filmproduktion. In weiterer Folge wird daher ausschließlich der Begriff Motion Tracking in dieser Diplomarbeit verwendet um sich von dieser eingeschränkten Branche abzuheben. Kurzum, Motion Tracking Technologien beschäftigen sich mit der kontinuierlichen Positionserfassung von Objekten (vgl. [Habibi 2003], S. 6).

2.1.2 Motion Sensing

Das Gebiet Motion Sensing kann als ein Vorläufer zu Motion Tracking betrachtet werden. Es geht dabei allerdings nicht um eine kontinuierliche Positionserkennung von Bewegungen, sondern darum Veränderungen im abgetasteten Bereich zu erkennen, folglich die Bewegungserkennung („motion detection") (vgl. [University of Southern California 2008]).

Der große Durchbruch von Motion Sensing erfolgte im zweiten Weltkrieg: Anfänglich wurde es für Radaranlagen verwendet um durch Überschallmessungen Veränderungen im Luftraum aufzuspüren. Heutzutage findet sich diese Technologie auch in gängigen Alarmanlagen wieder. Daneben werden auch häufig Infrarotsensoren in Bewegungsmeldern für Licht- und Türschaltungen eingesetzt. Die Erfindung dieser Infrarotsensoren erfolgte bereits in der Mitte des 19. Jahrhunderts und wurde ursprünglich ausschließlich in der Astronomie eingesetzt (vgl. [University of Southern California 2008]).

2.1.3 Natürliche Benutzerschnittstelle

Als natürliche Benutzerschnittstelle werden Technologien definiert, die eine intuitivere und dadurch natürlichere Interaktion mit Computern ermöglichen. Die effektive Dauer des Lernvorganges soll dramatisch verkürzt werden. Der Umfang der Auswirkungen dieser neuen Interfaces zu den bisherigen Interaktionskonzepten, könnte nach Villaroman u. a. in naher Zukunft eine ebenso große Bedeutung annehmen, wie einst der Sprung von Kommandozeile zur grafischen Oberfläche (vgl. [Villaroman u. a. 2011], S. 227).

Als primäres Eingabeinstrument bieten sich im Besonderen die Hände eines Benutzers an (vgl. [Garg u. a. 2009], S. 972). Motion Tracking Technologien werden daher als Hauptforschungsgebiet angesehen, um natürliche Benutzerschnittstellen zu verwirklichen (vgl. [Harshith u. a. 2010], S. 31). Aufgrund dieser Feststellungen befassen sich die nachfolgenden Abschnitte daher mit der Kombination aus Motion Tracking Technologien und natürlichen Benutzerschnittstellen.

2.2 Historie von HCI und natürlichen Benutzerschnittstellen

Die Bemühung, die Interaktion zwischen Mensch und Maschine zu verbessern, geht bis in den zweiten Weltkrieg zurück. Lochkarten und tastaturähnliche Benutzerschnittstellen schienen bereits damals als wenig angemessen, um mit Computern zu interagieren. Ein Ziel war geboren: Interaktion soll in einer natürlichen Weise möglich werden. Ende der 1960er wurden erste Studien veröffentlicht die sich mit der Erkennung von Mustern in Schrift und Sprache befassen. Kein halbes Jahrhundert später sind Touchscreens und Multitouch-Technologien längst im alltäglichen Leben verankert. Microsofts Kinect eröffnete die Welt der natürlichen Benutzerschnittstellen einer breiten Masse. Dieses Produkt wird in Universitätsprogrammen, die sich mit HCI befassen, mittler-

weile auch als einfachster Einstieg in das Forschungsgebiet angesehen (vgl. [Villaroman u. a. 2011], S. 227).

Der Weg bis hierhin kann jedoch als holprig angesehen werden. Der erste Handschuh um Bewegungen zu Tracken, der „Sayre glove“ wurde 1977 an der Universität von Illinois entwickelt. Er erlaubte optisches Tracking der Hand, unterstützte jedoch noch keine Gestenerkennung. In den frühen 1980ern wurde am Massachusetts Institute of Technology (MIT) ein Handschuh mit Leuchtdioden kombiniert und für erste Motion Capturing Verfahren verwendet. Ein weiterer Prototyp entsprang aus den Bell Telephone Laboratories 1983: Er erkannte Handgesten der amerikanischen Gebärdensprache ASL (American Sign Language). Keines dieser Experimente schaffte jedoch den Sprung aus den Laboren (vgl. [Sturman und Zeltzer 1994], S. 32).

Erste kommerziell verfügbare Produkte gab es Ende der 1980er: Nintendos Power Glove – ein Handschuh mit Controller für die Konsole Super Nintendo – war für seine Zeit revolutionär, scheiterte allerdings bei den Kunden und wird heutzutage als eines von vielen Experimenten von Nintendo abgetan. Nichtsdestotrotz wird das Unternehmen bis heute als großer Innovator und Retter der Videospiel-Industrie der 1980er angesehen. Es finden sich heutzutage immer noch neue Einsatzzwecke für den Power Glove, beispielsweise zur Erzeugung von Musik (vgl. [Watercutter 2013] und [Rochester Institute of Technology 2011]).

Abbildung 2.1 auf der nächsten Seite zeigt den Nintendo Power Glove mit seinen angebrachten Controller und Code-Eingabefeld zur Programmierung des Verhaltens beim jeweiligen Videospiel.

Im Jahr 2002 erschien der Film Minority Report, der mit einer neuartigen Motion Tracking Technologie aufzeigte, wohin sich die Forschung zu natürlichen Benutzerschnittstellen hinbewegen könnte. Steven Spielberg rekrutierte für die Produktion des Filmes wissenschaftliche Berater des MIT. Diese setzen sich das Ziel, einerseits die Mimik eines Benutzers und andererseits Handgesten, via eines Datenhandschuhs in Echtzeit zu erfassen und zu interpretieren. Das

Abbildung 2.1: Nintendo Power Glove von 1989 ([Wikimedia Commons 2011a]).

im Film eingesetzte Betriebsystem und Erkennungsverfahren war also auch real (teilweise) so einsetzbar. Es wurde später vom Hauptberater John Underkoffler unter dem Namen G-Speak weiterentwickelt (vgl. [Rochester Institute of Technology 2011] und [Oblong Industries Inc. 2013]). Der Film dient heute noch häufig als Maßstab, um die Vision von natürlichen Benutzerschnittstellen zu verdeutlichen.

Wie historisch ersichtlich wird, beeinflussen die Forschungen im Gebiet der Motion Tracking Technologien die Entwicklung von neuen natürlichen Benutzerschnittstellen. Es ist daher relevant, die Funktionsweise dieser Technologien zu betrachten. Dies findet im Rahmen der Findung eines Klassifikationsschemas im nachfolgenden Abschnitt statt.

2.3 Klassifikation von Motion Tracking Technologien

Dieser Abschnitt beschreibt ein Klassifikationsschema von Motion Tracking Technologien. Da im Rahmen dieser Diplomarbeit vorrangig Technologien zur Erfassung von Finger- und Handbewegungen relevant sind, spielen Produk-

te und Prototypen, die diese Anforderung nicht erfüllen keine weitere Rolle. Motion Tracking Technologien werden üblicherweise in die folgenden Bereiche eingeteilt (vgl. [Garg u. a. 2009], S. 972 und [Harshith u. a. 2010], S. 31):

- Datenhandschuhe bzw. tragbare Sensoren und
- sichtgesteuertes („vision-based“) Motion Tracking.

Bei den sichtgesteuerten Tracking Technologien muss zudem noch zwischen zwei prinzipiell unterschiedlichen Erkennungsmethodiken unterschieden werden. Daher bietet sich für diese Klassifikation noch die weitere Unterklassifikation in markerlose („barehand“) und markerunterstützte Techniken („marker-based“) an (vgl. [Chen 2008], S. 3).

In den nachfolgenden Unterabschnitten werden die einzelnen Bereiche detaillierter vorgestellt, deren Vorteile und Nachteile beschrieben und jeweils ein Prototyp bzw. Produkt präsentiert.

2.3.1 Datenhandschuhe bzw. tragbare Sensoren

Datenhandschuhe bestehen aus einem (bzw. zumeist mehreren) Sensoren um Hand- und Fingerbewegungen zu erfassen (vgl. [Harshith u. a. 2010], S. 31). Diese Art des Motion Trackings wird mittlerweile als untauglich für einen alltäglichen bzw. kommerziellen Einsatz angesehen. Laut Ren u. a. behindern sie Benutzer bei der Ausführung einer natürlichen Bewegung und erfordern häufig eine Kalibrierung und Abstimmung mit den unterschiedlich geformten Händen von Benutzern (vgl. [Ren u. a. 2011], S. 1093). Ein weiteres Problem stellen nach Chen die zumeist sehr hohen Anschaffungskosten derartiger Produkte dar (vgl. [Chen 2008], S. 2).

Als Vorteil wird jedoch angesehen, dass bei den meisten Produkten keine visuellen Sensoren eingesetzt werden. Dadurch spielen unterschiedliche Lichtstimmungen bzw. stille Hintergründe für die Erkennung keine Rolle. Das Tracken von Handbewegungen kann dadurch in einer robusteren Weise erfolgen (vgl. [Ren u. a. 2011], S. 1093).

Ein Beispiel eines kommerziellen Datenhandschuhs ist der CyberGlove der Immersion Corporation. Durch die angebrachten Sensoren können Parameter, wie der aktuelle Winkel der Gelenke und die räumlichen Koordinaten der Hand direkt ausgelesen werden. Die Kosten eines solchen Handschuhes sind für Normalanwender allerdings weit zu hoch (vgl. [Chen 2008], S. 2). Die Abteilung die den zuvor angesprochenen Handschuh entwickelte, wurde übrigens im Jahre 2009 verkauft und tätigt ihr Geschäft mittlerweile unter dem Namen CyberGlove Systems (vgl. [Immersion Corporation 2009]). Abbildung 2.2 zeigt die dritte Version des angesprochenen Datenhandschuhs.

Abbildung 2.2: CyberGlove III von CyberGlove Systems ([CyberGlove Systems LLC 2010]).

Trotz der pessimistischen Sichtweise zu Datenhandschuhen in zahlreichen wissenschaftlichen Publikationen (vgl. [Ren u. a. 2011], S. 1093, [Chen 2008], S. 2, [Garg u. a. 2009], S. 972 und [Harshith u. a. 2010], S. 31), könnten sich vor allem tragbare Sensoren in Zukunft zu einem profitablen Zukunftsmarkt entwickeln. Ein hervorragendes Beispiel ist hierfür eine Aussage von Tim Cook (Apple), in der er tragbaren Sensoren eine große Rolle für die nächsten Produkte von Apple zuschreibt (vgl. [Ovide und Rusli 2013]). Dies betrifft zwar nicht direkt den Bereich des Motion Trackings, könnte bei breiter Adaption je-

doch wieder indirekt darauf zurückführen.

Kim u. a. demonstrierten bereits 2012 einen am Handgelenk tragbaren Prototypen, der zum Tracking von Hand- und Fingerbewegungen eingesetzt werden kann. Das System ist explizit auf einen geringen Stromverbrauch und einer hohen Mobilität ausgelegt. Es werden sowohl eine Infrarot-Kamera und ein Feld von Infrarot-Leuchtdioden verbaut als auch ein Infrarot-Laser verwendet, um die Hand robust und sicher zu erfassen (vgl. [Kim u. a. 2012], S. 167ff.). Abbildung 2.3 zeigt den Aufbau der einzelnen Komponenten und die „Trageweise“ des Prototypen.

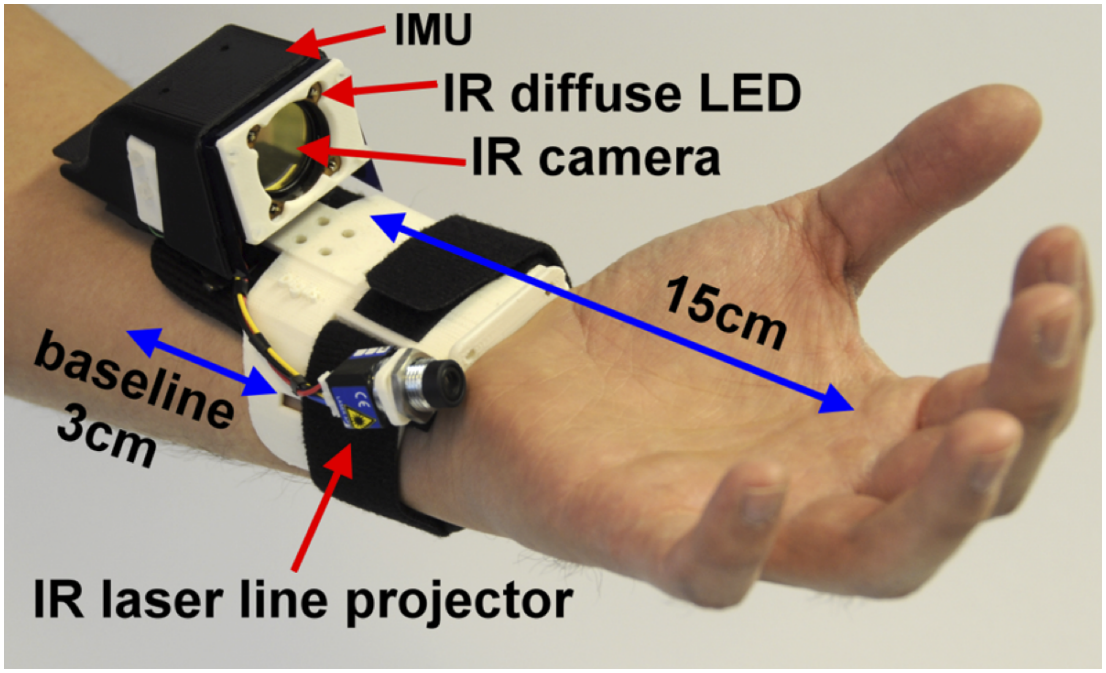

Abbildung 2.3: Mobiler, energieeffizienter und am Handgelenk tragbarer Prototyp zum Tracking von Hand- und Fingerbewegungen ([Kim u. a. 2012], S. 169).

Als Nachteil dieses Prototypen kann jedoch die Verwendung von visuellen Sensoren – wie in der Abbildung 2.3 ersichtlich, wird eine Infrarot-Kamera eingesetzt – angesehen werden. Gerade dies machte den zuvor angesprochenen vermeintlichen Vorteil von Datenhandschuhen bei der Genauigkeit der Erkennung aus. Anscheinend müssen zukünftig Kompromisse in einem Bereich gemacht werden. Darüber hinaus zeigt dies aber auch, dass die Grenzen zwischen den Klassifizierungen Datenhandschuh und dem, im nächsten Unterab-

schnitt vorgestellten, sichtgesteuerten Tracking nach und nach verschwimmen.

2.3.2 Sichtgesteuertes Motion Tracking

Als sichtgesteuertes Motion Tracking von Hand- und Fingerbewegungen werden Methoden klassifiziert, die durch Bildverarbeitung eines Videosignals Gesten identifizieren können (vgl. [Chen 2008], S. 2). Hierzu wird das Eingabe-Video per einzelnen Frame verarbeitet. Hände und Finger werden vom statischen Hintergrund isoliert. Die Analyse von Gesten kann somit über mehrere zusammenhängende Frames nach verschiedenen Parametern erfolgen (vgl. [Harshith u. a. 2010], S. 35).

Ursprünglich wurde sichtgesteuertes Tracking mit speziellen Markern oder eingefärbten Handschuhen realisiert. Mittlerweile wird mehr in Richtung markerloses Motion Tracking geforscht, das ohne diese zusätzlichen Hilfsobjekte auskommt. Beide Unterbereiche werden jedoch als bedeutend praktikabler im Vergleich zu Datenhandschuhen angesehen. Ein wesentlicher Grund sind die weniger sperrigen Hardware-Komponenten, die eingesetzt werden können (vgl. [Chen 2008], S. 2f.).

Die Methoden zur Erkennung innerhalb der Gesamtklassifizierung können sehr unterschiedlich ausfallen. Neben multiplen Video-Kameras gab es zudem auch Versuche Sensoren zur Tiefenmessung zu verwenden. Daraus entstanden später Produkte mit Spezialsensoren, wie beispielsweise Intel Gesture Camera und Leap Motion. Eine weitere übliche Methode ist die Kombination aus Tiefensensor und Videokamera, wie beispielsweise bei der Microsoft Kinect (vgl. [Sridhar 2013], S. 2756).

Sichtgesteuertes Motion Tracking wurde lange als nicht umsetzbar für Echtzeit-Anwendungen angesehen. Dies liegt an den Einschränkungen, die sich aus dem Einsatz von optischen Sensoren ergeben. Die Technologie muss unabhängig von unterschiedlichen Lichtbedingungen und Hintergründen robust funktionieren. Nach Garg u. a. ist es daher unumgänglich, dass derartige Techno-

logien stark optimiert werden müssen, damit sie durchgängig eine hohe Präzision erzielen. Ren u. a. sah diese Einschränkungen auch noch im Jahre 2011 als nicht überwunden an. Derartige Technologien müssten sehr hohe Anforderungen an die Robustheit, Genauigkeit und Effizienz der Erkennung erfüllen (vgl. [Ren u. a. 2011], S. 1093, 1096 und [Garg u. a. 2009], S. 972).

2.3.2.1 Markerunterstütztes Motion Tracking

Früher wurden beim markerunterstützten Tracking oft mehrere Videokameras für die Aufzeichnung der Marker verwendet. Die Meisten nutzten als Marker Retroreflektoren, damit die Lichtstrahlen unabhängig von der Ausrichtung des Reflektors zurück reflektiert werden können. Auch Leuchtdioden werden eingesetzt (vgl. [Wang und Popović 2009], S. 1f.). Seither gab es verschiedene Ansätze, die in zahlreichen Prototypen verwirklicht wurden. An dieser Stelle ist es wichtig festzuhalten, dass das Vorhandensein von elektronischen Bauteilen auf den Markern diese nicht automatisch als Datenhandschuh klassifiziert. Wichtig ist, dass der primäre Tracking-Vorgang durch ein visuelles Verfahren erfolgt.

Ähnlich wie bei den Datenhandschuhen müssen Benutzer beim markerunterstützten Tracking also mindestens ein zusätzliches Objekt am Körper tragen bzw. in der Hand halten. Der Fokus dieser Technologien liegt auf Genauigkeit, auf Kosten der Einfachheit in Aufbau und Konfiguration (vgl. [Wang und Popović 2009], S. 2). Nach Ghosh u. a. schränkt die Notwendigkeit von Markern die Effektivität in virtuellen Welten („immersive environments“) besonders ein (vgl. [Ghosh u. a. 2010], S. 323). Folglich könnte dieser Umstand auch erklären, warum die Forschungsbestreben aktuell mehr auf markerloses Tracking gerichtet sind.

Die eigentlichen Marker sind meist verschiedenfarbig und werden häufig als speziell eingefärbte Stoffhandschuhe verwendet. Auch Armbänder können als Indikatoren dienen (vgl. [Ghosh u. a. 2010], S. 324). Abbildung 2.4 auf der nächsten Seite zeigt einen typischen Vertreter für markerunterstütztes Tracking.

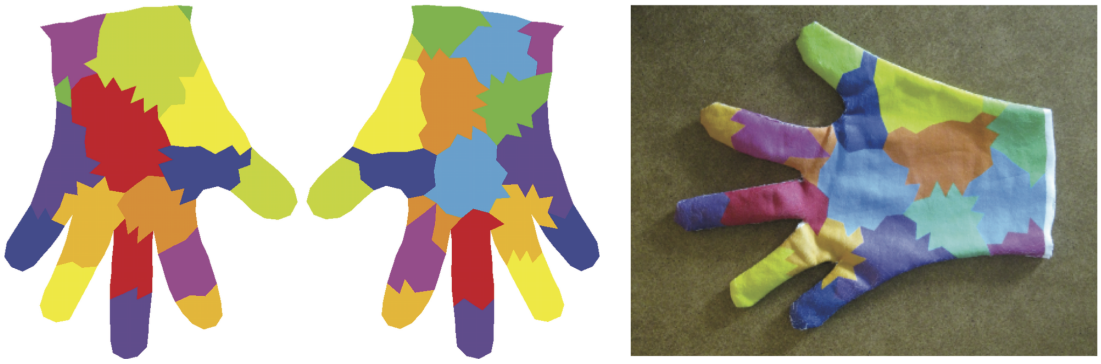

Abbildung 2.4: Eingefärbter Stoffhandschuh zum sichtgesteuerten markerunterstützten Motion Tracking ([Wang und Popović 2009], S. 3)

Eine einfache Methode zur Positionserfassung, beispielsweise der Fingerkuppen, kann hierbei durch Analyse der Häufigkeit von gleichfarbigen Farbpunkten ausgeführt werden. Derartige funktionierende Prototypen gab es bereits vor dem 21. Jahrhundert (vgl. [Pavlovic u. a. 1997], S. 684). Das Problem dieser simplen Erkennungsmethodik ist allerdings, dass sich das Tracking so auf nur eine zweidimensionale Ebene beschränkt. Das Ergebnis wäre folglich nur die Positionserkennung beispielsweise einer Fingerkuppe innerhalb eines Bildes, jedoch nicht im dreidimensionalen Raum (vgl. [Prisacariu und Reid 2012], S. 237). Die Komplexität von Erkennungsalgorithmen ist somit trotz der markerunterstützten Technologien als hoch anzusehen, sobald es um die Erfassung der Hände im dreidimensionalen Raum geht.

Ein bekannter kommerzieller Vertreter innerhalb dieser Klassifizierung ist übrigens die Nintendo Wiimote, der Controller der Nintendo Wii. Der Marker ist bei diesem Produkt nicht im oder am Controller verbaut, sondern tatsächlich in Form einer externen stationären Infrarot-Lichtquelle verwirklicht: Ein direkt im Controller verbauter Infrarot-Sensor erfasst eine externe Infrarot-Lichtquelle um die Position des Controllers im Raum zu errechnen (vgl. [Lee 2008], S. 39f.). Der Sensor wird in den Händen gehalten und befindet sich also in Bewegung. Die Erkennung findet folglich umgekehrt statt.

Eine „Immersion“ (das Eintauchen in die virtuelle Welt) ist mit markerunterstützten Technologien laut Ghosh u. a. nur eingeschränkt möglich. Ghosh u. a. impliziert damit, dass nur markerlose Tracking-Technologien für einen Einsatz

in virtuellen Welten geschaffen sein dürften (vgl. [Ghosh u. a. 2010], S. 324).

2.3.2.2 Markerloses Motion Tracking

Die Klassifizierung markerloses („bare-hand“) Motion Tracking umfasst Technologien, die Hände von Benutzern ohne Marker oder andere tragbare Geräte erkennen können. Daraus ergeben sich vor allem Vorteile für die Ausführung der eigentlichen Benutzer-Interaktionen: Sie sind natürlicher und einfacher (vgl. [Song u. a. 2008], S. 2). Da die Marker wegfallen, muss sich dieser Bereich auch nicht mit Problemen, wie die Verdeckung von Markern oder der robusten Identifikation dieser befassen (vgl. [Du u. a. 2012], S. 2).

Das sichtgesteuerte Tracken von Handbewegungen ohne Marker war lange Zeit mit Schwierigkeiten verbunden. Im Jahre 2009 zogen Wang und Popović noch den Schluss, dass Erkennungs-Algorithmen nur dann in interaktiven Geschwindigkeiten laufen können, wenn beträchtliche Einbusen bei der Auflösung und der Größe des Interaktionsbereichs gemacht werden (vgl. [Wang und Popović 2009], S. 2).

Das Interesse an markerlosen Motion Tracking Technologien riss aber nicht ab und seit 2010 wurden zahlreiche neue Prototypen vorgestellt (vgl. [Ghosh u. a. 2010], S. 324). Dies dürfte größtenteils mit der Veröffentlichung der Microsoft Kinect zusammenhängen, ein kommerzielles Produkt für die Microsoft Xbox 360, die etwas später auch am PC verwendet werden konnte.

Microsoft Kinect revolutionierte das Gebiet des markerlosen Motion Trackings und erlaubte erstmals die Erfassung des kompletten Körpers in Echtzeit. Als quelloffene Treiber zur kommerziellen Verwendung verfügbar wurden, konnten außerdem völlig neue Einsatzmöglichkeiten geschaffen werden, um das Produkt beispielsweise als 3D-Scanner einzusetzen (vgl. [König 2013], S. 118). Abbildung 2.5 auf der nächsten Seite zeigt das Produkt Kinect und die drei Hardware-Komponenten die zum Tracking maßgeblich zum Einsatz kommen.

Der Infrarot-Emitter der Kinect projiziert ein Punktgitter in den Raum, das vom Infrarot-Tiefensensor ausgelesen wird, um ein 320x240 Pixel großes Bild mit

Abbildung 2.5: Microsofts Kinect (vgl. [Wikimedia Commons 2011b] und [Holmquest 2012a]).

den Tiefen-Informationen zu erstellen. Dies kann danach mit den Bildern der Videokamera kombiniert werden, um schlussendlich mit speziellen Algorithmen die relevanten Positions- und Bewegungsdaten heraus zu extrahieren (vgl. [Holmquest 2012a]). Der Mindestabstand, damit Objekte überhaupt erkannt werden können, liegt dabei bei 40 Zentimeter (vgl. [Heise 2012b]).

Mit dem „Software Development Kit“ (SDK) der Kinect ist der Umgang mit den Tracking-Daten bereits abstrahiert möglich. Es ermöglicht die Formung eines Skeletts aus bestimmten getrackten Benutzerpunkten im Raum und befreit Entwickler somit davon, einen eigenen Erkennungsalgorithmus selbst zu entwerfen (vgl. [Holmquest 2012b]]).

Abbildung 2.6 auf der nächsten Seite zeigt dieses Skelett. Wie ersichtlich wird, ist es durch das Skelett zwar möglich, die Position bzw. Bewegungen der Hände zu erfassen, eine kleinere Differenzierung (in einzelne Finger) ist aber nicht möglich.

Die präzise Erfassung der Finger ist bei der Microsoft Kinect auch mit speziellen Algorithmen nur schwer möglich. Ren u. a. entwickelnden zwar eine Methode, um aus dem Tiefenbild die einzelnen Finger zu differenzieren, die Hand muss hierzu jedoch stets frontal positioniert werden und ein zusätzliches Armband zur Differenzierung muss getragen werden (vgl. [Ren u. a. 2011], S. 1093ff.). Aufgrund von hardwarespezifischen Limitierungen kann somit nicht davon ausgegangen werden, dass die Erkennung von individuellen Fingern

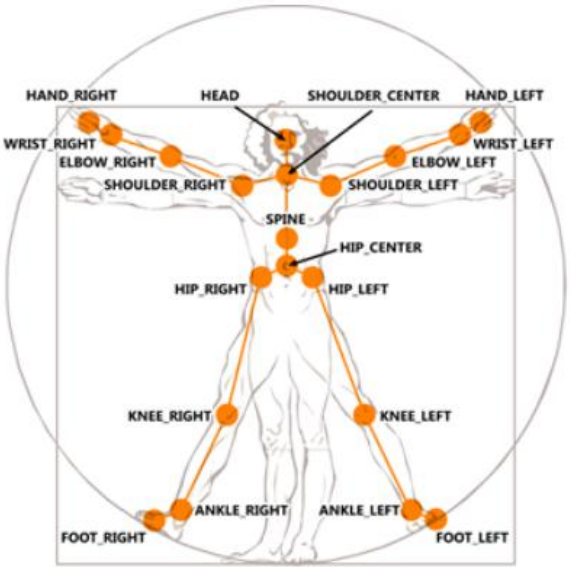

Abbildung 2.6: Kinect SDK Skelett ([Holmquest 2012b])

präzise, reaktionsschnell und robust mit der Kinect durchgeführt werden kann.

Genau diesem Mangel nimmt sich das Produkt Leap Motion an, das im nächsten Kapitel genauer vorgestellt wird.

2.4 Zusammenfassung

Die Begriffe Motion Tracking und natürliche Benutzerschnittstelle wurden in diesem Kapitel definiert und die Zusammenhänge zwischen ihnen klargestellt. Die Bemühungen, die Natürlichkeit der Interaktion zwischen Mensch und Maschine zu steigern, gehen weit zurück. Die Erforschung von neuen Motion Tracking Technologien beeinflusst dabei maßgeblich das Ziel natürliche Benutzerschnittstellen zu schaffen. Es war daher relevant den aktuellen Forschungsstand von Motion Tracking, im Rahmen der Klassifikation, zu beleuchten und die Probleme bzw. Einschränkungen der jeweiligen Technologien aufzuzeigen.

Datenhandschuhe werden sich in der Form, wie sie ursprünglich erdacht wurden, heute nicht mehr durchsetzen. Eventuell spielen tragbare Sensoren aber in der Zukunft eine Rolle, wenn auch nicht primär zum Motion Tracking. An-

ders sieht es beim sichtgesteuerten Motion Tracking aus. Sowohl beim markerunterstützten Tracking (Nintendo Wii) als auch beim markerlosen Tracking (Microsoft Kinect) sind zwei sehr erfolgreiche kommerzielle Produkte hervorgegangen. Insgesamt scheinen markerlose Motion Tracking Technologien aber am Zukunftsträchtigsten zu sein.

Microsofts Kinect wird heute als Einstieg in HCI und natürlichen Benutzerschnittstellen an den Universitäten verwendet. Ein Großteil des Erfolges ist hier sicherlich auch dem SDK von Microsoft, mit dem mitgelieferten Algorithmen zur Skelettprojektion zu verdanken. Im nächsten Kapitel erfolgt nun die Auseinandersetzung mit dem Produkt Leap Motion, das den Bereich der markerlosen Motion Tracking Technologien weiter vorantreiben könnte.

3 Leap Motion

Der Leap Motion Controller ist eine neue kommerzielle natürliche Benutzerschnittstelle des gleichnamigen Unternehmens. Im Gegensatz zur Microsofts Kinect, muss das Gerät nicht in einem größeren Abstand zu Objekten positioniert werden, sondern erlaubt das Tracken laut Hersteller bereits nach einem Mindestabstand von 25 Millimeter und dies weit präziser (vgl. [Leap Motion Inc. 2013c]).

Im nachfolgenden Abschnitt wird der Leap Motion Controller allgemein vorgestellt. Danach wird im Abschnitt 3.2 auf Seite 38, ein Blick ins Innere der Hardware geworfen und die Funktionsweise analysiert. Die Schnittstellen zur Programmierung des Leap Motion Controllers werden im Abschnitt 3.3 auf Seite 39 beleuchtet. In weiterer Folge werden im Abschnitt 3.4 auf Seite 41, die Bewegungsdaten vom Controller visualisiert und Einschränkungen bzw. Probleme bei der Erkennung erklärt. Anschließend wird eine Prototyp-Applikation mit der Adobe Flash Plattform implementiert und im Abschnitt 3.5 auf Seite 45 vorgestellt. Zum Abschluss des Kapitels wird schlussendlich eine Zusammenfassung der Erkenntnisse auf Seite 49 gegeben.

3.1 Allgemein

Das Produkt erlaubt das präzise Tracking von Händen, Fingern und Stiften. Laut Herstellerangaben ist die Erkennung bis 200 Mal präziser im Vergleich zur Microsofts Kinect (vgl. [Baldwin 2012]). Allerdings ist der Interaktionsraum auch nur ungefähr 50x50x50 cm groß. Innerhalb dieses Raumes kann jedoch eine Präzision von 0,01 mm erreicht werden. Das Produkt ist somit weniger als

eine Konkurrenz zu Microsofts Kinect – das sich dem Ganz-Körper-Tracking verschrieben hat – positioniert, sondern fokussiert sich auf einen Nischenmarkt. Der einzuhaltende Mindestabstand um ein Tracking zu ermöglichen ist minimal und eher auf kleinere Distanzen ausgelegt (vgl. [Heise 2012c]). Aufgrund der kleinen Bauweise könnte das Produkt in naher Zukunft auch direkt in Laptops integriert werden. Es besteht hierzu bereits eine Kooperation mit dem Computer-Hersteller Asus (vgl. [Leap Motion Inc. 2013a]).

Abbildung 3.1 zeigt die Größe und die Anschlüsse des Controllers. Das Gerät ist mit 8,0x3,0x1,2 cm besonders portabel und besitzt neben der grünen Leuchtdiode und dem Universal Serial Bus (USB) Anschluss äußerlich keine weiteren Merkmale. Es kommt ein USB 3.0 micro-B Port zum Einsatz. Der linke Teil des Ports muss jedoch nicht benutzt werden, sodass auch ältere USB 2.0 micro-Kabel angeschlossen werden können (vgl. [Leap Motion Inc. 2013d]).

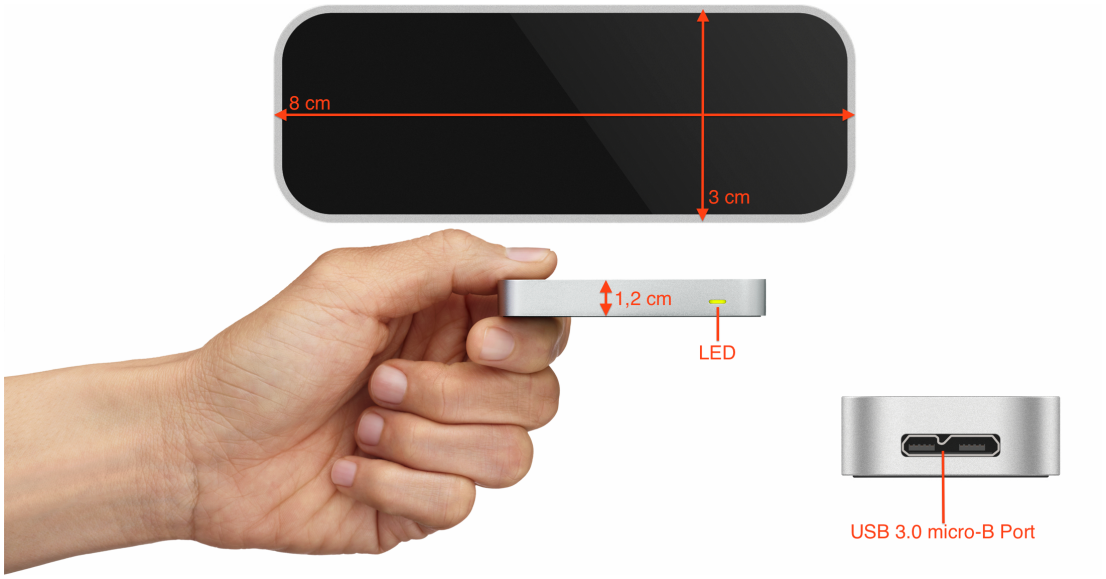

Abbildung 3.1: Größe und Anschlüsse des Leap Motion Controllers (vgl. [Leap Motion Inc. 2013d])

Das Produkt kann somit direkt über den USB-Anschluss am Computer betrieben werden. Der Datendurchsatz der Tracking-Informationen vom Gerät liegt dabei bei ca. 100 „Frames Per Second“ (FPS) über USB 2.0, bei USB 3.0 kann diese Rate nochmals auf rund 150 FPS gesteigert werden (vgl. [Vikram u. a. 2013], S. 1181). Die Datenrate kann somit als hoch angesehen werden

und dürfte sich in geringen Reaktionszeiten in den eigentlichen Applikationen äußern. Folglich dürfte Leap Motion auch zur Entwicklung von Applikationen geeignet sein, die eine minimale Latenz voraussetzen, beispielsweise Musikanwendungen (vgl. [Kiefer und Loclair 2013], S. 72).

3.2 Technologie und Software

Die Technologie hinter Leap Motion ist klar als sichtgesteuertes, markerloses Motion Tracking klassifizierbar. Allerdings ist das mathematische Erkennungsverfahren bzw. die Methodik patentiert, sodass aktuell leider wenig über tatsächliche Funktionsweise bekannt ist. Zum Abtasten des Interaktionsraums werden drei Infrarot-LEDs in Verbindung mit zwei Infrarot-Kameras eingesetzt. Dadurch können die Tiefen-Informationen erfasst werden (vgl. [Leap Motion Inc. 2013f]). Abbildung 3.2 zeigt die Hardware-Komponenten des Leap Motion Controllers.

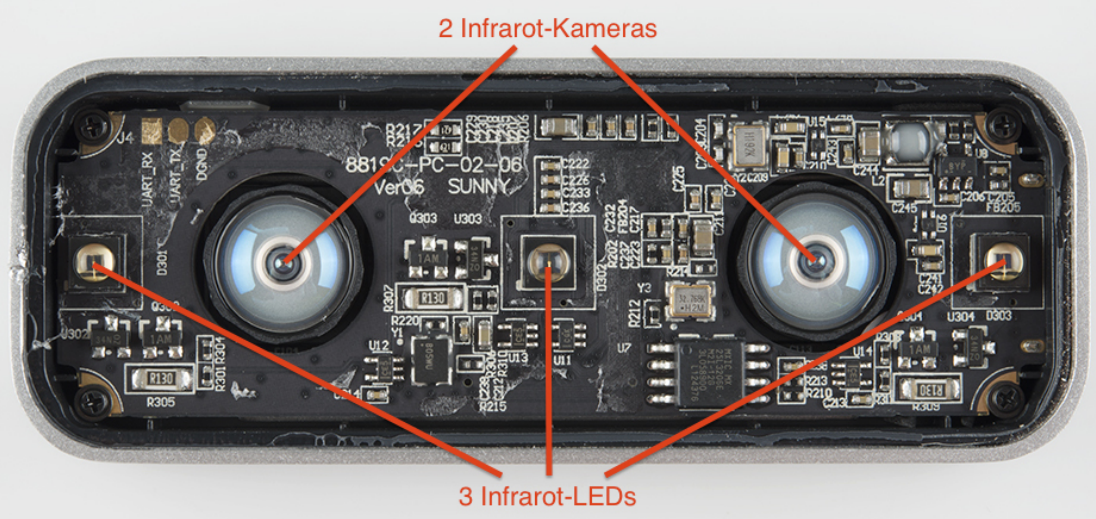

Abbildung 3.2: Hardware-Komponenten des Leap Motion Controllers (vgl. [SparkFun Electronics 2013])

Im Gerät selbst finden keine weiteren Berechnungen statt, die Rohdaten werden direkt über USB weitergeleitet. Der eigentliche Tracking-Algorithmus wird dabei von einem Dienstprogramm (dem „Leap Motion Service“) des Leap Mo-

tion Software-Pakets ausgeführt. Dieses Software-Paket ist für Windows, Mac und Linux verfügbar und muss zwingend installiert werden, um mit den Leap Motion Controller zu kommunizieren (vgl. [Leap Motion Inc. 2013f]).

Der Tracking-Algorithmus der Leap Motion ist am ehesten als ein sogenannter „3D Hand Model Based Approach“ klassifizierbar. Bei derartigen Algorithmen wird ein dreidimensionales Modell der Hand erstellt. Dies erfolgt via Vergleich und Projektion des Input-Bildes auf das schlussendliche Modell. Die Input-Parameter müssen folglich unbedingt präzise und richtig sein, um ein dreidimensionales Modell zu den Werten annähern zu können. Herausforderungen derartiger Algorithmen sind die mögliche Selbstverdeckung der Hände und das – zu kompensierende – Bildrauschen um Verfälschungen zu vermeiden (vgl. [Harshith u. a. 2010], S. 35).

Neben dem Konfigurations-Tool bietet das Software-Paket übrigens auch eine eigene Start-Oberfläche für Leap Motion Applikationen namens Airspace. Hier werden alle kostenlosen bzw. gekauften Applikationen aus dem Airspace Store angezeigt. Die Strategie des Herstellers ist es folglich mit Leap Motion ein komplettes Ökosystem zu schaffen, ähnlich wie es bei mobilen Plattformen (iOS, Android, usw.) schon lange der Fall ist.

3.3 Schnittstellen und Programmierung

Ähnlich wie Microsoft mit dem Kinect SDK, stellt auch Leap Motion ein eigenes SDK zur Verfügung, um das Gerät anzuprogrammieren. Es werden offiziell die Programmiersprachen C++, C#, Java, Python und JS unterstützt und Bibliotheken für diese Sprachen bereitgestellt. Die Programmierschnittstelle („application programming interface“, API) ist bei allen Programmiersprachen ähnlich einzusetzen und die Kommunikation kann sowohl nativ als auch über WebSockets erfolgen. Das Dienstprogramm der Leap Motion Software dient dabei als Vermittler und liefert die berechneten, komplett abstrahierten Informationen zu den getrackten Händen und Fingern (vgl. [Leap Motion Inc. 2013b]).

Abbildung 3.3 visualisiert diese Architektur.

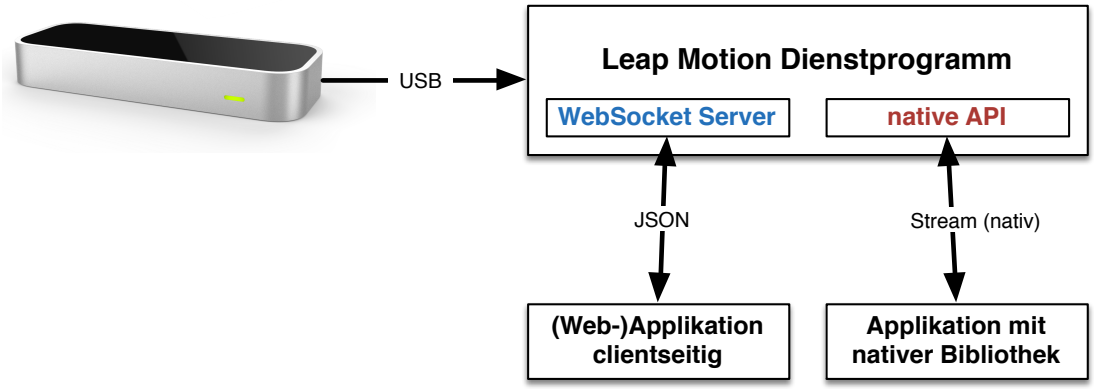

Abbildung 3.3: Architektur der Leap Motion API: WebSockets oder nativ (vgl. [Leap Motion Inc. 2013b])

Wie in Abbildung 3.3 ersichtlich, stehen somit zwei Möglichkeiten, um die Kommunikation mit Leap Motion in Anwendungen zu ermöglichen, zur Verfügung. Bei den nativen Varianten wird direkt mit der API der offiziellen C++-Bibliothek gearbeitet. Bei den WebSockets werden die Daten im „JavaScript Object Notation“-Format (JSON) gesendet, bzw. empfangen (vgl. [Leap Motion Inc. 2013b]). Dieses Format ist äußerst leichtgewichtig und mittlerweile weit verbreitet. JSON wird von den meisten Programmiersprachen unterstützt und ist übrigens auch tatsächlich gültiges JS (ein sogenanntes Objekt-Literal, mehr dazu im Abschnitt 5.1.1 auf Seite 63) (vgl. [Stefanov 2010], S. 49).

Es gibt auch inoffizielle Bibliotheken für andere Programmiersprachen, die auf die (offizielle) API von Leap Motion in nativer Weise zugreifen bzw. über WebSockets kommunizieren. Für die Entwicklung auf der Flash-Plattform kann beispielsweise die offene Bibliothek LeapMotionAS3 von Victor Norgren eingesetzt werden. Sie erlaubt sowohl eine native Kommunikation als auch über WebSockets (vgl. [Norgren 2013]). In wie weit derartige Bibliotheken für einen produktiven Einsatz geeignet sind, sollte jedoch jedenfalls vorab getestet werden. LeapMotionAS3 wurde beispielsweise im Rahmen der praktischen Arbeiten für einen ersten Prototypen (siehe Abschnitt 3.5 auf Seite 45) verwendet

und war zum Zeitpunkt des Testens noch höchst instabil. Soweit möglich, sollten folglich eher die offiziellen Bibliotheken verwendet werden.

Das direkte Auslesen des Tiefenbildes bzw. der Rohdaten des Leap Motion Controllers ist über das Dienstprogramm bzw. über die offiziellen APIs leider nicht möglich. Dies ist eine wesentliche Einschränkung im Vergleich zu den Programmiermöglichkeiten bei Microsofts Kinect. Features, wie beispielsweise Personenerkennung und dreidimensionale Punktwolken bzw. eigenentwickelte Erkennungs-Algorithmen auf Basis der Rohdaten sind mit der offiziellen Leap Motion Software somit nicht möglich (vgl. [Kiefer und Loclair 2013], S. 76).

Das Open-Source Projekt OpenLeap könnte in naher Zukunft bei diesen Problemen aber eventuell behilflich sein. In diesen Projekt wird versucht einen offenen Treiber für Leap Motion zu entwickeln, der auch auf anderen Architekturen wie beispielsweise auf „Advanced RISC Machines“ (ARM) lauffähig ist. Aktuell ist es bereits möglich auf die Rohdaten des Leap Motion Controllers zuzugreifen, der Nachbau der kompletten API der offiziellen Leap Motion Software dürfte laut den Entwicklern allerdings nahezu unmöglich sein. Die mathematische Komplexität des Erkennungsalgorithmus wird als äußerst hoch („*magic*“) eingestuft. Das Projekt befindet sich aktuell in einen sehr frühen Stadium, sodass es derzeit noch schwierig einzuschätzen ist, ob daraus eine produktiv einsetzbare offene API entsteht (vgl. [OpenLeap Initiative 2013]). Im Rahmen dieser Diplomarbeit wurde daher nur mit der offiziellen API von Leap Motion gearbeitet.

3.4 Analyse der Input-Daten

Input-Daten werden von der Leap Motion API in Form von „Frame“-Objekten bereitgestellt, die mehrmals in der Sekunde gesendet werden. Ein Objekt repräsentiert eine einzelne Tracking-Information zu einem bestimmten Zeitpunkt. Es beinhaltet eine Liste aller erfassten Hände, Finger, Stifte und Gesten, inklusive deren Eigenschaften wie beispielsweise Position, Ausrichtung, Geschwin-

digkeit und Rotation. Ein großer Vorteil der API ist, dass die getrackten Positionen bereits in der metrischen Maßeinheit Millimeter gesendet werden (vgl. [Leap Motion Inc. 2013c]). Die Bewegungs- und Positionswerte würden sich somit direkt für Auswertungen und Statistiken eignen.

Im Leap Motion Konfigurationstool kann ein Diagnose-Programm zur Visualisierung der Input-Daten gestartet werden. Dieses Tool wird in diesem Abschnitt eingesetzt, um die Input-Daten zu visualisieren bzw. die Robustheit der Erkennung bei verschiedenen Hand-Posen zu analysieren. Das Bildmaterial aller Abbildungen dieses Abschnitts wurde aus einem aufgezeichneten Video-Testlauf mit Leap Motion generiert. Hierzu wurden die Hand-Posen gleichzeitig mit einer Webcam gefilmt und die Visualisierung des Diagnose-Programms via eines Screen-Capture-Programms aufgezeichnet. Die Abbildungen bestehen aus mehreren alphabetisch durchnummerierten Bildteilen und zeigen jeweils links die Aufnahme der Webcam und rechts die Visualisierung des Diagnose-Programms zum Aufnahmezeitpunkt. Zur besseren Veranschaulichung wurden die Abbildungen zudem um Anmerkungen (in roter Farbe) ergänzt.

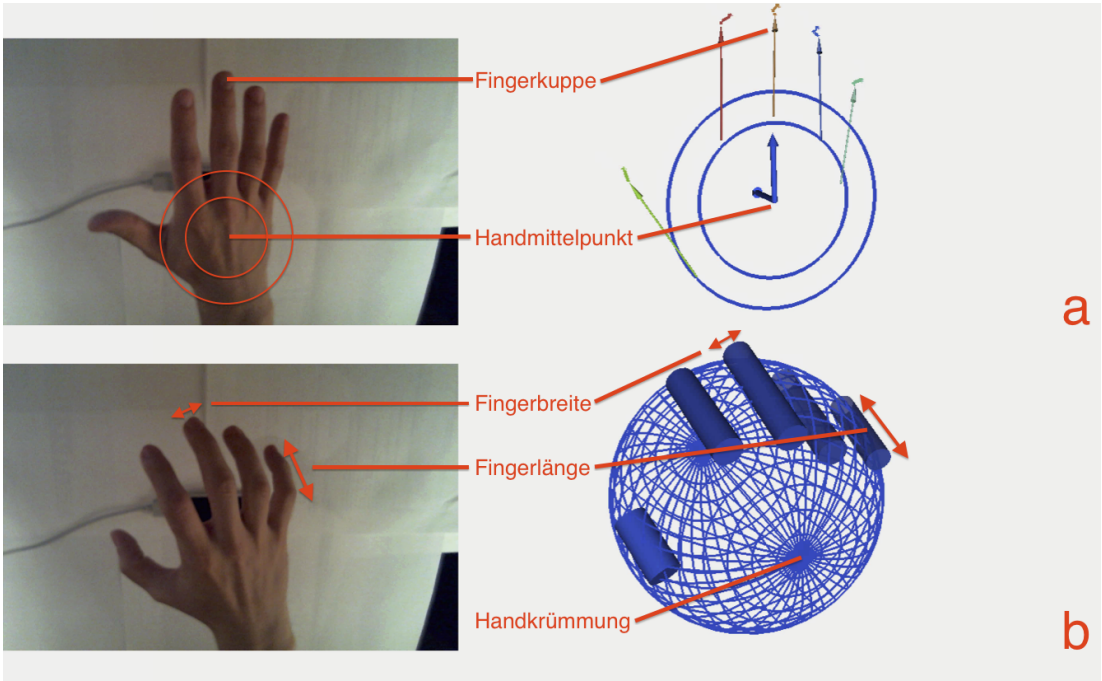

Abbildung 3.4: Analyse der Input-Daten: Hände und Fingerkuppen

Abbildung 3.4 auf der vorherigen Seite zeigt welche Input-Daten prinzipiell von Leap Motion zu erwarten sind. Wie (im Bildteil a der Abbildung) ersichtlich, werden sowohl Position („palmPosition“ bzw. „tipPosition“) als auch Ausrichtungsvektor („direction“) für den Handmittelpunkt und für die Fingerkuppen, durch den Erkennungsalgorithmus erfasst. Aus dem Handmittelpunkt ragt außerdem noch ein Normalvektor („palmNormal“). Die Krümmung der Hände (siehe b) wird durch eine Kugel ausgedrückt, dessen Radius („sphereRadius“) und Mittelpunkt („sphereCenter“) auslesbar sind. Ausgehend von der Position und Ausrichtung der Fingerkuppen, wird auch die Länge und die Breite der getrackten Finger mitgegeben. Weiters steht sowohl von Händen als auch von Fingern, die aktuelle Geschwindigkeit in Millimetern pro Sekunde, („palmVelocity“ bzw. „tipVelocity“) im Vergleich zum letzten „Frame“-Objekt zur Verfügung (vgl. [Leap Motion Inc. 2013c]).

Die vorherige Abbildung 3.4 repräsentiert eher eine Optimal-Situation bei der Erkennung. Der Winkel der Hand und die Spreizung der Finger spielen nämlich eine besonders große Rolle für die Erkennung. Diese Problematik soll in der nachfolgenden Abbildung 3.5 verdeutlicht werden.

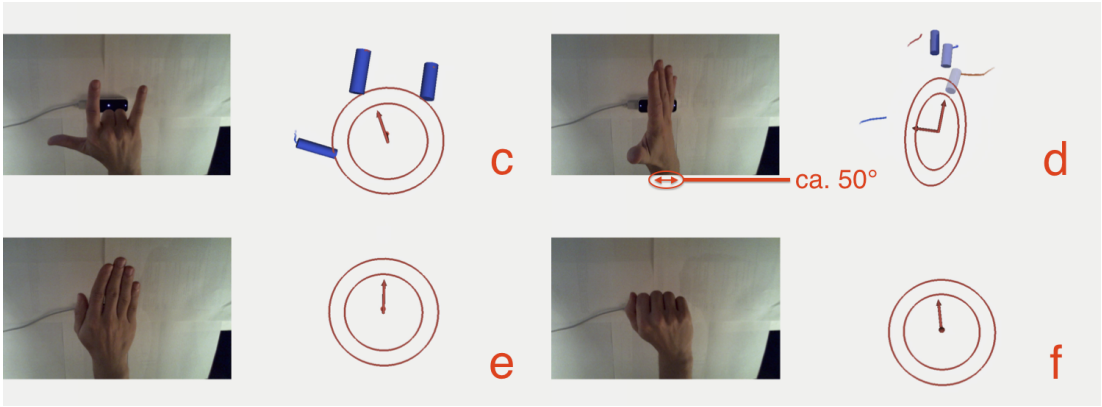

Abbildung 3.5: Analyse der Input-Daten: Probleme

Abbildung 3.5 zeigt vier unterschiedliche Hand-Posen. Während einzelne Finger bei einer parallelen Positionierung der Hand direkt über dem Gerät noch problemlos vom Algorithmus erkannt werden (siehe c), ist eine robuste Erkennung der Finger, ab einen gewissen seitlichen Hand-Rotations-Winkel nicht

mehr gegeben (siehe d). Nach Beobachtungen dürfte dieser tote Winkel, in dem keine Finger erkannt werden können, zwischen 50° und 130° liegen. Das Tracken von Fingern, wenn der Handrücken parallel zum Controller (180°) zeigt, ist jedoch möglich. Wie aus den unteren Bildteilen der Abbildung (e und f) ersichtlich wird, müssen die Finger für eine erfolgreiche Erkennung zudem unbedingt gespreizt werden. Eine flache Hand (siehe e) führt ansonsten zum selben Ergebnis wie eine Faust-Pose (siehe f). Auch das Auslesen des Radius der Sphäre (siehe b in der vorherigen Abbildung 3.4) erlaubt hier keine hinreichende Unterscheidung zwischen den zwei Zuständen.

Applikationen, die ausschließlich mit Leap Motion gesteuert werden, sollten die aufgezeigte Problematik aus der letzten Abbildung nicht unterschätzen. Wird mit Leap Motion beispielsweise die Position eines Cursors gesteuert, sollte dieser nicht ausschließlich von der erfolgreichen Erkennung der Finger abhängen, sondern jedenfalls auch die konstanter verfügbare Position des Handmittelpunkts miteinbeziehen. Ohne Spreizen der Finger, bzw. ab einer gewissen seitlichen Rotation der Hand, werden keine Finger mehr erkannt. Diese beiden Einschränkungen sollten vor Entwicklungsbeginn einer Anwendung mit Leap Motion daher unbedingt bedacht werden, da gewisse Interaktionen hierdurch nur eingeschränkt, bzw. unmöglich umsetzbar sind.

Die gleichzeitige Erkennung von zwei oder mehreren Händen ist zwar möglich, sollte jedoch kritisch betrachtet werden. Das Potential, dass sich die Hände im Interaktionsbereich gegenseitig verdecken ist groß. Der Hersteller empfiehlt zwar maximal zwei Hände gleichzeitig, nach der Einschätzung des Autors, ist der mögliche Raum aber selbst dafür zu klein und die Bewegungsfreiheit ist extrem eingeschränkt. Applikationen die mehrhändige Interaktionen mit Leap Motion verwirklichen wollen, sollten daher unbedingt vorher einen Prototypen erstellen und überprüfen ob genügend Input-Daten für eine Umsetzung zur Verfügung stehen. Die API bietet darüber hinaus auch keine direkte Möglichkeit, um zwischen linker und rechter Hand zu unterscheiden. Eine derartige Unterscheidung müsste in der Applikation mit einem eigenen Algorithmus vorgenommen werden (vgl. [Leap Motion Inc. 2013c]).

Neben der Positionserkennung von Händen und Fingern, werden auch Stifte und vier verschiedene Typen von Gesten im Interaktionsbereich erkannt (vgl. [Leap Motion Inc. 2013c]). Abbildung 3.6 visualisiert dies.

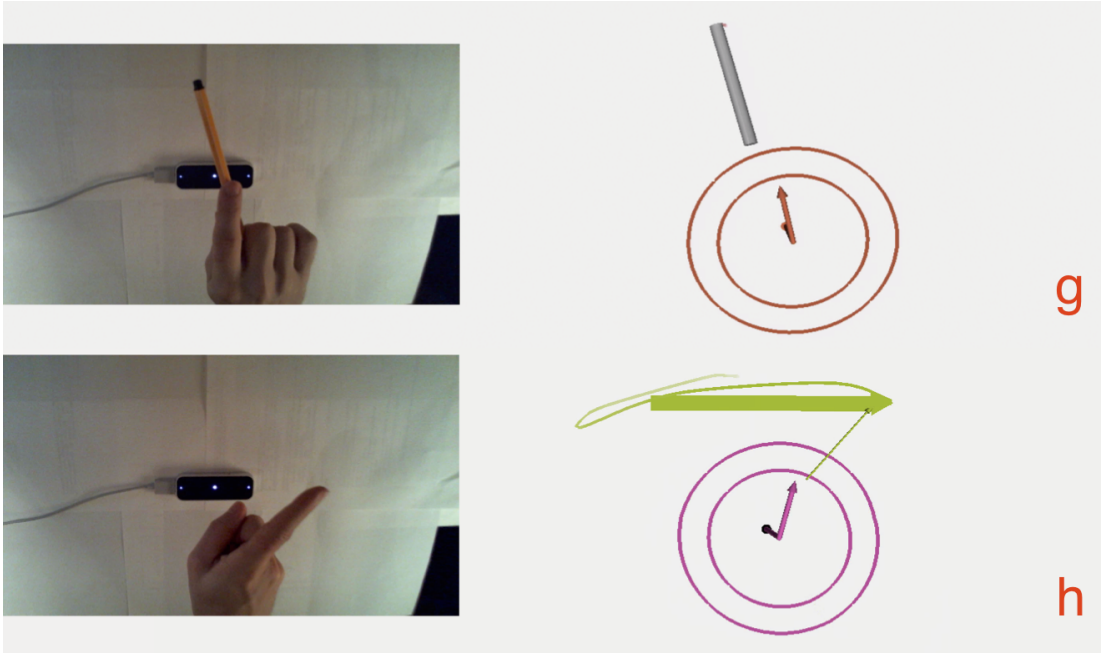

Abbildung 3.6: Analyse der Input-Daten: Stifte und Gesten

Wie in Abbildung 3.6 ersichtlich, werden Stifte (siehe g) von Fingern differenziert (andere Farbe). Sie werden als „Tools“ im „Frame“-Objekt gesendet, besitzen ansonsten aber dieselben Eigenschaften wie Finger. Gesten wie die hier aufgezeigte Wischgeste (siehe h) werden in einer eigenen Liste „Gestures“ im „Frame“-Objekt mitgesendet. Neben der Wischgeste („Swipe“), werden auch Kreisbewegungen („Circle“) und zwei Arten von Tipp-Bewegungen – zum Bildschirm hin („Screen Tap“) und Klicks in der Luft („Key Tap“) – erkannt (vgl. [Leap Motion Inc. 2013c]).

3.5 Erstellung einer Prototyp-Applikation

Im Rahmen dieser Diplomarbeit soll die Applikation Physiogame aufgrund von klaren Anforderungen konzipiert und entwickelt werden. Bevor diese Arbeiten

jedoch begonnen werden konnten, musste getestet werden, ob Adobe Flash eine bessere Leistung zur Umsetzung der Applikation, als JS bieten würde. Es wurde daher eine Prototyp-Applikation mit der inoffiziellen Bibliothek LeapMotionAS3 in Adobe Flash Builder 4.7 erstellt. Diese Entwicklungsumgebung wird im Rahmen dieses Abschnitts außerdem dazu eingesetzt, um das „Frame“-Objekt abschließend zu analysieren. Abbildung 3.7 zeigt einen Ausschnitt des Prototyps.

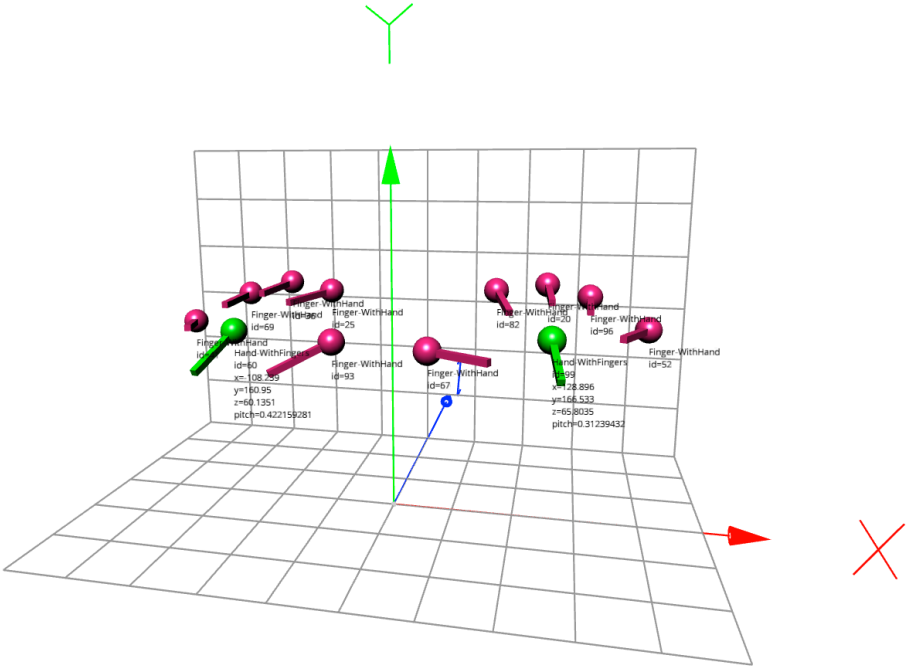

Abbildung 3.7: Leap Motion Prototyp-Applikation mit Adobe Flash

Wie in Abbildung 3.7 ersichtlich wird, wurde eine Art Diagnose-Applikation entwickelt. Es wurde ein Gitter im dreidimensionalen Raum erstellt, das den Interaktionsbereich von Leap Motion repräsentieren soll. Die Position der Fingerkuppen (in der Farbe pink) und der Handmittelpunkte (in Grün) werden durch eine Kugel visualisiert. Aus diesem ragen die Richtungsvektoren der jeweiligen getrackten Punkte heraus. Daneben können noch einige Tracking-Daten interaktiv eingeblendet werden. Da sich die API der offiziellen Bibliotheken und von LeapMotionAS3 ähneln, konnten durch diesen Prototyp somit jedenfalls Erfahrungen gesammelt werden, unabhängig davon ob die Plattform für eigentlichen Entwicklung von Physiogame tatsächlich eingesetzt wird.

3.5.1 Probleme

Zur 3D-Visualisierung wurde die Bibliothek Away3D, für alle zweidimensionalen Informationen (Text) wurde Starling eingesetzt. Dadurch werden alle visuellen Berechnungen auf die „Graphics Processing Unit“ (GPU) ausgelagert, wodurch dem Erkennungsalgorithmus von Leap Motion (im Dienstprogramm) folglich die volle Leistung des Hauptprozessors zur Verfügung stehen müsste. Die Datenrate vom Leap Motion Dienstprogramm blieb jedoch sowohl bei einer nativen Kommunikation (durch eine „AIR Native Extension“) als auch über WebSocket beim Prototyp nie konstant, ganz im Gegensatz zu den offiziellen Beispiel-Projekten von Leap Motion mit JS.

Weiters stützte die Applikation aufgrund von Fehlern in der inoffiziellen Bibliothek LeapMotionAS3 häufig ab. Zum Zeitpunkt der Entwicklung dieser Prototyp-Applikation am 07. Juni 2013, war die Flash-Plattform folglich nicht für produktive Applikationen mit Leap Motion geeignet. Es wurde daher beschlossen, zur Entwicklung der Applikation Physiogame im Rahmen dieser Diplomarbeit JS zu verwenden.

3.5.2 Analyse des „Frame“-Objekts

Da sich die APIs der Bibliotheken wie bereits angesprochen gleichen, lohnt es sich die Struktur des „Frame“-Objekts, im Rahmen der Entwicklung des Prototypen zu analysieren. Dies erfolgt über die Debug-Sicht im Adobe Flash Builder 4.7. Abbildung 3.8 auf der nächsten Seite zeigt die Eigenschaften des „Frame“-Objekts (linker Bildteil). Die Eigenschaften der getrackten Hände und Finger des aufgezeichneten „Frame“-Objekts, werden in der Mitte bzw. auf der rechten Seite der Abbildung dargestellt.

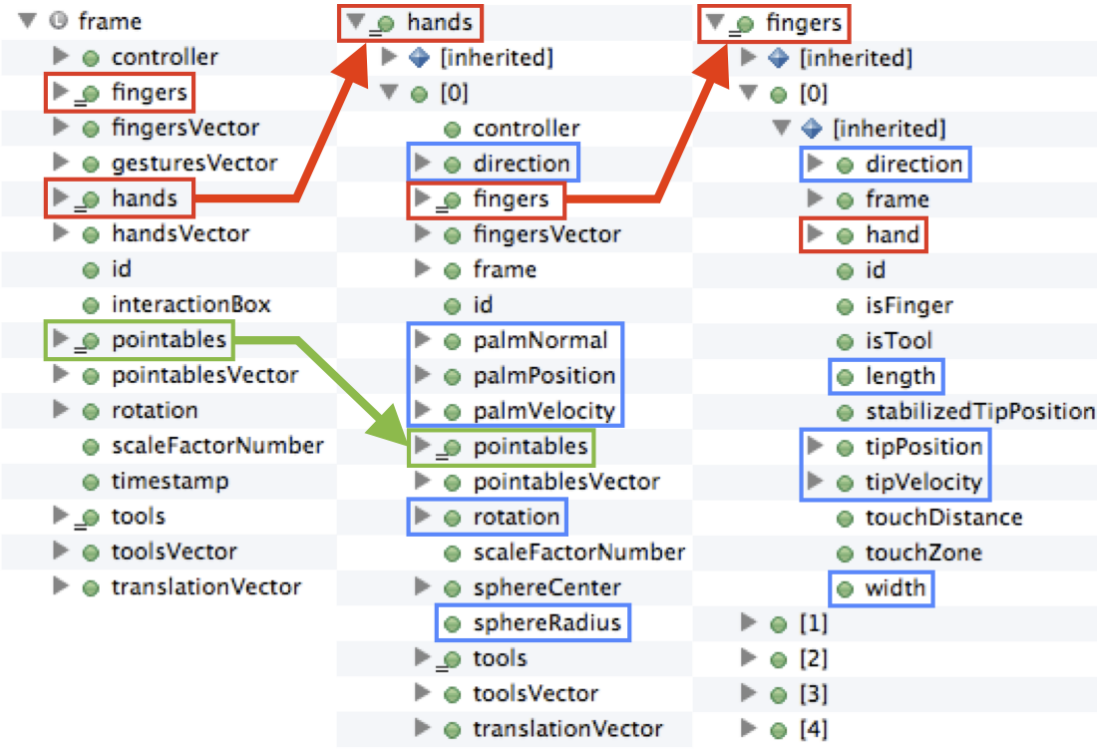

Abbildung 3.8: Das „Frame“-Objekt der Leap Motion API

Die blauen Markierungen aus der oberen Abbildung 3.8 zeigen die Eigenschaften einer Hand bzw. eines Fingers, die bereits im Rahmen die Analyse der Input-Daten (siehe vorheriger Abschnitt) vorgestellt wurden. Wie durch die rote Markierung ersichtlich wird, werden die Finger mit den Händen automatisch assoziiert, können aber auch direkt aus dem „Frame“-Objekt abgefragt werden. Es können also direkt alle Finger des getrackten Raumes, bzw. alle Finger einer einzelnen Hand ausgelesen werden. Sowohl Finger als auch Stifte („Tools“) sind sogenannte „Pointables“ und ebenso im „Frame“- als auch im Hand-Objekt verfügbar (siehe grüne Markierung).

Schlussendlich ist es wichtig anzumerken, dass jedes „Finger“- bzw. „Hand“-Objekt eine eindeutige ID besitzt. Sie behalten dieselbe ID bis die Hand, bzw. der Finger den Interaktionsbereich verlässt (vgl. [Leap Motion Inc. 2013c]). Hierdurch können einzelne Finger beispielsweise leicht über mehrere „Frames“ nachverfolgt werden.

3.6 Zusammenfassung

Da die Technologie von Leap Motion patentiert ist, können wenig Aussagen über den Erkennungsalgorithmus, der vom Dienstprogramm ausgeführt wird, getroffen werden. Bei der Analyse der Input-Daten wurden einige Schwächen (eingeschränkter Winkel, Spreizen der Finger nötig, Verdeckung bei mehreren Händen) gefunden, die bei der Definition der Anforderungen für die Applikation Physiogame berücksichtigt werden müssen. Die offizielle API wirkt insgesamt durchdacht und liefert ähnliche Möglichkeiten, wie die Skelettprojektion des Kinect SDKs. Aufgrund der Leistungsprobleme mit der Flash-Plattform in Kombination mit der inoffiziellen LeapMotionAS3 Bibliothek, ist die Umsetzung von Physiogame mit JS jedenfalls die bessere Wahl, da offizielle Bibliotheken bereitstehen.

Im nächsten Kapitel erfolgt nun die Definition der Anforderungen an die Applikation Physiogame.

4 Anforderungen an die Applikation Physiogame

Physiogame soll zur Bewegungstherapie eingesetzt werden können. Derartige Applikationen bzw. Systeme sind im medizinischen Bereich unter dem Begriff „Virtual Rehabilitation“ bekannt. Es ist daher von Bedeutung diesen Begriff zu beleuchten und die wichtigsten Kriterien, die derartige Systeme aufweisen müssen, zu erläutern. Dies erfolgt im nächsten Abschnitt.

In zweiten Abschnitt 4.2 auf Seite 53 werden die Anforderungen an die Applikation Physiogame präsentiert und der Hintergrund zu selbigen erläutert. Schlussendlich werden die Erkenntnisse dieses Kapitels auf Seite 59 zusammengefasst gegeben und auf das nächste Kapitel übergeleitet.

4.1 Virtuelle Rehabilitation

„Virtual Reality“ (VR) hat das Ziel die sensorische Wahrnehmung von Benutzern von der realen Welt in eine virtuelle umzulenken. Zur Interaktion mit virtuellen Welten bieten sich dabei die Hände besonders an, daher sind vor allem natürliche Benutzerschnittstellen zur Schaffung von VR interessant (vgl. [Ghosh u. a. 2010], S. 323). Interaktion ist tatsächlich das wichtigste Feature jeder VR-Applikation (vgl. [Sveistrup 2004]). Mittlerweile sind VR-Technologien auch in der Medizin relevant, sie werden beispielsweise zur Umsetzung von virtuellen Eingriffen und für Lehrzwecke eingesetzt. Eher neu ist jedoch der Einsatz von VR-Technologien in der Therapie, die sogenannte virtuelle Rehabilitation (vgl. [Burdea 2003], S. 519 und [Holden 2005], S. 187).

Klassische Therapien werden häufig schnell als langweilig empfunden, da viele Übungen aus wiederkehrenden Abläufen bestehen und die Motivation von Patienten somit nach kurzer Zeit sinkt. In der virtuellen Rehabilitation können interaktive Applikationen dazu eingesetzt werden, die Motivation des Patienten zu steigern. Eine Steigerung tritt vor allem dann ein, wenn die Therapie in einer spielerischen Weise erfolgen kann. Grafik- und Audio-Effekte können zusätzlich zur Belohnung des Patienten genutzt werden. Motivation ist also eines der Hauptziele, die in virtuellen Rehabilitationen erreicht werden soll. Auch der Wettbewerb zwischen Patienten könnte in der Zukunft eine Rolle spielen und weiters motivierend wirken (vgl. [Burdea 2003], S. 520).

Abbildung 4.1 zeigt einige typische Applikationen eines Systems zur virtuellen Rehabilitation, das sogenannte SeeMe VR-System. Das System benötigt lediglich eine handelsübliche Webcam. Bewegungen werden direkt aus dem Bild-Material, durch einen Motion Tracking Algorithmus extrahiert. Das Kamerabild wird zusätzlich am Bildschirm angezeigt um eine virtuelle Umgebung zu simulieren. Bei der Applikation SeeMe Cleaner muss der Bildschirm durch „Wisch“-Bewegungen mit den Händen freigemacht werden. Das Ziel der Applikation React ist es die (blauen) Bälle so schnell wie möglich mit den Händen zu erreichen, während bei SeeMe Ball, Bälle (virtuell) gefangen und anderen Objekten ausgewichen werden muss (vgl. [Sugarman u. a. 2012], S. 2f.). Wie dabei ersichtlich wird, ist der Spielablauf und die Art der Interaktion eher simpel, trotz der wiederkehrenden Bewegungsabläufe kann mit derartigen Applikation aber die Motivation signifikant gesteigert werden.

Abbildung 4.1: VR-System zur virtuellen Rehabilitation: SeeMe (vgl. [Sugarman u. a. 2012], S. 3)

Die sich wiederholenden Aufgaben müssen mit einem Ziel bzw. mit einem Erfolg verbunden werden. Der Spielablauf sollte zudem an den Fortschritt der Patienten anpassbar sein. Bei interaktiven Applikationen zur Bewegungstherapie ist im Übrigen das Geben von Feedback (wenn möglich in Echtzeit) von außerordentlicher Bedeutung und beeinflusst die Lernkurve positiv. (vgl. [Holden 2005], S. 189 und S. 192). Somit müssen sowohl die Intensität des Feedbacks als auch der Schwierigkeitsgrad der Trainingseinheit, auf die jeweilige Behandlungssituation angepasst werden können (vgl. [Jack u. a. 2000], S. 56).

Übungen in der virtuellen Rehabilitation bieten den Vorteil, dass sie leichter standardisiert und ausgewertet werden können (vgl. [Sveistrup 2004]). Eine weitere Möglichkeit die sich ergibt, ist die Tele-Rehabilitation. Patienten können Übungen so auch zu Hause ausführen. Die Daten sollen dabei automatisch mit dem Servern des Krankenhauses synchronisiert werden (vgl. [Burdea 2003], S. 521). Das Training mit interaktiven Applikationen kann so leichter auf größere Zeiträume nach der Hospitalisierung und Rehabilitation erweitert werden (vgl. [Jack u. a. 2000], S. 57).

Die wichtigsten Kriterien zur Erstellung von interaktiven Applikationen für den Einsatz in der Therapie sind in der nachfolgenden Auflistung zusammengefasst (vgl. [Holden 2005], S. 189 und S. 192, [Burdea 2003], S. 520):

- Die **Motivation** in Therapien soll durch den Einsatz von spielerischen und herausfordernden Abläufen gesteigert werden.
- Die **Wiederholung** von Übungen (Bewegungsmuster) wird durch die Definition von (Spiel-)Zielen motivierender und muss durch unmittelbares **Feedback** unterstützt werden.
- Applikationen müssen über umfangreiche **Konfigurations-Möglichkeiten** verfügen, sodass beispielsweise der Schwierigkeitsgrad und die Interaktion auf den jeweiligen Patienten anpassbar sind.
- Der Fortschritt von Patienten muss durch **Monitoring-Methoden** überwacht werden und diese Daten können für **Auswertungen** verwendet werden.

- Die **Portabilität** einer Applikation muss gegeben sein, um die Software auch zur Tele-Rehabilitation einsetzen zu können.

Physiogame muss die hervorgehobenen Kriterien folglich unterstützen. Der Leap Motion Controller dürfte sich für den medizinischen Einsatz besonders eignen. Da er kontaktlos funktioniert, stellen typische Desinfektionsmaßnahmen kein großes Problem dar. Durch die kleinen Ausmaße und der unkomplizierten Installation kann er als portabel angesehen werden, was im Fall von Tele-Rehabilitation ein großer Vorteil ist. Auch der Preis der Benutzerschnittstelle fällt mit rund 80 Dollar – im Gegensatz zu gängigen medizinischen VR-Spezial-Equipment – äußerst niedrig aus (vgl. [Burdea 2003], S. 522).

4.2 Anforderungen

Um den Hintergrund der Anforderungen zur Entwicklung der Applikation Physiogame zu verstehen, ist es von integraler Bedeutung den Ablauf der Konzeption und Entwicklung zu beleuchten. Dies erfolgt daher gleich im nächsten Unterabschnitt, während die nachfolgenden Unterabschnitte dann die tatsächlichen Anforderungen an die Applikation Physiogame behandeln.

4.2.1 Beschreibung des Entwicklungsablaufs

Die Planung zur Entwicklung einer Applikation mit dem Leap Motion Controller begann bereits im Jänner 2013. Das Produkt konnte zu diesem Zeitpunkt ausschließlich als „Dev-Kit“ und auf Einladung des Unternehmens bezogen werden. Die „Digital Media Technologies“-Abteilung der FH JOANNEUM beschloss daher den Autor bei der Beschaffung zu unterstützen und es wurden Dokumente zur Einladung in das „Developer Program“ an das Unternehmen gesendet. Am 15. Mai 2013 erhielt der Autor schlussendlich den Leap Motion Controller, rund zwei Monate bevor er offiziell veröffentlicht wurde. Das

Produkt konnte sodann analysiert und im Rahmen einer Prototyp-Applikation (siehe Abschnitt 3.5 auf Seite 45 des vorherigen Kapitels) getestet werden.

Eine Kooperation mit Lehrenden der FH-Studiengänge Ergo- und Physiotherapie wurde fixiert und am 8. Juli 2013 wurde ein erstes Meeting zur Besprechung der Anforderungen der Applikation Physiogame durchgeführt. Die Konzeption der Anwendung Physiogame konnte darauf starten und am 17. September 2013 wurde ein erster Prototyp der Applikation vorgestellt. Mit dem Feedback wurden folglich die schlussendlichen Fix-Anforderungen bestimmt und die Applikation konnte am 24. Oktober 2013 fertiggestellt werden.

Die Anforderungen an Physiogame wurden in Rücksprache mit dem Betreuer der Diplomarbeit Prof. DI Dr. Alexander Nischelwitzer, den Ergotherapie-Lehrenden Romana Toriser und Julia Unger, MSc und den Physiotherapie-Lehrenden Mag. Dr. rer. nat. Petra Gröbl und Barbara Gödl-Purrer, MSc getroffen. Sie repräsentieren die finalen Endanforderungen an Physiogame.

4.2.2 Interaktion

Physiogame soll zur Bewegungstherapie der Hände, der Finger bzw. des Ellbogengelenks eingesetzt werden können. Prinzipiell soll durch die Applikation, das Timing und die Präzision von Bewegungen trainiert werden. Bewegungen im Interaktionsbereich sollen zielgerichtet erfolgen. Die Applikation soll Patienten durch die natürliche Interaktion mit Spielobjekten zum Initiieren und Stoppen von Bewegungen animieren. Das übergeordnete Ziel ist ferner sowohl Koordinations- als auch Kraftprobleme in Verbindung mit den Armen, Händen und Fingern mit Physiogame trainieren zu können.

Die Interaktion mit den angesprochenen Spielobjekten in Physiogame soll auf drei verschiedenen Weisen (auch „Interaktionsarten“ bzw. „Interaktionsformen“ genannt) erfolgen können:

1. Die Hand des Patienten soll sich – im Interaktionsbereich von Leap Motion – zu einem (virtuellen) Spielobjekt bewegen, in dieser Position für

eine einstellbare Zeitspanne verweilen und dann zum nächsten Spielobjekt weiterbewegen. Nur die Abfrage der X- und Y-Achse von Leap Motion ist bei dieser Interaktion also nötig.

2. Ähnlich zur vorherigen Interaktionsart, soll ein Spielobjekt durch eine Hand erfasst werden, danach jedoch zusätzlich eine Vorwärts- und Rückwärtsbewegung (zum Bildschirm hin) ausgeführt werden, bevor die Weiterbewegung zum nächsten Spielobjekt erfolgt. Hierfür ist folglich die Abfrage der Z-Achse zusätzlich relevant und die Interaktion erfolgt tatsächlich in einem virtuellen Raum.
3. Wiederum soll ein Spielobjekt erfasst werden, dabei jedoch eine bestimmte Anzahl an Fingern für eine einstellbare Zeitspanne im Interaktionsbereich der Leap Motion gezeigt werden.

Ein spezielles Augenmerk soll auf die Länge der Handbewegungen gelegt werden. Die Zeitspanne, zwischen Initiieren und Stoppen einer Handbewegung, soll zum Beispiel wechselweise in fünf bis zehn Sekunden stattfinden können, bzw. im besten Fall konfigurierbar sein. Das Starten und Stoppen von Spielrunden soll zudem selbstständig von Patienten vorgenommen werden können, das Verändern von Einstellungen jedoch den Therapeuten vorbehalten sein.

Die angestrebte Patienten-Interaktion ist eher als einfach zu bezeichnen, ähnlich zu den im Abschnitt 4.1 auf Seite 50 vorgestellten SeeMe Applikationen. Dies ist auch absolut erwünscht und wird ebenso von den „User Experience Guidelines“ von Leap Motion empfohlen. Das Erlernen von komplexen Handgesten soll vermieden werden. Je weniger Lernaufwand erforderlich ist und je näher die Interaktion der realen Welt nachempfunden ist, desto intuitiver und natürlicher fühlt sich die Applikation für Benutzer an. Der erste Gedanke eines Benutzers bzw. einer Benutzerin wie die Interaktion erfolgen könnte, soll bereits der Richtige sein (vgl. [Leap Motion Inc. 2013e]).

4.2.3 Gestaltung, Spielablauf und Motivation

Physiogame zielt primär auf die Zielgruppe der bewegungseingeschränkten (und eventuell auch hyperaktiven) Kinder ab und soll der Wiederherstellung der Schulter-Arm-Funktion und dem Training der Finger-Augen-Koordination dienen. Ursprünglich wurde die Zielgruppe von Physiogame noch weiter ausgelegt, beispielsweise auch zum Einsatz in der Therapie von älteren Patienten, um die Selbstständigkeit nach schweren, bzw. multiplen Erkrankungen wiederherzustellen. In diesem Zusammenhang war es daher wichtig, dass die virtuelle Spielwelt – die visuelle Gestaltung – von Physiogame austauschbar ist. Bereits anfänglich wurde zudem definiert, dass sich die visuelle Umsetzung auf den zweidimensionalen Raum beschränken soll, da Anwender mit der Interpretation einer dreidimensionalen Spielwelt Schwierigkeiten haben könnten.

Physiogame darf keine gewalttätigen Handlungen implizieren, zur visuellen Gestaltung der Spielobjekte wurde daher primär die Metapher eines Luftballons ausgewählt. Diese können, durch eine der – im vorherigen Abschnitt vorgestellten – drei Möglichkeiten zur Interaktion, zum Platzen gebracht werden. Bei der dritten Form der benötigten Interaktion wird zusätzlich eine Zahl am Spielobjekt, für die Visualisierung der Anzahl der zu zeigenden Finger, eingeblendet. Da sich in der Regel mehrere Spielobjekte am Spielfeld befinden, werden Patienten dazu animiert ihre Bewegungen zu planen, beispielsweise um insgesamt den kürzesten Bewegungsweg zurückzulegen. Die Spielmodi konnten ferner auf zwei Möglichkeiten fixiert werden:

- Der Spielmodus **„nach Anzahl“,** mit dem Ziel eine definierte Anzahl von Spielobjekten so schnell wie möglich zu erwischen (bzw. Luftballons zum Platzen zu bringen).
- Der Spielmodus **„nach Zeit“,** mit dem Ziel so viele Spielobjekte wie möglich innerhalb eines definierten Zeitfensters zu erreichen. Neue Spielobjekte werden hierbei wellenweise in bestimmten Zeitabständen wieder zur Spielfläche hinzugefügt.

Somit sind zwei mögliche Spielmodi und drei Formen der Interaktion unabhän-

gig voneinander wählbar.

Feedback ist bei Applikationen zur virtuellen Rehabilitation besonders wichtig und wird zudem auch explizit in den Best Practices, zur Entwicklung von Applikationen mit den Leap Motion Controller, erwähnt. Je mehr Feedback, desto eher wird die Art der Interaktion von Benutzern angenommen. Das Feedback soll dabei simpel sein und seinen lediglich Zweck erfüllen, ohne ablenkend zu wirken. Es soll die Lernkurve unterstützen und motivieren (vgl. [Leap Motion Inc. 2013e]).

Um die Metapher des Luftballons aufrecht zu halten, soll das „Platzen“ mit Audio- und Bild-Effekten simuliert werden. Um die Motivation weiter zu steigern, sollen die Spielobjekte möglichst mittig getroffen werden. Somit kann die Treffergenauigkeit als sofortiges Feedback zurückgegeben werden. Der Interaktionsbereich des Leap Motion Controllers ist begrenzt. Es ist daher auch wichtig Feedback bereitzustellen, wenn sich die Hand außerhalb des erfassbaren Bereichs befindet.

4.2.4 Konfiguration

Eine Applikation, die in der virtuellen Rehabilitation eingesetzt werden soll, muss zahlreiche Möglichkeiten zur Konfiguration bieten. Dies betrifft nicht nur das Aussehen und die Art der Interaktion, sondern auch das Verhalten der Spielobjekte (zum Beispiel die Geschwindigkeit), die Spielweise und das Eingabeverhalten des Leap Motion Controllers (Kalibrierung). Bei der Entwicklung von Physiogame muss daher darauf geachtet werden, dass die schlussendliche Applikation in zahlreichen Belangen – vorzugsweise interaktiv – verändert werden kann und somit den Fähigkeiten der Patienten entspricht (vgl. [Sugarman u. a. 2012], S. 3).

4.2.5 Statistiken

Damit der Trainingserfolg von Patienten ausgewertet werden kann, muss in Applikationen aus dem Gebiet der virtuellen Rehabilitation, jedenfalls ein Monitoring der Bewegungen vorgenommen werden. Hier kann die Leap Motion API ihren Vorteil ausspielen: Positionsdaten werden in der Maßeinheit Millimeter gesendet und dadurch muss keine – ansonsten sicherlich aufwendige – Umwandlung in das metrische Einheitensystem, von den Input-Daten vorgenommen werden (vgl. [Leap Motion Inc. 2013c]). Neben den Bewegungsdaten ist auch die Erfassung von spiel-spezifischen Daten, wie beispielsweise Treffergenauigkeit, Spielzeit, Anzahl der Punkte, usw. für spätere Auswertungen wichtig.

Physiogame muss Möglichkeiten bieten, um die Daten einer jeweiligen Spielrunde zu sammeln und für eine spätere Verwendung zu speichern. Die Entwicklung eines Speicherungs-Mechanismus, ist dabei auch für die Wiederherstellung der Einstellungen von früheren Spielrunden wichtig. Damit eine Analyse der Statistik-Daten erfolgen kann, muss es außerdem noch möglich sein, die Daten aus dem Programm in ein gängiges Format zu exportieren. Von hoher Relevanz ist dabei das „Comma-separated Values“-Format (CSV), wodurch die Daten später in Microsoft Excel importiert werden können.

4.2.6 Portabilität

Da die Möglichkeit einer Tele-Rehabilitation prinzipiell gegeben sein soll, muss dieser Faktor bei der Entwicklung von Physiogame berücksichtigt werden. Die Portabilität wirkt sich auch effektiv auf die Auswahl der Programmiersprache bzw. der Plattform aus. Physiogame soll direkt im Browser lauffähig sein und auch als Desktop-Version ausgeliefert werden können, um die Anforderungen an die Portabilität zu erfüllen. Sowohl eine Web-Applikation in JS als auch die Flash-Plattform würden sich hierfür eignen. Eine häufige Anforderung – das Synchronisieren der Therapiedaten mit einem Online-Server – muss jedoch

nicht durch Physiogame erfüllt werden, da dies den Umfang der Applikation sprengen würde.

4.2.7 Anforderungen an die Software-Entwicklung

Die Entwicklung von Applikationen – in der Größe von Desktop-Applikationen – mit JS, ist noch ein sehr junges Gebiet und es muss daher geprüft werden, wie Applikationen mit dieser Sprache konzipiert und implementiert werden können. Dies umfasst beispielsweise Punkte, wie gängige Konventionen, das Testen des Codes, Modularisierung und Patterns mit JS verwendet werden. Auch die neuen Einsatzgebiete von JS, Node.js und Node-webkit müssen betrachtet werden. Die Anforderung diese Applikation mit JS zu implementieren, ist ein zentrales Thema dieser Diplomarbeit und das Kapitel 5 auf Seite 61 und Kapitel 6 auf Seite 80 befasst sich ausführlich mit dieser Anforderung.

4.3 Zusammenfassung

Im Rahmen dieses Kapitels wurde der Bereich der virtuellen Rehabilitation beleuchtet und Kriterien, die derartige Systeme bzw. Applikationen erfüllen müssen, vorgestellt. Danach wurde sowohl der Feedback- und Planungsablauf als auch die eigentlichen Anforderungen zur Entwicklung der Applikation Physiogame präsentiert.

Abbildung 4.2 auf der nächsten Seite zeigt die wichtigen Anforderungen für Physiogame aus den vorherigen Unterabschnitten übersichtlich zusammengefasst. Physiogame soll drei verschiedene Formen der Interaktion durch den Leap Motion Controller bieten. Luftballons werden als Metapher eingesetzt und zwei Spielmodi sollen verfügbar sein. Sowohl das Spielverhalten, die Kalibrierung des Leap Motion Controllers als auch die visuellen Inhalte, müssen interaktiv veränderbar sein. Es muss ermöglicht werden Statistiken speichern

und exportieren zu können. Weiters soll Physiogame als Web- und Desktop-Applikation lauffähig sein.

Abbildung 4.2: Übersicht der Anforderungen zu Physiogame

Physiogame wird ausschließlich mit Web-Technologien entwickelt. JS zur Entwicklung von derartig komplexen Applikationen einzusetzen, kann aktuell noch als „Neuland“ bezeichnet werden. Im nächsten Kapitel erfolgt daher eine grundlegende Analyse der Programmiersprache JS.

5 Die mannigfaltigen Gesichter von JavaScript

JS ist vermutlich eine der unterschätztesten Programmiersprachen auf der Welt. Aus der ursprünglichen Intention, bestimmte Elemente des „Domain Object Models“ (DOM) von HTML-Seiten zu manipulieren, entstand eine komplexe Sprache, die sich von traditionellen klassenbasierten Sprachen sehr stark unterscheidet. Umso erstaunlicher ist, dass die Einstiegsbarriere um einige Zeilen Code in JS zu produzieren, derart niedrig ist, sodass die Annahme, JS müsste gar nicht erlernt werden, sehr weit verbreitet ist. Dies kann wohl aber ebenso der Ausdruckskraft der Sprache zu Gute gelegt werden. Abseits von kleinen Codespielereien, dürfte die Ignoranz, Code in JS ohne Wissen der Spracheigenheiten programmieren zu können, jedoch schnell zu Frustration führen (vgl. [Crockford 2008], S. 2).

Der Zweck dieses Kapitels ist ein Basisverständnis für JS herzustellen. Im nächsten Abschnitt „Signifikanz trotz Eigenheiten“, erfolgt eine Einführung in die Eigenheiten und Relevanz von JS. In den nachfolgenden Abschnitten werden zuerst die Verbreitung (Abschnitt 5.2 auf Seite 67) und dann die neuen Einsatzmöglichkeiten Node.js und Node-webkit (Abschnitt 5.3 auf Seite 69) unter die Lupe genommen. JS wird bekanntlich von verschiedensten Applikationen (zumeist Browsern) unterschiedlicher Hersteller implementiert. Im Abschnitt „Weiterentwicklung und Kompatibilität“ auf Seite 76 wird der Status der Weiterentwicklung des JS-Standards ECMAScript erläutert und eine Maßnahme zur Schaffung von Kompatibilität in älteren Umgebungen vorgestellt. Die Erkenntnisse dieses Kapitels werden abschließend zusammengefasst und mit der weiteren Vorgehensweise des darauffolgenden Kapitels verknüpft.

5.1 Signifikanz trotz Eigenheiten

JS wurde ursprünglich von Brendan Eich im Unternehmen Netscape für den Browser Navigator 2.0 entwickelt und später von Microsoft eigenständig unter dem Namen „JScript“ auch im Internet Explorer 3.0 implementiert. Zur Standardisierung der Sprache wurde bei der Normungsorganisation Ecma International, die Spezifikation ECMAScript entwickelt und im Jahre 1997 erstmalig veröffentlicht. Die ursprüngliche Ausrichtung von ECMAScript war eine Skriptsprache für das Web zu definieren, die sowohl von professionellen Programmierern als auch von Laien benutzt werden kann. Das Web sollte interaktiv werden und der Browser, als Hostplattform, die nötige Umgebung dafür bereitstellen (vgl. [ECMA International 2011], S. vii und S. 2). Komplexe JS-Web-Applikationen waren folglich nie vorgesehen. Trotz der ungewöhnlichen Herkunft ist JS heute eine der wichtigsten Programmiersprachen und dies ist sicherlich den damals getroffenen konzeptionellen Entscheidungen zu verdanken.

Die Dynamik von JS erlaubt es in den verschiedensten Weisen zu programmieren. Auch traditionelle Strukturen aus klassenbasierten Sprachen sind möglich, wenn auch unvorteilhaft. Dies ist insbesondere deswegen beachtenswert, da die prototypische Natur von JS eigentlich komplett ohne Klassen auskommt. In JS ist (fast) alles ein Objekt, selbst Funktionen sind Objekte. Es gibt nur fünf Primitive: „number“, „string“, „boolean“, „null“ und „undefined“ (vgl. [Stefanov 2010], S. 1 und S. 3 und [ECMA International 2011], S. 2). Die Spezialstellung von Objekten inklusive die prototypische Vererbung mag ungewöhnlich erscheinen, der Ansatz ist jedoch „lower-level“ im Vergleich zu klassenorientierte Sprachen. Eben dieser Ansatz ist es auch, der JS zu so einer eleganten und dynamischen Sprache macht (vgl. [Fogus 2013], S. 32).

5.1.1 Objekte ohne Klasse

Ein Objekt in JS ist nichts weiteres als ein Bündel von benannten, veränderbaren Eigenschaften („properties“) und muss nicht erst von einen Rezept gebaut, bzw. von einer Klasse instanziert werden. Ein Objekt ist nur eine Sammlung von Schlüssel-Wert Paaren („key-value pairs“). Diese Leichtgewichtigkeit kann jedenfalls als Vorteil angesehen werden. Sie prägte beispielsweise die Entstehung von JSON als Datenaustausch-Format, das ebenfalls valides JS ist. Der einzige syntaktische Unterschied zwischen JS und JSON ist, dass die Namen der Eigenschaften bei JSON unbedingt in Anführungszeichen eingekapselt werden müssen, was ist bei JS ansonsten optional ist (vgl. [Stefanov 2010], S. 3f. und S. 49).

Vererbung wird in JS durch die „prototype“-Eigenschaft ausgedrückt, die jedes Objekt automatisch besitzt. Der Prototyp eines Objekts „a“ kann beispielsweise ein Objekt „b“ sein. Das Objekt „a“ erbt folglich die Eigenschaften des Objekts „b“ in seinem Prototypen und „a“ kann die Eigenschaften von „b“ als seine Eigenen verwenden. Auch Funktionen sind Objekte, mit dem Unterschied, dass sie ausführbar sind. Falls die Funktion eine Eigenschaft eines Objekts ist, wird sie als eine Methode des Objekts bezeichnet (vgl. [Stefanov 2010], S. 3f. und S. 41).

Abbildung 5.1 zeigt den Quellcode zur Erstellung eines simplen Objekts, mittels des Objekt-Literals „{}“. Es besitzt zwei Eigenschaften: die „number“ namens „callCount“ und die Methode „print“. Die Methode „print“ erhöht den „callCount“ und gibt einen String zurück.

```
1 // An object in JavaScript is not more than a list of key-values pairs
2 var myObject = {
3   callCount: 0,
4   print: function() { // increases callCount and returns info string
5     this.callCount += 1;
6     return "print was called " + this.callCount + " times.";
7   }
8 };
```

Abbildung 5.1: Erstellung eines simplen Objekts in JavaScript

Dieses Skript wird nun in der JS-Konsole von Google Chrome ausgeführt. Abbildung 5.2 zeigt die Ergebnisse dazu.

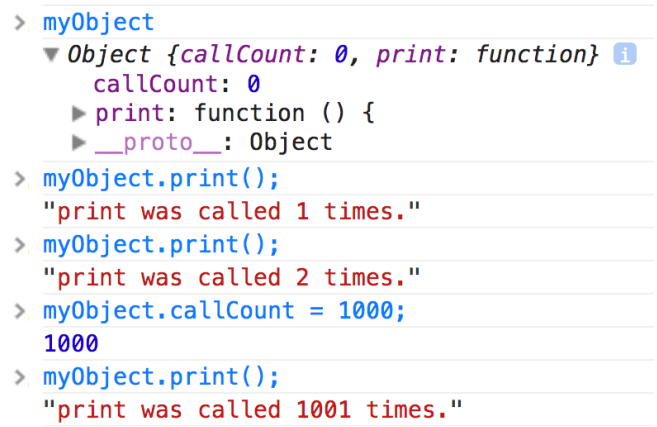

Abbildung 5.2: Ausführung und Manipulation des simplen JS-Objekts in der Google Chrome Konsole

In der Abbildung wird nun ersichtlich, dass das Objekt neben den beiden Eigenschaften noch die zusätzliche Eigenschaft „__proto__“ besitzt. Dies ist der Prototyp unseres neu erstellten Objekts, der ebenfalls ein Objekt ist. Außerdem wird hier demonstriert, dass alle sichtbaren Eigenschaften eines Objekts auch jederzeit beschreibbar sind. Dies wird häufig als eines der großen Probleme in JS angesehen, da angenommen wird, dass keine privaten Eigenschaften in JS möglich sind. Ein Lösungsansatz hierzu wird aber im nächsten Kapitel, im Abschnitt „Patterns“ auf Seite 115, vorgestellt.

5.1.2 Sprachdesign

Das Wort „Java“ in JS ist eher irreführend. Bis auf die Ähnlichkeit einiger Schlüsselwörter in der Syntax, hat JS nicht viele Gemeinsamkeiten mit Java. JS ist zudem keine Sprache mit starker Typisierung („strong typing“) wie Java, sondern ist schwach typisiert („loose typing“). Es ist eine funktionale Sprache und orientiert sich stark an den Sprachen Lisp und Scheme (vgl. [Crockford 2008], S. 3). Diese Umstände wurden für eine lange Zeit eher negativ aufgenommen, da sich die Sprache weit anders verhält, als es die Syntax vermuten

lässt. Mittlerweile wurden die Vorteile von immer schon da gewesenen JS Features, wie anonyme Funktionen und Funktionsabschlüsse („Closures“) aber erkannt und werden sogar in den Sprachraum von Java und PHP aufgenommen (vgl. [Stefanov 2010], S. 1).

Der Reiz des Sprachdesigns von JS geht von der Idee der sogenannten funktionalen Programmierung („functional programming“) aus. Funktionale Programmierung verfolgt einen grundlegend anderen Ansatz und unterscheidet sich von der klassischen objektorientierten Programmierung. Sie hat das Ziel, werttransformierende Funktionen in ihre abstrakten Einzelteile zu brechen, um ein System – und damit die Funktionalität der Applikation – zu schaffen. Nicht die Vererbung zwischen Klassen und deren Instanzen (kurzum keine Nomen) beschreiben ein System, sondern die Funktionen (die Verben) des Systems an sich. Dies deckt sich noch immer mit den Überlegungen zu wiederverwendbaren Softwaredesign der „Gang of Four“. Gewisse Patterns, die in starktypisierten Sprachen wie C++ und Java nötig sind, sind in JS jedoch aufgrund der Dynamik und des funktionalen Stil nicht mehr nötig. In JS können somit sowohl die Vorteile von objektorientierter als auch von funktionaler Programmierung genutzt werden (vgl. [Fogus 2013], S. 4 und S. 7 und [Stefanov 2010], S. 2).

Nichtsdestotrotz gibt es einen Grund, warum JS für die Entwicklung von komplexeren Applikationen für lange Zeit nicht in Erwägung gezogen wurde. Neben der ungewöhnlichen prototypischen Natur und der schwachen Typisierung von JS, die bei richtiger Anwendung allerdings enorme Vorteile bringen können, gibt es auch wirklich schlechte und unzuverlässige Teile in der Sprache. Unangefochten an erster Stelle muss die Notwendigkeit von globalen Variablen in JS erwähnt werden. Dies ist bei komplexen Applikationen ein absolutes No-Go (vgl. [Crockford 2008], S. 3). Ohne klare Namensräume sind Kollisionen vorprogrammiert (vgl. [Stefanov 2010], S. 11).

Spätestens seit dem Buch „JavaScript: The Good Parts“ von Crockford hat sich die Sichtweise auf JS jedoch grundlegend verändert. JS bietet nämlich

sehr wohl Wege zur Minimierung von globalen Variablen. Mehr noch, durch die geschickte Anwendung von Funktionsabschlüssen können auch private Objekte und öffentliche Schnittstellen verwirklicht werden. Aus diesem Grund sind Patterns – empfohlene und etablierte Codestrukturen zur Bewältigung von bestimmten Architekturproblemen – in JS besonders relevant (vgl. [Stefanov 2010], S. 1 und S. 11). Die restlichen unzuverlässigen Teile der Sprache können tatsächlich vermieden werden. Crockford definiert hierzu ein kleineres aber zuverlässiges Subset von JS. Dies wird im Rahmen dieser Diplomarbeit eingesetzt werden (vgl. [Crockford 2008], S. 3f.).

5.1.3 Alternativen

Es bleibt die Frage, ob der Aufwand JS zu verstehen und effektiv zu verwenden gerechtfertigt ist, bzw. ob nicht eine alternative Programmiersprache der bessere Weg zur Implementierung von (Web-)Applikationen wäre. Die Sichtweise von Crockford ist hier sehr eindeutig: Es gibt keine Alternative! JS Apps haben sich gegenüber Java Applets in der Vergangenheit klar durchgesetzt und heutzutage findet sich JS in jedem Browser (vgl. [Crockford 2008], S. 4). JS ist nun die einzige Sprache, die unabhängig vom Gerät und Hersteller – Smartphone, Tablet, Konsole oder PC – im Browser funktioniert.

Die neuen Standards in HTML5 und CSS3 bieten in Kombination mit JS schon heute die Möglichkeit komplexe interaktive Applikationen zu erstellen. Folglich dürfte auch die proprietäre Flash-Technologie von Adobe, langfristig aus dem Browser verschwinden. Diese düstere Zukunftsvision für Flash wurde in den letzten Jahren noch angezweifelt. Spätestens seit dem Ende der Weiterentwicklung von Flash für mobile Browser und der Neufokussierung von Flash, als Plattform für 3D-beschleunigte Spiele am Desktop und auf Konsolen, dürften aber die Weichen im Sinne von JS bereits gestellt sein (vgl. [Adobe Systems Incorporated 2011]).

5.2 Verbreitung und Popularität

Die Popularität von JS hat in den letzten Jahren kontinuierlich zugenommen. Der TIOBE Index versucht diese Popularität monatlich zu quantifizieren (primär über die Analyse der Suchanfragen im Web). Im Oktober 2013 befand sich JS dabei auf dem zehnten Platz (hinter C, Java, Objective-C, C++, PHP, C#, Basic, Python und Transact-SQL) (vgl. [Tiobe Software 2013]). Die Plattform Github wird heute häufig zur Versionierung und Kollaboration in Software-Projekten eingesetzt und eignet sich daher ebenfalls sehr gut, um die tatsächliche Einsatzhäufigkeit einer Programmiersprache zur Entwicklung von neuen Projekten in Zahlen auszudrücken. Die Analyse der erstellen Projekte auf Github zwischen Jänner und August 2013 bescheinigt JS dabei eine Anzahl von 264.131 Projekten, was JS in diesem Zeitraum klar auf den ersten Platz aller eingesetzten Programmiersprachen heben würde (vgl. [Bard 2013]).

Die starke Diskrepanz zwischen den erhobenen Rängen bzw. Plätzen lässt sich wohl nur durch die unterschiedliche Erhebungsmethodik erklären. Es gibt eben keinen vollkommenen Weg, um die Popularität einer Programmiersprache zu messen. Als die einzige Programmiersprache, die von den meisten Browsern unterstützt wird, dürfte JS aber jedenfalls in den Top-10 vertreten sein (vgl. [Johnson 2013]).

Tabelle 5.1 auf der nächsten Seite zeigt die Ergebnisse der Popularitätsanalyse klar aufgelistet im Vergleich zu den anderen Sprachen. Der Rang im TIOBE-Index ist in der mittleren Spalten (grüner Hintergrund) aufgeführt, in den rechten Spalten wird die Anzahl an Github-Projekten aufgelistet und diese Anzahl bestimmt den Platz (in der letzen Spalte). Reihen innerhalb der Tabelle, die grau eingefärbt und mit „kA“ (keine Angabe) gekennzeichnet wurden, sind leider nicht aus den Github Analysen erhebbar. Wie ersichtlich wird, ist JS im Vergleich zu ActionScript in beiden Analysen (TIOBE-Index 49 bzw. Github-Platz 25) weiter vorne platziert. Im Gegensatz zur (in der Github-Analyse) zweitplatzierten Programmiersprache Ruby, wurden im analysierten Zeitraum rund 45.000 mehr Projekte in JS erstellt.

	TIOBE Index (Oktober 2013)	Github Repositories (Jänner bis August 2013)	
Programmiersprache	**Rang im Index**	**Anzahl erstellte Projekte**	**Platz nach Anzahl**
C	1	67.706	7
Java	2	157.618	3
Objective-C	3	36.344	8
C++	4	78.327	6
PHP	5	114.384	4
C#	6	32.170	9
(Visual) Basic	7	1.854	30
Python	8	95.002	5
Transact-SQL	9	kA	kA
JavaScript	**10**	**264.131**	**1**
Visual Basic .NET	11	kA	kA
Perl	12	15.412	12
Ruby	13	218.812	2
Pascal	14	kA	kA
PL/SQL	15	kA	kA
Lisp	16	kA	kA
Delphi / Object Pascal	17	901	39
Groovy	18	3.372	22
MATLAB	19	2.395	26
COBOL	20	kA	kA
Bash / Shell	27	28.561	10
ActionScript / Adobe Flash	49	2.413	25

Tabelle 5.1: Popularität von Programmiersprachen nach TIOBE Index vom Oktober 2013 und erstellen Projekten auf Github zwischen Jänner und August 2013 (vgl. [Tiobe Software 2013] und [Bard 2013])

Wie ersichtlich wird, befindet sich JS im TIOBE-Index nur auf Rang zehn und muss sich vorrangig älteren Programmiersprachen, wie C, Java, Objective-C und C++, geschlagen geben. Diese Programmiersprachen dürften auch abseits von Open-Source-Projekten (auf Github) sehr häufig eingesetzt werden, was die Diskrepanz zwischen den Rängen des TIOBE-Index und der Github-Analyse erklären würde. Äußerst interessant ist zudem der Rang fünf von PHP im TIOBE Index. JS dürfte sich durch die Plattform Node.js in naher Zukunft vor allem von dieser Programmiersprache Popularität abzweigen.

Die Analyse der Popularität ist bei Programmiersprachen relevant, da eine hohe Popularität in der Regel auf eine florierende Community umgelegt werden kann, die sich an Problemlösungen und der Weiterentwicklung beteiligt. Bei JS dürfte das Bedürfnis nach zusätzlichen Bibliotheken nochmals höher als bei anderen Programmiersprachen sein. Dies liegt daran, dass JS in seiner Basis nur wenige spezialisierte Funktionen für alltägliche Probleme bietet. Beispielsweise gibt es in JS keine Standardimplementierung für Events und auch keinen Support zur Internationalisierung von „Date“-Objekten. In JS zu Pro-

grammieren, ohne zusätzliche Bibliotheken zu verwenden, führt daher schnell dazu Arbeit zu machen, die bereits getan wurde.

Fünfzehn von den zwanzig aktuell populärsten Open-Source-Projekten auf Github stehen übrigens direkt in Verbindung mit JS. Darunter befinden sich äußerst bekannte Namen wie jQuery, Node.js, Bootstrap und D3 (vgl. [Nanni 2013]).

5.3 Neue Einsatzgebiete

Neben dem ursprünglichen Haupteinsatzgebiet von JS, als clientseitige Skriptsprache, die vom Browser ausgeführt wird, hat sich die Anzahl an zusätzlichen Einsatzmöglichkeiten in den letzten Jahren massiv erhöht. JS wird nun auch serverseitig (Node.js und .NET), in Desktop-Applikationen und mobilen Applikationen, in Erweiterungen und zur Programmierung von Kommandozeilen-Tools eingesetzt (vgl. [Stefanov 2010], S. 1).

In den nachfolgenden beiden Unterabschnitten werden Node.js und Node-webkit vorgestellt und die Vorteile dieser Technologien erläutert. Abschließend erfolgt im Unterabschnitt „Vertiefung der Zusammenhänge“ auf Seite 73 eine Analyse, wie sich diese Technologien auf die Laufzeitumgebung von JS auswirken.

5.3.1 Node.js

Node.js basiert auf der Open-Source JS-Engine V8 von Google, die in den Browsern Chromium und Chrome eingesetzt wird. Node.js ist sowohl Laufzeitumgebung als auch Bibliothek, die die Ausführung von JS außerhalb des Browsers ermöglicht. Die Plattform hat das Ziel, die Entwicklung von schnellen und skalierbaren Netzwerk-Applikationen zu vereinfachen (vgl. [Joyent Inc. 2013a]). Sie bietet dadurch den Vorteil, dass JS ganzheitlich – sowohl am Frontend als auch am Backend – bei Webapplikationen eingesetzt werden

kann. Die Anzahl an zusätzlich benötigten Programmiersprachen innerhalb eines großen Projekts minimiert sich, was der Wartbarkeit sicherlich zu Gute kommt. Die zahlreichen Vorteile von Node.js scheinen mittlerweile auch viele Großunternehmen erkannt zu haben: Yahoo!, Ebay, Microsoft und LinkedIn nutzen Node.js produktiv für ihre geschäftskritischen Anwendungen (vgl. [Joyent Inc. 2013b]).

Ein weiterer Vorteil von Node.js ergibt sich durch die JS-Engine V8. Sie erschafft nicht nur die Laufzeitumgebung für JS, sondern kann auch als Bibliothek verwendet werden. Dies ermöglicht es, nativ ausgeführte Addons in C und C++ zu schreiben, die mit der JS-Laufzeitumgebung und vice versa kommunizieren können. Der C bzw. C++ Quellcode wird hierzu mit dem Node.js Kommandozeilentool „node-gyp“ kompiliert. Vorab ist es wichtig, die Build-Konfiguration innerhalb einer separaten Datei „binding.gyp“, in einem JSON-ähnlichen Format zu beschreiben (vgl. [Joyent Inc. 2013c]). Daneben muss eine Schnittstelle via C++ definiert werden, die spezielle Funktionen der V8-Bibliothek verwendet und als „Glue-Code“ zwischen JS und dem nativen Code fungiert (vgl. [Farrell 2013]). Häufige Praxis ist es beispielsweise ältere Bibliotheken, die in C bzw. C++ geschrieben sind, auf diesem Wege für Node.js verfügbar zu machen (vgl. [Rajlich 2012]).

Node.js kommt mit seinem eigenen Paketmanager NPM („Node.js Package Manager“). Dies ist ein Kommandozeilen-Tool, das die Installation von Paketen und das automatische Auflösen der Abhängigkeiten zu anderen Paketen ermöglicht. Pakete müssen beim zentralen Repository npmjs.org registriert werden, um öffentlich zur Verfügung zu stehen. Die einzige Voraussetzung zur Erstellung eines derartigen Pakets ist das Vorhandensein einer package.js Datei im Projekt-Wurzelverzeichnis. Diese muss dem JSON-Format folgen und dient der Definition des Projekts (Name, Beschreibung und Version) und zur Auflistung der Abhängigkeiten zu anderen Paketen (vgl. [Reed 2011]).

5.3.2 Node-webkit

Node-webkit kombiniert Node.js mit dem Browser Chromium. Hierdurch wird die Entwicklung von nativen Desktop-Applikationen für Windows, Mac und Linux mit den Webtechnologien HTML, CSS und JS möglich (vgl. [Wang 2013a]). Das heißt: Es können Applikationen mit JS entwickelt werden, die via Node.js Zugriff auf die nativen Betriebssystemfunktionen haben und gleichzeitig die bereits etablierten Technologien von Webapplikationen nutzen. Ein weiteres großes Manko von modernen Webapplikationen kann mit Node-webkit umgangen werden: Der Support der Applikation für unterschiedliche Browser-Umgebungen muss nicht mehr berücksichtigt werden.

5.3.2.1 Erstellung eines Prototypen

Um das Potential von Node-webkit zu evaluieren wurde ein Prototyp erstellt. Ein Hauptziel dieses Prototypen ist es zu testen, ob die nativen Addons von Node.js ebenfalls in Node-webkit funktionieren. Um dies zu beweisen, wurde eine Applikation entworfen, die mit dem „Roboterball“ Sphero kommuniziert. Die Beschleunigungssensoren des Balles werden ausgelesen und beeinflussen bei Veränderungen die Farbe des Balles. Zudem wird über die Motoren des Balles, ein „schlagendes Herz“ simuliert. Dieser Herzschlag wird mit einer Partikel-Visualisierung als zusätzliches Feedback synchronisiert.

Abbildung 5.3 auf der nächsten Seite zeigt einen Ausschnitt des Prototyps. Im Ruhezustand sind die grünen Partikel klein und fast durchsichtig, die Intensität der Farbe des Balles ist reduziert. Beim simulierten Herzschlag vergrößern und erhellen sich die Partikel, die Motoren des Balles schlagen kurz an und die Farbe des Balles maximiert sich. Der Ausschnitt ist allerdings nur eine eingeschränkte Sichtweise auf die Applikation, da die Wirkung der Motoren ebenso die Beschleunigungssensoren beeinflusst. Darauf wird im Rahmen dieser Diplomarbeit allerdings nicht weiter eingegangen.

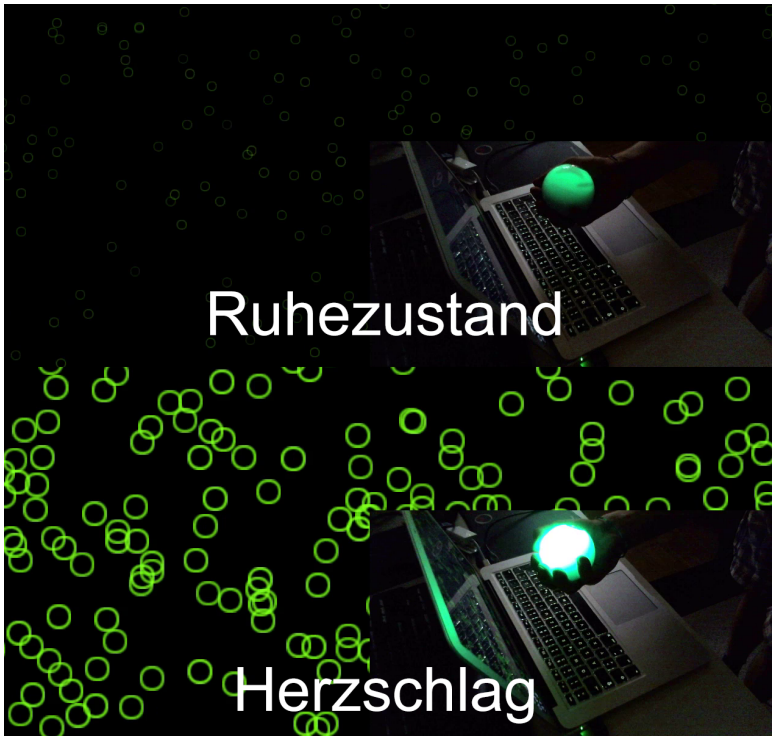

Abbildung 5.3: Prototyp in Node-webkit kommuniziert via nativen Addon in „node-sphero“ mit dem „Roboterball“ Sphero

Zur Programmierung dieses Prototypen wurde ausschließlich JS verwendet. Die Visualisierung wird mittels der Bibliothek cocos2d-html5 erstellt und kontrolliert. Sphero muss sich über Bluetooth direkt mit dem Betriebssystem verbinden. Daher wird die Node.js Bibliothek node-sphero eingesetzt. Diese ermöglicht – über ein natives Addon – die direkte Kommunikation mit der Bluetooth-Schnittstelle und stellt Methoden zum Auslesen und Senden von Parametern, vom bzw. zum Ball, zur Verfügung (vgl. [Thompson 2013]). Um native Addons in Node-webkit verwenden zu können, müssen diese zuvor mit dem Kommandozeilen-Tool „nw-gyp“ auf der tatsächlichen Plattform (jeweils einmalig unter Windows, Mac und Linux) kompiliert werden. Danach können diese im Script „required“ werden, sie sind danach jedoch nicht mehr mit der reinen Node.js Plattform kompatibel (vgl. [Wang 2013b]).

5.3.2.2 Fazit

Node-webkit vervollständigt die Vision, dass JS und moderne Webtechnologien im Allgemeinen, als vollwertiger Flash-Ersatz geeignet sein könnten. Ähnlich wie die nativen Erweiterungen in Adobe AIR („AIR Native Extensions“), kann auch Node-webkit derartige native Funktionalität durch Node.js Addons einbinden. Diese müssen jedoch auch hier pro Plattform kompiliert werden, um einsatzfähig zu sein (vgl. [Adobe Systems Incorporated 2013]).

Auch grafikintensive Desktop-Applikationen werden durch Hardwarebeschleunigung mit der „Web Graphics Library“ (WebGL) mit Node-webkit möglich. Für mobile Plattformen gibt es hierzu leider noch keine Lösung. Der bekannte Wrapper PhoneGap ermöglicht zwar das Verpacken, die Performance des Canvas-Elements, das für immersive Applikationen besonders wichtig ist, ist jedoch limitiert. Speziallösungen wie CocoonJS und Ejecta nehmen sich diesen Problem aktuell an, sind jedoch leider noch nicht für alle mobilen Plattformen verfügbar (vgl. [Szablewski 2013] und [Ludei Inc. 2013]).

5.3.3 Vertiefung der Zusammenhänge

Um die Unterschiede zwischen den zuvor vorgestellten unterschiedlichen Technologien zu verdeutlichen, wird an dieser Stelle ein Skript vorgestellt, das das Vorhandensein der Objekte „navigator“ und „process“ im globalen Namensraum überprüft und Informationen zu diesen am Bildschirm ausgibt. Es soll damit aufgezeigt werden, dass jede Plattform eine andere Laufzeitumgebung zur Verfügung stellt. Dieses Skript kann daher sowohl im Browser (Google Chrome), unter Node.js (im Terminal) als auch in Node-webkit ausgeführt werden.

```
1  (function() {
2
3    // Shortcut to log output
4    function log(text) {
5      console.log(text);
6    }
7
8    // global object "navigator" is only available in browsers
9    if (typeof navigator === "object") {
10     log("browser detected.");
11     log("navigator.userAgent: " + navigator.userAgent);
12   } else {
13     log("browser NOT detected.");
14   }
15
16   // global object "process" is only available in node.js
17   if (typeof process === "object") {
18     log("node detected.");
19     log("process.title: " + process.title);
20     log("process.version: " + process.version);
21     log("process.arch: " + process.arch);
22     log("process.platform: " + process.platform);
23   } else {
24     log("node NOT detected.");
25   }
26
27 }()); // immediate function (self-executing)
```

Abbildung 5.4: Code: Plattformbestimmung. Ausgabe von Informationen zur Bestimmung der aktuellen JS Umgebung

Abbildung 5.4 zeigt das angesprochene Skript im Detail. Zwischen Zeile vier und sechs erfolgt die Definition einer Hilfefunktion, um die Informationen durch den kürzeren Aufruf „log()“ direkt an die Ausgabe zu leiten. Danach erfolgen zwei Bedingungsblöcke, wo darauf geprüft wird, ob der Typ der Variablen „navigator“ und „process“, dem String „object“ entspricht. Trifft dies zu, werden Informationen zu dem jeweiligen Objekt ausgegeben, ansonsten ist dieses Objekt im globalen Kontext nicht verfügbar. Wenn „navigator“ verfügbar ist, handelt es sich um einen Browser und der „userAgent“ kann ausgegeben werden. Wenn „process“ verfügbar ist, können Informationen zu Node.js ausgegeben werden.

Um den Codeblock (Zeile eins und 27) befindet sich übrigens eine selbstausführende Funktionsexpression. Das ist ein sehr bekanntes Pattern in JS und wird im Unterabschnitt „Selbstausführende Funktionen“ auf Seite 116 noch genauer behandelt. Zu diesem Zeitpunkt ist es nur wichtig zu wissen, dass diese Funktion nachdem sie interpretiert wurde, sofort ausgeführt wird.

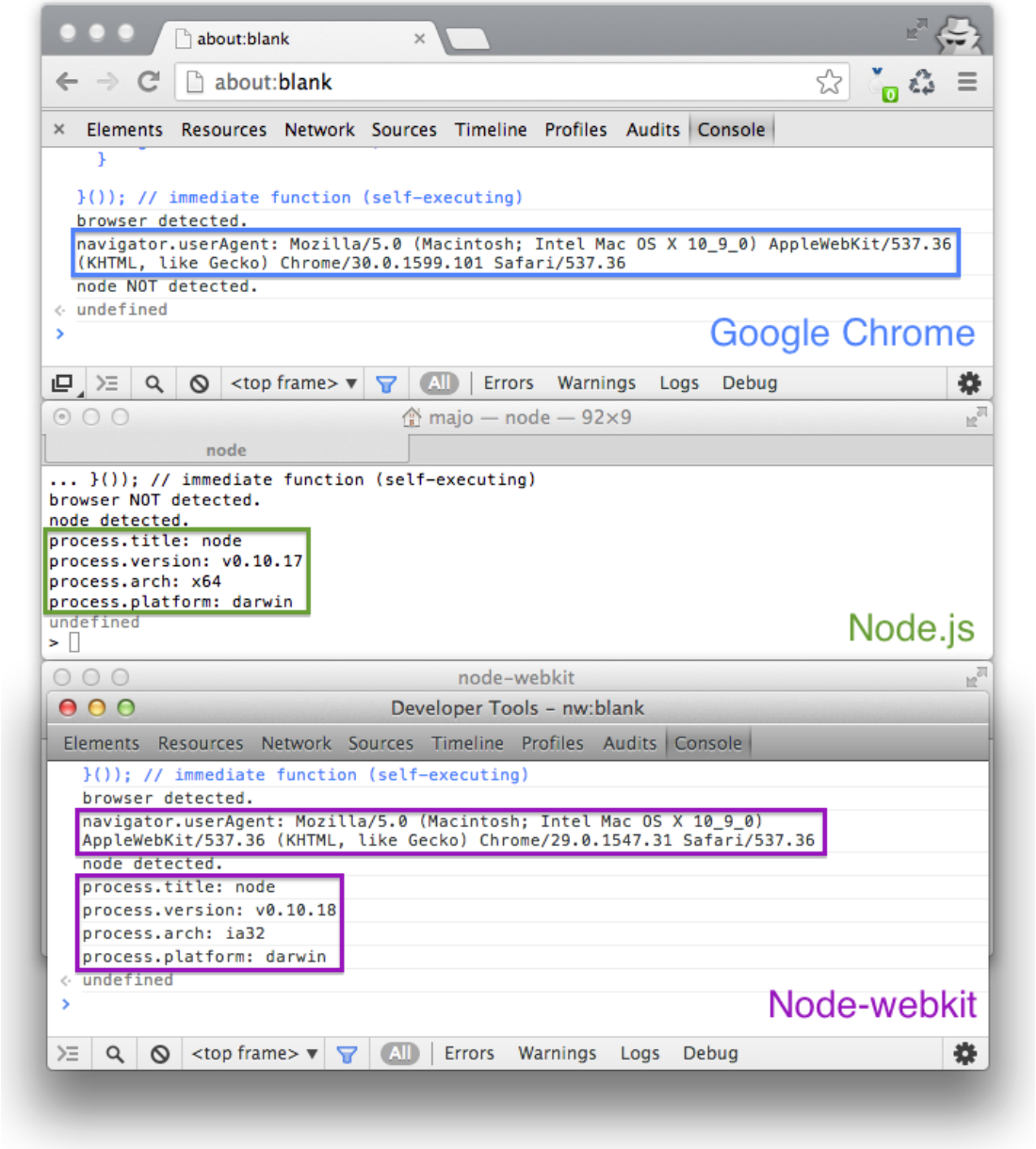

Abbildung 5.5: Ergebnisse des Skripts Plattformbestimmung bei Ausführung unter Google Chrome, Node.js und Node-webkit

Abbildung 5.5 zeigt die Ergebnisse der Ausführung unter Google Chrome, Node.js und Node-webkit. Bei Betrachtung der Ausgabe von Node-webkit wird die Kombination der beiden Ausführungsumgebungen nun klar ersichtlich. In den anderen beiden Umgebungen ist entweder nur „navigator“ oder „process“ verfügbar. Außerdem lässt sich so einfach beweisen, dass Node-webkit eine eigene Node.js Laufzeitumgebung verwendet und nicht die globale Instal-

lation des Betriebssystems verwendet. Sowohl die verwendete Version („process.version“) als auch die Architektur („process.arch“) unterscheiden sich. Die Applikation ist folglich in sich geschlossen und benötigt keine externen Abhängigkeiten im System.

5.4 Weiterentwicklung und Kompatibilität

JS basiert auf den Standard ECMAScript, der aktuell in der fünften Version (5.1) zur Verfügung steht (vgl. [ECMA International 2011], S. 1). Mittlerweile implementieren alle modernen Browser JS in dieser Version: Beispielsweise Microsoft Internet Explorer ab Version zehn und Mozilla Firefox ab Version vier (vgl. [Zaytsev 2013]).

Die nächste Iteration von ECMAScript ist die Version sechs (Codename Harmony), die bereits in mehreren „Drafts“ vorliegt. Die Unterstützung in aktuellen Browsern (beispielsweise Google Chrome) ist übrigens schon eingeschränkt möglich (vgl. [Allardice 2013]). Bis zur finalen Spezifikation und der Freigabe der Implementierungen der Browser-Hersteller, sollten die kommenden Features von JS aber noch als experimentell angesehen werden. Einen hervorragenden Überblick über die Kompatibilität verschiedener Browsern, zu bestimmten HTML5-, CSS- und JS-Features, bietet die Webseite caniuse.com.

Ein interessanter Ansatz sind zudem sogenannte Polyfills, JS-Bibliotheken, die zum Zwecke der Herstellung von Kompatibilität für gewisse Web-Features eingesetzt werden können. Diese werden im darauffolgenden Unterabschnitt vorgestellt.

Aufgrund der hohen Bedeutung von JS als Web-Technologie, gibt es mittlerweile auch Programmiersprachen, die JS-Code bei der Kompilierung produzieren. Dieser Ansatz wird im Unterabschnitt „JavaScript als Compilerziel“ auf der nächsten Seite vorgestellt.

5.4.1 Kompatibilität durch Polyfills

Kompatibilität zu neuen Web- bzw. JS-Standards kann häufig auch in älteren Browsern, durch den Einsatz sogenannter Polyfills, erreicht werden. Ein Polyfill ist definiert als *„ein Zwischenstück („shim"), das eine zukünftig verfügbare Schnittstelle nachahmt und einen Fallback-Modus für ältere Browser zur Verfügung stellt"*[1] ([Sharp 2010]).

Ein großer Vorteil durch Polyfills ist, dass einerseits auch mit älteren Browsern mit neuen Standards gearbeitet werden kann und andererseits ein nahtloser Übergang auf die nativen Implementierungen von zukünftigen Funktionen erfolgen kann. Beispielsweise gibt es bereits jetzt Polyfills für bestimmte Funktionen des kommenden ECMAScript6 Standards. Ein besonders häufiger Anwendungsfall dürfte aber aktuell die Herstellung von HTML5 Features (zum Beispiel Canvas-, Video- und Audio-Elemente) in älteren Browsern sein (vgl. [Irish 2013]).

5.4.2 JavaScript als Compilerziel

JS hat sich als Sprache des Webs durchgesetzt und wird von allen Browsern verstanden. Eine alternative Sprache hätte folglich den entscheidenden Nachteil, dass erst eine Unterstützung auf den Plattformen ermöglicht werden muss. Die Programmiersprachen CoffeeScript, TypeScript und Dart verfolgen daher den Ansatz, JS als Compilerziel zu verwenden. Somit kann der eigentliche Quellcode in der jeweiligen Sprache erstellt werden. Damit er ausgeführt werden kann, wird dieser von speziellen Tools in JS-Code umgewandelt.

CoffeeScript ist die älteste dieser alternativen Sprachen und kann auch als einzige Sprache als stabil für den produktiven Einsatz (Version 1.6.3) angesehen werden. Sie wurde besonders von Python und Ruby geprägt (vgl. [Ashkenas 2013]). In Diskussionen innerhalb der Community wird CoffeeScript häufig ein schlechtes Sprachdesign zur Definition des Gültigkeitsbereichs („scope")

[1] Übersetzung durch den Autor

von Variablen vorgeworfen. Das Entwicklungsteam musste hier anscheinend Kompromisse eingehen (vgl. [Donat 2013]).

Hinter der Entwicklung von TypeScript steht Microsoft und Anders Hejlsberg, der als Schöpfer von Turbo Pascal und C# gilt. Die Sprache basiert hauptsächlich auf den kommenden ECMAScript6-Standard, führt daneben jedoch auch eine (optionale) starke Typisierung von Variablen und eine erweitertere Syntax zur Strukturierung des Quellcodes ein. Gängiges JS ist also bereits valider TypeScript-Code, im Gegensatz zu CoffeeScript und Dart. Das Hauptziel der Sprache ist es, den Entwicklungsprozess durch die Features von TypeScript zu unterstützen. Vor allem die statische Typisierung soll die Fehleranfälligkeit reduzieren, da Aufrufe von falschen Typen bereits beim Kompilieren bzw. innerhalb eines „Integrated Development Environments“ (IDE) entdeckt werden können (vgl. [Golem.de 2013]). Die Sprache steht derzeit (Anfang November 2013) kurz vor der Veröffentlichung der ersten stabilen Version 1.0 (vgl. [Kalenda 2013]).

Dart von Google kompiliert im Gegensatz zu CoffeeScript und TypeScript nicht nur zu JS, sondern verfügt auch über eine eigene virtuelle Maschine (VM) zum Ausführen des Quellcodes. Die Syntax der Sprache ist signifikant anders als JS und das Sprachdesign ist klar auf die Prinzipien der klassischen Objektorientierung ausgelegt (vgl. [Heise 2012a). Sie steht aktuell ebenfalls (Anfang November 2013) kurz vor der Veröffentlichung der ersten stabilen Version 1.0 (vgl. [Heise 2013]).

Übrigens, diese drei Sprachen sind nur die Spitze des Eisbergs. Auch ClojureScript, Roy, Elm, Flapjax, Java und JS selbst, können dazu verwendet werden nach JS zu kompilieren (vgl. [Fogus 2013], S. 3). Für den praktischen Teil der Diplomarbeit wurde keine dieser alternativen Sprachen eingesetzt. Das Risiko für einen produktiven Einsatz erschien aufgrund der Neuartigkeit noch als zu hoch bzw. im Fall von CoffeeScript wird die Syntax der Sprache (subjektiv) als unattraktiv vom Autor angesehen. Es macht zudem Sinn, vorab JS in allen seinen Facetten kennenzulernen, bevor eine der Alternativen eingesetzt wird. Es

darf nämlich nicht vergessen werden, dass beim Debuggen ebenso nur JS zur Verfügung steht.

5.5 Zusammenfassung

JS ist seinen Wurzeln entwachsen und hat sich gegen Java und neuerdings auch Flash als die Programmiersprache des Webs durchgesetzt. Die Einsatzgebiete sind vielfältig und nicht mehr nur auf den Browser beschränkt. Node.js ermöglicht die serverseitige Programmierung mit JS und wird mittlerweile in zahlreichen Top-Unternehmen produktiv eingesetzt. Node-webkit könnte JS und andere Webtechnologien wie HTML und CSS den großen Sprung zu nativen Desktop-Apps ermöglichen. Die Community hinter JS ist riesig, Bibliotheken auf Github sind zu großer Zahl vorhanden und der Standard hinter JS, ECMAScript wird aktiv weiterentwickelt.

Die Kombination aus HTML, CSS und JS ist am Desktop auch zur Programmierung von immersiven Applikationen geeignet. Nichtsdestotrotz bringt JS auch Fallen und schlechte Eigenheiten mit. Die Unterschiede zu klassischen Sprachen sollten nicht unterschätzt werden. Zur Konzeption einer Applikation mit JS ist daher eine Analyse der Sprache, Patterns und der prinzipiellen Anforderungen (beispielsweise Modularisierung und Automatisierung) erforderlich. Dies wird im nächsten Kapitel vorgenommen.

6 Konzeption von JavaScript-Applikationen

JS ist eine Skriptsprache und zur Ausführung im Browser werden üblicherweise „script“-Tags im HTML-Dokument „head“ oder „body“ eingebunden. Es ist heute gängige Praxis einfach mehrere Skripts hintereinander so hinzuzufügen. Das mag für kleine Spielereien im Browser ausreichen, für komplexere Applikationen mit mehreren tausend Zeilen Quellcode wird eine angemessene Struktur der Applikation jedoch zur Pflicht. Eine „Try and Error“-Vorgangsweise reicht nicht mehr.

Um ansprechenden Code zu produzieren und die Stabilität der Applikation zu gewährleisten, müssen alle Register gezogen werden. Dies gilt besonders, wenn nicht alleine, sondern in einem Team gearbeitet wird. Überlegungen zur Auswahl der Entwicklungsumgebung, der einzusetzenden Code-Konventionen, Analyse-Tools und Testumgebungen müssen getroffen werden. Eine Modularisierung, Automatisierung und die Verwaltung von externen Abhängigkeiten werden zu einer Notwendigkeit für die erfolgreiche Entwicklung der Applikation. Ebenso müssen aber auch die Fallen der Programmiersprache und bewährte Lösungsschablonen (Patterns) immer im Hinterkopf behalten werden.

In den nachfolgenden Abschnitten werden nun diese verschiedenen Anforderungen bei der Entwicklung von JS-Applikationen analysiert und Lösungen – aus Sicht der Applikation Physiogame – präsentiert. Es wird dabei etwas weiter ausgeholt, damit die Lösungsansätze übertragbar sind und als Leitfaden für neue Projekte verwendet werden können. Alle vorgestellten Punkte haben ihre maßgebliche Relevanz in der Konzeptionsphase, also vor dem Start der

eigentlichen Implementierung eines Projekts. Abschließend wird eine Zusammenfassung der vorgestellten Lösungen gegeben.

6.1 Entwicklungsumgebung

Eine generelle Basis-Voraussetzung zum Programmieren ist ein simpler Texteditor, auch wenn die Grenzen damit wohl schnell erreicht sein dürften. Zur Entwicklung werden daher häufig IDEs eingesetzt, die weit mehr Features bieten. Im Rahmen der Entwicklung von Physiogame wurde das Programm Sublime Text 2 verwendet, ein kommerzielles Produkt, das besonders auf Geschwindigkeit und Erweiterbarkeit durch Plugins ausgelegt ist. Folgende Plugins werden verwendet:

- Das Plugin jsFormat zum Formatieren des JS-Quellcodes via Tastaturkürzel.
- JSHint zur Prüfung („linten“) des JS-Quellcodes auf Konformität zu gewissen Best Practices und zur Vermeidung der „bad parts“ von JS (Qualitätssicherung). JSHint und das Original JSLint werden im Abschnitt „Code-Analyse und Testumgebung“ auf Seite 86 noch genauer vorgestellt.
- SublimeSaveOnBuild um JSHint automatisch beim Speichern des Quellcodes auszuführen.

Sublime Text 2 ist in seiner Basis nicht vergleichbar mit großen IDEs wie Eclipse. Das Programm ist aber sowohl für Windows, Mac und Linux verfügbar und durch Plugins soweit erweiterbar, dass quasi eine eigene IDE von Grund aufgebaut werden kann (vgl. [Sublime HQ Pty Ltd. 2013]). Vorweg muss hier allerdings darauf aufmerksam gemacht werden, dass eine automatische Vervollständigung des Codes („Code-Completion“) oder „Refactoring“ – wohl zwei der wichtigsten Features einer IDE – durch JS extrem erschwert werden. Diese Features sind durch die dynamische Natur von JS nur schwer realisierbar und sehr von einer statischen Typisierung abhängig. Eine angemessene Unterstützung durch IDEs, ist somit nur mit den JS-Sprachalternativen TypeScript und

Dart wirklich möglich, die eine (optionale) statische Typisierung ermöglichen (vgl. [Rendle 2012]).

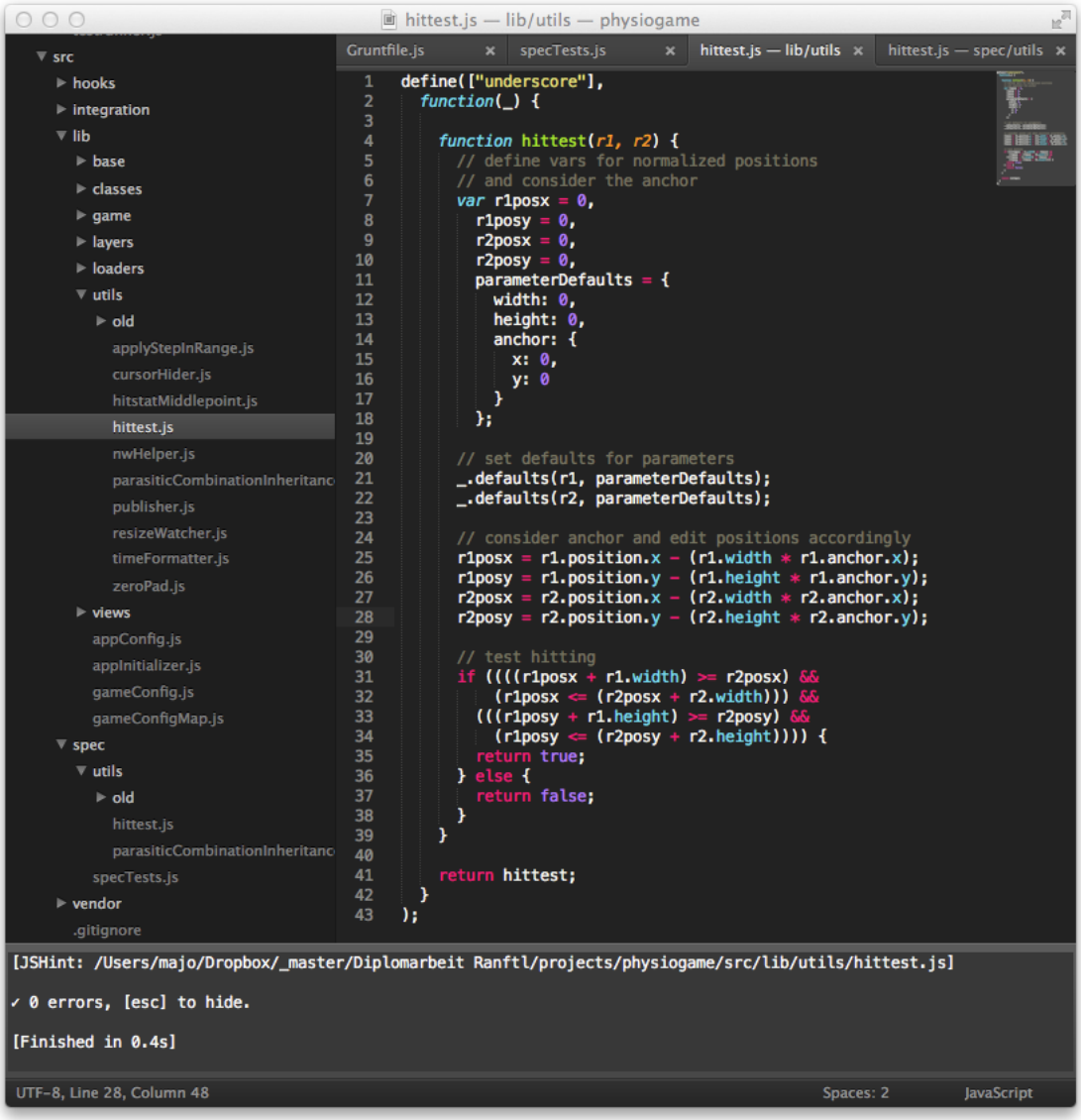

Abbildung 6.1: Sublime Text 2 nach einer Dateiänderung mit JSHint

Abbildung 6.1 zeigt eine Beispiel-JS-Datei aus Physiogame in Sublime Text 2, die gerade mit JSHint überprüft wurde. Wie hier ersichtlich wird, können Dateien in Reitern geöffnet werden, die Hervorhebung der Syntax wird unterstützt und die Ordnerstruktur kann eingeblendet werden. Zudem wird rechts, statt einer normalen Bildlaufleiste, eine „Minimap“ des Codes anzeigt, die bei der Navigation in längeren Dateien besonders hilfreich sein kann.

Es ist an dieser Stelle wichtig klarzustellen, dass die Auswahl eines Editors bzw. einer IDE immer von subjektiven Präferenzen geprägt ist, die sich von EntwicklerIn zu EntwicklerIn unterscheiden. Ähnlich verhält es sich bei der Auswahl des Version-Control-Systems. Bei Physiogame wird Git verwendet, hauptsächlich aufgrund des Vorteils, nicht von einem zentralen Repository abhängig zu sein. Alle Operationen (einchecken, zurücksetzen, usw.) finden dabei direkt im Terminal (Kommandozeilen-Tool) statt, könnten aber ebenso auch durch Plugins direkt in Sublime Text 2 erfolgen.

6.2 Management von Abhängigkeiten

Bei kleinen JS-Experimenten reicht es externe Bibliotheken direkt im Code bzw. via eines „Content Delivery Networks“ einzubinden (vgl. [Stefanov 2010], S. 197). Die Anzahl an fremder Funktionalität kann aber sehr schnell zunehmen und folglich geht der Überblick verloren. Es ist wichtig zu erkennen, dass beim Einbinden von zusätzlichen Bibliotheken, Abhängigkeiten („dependencies“) entstehen und diese müssen verwaltet werden. Dies ist insbesondere auch deswegen relevant, da deren Weiterentwicklung nie still steht und eine Kontrolle über die genaue Version bzw. des „Commit-Hashes“ der jeweiligen Abhängigkeit erforderlich ist, da frühere bzw. spätere Versionen derselben Bibliothek vielleicht nicht mehr reibungslos mit der eigenen Applikation zusammenarbeiten.

Um dieses Problem bei der Entwicklung von JS-Applikationen zu lösen, werden die Paketmanager NPM und Bower eingesetzt. Beide haben zwar das selbe Ziel, die Installation und Verwaltung von Paketen zu ermöglichen, unterscheiden sich jedoch in der Art der verfügbaren Pakete. Registrierte Pakete aus dem NPM-Repository sind vorrangig für die Ausführung in der Node.js Umgebung bestimmt, während Bower selbst ein Paket vom NPM-Repository ist und sich auf Verwaltung von clientseitigen Paketen für den Browser spezialisiert hat. Bower ist dabei allerdings nicht nur auf JS-Pakete reduziert, sondern

akzeptiert alle Pakete, die via Git ausgecheckt werden können (vgl. [Twitter Inc. 2013]).

NPM wird daher bei der Entwicklung von Physiogame vor allem für die Verwaltung von Paketen, die für den Entwicklungsprozess relevant sind (zum Beispiel im Build- und Testprozess) eingesetzt, während Bower, die (clientseitigen) Bibliotheken verwaltet, die in der eigentlichen Applikation eingesetzt werden. Das Format, um die benötigten Pakete zu beschreiben, ist für beide Paketmanager fast identisch. Sie müssen im JSON-Format vorliegen. Abbildung 6.2 zeigt die beiden Dateien „package.json“ (für NPM) und „bower.json“ (für Bower) im Vergleich.

```
// package.json
{
  "name": "physiogame",
  "version": "1.0.1",
  "dependencies": {
    "requirejs": "~2.1.9"
  },
  "devDependencies": {
    "grunt-cli": "~0.1.9"
  }
}

// bower.json
{
  "name": "physiogame",
  "version": "1.0.1",
  "dependencies": {
    "keymaster": "https://github.com/madrobby/keymaster.git#a2b06f556cc1f219251c55ea0cf0e5d7224987f8",
    "loglevel": "~0.5.0"
  }
}
```

Abbildung 6.2: Vergleich „package.json“ (NPM) und „bower.json“

Wie aus der Abbildung 6.2 ersichtlich wird, ist sind die Namen der Eigenschaften „name“, „version“ und „dependencies“ identisch. Auch Bower unterstützt übrigens ebenfalls „devDependencies“, Abhängigkeiten die nur für die Entwicklung, aber nicht für das eigentliche Ausführen der Applikation relevant sind. Die Unterschiede von Bower werden bei der Abhängigkeit „keymaster“ auffällig. Diese ist nicht im Repository registriert, da Bower aber Pakete von Git akzeptiert, kann diese direkt „ausgecheckt“ werden. Der „Commit-Hash“ nach „#“ gibt die Version dabei an.

Die Installation der Abhängigkeiten erfolgt übrigens in der Kommandozeile über „npm install -d“ bzw. „bower install -d“. Es gibt noch zahlreiche zusätzliche

Kommandos, die Beschreibung dieser würde den Rahmen dieser Diplomarbeit allerdings sprengen.

6.3 Code-Konventionen

Das Einhalten von Konventionen ist in JS alleine schon deswegen eine absolute Notwendigkeit, da die Syntax weit weniger Schlüsselwörter als andere Sprachen besitzt. Eine Variable in JS kann vieles sein, die reine Betrachtung des Schlüsselworts „var“ reicht nicht. Üblicherweise wird deren „Bedeutung“ daher durch die unterschiedliche Groß- bzw. Kleinschreibung ausgedrückt. Die nachfolgende Abbildung zeigt einen Auszug ausgesuchter Konventionen in JS, die auch während der Entwicklung von Physiogame angewendet wurden.

```
1  // constants are completely in CAPITAL LETTERS
2  var MY_CONSTANT = 1;
3
4  // objects and primitives start with a lower case letter
5  var myObject = {};
6
7  // functions start with a lower case letter too!
8  var myFunction = function() {
9    return true;
10 };
11
12 // constructors start with a CAPITAL LETTER
13 var MyConstructor = function(options) {
14   this.options = options;
15 };
16
17 // objects build by a constructor are also objects, thus lower case
18 var myConstructorObject = new MyConstructor({
19   name: "DA"
20 });
21
22 // always use triple === or !== in conditions, == and != have sideeffects!
23 if (myFunction() === true) {
24   MY_CONSTANT = 1000; // possible, no true constant!
25 }
```

Abbildung 6.3: Auszug einiger benutzter Code-Konventionen während der Entwicklung von Physiogame

Wie in Abbildung 6.3 ersichtlich, wären konstante Werte und Objekte ansonsten voneinander ununterscheidbar. An dieser Stelle muss unbedingt auf den

großen Anfangsbuchstaben der „Constructor“-Funktion „MyConstructor“ hingewiesen werden. Diese Funktion erzeugt ein neues Objekt und muss daher unbedingt mit „new“ aufgerufen werden. Die unterschiedliche Schreibweise kann also auch darauf hinweisen, wie eine Variable zu verwenden ist. Crockford empfiehlt daher übrigens, „Constructor“-Funktionen so oft wie möglich zu vermeiden, da sie eine häufige Fehlerquelle sind. Vergleiche in Bedingungen sollten übrigens immer mit „===“ oder „!==“ erfolgen. Die schlimmen Geschwister „==“ und „!=“ versuchen nämlich automatisch die primitiven Typen zweier unterschiedlicher Variablen anzugleichen, was zu unerwarteten Ergebnissen führen kann. Beispielsweise ist die Bedingung „if (0 == ””)“ wahr (vgl. [Crockford 2008], S. 30 und S. 109).

Die vorgestellten Konventionen müssen nicht blind befolgt werden. Viel wichtiger ist, dass sie einmalig definiert und im innerhalb des gesamten Teams auch so kommuniziert werden. Das vorrangige Ziel soll sein, konsistenten Code zu produzieren, damit dieser wenn nötig auch langjährig gewartet werden kann. Nochmals wichtig ist an dieser Stelle festzuhalten, dass das Vorgestellte nur eine ausgesuchte Teilmenge aller möglichen Konventionen für JS ist. Einen hervorragenden Überblick zum aktuell bevorzugten Stil innerhalb der JS-Community, bietet das Open-Source-GitHub-Projekt „idiomatic.js: Principles of Writing Consistent, Idiomatic JavaScript“ (vgl. [Waldron 2013]).

6.4 Code-Analyse und Testumgebung

Korrekten Code zu produzieren, der sich wie vorgesehen verhält, ist in jeder Programmiersprache schwierig. JS ist aber zusätzlich noch eine interpretierte (bzw. „just-in-time“-kompilierte) Sprache und hat prinzipiell keinen Compiler, der vor Fehlern vor Ausführung warnen könnte. Umso wichtiger ist es hier also, Hilfstools einzusetzen, die den Code analysieren können. Eine korrekte Syntax heißt aber lange noch lange nicht, dass ein Programm so funktioniert, wie es ursprünglich geplant war. Neben der Code-Analyse muss die tatsächliche

Funktionalität mit Tests überprüft werden.

6.4.1 JSLint und JSHint

Bereits im ersten Abschnitt wurde das Plugin JSHint für Sublime Text 2 erwähnt. Die Ursprünge zu diesem Plugin liegen aber eigentlich beim Tool JSLint von Douglas Crockford. Crockford war auch vor der Veröffentlichung des Buches „JavaScript: The Good Parts“ kein Unbekannter in der JS-Community. Er veröffentlichte JSLint im Jahre 2002. Es handelt sich dabei um ein Tool zur statischen Analyse von JS-Code nach bekannten Schwachstellen (die späteren „bad parts“) und generellen Syntaxfehlern. Das namensgebende „lint“ kommt aus den Anfängen von C, wo es ein gleichnamiges „accessory program“ gab, dass im Laufe der Weiterentwicklung der C-Compiler jedoch überflüssig wurde. Das Programm ist sowohl im Web als auch auf der Kommandozeile lauffähig (vgl. [Crockford 2002] und [Crockford 2008], S. 115).

Crockford entwickelte JSLint seither stetig weiter, im Laufe der Zeit wurden in der JS-Community die Stimmen jedoch lauter, dass es zur sehr von seinen eigenen Stil beeinflusst wäre und zu wenig Optionen zur Konfiguration zulässt. Aus diesem Unmut entstand JSHint – ein Fork von JSLint – und wird seither ähnlich wie die zuvor angesprochene Code-Konvention „idiomatic.js“ von der JS-Community Open-Source weiterentwickelt. (vgl. [Kovalyov 2011]). Mittlerweile verlassen sich viele Top-Unternehmen auf dieses Tool, wie beispielsweise Mozilla, Facebook, Twitter und Yahoo! (vgl. [Kovalyov 2013]). Aus diesem Grunde wird auch für die Entwicklung für Physiogame ausschließlich auf JSHint zur Code-Analyse vertraut. Für ein Zusammenspiel mit – der später vorgestellten Bibliothek – RequireJS wäre dies auch nicht anders möglich gewesen.

Abbildung 6.4 auf der nächsten Seite zeigt wie die Fehlerausgaben von JSHint innerhalb Sublime Text 2 aussehen. Hier wurde ein Strichpunkt in der Zeile drei, bei der Deklaration und Initialisierung der Variable „noSemicolon“, vergessen. Die Zahl 24 in der Fehlerbeschreibung ist dabei die Stelle in der Zeile,

wo das Symbol erwartet wird.

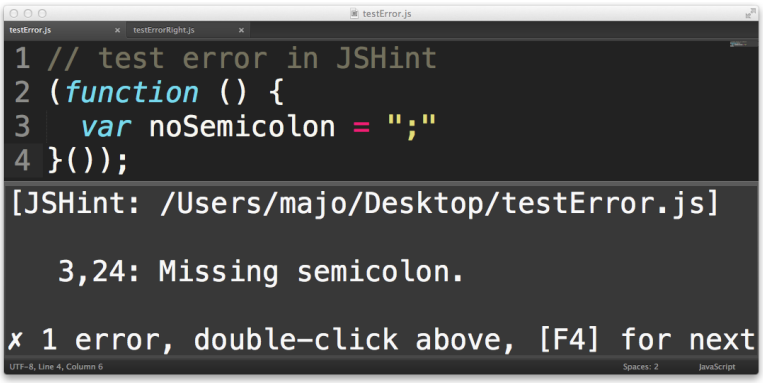

Abbildung 6.4: Ausführung von JSHint via Plugin in Sublime Text 2

Code-Analyse alleine reicht allerdings bei Weitem nicht, um den korrekten Ablauf eines Programms zu überprüfen (vgl. [Kovalyov 2013]). Aus diesem Grund lohnt es sich eine entsprechende Testumgebung aufzusetzen.

6.4.2 Aufbau einer Referenz-Testumgebung

Für JS gibt viele verschiedene Bibliotheken, sowohl clientseitig als auch für Node.js, um automatisierte Unit- und Integration-Tests in JS zu verwirklichen. Diese sind auch absolut notwendig, da es neben JSHint bzw. JSLint keine Möglichkeit gibt, Code-Fehler vor der Ausführung zu erkennen. Hier wird die wohl größte Einschränkung der schwachen Typisierung in JS sichtbar: Es können zur Kompilier-Zeit (die in JS nicht vorhanden ist) keine Fehler aufgrund von falscher Typisierung durch einen Compiler festgestellt werden.

Die Verantwortung zur Produktion von fehlerfreien Code, liegt vollständig beim Entwicklungsteam. Eine hervorragende Idee ist daher, gleich zu Beginn eines neuen Projekts mit einer agilen Software-Entwicklungsmethode, wie beispielsweise „Test-Driven Development“ (TDD) zu starten. TDD verlangt, dass (Unit-) Tests vor der eigentlichen Implementierung einer Funktionalität geschrieben werden (vgl. [Trostler 2013], S. 5). Es werden also vorab Testfälle definiert, wie sich etwas zu verhalten hat. Die Implementierung muss danach diese Testfälle erfüllen.

Dieses Paradigma wurde auch während der Entwicklung von Physiogame angewandt. Eine vollständige Abdeckung durch Tests hätte der pünktlichen Fertigstellung des Projekts jedoch im Wege gestanden, daher wurden nur die essentiellsten Teile der Applikation mit Unit-Tests getestet und auf Integration-Tests wurde vollständig verzichtet.

6.4.2.1 Komponenten

Die eingesetzte Testumgebung bei Physiogame besteht aus mehreren Komponenten. Dieser Aufbau der Testumgebung ist als Referenz-Umsetzung anzusehen. Jede Komponente kann bei der Entwicklung von eigenen Applikationen durch andere Teile ersetzt werden, falls dies gewisse Anforderungen voraussetzen. Die einzelnen Teile und die Ausführung im Gesamten wird in den nachfolgenden Paragraphen vorgestellt.

Http-server

Http-server ist ein Kommandozeilen-Tool, das in JS geschrieben wurde und in Node.js läuft. Es stellt einen kompletten HTTP-Server bereit. Die Besonderheit dieses Tools ist, dass es ohne komplizierte Konfigurationsschritte auskommt. Nach der Installation über NPM, reicht die Ausführung des Befehls „http-server“ um einen Webserver zu starten, der das aktuelle Verzeichnis als seine Wurzel nimmt (vgl. [Nodejitsu Inc. 2013]). Der Server stellt somit die Einsprungs-Adresse, sowohl zum Testen als auch zur normalen Ausführung von Physiogame im Browser.

PhantomJS

PhantomJS ist ein Kommandozeilen-Tool, das auf die Browser-Engine WebKit basiert und eine kompletten Browser-Umgebung bereitstellt. Der große Unterschied ist, dass es dieser Browser keine grafische Oberfläche („headless“)

besitzt und sich somit vor allem zur Automatisierung bzw. für serverseitige Ausführungen eignet (vgl. [Hidayat 2013]). Hierdurch kann eine schnelle Umgebung zur Ausführung der Tests zur Verfügung gestellt werden.

Mocha

Die JS-Bibliothek Mocha stellt ein komplettes Testframework bereit. Tests, die durch diese Bibliothek definiert werden, können sowohl in Node.js als auch im Browser ausgeführt werden (vgl. [Holowaychuk 2011]). Die Bibliothek ist also für die Definition und spätere Ausführung der Unit- bzw. Integration-Tests zuständig.

Chai

Chai ist eine JS-Bibliothek die das Arbeiten mit „assertions“ erleichtert und die Lesbarkeit erheblich erhöht. Die Überprüfung der Aussage, dass die Variable „myVar“ ein primitiver Typ „string“ ist, sieht beispielsweise folgendermaßen aus (vgl. [chaijs 2013]): „myVar.should.be.a('string')“.

Grunt-contrib-watch

Grunt-contrib-watch ist ein Plugin für Grunt um die Ausführung von Phantomjs zu automatisieren. Hierdurch kann erzielt werden, dass die Tests bei einer Änderung des JS-Quellcodes sofort (in PhantomJS) ausgeführt und die Ergebnisse im Terminal angezeigt werden (vgl. [GruntJS 2013a]). Grunt wird im Abschnitt „Automatisierung“ auf Seite 107 noch genauer vorgestellt.

6.4.2.2 Zusammenspiel

Die Testumgebung kann mit Grunt komplett automatisiert werden. Durch den Einsatz von PhantomJS ist die Ausführungszeit zudem derart beschleunigt, dass tatsächlich effizient nach den Prinzipien von TDD gearbeitet werden kann.

Abbildung 6.5 zeigt die automatisierte Ausführung von drei Testfällen für die Hilfefunktion „hittest“, nachdem sich der Quellcode geändert hat.

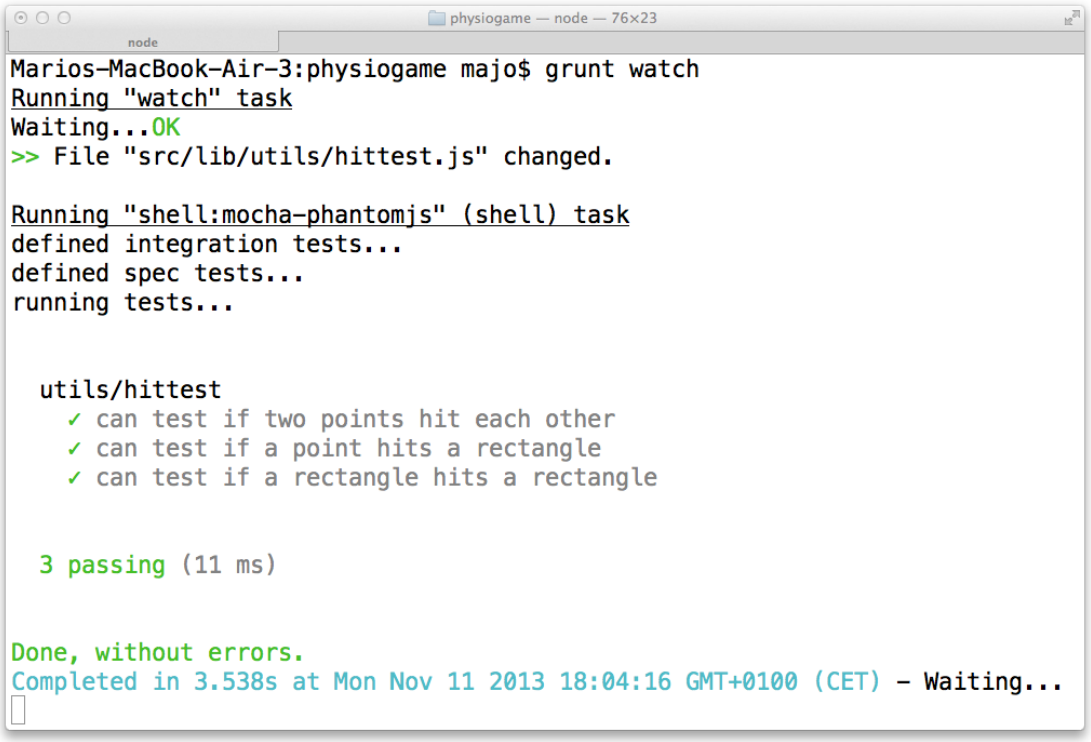

Abbildung 6.5: Automatisierte Ausführung von Tests nach Dateiänderungen

Wie ersichtlich wird, werden die eigentlichen Tests in elf Millisekunden abgearbeitet. Lediglich das Starten von PhantomJS benötigt auf der Hardware des Autors etwas mehr als drei Sekunden. Die Ergebnisse der Tests können somit nach jeder Änderung des Quellcodes fast augenblicklich untersucht werden, was die Implementierung von neuer Funktionalität auf Basis von definierten Testfällen vereinfacht. Es macht daher Sinn diesen Grunt-Task bei Entwicklungsarbeiten immer aktiviert zu lassen.

6.5 Modularisierung

JS hat von Haus aus kein System, um Module zu definieren und dies führte dazu, dass viele Eigenimplementierungen hierfür existieren (vgl. [Fogus 2013], S. 3). Wie der eigentliche Quellcode, zu einem derartigen Modul aussieht, wird

erst später im Abschnitt „Patterns“ auf Seite 115 erläutert. Vorab wird nämlich erst ein Mechanismus benötigt, um mit Modulen effektiv arbeiten zu können. In den nachfolgenden Unterabschnitten werden daher zuerst die Anforderungen und Probleme einer Modularisierung in JS erläutert und aktuelle Standards beleuchtet. Die Bibliothek RequireJS bringt für die Modularisierung entscheidende Vorteile. Sie wird daher zu einem zentralen Thema der letzten beiden Unterabschnitte und sowohl theoretisch als auch beim praktischen Einsatz untersucht.

6.5.1 Anforderungen und Probleme

Modularität ist von sehr großer Bedeutung, vor allem wenn es um eine spätere Wiederverwertbarkeit von Funktionalität geht. Ein Modul in JS ist definiert als *„eine Funktion oder ein Objekt, dass eine Schnittstelle zur Verfügung stellt, den internen Status („state“) und die Implementierung jedoch verbirgt“*[2] ([Crockford 2008], S. 40).

Alle Module in einer einzigen JS-Datei zu definieren, würde der Struktur eines Projekts schaden, insbesondere bei komplexeren Applikationen. Die simple Aufteilung in separate Dateien reicht aber alleinig nicht, wie aus der nachfolgenden Abbildung 6.6 auf der nächsten Seite ersichtlich werden sollte. Module müssen meist in einer bestimmten Reihenfolge geladen werden. Diese Reihenfolge händisch via „script“-Tags direkt im HTML-Dokument zu setzen, mag für den Anfang reichen, wird jedoch schnell zum Flaschenhals.

[2] Übersetzung durch den Autor

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Script ordering is insane for big applications!</title>
  </head>
  <body>
    <script src="utils/resizer.js"></script>
    <script src="utils/formatter.js"></script>
    <script src="models/model1.js"></script>
    <script src="models/model2.js"></script>
    <script src="layers/Layer.js"></script>
    <script src="layers/welcomeLayer.js"></script>
    <script src="layers/debugLayer.js"></script>
    <script src="controllers/soundController.js"></script>
    <script src="controllers/appController.js"></script>
    <script src="initializer.js"></script>
  </body>
</html>
```

Abbildung 6.6: Manuelle Script-Tags hindern bei der Entwicklung von modularen JS-Applikationen

Die Beispiel-Applikation der Abbildung 6.6 besteht lediglich aus zehn Skripts und ist von der exakten Ausführungsreihenfolge, die hier definiert wurde abhängig. Diese Vorgehensweise funktioniert zwar, ist jedoch extrem fehleranfällig. Beispielsweise besitzt die Applikation Physiogame über 80 eigene Module (ohne externe Bibliotheken) und es darf nicht vergessen werden, dass sich während der Entwicklung, die Struktur einer Applikation häufig verändern kann. Daher ist es extrem wichtig ein System zu finden, welches das automatische Laden der Skripte in der richtigen Reihenfolge ermöglicht.

Im Abschnitt „Management von Abhängigkeiten“ auf Seite 83 wurden mit NPM und Bower bereits zwei Tools vorgestellt um die Abhängigkeiten von externen Bibliotheken für eine Applikation zu verwalten. Was jetzt noch fehlt, ist eine Möglichkeit, um die Abhängigkeiten zwischen den eigenen und fremden Modulen klar zu definieren und daraus die benötigte Ladereihenfolge abzuleiten. Für den produktiven Einsatz der JS-Applikation im Browser muss zudem noch eine weitere Anforderung erfüllt werden: Das Laden der kompletten JS-Applikation aus einer einzigen Datei, da ansonsten jeder Ladevorgang eines Modules einen HTTP-Request darstellen würde. Eine hohe Anzahl an HTTP-Requests würde sich ansonsten negativ auf die Ladezeit der Applikation auswirken (vgl. [Stefanov 2010], S. 196).

6.5.2 CommonJS und AMD

Eine „import“- bzw. „include“-Anweisung wie in C++, Java oder ActionScript existiert in JS (noch) nicht, kommt aber mit der nächsten Version von ECMA-Script „Harmony“. Da diese Funktionalität von äußerster Bedeutung ist, um eine ansprechende Code-Struktur zu verwirklichen, haben sich inzwischen bereits eigene Standards gebildet, die sich mit der Modularisierung in JS beschäftigen. Unter dem Namen CommonJS ist in der JS-Community eine offene Gruppe entstanden, um den gleichnamigen Standard für das Deklarieren und Laden von Modulen zu formen (vgl. [Osmani 2011]).

Mittlerweile hat sich CommonJS in serverseitigen Umgebungen (Node.js) bereits durchgesetzt, im Browser jedoch nicht. Der Hauptgrund dafür, ist die Art des Ladevorganges von CommonJS-Modulen. In Server-Umgebungen macht es nämlich Sinn, Module in einer synchronen Weise zu laden, da davon ausgegangen werden kann, dass diese unmittelbar zu Verfügung stehen, in Browser-Umgebungen aber nicht. Aus diesem Grund formte sich eine weiterer Standard, die „Asynchronous Module Definition“ (AMD). Der Standard baut auf den CommonJS-Standard auf, spezifiziert ihn aber in einer asynchronen Weise, speziell für Browser-Umgebungen (vgl. [Osmani 2011]).

Abbildung 6.7 auf der nächsten Seite zeigt die Umsetzung derselben Funktionalität in Form eines Moduls nach CommonJS und nach AMD. Die Module „moduleCommonJS“ und „moduleAMD“ sind beide vom einem Modul mit dem „Identifier“ „package/lib“ abhängig. Diese Abhängigkeit muss zuerst aufgelöst werden, bevor die eigene öffentliche Schnittstelle „something“ des Moduls exportiert werden kann, da die öffentliche Methode „publicMethod“ des abhängigen Moduls hierzu benötigt wird.

Die Unterschiede zwischen der synchronen Ladeweise bei CommonJS und der asynchronen bei AMD werden in der Zeile sechs und 17 auffällig. JS-Code wird nämlich sowohl im Browser als auch unter Node.js immer von oben nach unten ausgeführt. CommonJS verlangt folglich, dass die Abhängigkeit „package/lib“ sofort verfügbar sein muss, damit die nachfolgenden Zeilen (ab

```
1  // ------------------------------------------------------------------------
2  // CommonJS module "moduleCommonJS.js"
3  // ------------------------------------------------------------------------
4
5  // require dependencies for this module (synchronous, instantly available)
6  var lib = require('package/lib');
7
8  // expose public API of this module via special "exports" object
9  exports.something = lib.publicMethod();
10
11
12 // ------------------------------------------------------------------------
13 // AMD module "moduleAMD.js"
14 // ------------------------------------------------------------------------
15
16 // require dependencies for this module (asynchronous, available in callback)
17 define(['package/lib'], function(lib) {
18
19   // expose public API of this module via normal return
20   return {
21     something: lib.publicMethod()
22   };
23 });
```

Abbildung 6.7: Unterschiedliche Struktur von CommonJS- und AMD-Modulen

der Zeile sechs) direkt ausgeführt werden können. AMD geht hier einen anderen Weg und nimmt eine Einkapselung in einer Funktion vor. Die Zeilen 19 bis 22 werden erst ausgeführt, nachdem das Modul „package/lib“ asynchron geladen wurde. Daher müssen Module im Stil von AMD innerhalb einer Rückruf-Funktion („Callback“) definiert werden, die erst aufgerufen wird, sobald die Abhängigkeiten tatsächlich zur Verfügung stehen.

Übrigens ist auch die Definition der öffentlichen Schnittstelle (Zeile neun bzw. 20 bis 22) zwischen CommonJS und AMD unterschiedlich. CommonJS stellt ein eigenes „exports“-Objekt bereit, während bei AMD direkt mit dem Rückgabewert der Rückruf-Funktion („return“) gearbeitet werden kann.

6.5.3 RequireJS

Da Physiogame vorrangig eine clientseitige JS-Applikation ist, die sowohl im Browser als auch unter Node-webkit lauffähig sein soll, wird zur Modularisierung auf AMD gesetzt. AMD ist – ähnlich wie CommonJS – jedoch nur ein einheitliches Format um Module auszudrücken, was daher noch fehlt, ist eine Bibliothek – ein sogenannter „Script-Loader“ – die den Standard implementiert.

Hier gibt es mehrere Möglichkeiten, beispielsweise RequireJS, curl.js, bdLoad, Yabble, PINF, wobei RequireJS von James Burke mittlerweile aber am Populärsten ist und dies aus guten Gründen (vgl. [Osmani 2011]).

RequireJS bietet neben der Modularisierung, die hauptsächlich während des Entwicklungsprozesses von großer Bedeutung ist, durch das Kommandozeilen-Tool r.js, auch Methoden zur Optimierung des JS-Codes in eine einzelne Datei („Concatenation“ und „Minification“) für den produktiven Einsatz. Für diese optimierte Datei ist dann nur mehr ein HTTP-Request erforderlich und die Dateigröße kann durch verschiedene Komprimierungsverfahren weiter reduziert werden. RequireJS dient somit sowohl zur Modularisierung als auch zur Verpackung der Applikation für die Veröffentlichung (vgl. [Osmani 2011] und [Burke 2013c]).

Der Fokus auf den AMD-Standard, versperrt RequireJS übrigens nicht die Möglichkeit auch unter Node.js eingesetzt zu werden. Mit r.js können bisherige CommonJS-Module außerdem direkt in AMD-Module konvertiert werden (vgl. [Burke 2013c]).

Zur Entwicklung von JS-Applikationen ist RequireJS heute sicherlich die erste Wahl und dürfte zudem auch mit dem Release des nächsten Version von ECMAScript nicht obsolet werden. Ein Hauptgrund hierfür ist sicherlich die Abwärts-Kompatibilität, da mit RequireJS keine neue JS-Syntax zur Definition von Modulen benötigt wird und somit wirklich alle Plattformen (von ECMAScript in der dritten Version bis zu Node.js ist alles möglich) anvisiert werden kann (vgl. [Osmani 2011] und [Burke 2012]). Für die Entwicklung von Physiogame wurde zur Modularisierung und Optimierung daher ausschließlich auf RequireJS gesetzt. Im nächsten Unterabschnitt wird eine Beispiel-Applikation mit RequireJS vorgestellt, um die Vorteile beim praktischen Einsatz aufzuzeigen.

6.5.4 Beispiel-Applikation mit RequireJS

Der prinzipielle Aufbau eines AMD-Moduls wurde bereits in der Abbildung 6.7 auf Seite 95 kurz vorgestellt. Die Syntax und die Schlüsselwörter („define“) wurden damals aber noch außen vor gelassen. Weiters fehlt bis dato eine Erläuterung wie sich RequireJS ins Gesamtbild einer Applikation einfügt und wie Module funktionieren und zusammenarbeiten. Daher macht es an dieser Stelle Sinn eine Beispiel-Applikation zu erstellen und die Fragestellungen in den nachfolgenden Unterabschnitten separat zu beantworten.

6.5.4.1 Vorbereitung und Konfiguration

Abbildung 6.8 zeigt die gewählte Ordner-Struktur für die Beispiel-Applikation. Die Datei „index.html“ dient dabei als Einstieg für den Browser. Innerhalb der Datei „main.js“ befindet sich die Konfiguration für RequireJS, jene Bibliothek die in ihrer komprimierten Form als „require.js“ verfügbar ist. Alle Module unserer Applikation liegen separat im Ordner „src“.

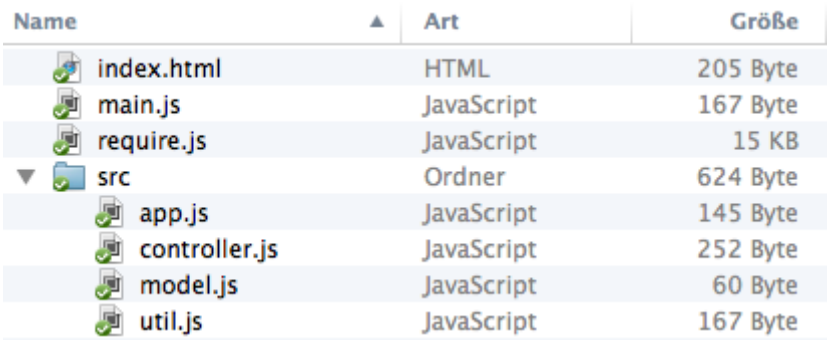

Abbildung 6.8: RequireJS Beispiel-Applikation: Ordner-Struktur

Durch die Abbildung 6.9 auf der nächsten Seite werden erstmals die Auswirkungen von RequireJS ersichtlich. Obwohl unsere Applikation modular aufgebaut ist und aus mehreren Dateien besteht, wird nur ein einziger Tag benötigt. Das Attribut „data-main“ im „script“-Tag zeigt dabei auf die Haupt-Konfigurationsdatei „main.js“ für RequireJS. Der Inhalt dieses Skripts wird direkt nach den Laden

der Bibliothek ausgeführt. Der beste Platz für das „skript“-Tag ist übrigens bevor „body“ geschlossen wird (vgl. [Stefanov 2010], S. 199).

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="utf-8" />
5      <title>Script loading with requireJS</title>
6    </head>
7    <body>
8      <script data-main="main.js" src="require.js"></script>
9    </body>
10 </html>
```

Abbildung 6.9: RequireJS Beispiel-Applikation: index.html

RequireJS ruft die Datei „main.js“ (siehe Abbildung 6.10) automatisch auf und erwartet, dass dieses Skript ein „config“-Objekt für „require“ mit den Einstellungen zum Laden der Module der Applikation bereitstellt.

```
1  // ------------------------------------------------------------------------
2  // main.js
3  // ------------------------------------------------------------------------
4
5  require.config({ // requireJS config parameters
6    baseUrl: "src/"
7  });
8
9  require(["app"], // main entry point (main module)
10   function(app) {
11     console.log("main: require complete");
12   }
13 );
```

Abbildung 6.10: RequireJS Beispiel-Applikation: main.js

Wie in Abbildung 6.10 ersichtlich, reicht es für unsere Beispiel-Applikation, die „baseUrl“ in Zeile sechs zu setzen. Der gesetzte Wert weist den Ordner „src“ als Wurzelverzeichnis für unsere Module aus. In der neunten Zeile wird das Modul „app“ schließlich mit dem Schlüsselwort „require“ als Hauptmodul ausgewiesen – das heißt diese Abhängigkeit wird für die Gesamt-Applikation benötigt – und von hier aus startet der Ladevorgang der abhängigen Module von „app“. Bei Betrachtung des Aufbaus von „require“ wird dabei folgendes ersichtlich:

1. Im ersten Parameter von „require“ müssen die benötigten Abhängigkeiten in einem Array aus Strings spezifiziert werden. Beispielsweise „[’util’, ’model’]“.
2. Der zweite Parameter ist eine Rückruf-Funktion („callback“), die mit den geladenen Abhängigkeiten in der selben Reihenfolge ausgeführt wird, wie im Array spezifiziert.

Die Ausgabe zur Konsole in der elften Zeile dient übrigens nur zur späteren Veranschaulichung des Ladevorganges. Diese Anweisung wird erst ausgeführt, sobald „app“ und alle Module, von denen „app“ abhängt, initialisiert wurden.

6.5.4.2 Funktionalität

Bevor wir den eigentlichen Code der vier Module betrachten, ist es wichtig die Funktionalität der Applikation zu erläutern. In der nachfolgenden Auflistung werden die Features der Module vorgestellt:

- Das Modul „**model**“ hält lediglich die zwei Strings „name“ und „version“ auf die zugegriffen werden kann.
- Das Hilfsmodul **„util“** stellt die Methode „concatString“ bereit, um zwei Strings zu einer schöneren Ausgabe zu kombinieren.
- Das Modul **„controller“** dient als Schnittstelle zu den Daten und besitzt die öffentliche Methode „getDisplayName“. In dieser Methode wird „concatString“ von „util“ verwendet um die Daten von „model“ aufbereitet zurückzugeben.
- Das Modul **„app“** ist unsere Hauptapplikation und greift bei Initialisierung auf „getDisplayName“ vom Controller zu und leitet die Ausgabe an die Konsole.

Jedes Modul – außer „model“ – gibt zudem beim ersten Aufruf einen String „init“, inklusive des Modulnamens aus, damit der Ausführungsvorgang besser verfolgt werden kann.

Abbildung 6.11 zeigt nun den tatsächlichen Quellcode der zuvor vorgestellten Features.

```
// ------------------------------------------------------------------------------
// app.js
// ------------------------------------------------------------------------------
define(["controller"], // definition function with dependencies
  function(controller) {
    console.log("app: init");
    console.log("app info: " + controller.getDisplayName());
  }
);

// ------------------------------------------------------------------------------
// controller.js
// ------------------------------------------------------------------------------
define(["model", "util"], // definition functions with dependencies
  function(model, util) {
    console.log("controller: init");

    function getDisplayName() {
      return util.concatString(model.name, model.version);
    }

    return { // public API
      getDisplayName: getDisplayName
    };
  }
);

// ------------------------------------------------------------------------------
// util.js
// ------------------------------------------------------------------------------
define(function() { // definition function
  console.log("util: init");

  function concatString(a, b) {
    return a + " -- " + b;
  }

  return { // public API
    concatString: concatString
  };
});

// ------------------------------------------------------------------------------
// model.js
// ------------------------------------------------------------------------------
define({ // simple key/value pairs (object-literal)
  name: "Simple Require Test",
  version: "1.0"
});
```

Abbildung 6.11: RequireJS Beispiel-Applikation: Module

RequireJS ist ein Verfechter des „Don't Repeat Yourself"-Prinzips (vgl. [Osmani 2011]). Dies zeigt sich in Abbildung 6.11 dadurch, dass der Modulname im Quellcode nirgendwo (abseits des Kommentare, die für diese Abbildung

eingefügt wurden) extra definiert werden muss. Der Name eines Moduls ergibt sich automatisch durch den Pfad – vom Wurzelverzeichnis der Module (die „baseUrl“ in der RequireJS-Konfiguration) – plus dem Dateinamen. Das Schlüsselwort „define“ von RequireJS weist den nachfolgenden Code folglich als ein Modul aus.

Bei Betrachtung des Codes in Zeile vier und 14, fällt weiters auf, dass sich Abhängigkeiten zu anderen Modulen ähnlich, wie beim zuvor vorgestellten „require“ definieren lassen. Die Module „util“ und „model“ sind allerdings nicht von anderen Modulen abhängig, daher kann der erste Parameter (Array aus Strings) in Zeile 31 und 46 verworfen werden. Für simple Schlüssel-Wert-Paare („key-value pairs“) reicht es ein Objekt-Literal zu verwenden, wie im „model“ ersichtlich. Das Modul „util“ dagegen, definiert Funktionalität, wird also wie bisher in einer Funktion eingekapselt.

6.5.4.3 Initialisierung und Ladereihenfolge

Abbildung 6.12 visualisiert die Abhängigkeiten zwischen den Modulen zur besseren Veranschaulichung und erklärt wie dieses Geflecht die Ausführungsreihenfolge beeinflussen wird.

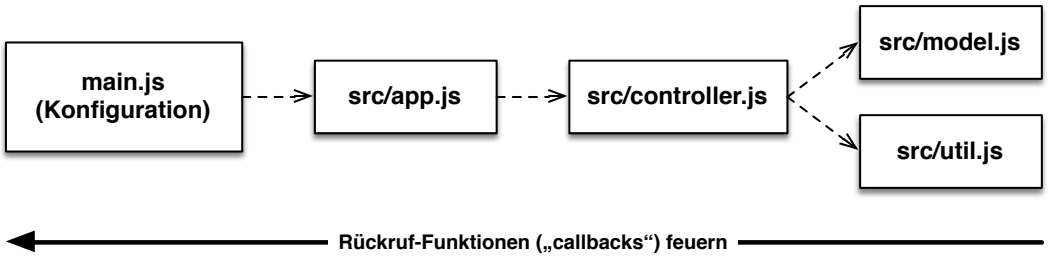

Abbildung 6.12: RequireJS Beispiel-Applikation: Abhängigkeiten der Module und Ladereihenfolge

Die gestrichelten Linien der Abbildung 6.12 zeigen mit ihrer Pfeilspitze auf die benötigten Abhängigkeiten. Beispielsweise ist das Modul „controller“ sowohl von „model“ als auch von „util“ abhängig. Die tatsächliche Ausführungsreihenfolge innerhalb der Laufzeitumgebung ist nun besonders spannend. In der

Konfigurations-Datei „main“ wird das Modul „app“, als erstes „required“. Alle „define“-Funktionen innerhalb der abhängigen Module von „app“ werden dadurch ausgeführt, also wirklich in der Reihenfolge von links nach rechts ausgeführt. Der eingekapselte Inhalt der Rückruf-Funktionen innerhalb der „define“-Aufrufe, wird jedoch erst dann aufgerufen, wenn die Abhängigkeit zum jeweiligen Modul erfüllt ist. Die tatsächliche Initialisierung der jeweiligen Module findet daher also in umgekehrter Reihenfolge, von rechts nach links, statt. Da in jedem Modul (außer in „model“) während der Initialisierung eine Ausgabe zur Konsole stattfindet, lässt sich die angesprochene Initialisierungsreihenfolge durch die Ausführung unserer Applikation leicht beweisen.

6.5.4.4 Ausführung

Abbildung 6.13 zeigt die Ausführung der Beispiel-Applikation im Google Chrome Browser.

```
util: init                                    util.js:6
controller: init                         controller.js:7
app: init                                     app.js:7
app info: Simple Require Test -- 1.0          app.js:8
main: require complete                       main.js:11
```

Abbildung 6.13: RequireJS Beispiel-Applikation: Ausführung

Wie in der Abbildung 6.13 ersichtlich wird, trifft die angesprochene Ladereihenfolge exakt zu. Die Information „Simple Require Test – 1.0“ wurde zudem erfolgreich vom Modul „app“ bei seiner Initialisierung über die Methode „getDisplayName“ von „controller“ (der „util“ und „model“ verwendet) generiert.

6.5.4.5 Erweiterung mit externen Bibliotheken

Unsere Beispiel-Applikation verfügt zwar über keine bahnbrechende Funktionalität, durch die Strukturierung in Modulen ist sie jedoch besonders einfach erweiterbar. Aktuell verwendet die Applikation beispielsweise noch überhaupt

keine externen Bibliotheken abseits von RequireJS. Diese können aber einfach innerhalb des „config“-Objekts („main.js“) mit ihrem eigenen „Identifier“ hinzugefügt werden. Einzig bei Bibliotheken, die noch nicht direkt zu AMD kompatibel sind, bedarf es eines zusätzlichen Schrittes um diese mit RequireJS zu laden und deren Funktionalität in den eigenen Modulen zu verwenden.

```
1  // ------------------------------------------------------------------------
2  // main.js
3  // ------------------------------------------------------------------------
4
5  require.config({ // requireJS config parameters
6    baseUrl: "src/",
7    paths: { // define identifier for external libs
8      "AMDCompliant": "../path/to/AMDCompliantjs",
9      "nonAMD": "../path/to/nonAMDjs"
10   },
11   shim: { // glue global objects of external libs to identifier
12     "nonAMD": {
13       exports: "nonAMDGlobalObject"
14     }
15   }
16 });
```

Abbildung 6.14: RequireJS Konfiguration mit externen Bibliotheken

Abbildung 6.14 zeigt eine erweiterte Beispiel-Konfiguration („main.js“) mit zwei unterschiedlichen fiktiven Bibliotheken. Die Bibliothek „AMDCompliant“ unterstützt AMD-Modulsysteme. Es muss bei dieser lediglich der Pfad zur Quelldatei definiert und ein frei wählbarer „Identifier“ gesetzt werden. Der Pfad zur Quelldatei im Objekt „paths“ ist dabei immer relativ vom der angegebenen „baseUrl“ (das Wurzelverzeichnis in dem unsere eigenen Module liegen). Die meisten populären JS-Bibliotheken – beispielsweise jQuery und lodash – sind in dieser Weise einzubinden.

Bibliotheken, die keinen Support für AMD besitzen, setzen meist ein globales Objekt, über das, deren Funktionalität aufgerufen werden kann. Diese müssen im „shim“-Objekt weiter definiert werden, wie die fiktive Bibliothek „nonAMD“ zeigt. In Zeile zwölf bis 14 wird ersichtlich, wie ein „Identifier“ expliziert mit dem globalen Objekt der Bibliothek verknüpft („Gluecode“) werden kann. In unserem Fall exportiert „nonAMD“ das globale Objekt „nonAMDGlobalObject“. Nach diesem Schritt kann auch diese externe Bibliothek über ihren „Identifier“,

wie bei unseren eigenen Modulen, als Abhängigkeit im „require“- bzw. „define“-Aufruf verwendet werden.

6.5.4.6 Optimierung mit r.js

Die vorgestellte Beispiel-Applikation kann mit r.js in eine einzelne Datei optimiert werden. Hierzu macht es Sinn, das Paket „requirejs“ mit NPM zuvor zu installieren, da dieses Paket das Kommandozeilen-Tool r.js zur Ausführung der Optimierungsaufgaben beinhaltet. Zusätzlich wird das Paket „almond“ heruntergeladen, ein alternativer Skript-Loader, der besonders leichtgewichtig ist und nur mit optimierten AMD-Modulen, die in einer einzelnen Datei definiert sind, funktioniert. Hierdurch lässt sich die Größe der optimierten Datei noch weiter reduzieren, da nicht mehr die vollständige Funktionalität von RequireJS eingebunden werden muss (vgl. [Burke 2013a]).

```
1 {
2   "name": "requireJS-simple-intro",
3   "version": "0.0.1",
4   "dependencies": {
5     "requirejs": "~2.1.9",
6     "almond": "~0.2.6"
7   }
8 }
```

Abbildung 6.15: RequireJS Optimierung: package.json

Abbildung 6.15 zeigt die Datei package.json, in der die abhängigen Node.js-Pakete definiert werden. Durch das Kommando „npm install -d“ werden diese dann automatisch heruntergeladen und im Projektverzeichnis installiert.

Somit fehlt nur noch die Definition eines Skripts, mit dem der Vorgang der Optimierung konfiguriert und die Datei schließlich erstellt wird. Abbildung 6.16 auf der nächsten Seite zeigt dies am Beispiel des alternativen Skript-Loaders Almond. Da das Skript mit Node.js ausgeführt wird, müssen die benötigten Abhängigkeiten synchron mit „require“ (Zeile sechs bis sieben) geholt werden. Im Konfigurationsobjekt (zwischen Zeile zehn bis 19) von RequireJS kann direkt

auf die Haupt-Konfigurationsdatei „main.js“ verwiesen werden. Somit müssen hier tatsächlich nur die Informationen zum Optimieren spezifiziert werden. Dies beinhaltet beispielsweise das zu verwendende Komprimierungsverfahren „uglify2“ und der Name der Ausgabedatei. Die eigentliche Ausführung und der Schreibvorgang findet danach zwischen Zeile 22 und 24 statt.

```
1  // ------------------------------------------------------------------------
2  // build-almondjs.js
3  // ------------------------------------------------------------------------
4
5  // Node.js require RequireJS und filestream to output a file
6  var requirejs = require('requirejs');
7  var fs = require('fs');
8
9  // Configure RequireJS
10 var config = {
11   baseUrl: "src/", // Base URL
12   mainConfigFile: "main.js", // include all dependencies from main
13   optimize: "uglify2",
14   include: ["app"],
15   logLevel: 0,
16   name: "../node_modules/almond/almond", // start building from where?
17   insertRequire: ["app"], // modules to require on start?
18   out: 'build/optimized-build-almond.js' // target file
19 };
20
21 // Optimize our script
22 requirejs.optimize(config, function(buildResponse) {
23   var contents = fs.readFileSync(config.out, 'utf-8');
24 });
```

Abbildung 6.16: RequireJS Optimierung: build-Datei „build-almondjs.js“

Das Build-Skript kann nun mit „node build-almondjs.js“ aufgerufen werden. Die erhaltene optimierte Datei behaltet jeglichen Code aller eigenen Module, der externen Bibliotheken und des eigentlichen Skript-Loaders (RequireJS oder Almond). Die gesamte Applikation kann somit mit einem einzigen Skript-Tag im HTML-Dokument eingebunden werden. Bleibt die Frage welche Bibliothek – Almond oder RequireJS – für optimierten Files verwendet werden soll. In der Regel wird das dynamische Nachladen (von externen Dateien) in der produktiven Version nicht benötigt und dies stellt einen großen Teil von RequireJS dar. Daher macht es Sinn, in den meisten Fällen Almond einzusetzen und die Dateigröße so am Geringsten zu halten.

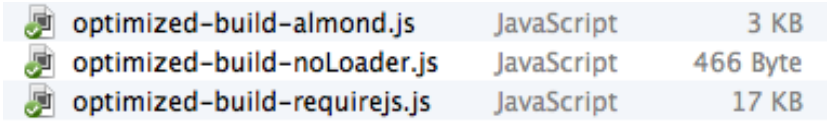

Abbildung 6.17: RequireJS Optimierung: zu erwartende Dateigröße

Abbildung 6.17 zeigt die Unterschiede in der Dateigröße der optimierten Dateien auf. Bei „optimized-build-noLoader“ wurde weder RequireJS noch Almond als Modul-Loader hinzugefügt. Dieses komprimierte Skript benötigt also noch immer ein externen Modul-Loader, um ausgeführt zu werden, ist aber auch am Kleinsten. Wird Almond inkludiert, vergrößert sich das Skript um rund 2,5 Kilobyte. Dies ist aber noch immer bedeutend kleiner, als das Einfügen der gesamten RequireJS-Bibliothek mit zusätzlichen 16,5 Kilobyte. Da die beiden Skripts aber keinen externen Modul-Loader mehr benötigen, sind sie komplett in sich geschlossen und tatsächlich fertig für die Auslieferung.

6.5.4.7 Spezialfall: Zirkelverweise

Innerhalb der „define“-Funktion werden die Abhängigkeiten zu anderen Modulen definiert. Alle Module hatten dabei eine einseitige Abhängigkeiten zueinander, wie in der früheren Abbildung 6.12 auf Seite 101 visualisiert wurde. Bleibt die Frage wie sich eine gegenseitige Abhängigkeit zwischen zwei Modulen auswirken würde, bzw. ob diese überhaupt möglich sind. Vorab muss aber angemerkt werden, dass diese sogenannten Zirkelverweise häufig eine Fehlentscheidung im Design der Applikation darstellen und konsequenterweise (wenn möglich) zu vermeiden sind. Für den Fall, dass eine Änderung im Design allerdings nicht möglich ist, helfen dynamische „require“-Anweisungen in diesem Fall weiter (vgl. [Burke 2013b]).

Was passiert, ist folgendes: Eines der beiden Module wird jedenfalls vorher initialisiert werden, bevor das andere Modul des Zirkelverweises geladen wurde. Es ist dabei nicht wichtig, welches Modul zu früh geladen wird, sondern es ist essentiell die jeweiligen Funktionen des Moduls, die auf das andere Modul zugreifen, abzusichern. Sobald eine (spätere) dynamische „require“-Anweisung

in einer Funktion des ersten Moduls zu einem Objekt führt, das nicht „undefined“ ist, handelt es sich um das tatsächliche zweite Modul. Um mit Zirkelverweisen zwischen Modulen umzugehen, ist also jedenfalls „Overhead“ bei der Funktionalität nötig, aber es ist prinzipiell möglich.

6.5.4.8 Reflexion

Insgesamt sollte durch die letzten Abschnitte nun aufgezeigt worden sein, dass RequireJS die Entwicklung von JS-Applikationen immens vereinfacht. Mit Bower und NPM können externe Abhängigkeiten in ihrer jeweiligen Version installiert und verwaltet werden, während sich RequireJS um die Struktur und Modularität der Applikation kümmert. Unsere Beispiel-Applikation besitzt eine modulare Struktur, lässt sich einfach erweitern und kann für den produktiven Einsatz gebündelt ausgeliefert werden. Neue externe Bibliotheken können transparent in der Konfiguration „main.js“ eingebunden werden bzw. ebenso einfach wieder ausgetauscht werden.

Eine große Falle von JS, globale Variablen und die Verschmutzung des globalen Namensraumes, tritt mit RequireJS gar nicht erst auf. Jedes Modul ist abgeschlossen und hat seinen eigenen Namensraum. Die Bibliothek RequireJS kann somit – aus der Sicht des Autors – aktuell tatsächlich als Basisanforderung für die Entwicklung von JS-Applikationen angesehen werden und dies wird sie wohl auch mit dem nächsten ECMAScript-Standard bleiben, solange Abwärts-Kompatibilität von Bedeutung ist.

6.6 Automatisierung

Im Laufe der Entwicklungsarbeiten fallen häufig sich wiederholende Tätigkeiten an, beispielsweise das Ausführen von Tests oder das Kombinieren („concatenation“) und Komprimieren („minification“) von Quelldateien. Es macht Sinn diese Arbeiten zu automatisieren, einerseits damit diese nicht mehr händisch

und damit zeitaufwendig ausgeführt werden müssen, andererseits damit die automatisierten Aufgaben in anderen Projekten wiederverwendet werden können.

6.6.1 Grunt

Grunt ist ein Kommandozeilen-Tool für Node.js zur automatisierten Ausführung von Aufgaben (sogenannte „Tasks“). Die Tasks werden direkt in JS definiert und können beliebig miteinander kombiniert werden. Für viele Anwendungsfälle stehen bereits fertige Plugins zur Verfügung, die direkt via NPM installiert werden können. Sie müssen danach nur mehr im sogenannten „Gruntfile“ für das jeweilige Projekt konfiguriert werden (vgl. [GruntJS 2013b]). Ein derartiges Plugin ist beispielsweise „Grunt-contrib-watch“, das im Rahmen der vorgestellten Referenz-Testumgebung aus Abschnitt 6.4.2 auf Seite 88 zum Einsatz kommt.

Aktuell existieren für die Entwicklung von JS-Applikationen keine nennenswerten Alternativen zu Grunt. Dies dürfte auch damit zusammenhängen, dass Grunt in der JS-Community extrem populär ist und das System hervorragend skalierbar ist. Grunt kann sowohl für kleine Automatisierungen beim Arbeitsablauf eingesetzt werden als auch einen kompletten Veröffentlichungsprozess übernehmen (vgl. [Ackerman 2013]).

6.6.2 Praktische Umsetzung von Grunt-Tasks

Um die Vorzüge von Grunt kennenzulernen, eignet sich ein praktisches Beispiel am Besten. Ziel dieses Beispiels soll sein mehrere Grunt-Tasks bereitzustellen, die das Veröffentlichen der Applikation automatisieren. Damit die – für den produktiven Einsatz bestimmte – Applikation in sich geschlossen ist, soll eine eigene Version im separaten Ordner „build“ erstellt werden. Dorthin müssen neben den eigentlichen Quellcode auch die Bilder („assets“) und „Cascading Style Sheets“ (CSS) exportiert werden. Der Quellcode wird zwar auto-

matisch durch r.js optimiert, muss aber durch den Grunt-Task über die Kommandozeile angestoßen werden. Die Tasks sollen sowohl separat als auch gesammelt in einem Durchlauf ausgeführt werden können.

```
1  // ---------------------------------------------------------------------------
2  // Gruntfile.js (example config)
3  // ---------------------------------------------------------------------------
4
5  module.exports = function(grunt) {
6    grunt.initConfig({
7      copy: {
8        "assets": { // copy matching assets to build directory
9          files: [{
10           src: ["assets/**/*.json", "assets/**/*.png"],
11           dest: "build/"
12         }]
13       }
14     },
15     cssmin: { // minify all css to build/app.min.css
16       css: {
17         src: ["bootstrap.css", "fonts.css", "app.css"],
18         dest: "build/app.min.css"
19       }
20     },
21     shell: { // run a shell task, uses node-script to run r.js with almond
22       "build-almond": {
23         command: "node build-almondjs.js"
24       }
25     }
26   });
27
28   // load grunt plugins (previously installed via npm)
29   grunt.loadNpmTasks("grunt-contrib-copy");
30   grunt.loadNpmTasks("grunt-contrib-cssmin");
31   grunt.loadNpmTasks("grunt-shell");
32
33   // define and chain tasks, run them in shell with "grunt name"
34   grunt.registerTask("optimize", "shell:build-almond");
35   grunt.registerTask("assetsminify", ["copy:assets", "cssmin"]);
36   grunt.registerTask("default", ["optimize", "assetsminify"]);
37 };
```

Abbildung 6.18: Beispiel-Konfiguration von Grunt

Abbildung 6.18 zeigt die Implementierung der vorherigen Anforderungen in einer einzigen Datei. Die Konfiguration von Grunt hat dabei unbedingt in einer Datei mit dem Namen „Gruntfile.js“ zu erfolgen muss im Wurzelverzeichnis des Projekts platziert werden. Im Beispiel werden drei Plugins verwendet, deren Initialisierung aber erst zwischen Zeile 29 bis 31 erfolgt. Die Konfiguration der Funktionalität findet gleich am Anfang in einem Objekt, das der „initConfig“-Funktion (siehe Zeile 6) übergeben wird, statt. Um die statischen Dateien der Applikation („assets“) zu kopieren, wird das Plugin „grunt-contrib-

copy“ zwischen Zeile sieben und vierzehn konfiguriert. Ähnlich verhält es sich mit der Konfiguration der anderen Plugins. Die eigentlichen Tasks werden zwischen Zeile 34 bis 36 registriert. Sie können dann in der Kommandozeile, beispielsweise durch „grunt optimize“ ausgeführt werden. Der „default“-Task ist übrigens ein Spezialfall und kann nur durch den Befehl „grunt“ gestartet werden kann. Wie in Zeile 36 ersichtlich, werden bei diesem Vorgang sowohl der „optimize“- als auch der „assetsminify“-Task hintereinander ausgeführt.

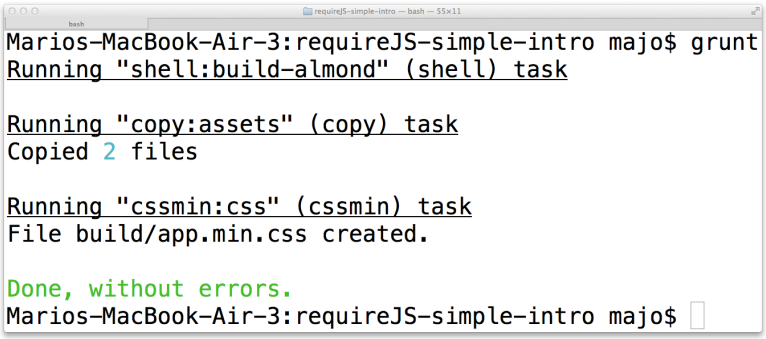

Abbildung 6.19: Ausführung der Beispiel-Konfiguration von Grunt

Abbildung 6.19 zeigt wie die Ausführung des „default“-Tasks der zuvor vorgestellten Konfiguration aussieht. Die drei Aufgaben („shell“, „copy“ und „cssmin“) werden hintereinander ausgeführt und die eigenständige Version der Applikation findet sich nun im Ordner „build“. Diese Konfiguration könnte nun noch weiter ausgebaut werden, um weitere Tasks einzuschließen.

Um den Build-Prozess von Physiogame zu automatisieren wurden noch andere Plugins verwendet. Beispielsweise wurde ein Task definiert um Erstellung der Release-Versionen (für Node-webkit, via Plugin „grunt-node-webkit-builder“) für Windows, Mac und Linux zu vereinfachen. Ergebnis ist somit eine vollständig verpackte Desktop-Applikation, die in dieser Form ausgeliefert werden kann. Insgesamt erwies sich Grunt daher als unerlässlich für die Entwicklung von Physiogame und findet sicherlich in jedem JS-Projekt einige Anwendungsfälle.

6.7 Fallen

JS unterscheidet sich in vielerlei Hinsicht von anderen Programmiersprachen und besitzt einige Fallen, die verstanden und vermieden werden müssen. Die nachfolgenden Unterabschnitte behandeln einige äußerst kritische Fallen, deren Kenntnis für die Entwicklung von JS-Applikationen vorausgesetzt werden sollte, inklusive einiger Best Practices um nicht hinein zu tappen.

6.7.1 Globale Variablen

Beim Sprachdesign von JS wurden einige schlechte Entscheidungen getroffen, aber *„eine Entscheidung war besonders schlecht: JavaScript ist auf globale Variablen zum Linken angewiesen. Alle Top-Level Variablen werden in einem gemeinsamen Namensraum, dem globalen Objekt, geschmissen“*[3] ([Crockford 2008], S. 3).

Globale Variablen sind überall sichtbar und während dies für kleine Spielereien nützlich sein kann, beispielsweise um Variablen zur Laufzeit direkt in der Konsole zu verändern, erweist sich dies als Flaschenhals bei komplexeren JS-Applikationen. Eine zusätzliche Bürde kommt durch den Umstand hinzu, dass bei vergessenen „var“-Deklarationen – wenn eine Variable im derzeitigen Gültigkeitsbereich also nicht existiert – die Variable automatisch als Eigenschaft zum globalen Objekt hinzugefügt wird. Faustregel vor allen anderen Patterns ist daher, alle Variablen mit „var“ zu deklarieren, damit dies nicht passieren kann (vgl. [Crockford 2008], S. 101f. und [Stefanov 2010], S. 11).

Um unabsichtliche globale Variablen zu erkennen, sollten unbedingt die Code-Analyse-Tools JSHint bzw. JSLint eingesetzt werden. Weiters kann der spezielle „strict mode“ (ECMAScript ab Version 5) Abhilfe schaffen. Dort tritt das angesprochene Problem nämlich nicht auf, sondern führt zu einem Laufzeitfehler. Der Modus muss allerdings in jeder JS-Datei durch die Anweisung „use strict;“ explizit aktiviert werden (vgl. [Stefanov 2010], S. 13).

[3] Übersetzung durch den Autor

Ein äußerst bekanntes Pattern um einen eigenen Namensraum zu erstellen, ist das „Namespace Pattern“. Es empfiehlt, dass eine Applikation aus lediglich einen einzigen globalen Objekt bestehen sollte, das somit den Namensraum der Applikation bereitstellt. Alle restlichen Objekte sollen dann als Eigenschaft dieses globalen Objekts ihren Platz finden (vgl. [Stefanov 2010], S. 87f.).

Erfreulicherweise geht die zuvor vorgestellte Bibliothek RequireJS noch einen besseren Weg. Alle mit „define“ festgelegten Module besitzen nämlich automatisch ihren eigenen Namensraum und somit hängt kein einziges Modul direkt am globalen Objekt. Einzig „define“ und „require“ sind dort als Funktionen definiert um die jeweiligen Module zu laden. Das „Namespace Pattern“ und seine vielen Erweiterungen sind bei Verwendung von RequireJS daher nicht relevant und werden im späteren Abschnitt „Patterns“ auf Seite 115 auch nicht berücksichtigt.

6.7.2 Gültigkeitsbereich und „Hoisting“

Im Gegensatz zu den meisten Programmiersprachen verhält sich der Gültigkeitsbereich („scope“) in JS nicht so wie die Syntax eigentlich vermuten lässt. Zwar existiert Syntax um Code in Blöcken mit geschweiften Klammern einzukapseln „{}“, diese beeinflusst den Gültigkeitsbereich aber nicht. Tatsächlich definiert sich ein eingeschlossener Block nur durch das „function“-Schlüsselwort. Es gibt also nur „Function-Scope“ und keinen „Block-Scope“ in JS. (vgl. [Crockford 2008], S. 102). Beispielsweise sind alle Variablen die innerhalb eines Bedingungsblocks oder einer Schleife deklariert werden, tatsächlich innerhalb der gesamten Funktion verfügbar und nicht nur im Block zwischen den geschweiften Klammern.

Eine der gängigen Best Practices beim Programmieren, Variablen erst dort zu deklarieren, wo sie als Erstes gebraucht werden, muss bei JS absolut vermieden werden. Das „var“-Schlüsselwort kann innerhalb einer JS-Funktion zwar überall verwendet werden, zur Laufzeit verhalten sich aber alle eingeschlossenen Variablen innerhalb einer Funktion jedoch so, als ob sie bereits am Anfang

der Funktion deklariert wurden. Aufgrund dieses – unter dem Namen „Hoisting“ bekannten – Verhaltens sollten alle Variablen-Deklarationen in einer Funktion immer als Erstes erfolgen, auch wenn dies Veränderungen im Design der Applikation erfordert (vgl. [Stefanov 2010], S. 14 und [Crockford 2008], S. 102). Fehlfunktionen, die durch dieses – im Quellcode quasi unsichtbare – Verhalten verursacht werden, sind nämlich besonders schwer auffindbar.

Der eigentliche „Function-Scope“ in JS ist aber nicht nur ungewöhnlich, sondern auch sehr nützlich, da sich durch ihn Privatsphäre schaffen lasst! Ein sehr gebräuchlicher Anwendungsfall sind selbst-ausführende Funktionen („immediate functions“) die im Abschnitt 6.8.1 auf Seite 116 genauer vorgestellt werden. Das Wissen um den Gültigkeitsbereich in JS, sollte daher jedenfalls immer im Hinterkopf behalten werden.

6.7.3 Automatisches Setzen von Strichpunkten

In JS werden Strichpunkte verwendet um den Abschluss eines Statements zu markieren. Dies sollte auch unbedingt geschehen und die Code-Analyse-Tools JSHint und JSLint prüfen dies automatisch. Bedauerlicherweise funktioniert Quellcode ohne Strichpunkte aber auch. Der Standard ECMAScript definiert dafür einen Mechansimus, der die Strichpunkte automatisch nachsetzt, leider werden aber viele Spezialfälle nicht berücksichtigt! So kann es passieren, dass ein Strichpunkt einige Zeilen zuvor gesetzt wird, obwohl er erst weiter unten tatsächlich gewünscht ist. Statements ohne Strichpunkte sind also jedenfalls zu vermeiden (vgl. [Crockford 2008], S. 102).

6.7.4 Prüfung des (primitiven) Typs

Die Überprüfung ob beispielsweise ein bestimmter Wert vom primitiven Typ „number“ ist und einen gültigen Wert (nicht „NaN“ („Not A Number“) oder „Infinity“) besitzt, ist leider weit schwieriger als erwartet. Crockford empfiehlt für viele

derartige Prüfungen, eine eigene Funktion zu erstellen um einige Inkonsistenzen im Sprachdesign zu begegnen. Er definiert beispielsweise die nachfolgende Bedingung für Array: „if(Object.prototype.toString.apply(theVariableToCheck) === '[object Array]')“ (vgl. [Crockford 2008], S. 105f.). Aus der Sicht des Autors, ist dies bei Weitem zu kompliziert und diese Probleme sollten gar nicht mehr selbst gelöst werden.

Es haben sich mehrere Bibliotheken gebildet, die als „Werkzeug-Gürtel“ („utility-belts“) bei der Entwicklung mit JS dienen sollen. Das ursprüngliche Ziel, war dabei eine funktionale Programmierung mit JS zu erleichtern. Mittlerweile finden sich dort auch viele andere Hilfs-Funktionen, wie eben jene zur konsistenten Prüfung des Typs. Ursprünglich etablierte sich die Bibliothek Underscore, heutzutage sollte sie aber durch die Bibliothek Lo-Dash, die schneller und gleichzeitig aber vollkommen mit den Schnittstellen von Underscore kompatibel ist, abgelöst werden (vgl. [Fleming 2013] und [Dalton 2013]). Die Bibliothek Lo-Dash wurde in Physiogame von Anfang an verwendet, da sie im Gegensatz zu Underscore AMD-kompatibel ist und damit einfacher mit RequireJS zusammenspielt.

6.7.5 Objekte, Arrays und primitive Wrapper

Die Erstellung von Objekten mit dem eigenen Literal „{}“ ist nicht nur möglich, sondern auch absolut empfohlen. Die Verhaltensweise der nativen „Constructor“-Funktion von „Object“ verhält sich bei unterschiedlich übergebenen Typen nämlich inkonsistent. Beispielsweise führt das Statement „var a = new Object(1);“ nicht zu einem Objekt, sondern tatsächlich zum nativen Wrapper-Objekt „Number“ (Großbuchstabe!) und erhält sämtliche Eigenschaften von „Number“. Das „new“-Schlüsselwort sollte jedenfalls nur zur Erstellung von Objekten aus eigenen „Constructor“-Funktionen verwendet werden (vgl. [Stefanov 2010], S. 41f.).

Die primitiven Typen „number“, „boolean“, „string“ besitzen jeweils auch eine Repräsentation als Objekt, sogenannte Wrapper. Dies bietet prinzipiell einige Vorteile. Auf String-Objekte können dadurch beispielsweise Methoden

wie „toLowerCase“ ausgeführt werden. Statements wie „’Test’.toLowerCase()“ funktionieren aber ebenso ohne dass explizit ein Wrapper vorher mit „new String(’Test’)“ initialisiert werden muss. Die primitiven Typen werden hierzu temporär zur Ausführung der Wrapper-Methode konvertiert. Aus diesem Grund sind die primitiven Typen daher absolut zu bevorzugen. Ein großer Nachteil von Wrappern zeigt sich bei der Typprüfung: Es wird nicht mehr „number“, „string“ oder „boolean“ zurückgeliefert, sondern immer „object“ (vgl. [Stefanov 2010], S. 52f.).

Ähnlich wie Objekte sollten auch Arrays nur mit ihrem eigenen Literal „[]“ initialisiert werden sollte. Das Verhalten des „Array-Constructors“ ist nämlich inkonsistent: Während das Statement „new Array(3,2);“, wie erwartet zu einem Array mit den zwei primitiven „numbers“ „3“ und „2“ führt, führt „new Array(3);“ zu etwas Unerwarteten: Ein Array mit drei Elementen, die alle „undefined“ sind (vgl. [Stefanov 2010], S. 46f.).

6.8 Patterns

Der weise Einsatz von Patterns ist in JS von enormer Bedeutung. Innerhalb dieses Abschnittes werden jene vorgestellt, die sich im Rahmen der Entwicklung der Applikation Physiogame als besonders nützlich erwiesen und bei der Entwicklung von Applikationen mit JS daher jedenfalls eine Rolle spielen werden.

Die Vermeidung des globalen Namensraums bzw. ein „Namespace“ Pattern ist übrigens durch den Einsatz von RequireJS nicht mehr erforderlich, da dies bereits mit „define“ und „require“ geregelt wird. Patterns in den nachfolgenden Unterabschnitten beschränken sich folglich auf die Struktur von einzelnen Objekten, während sich RequireJS um die Modularität kümmert.

6.8.1 Selbstausführende Funktionen

Die Betrachtung der sogenannten „immediate functions“ ist alleine schon deswegen wichtig, da JS zwischen „Function Statements“ und „Function Expressions“ unterscheidet. Statements werden zur Deklaration von Funktionen verwendet. Mit Expressions werden allerdings Funktionen geformt, die nur einmalig ausgeführt werden und in ihren jeweiligen Gültigkeitsbereich (siehe „Function Scope“ aus Unterabschnitt 6.7.2 auf Seite 112) hervorragend private Objekte einkapseln können. Das Vorhandensein von „Function Expressions“ macht Module in JS erst möglich, wie im nachfolgenden Abschnitt noch weiter ersichtlich wird. Die ähnliche Syntax erschwert jedoch deren Unterscheidung (vgl. [Crockford 2008], S. 113 und S. 40f.).

```
1  // function statements
2  var aStatement = function() {
3    return "Hello World";
4  };
5
6  // function expressions
7  var anExpression = (function() {
8    return "Hello World";
9  }()); // immediate invokation!
10
11 console.log(typeof aStatement); // prints "function"
12 console.log(typeof anExpression); // prints "string"
```

Abbildung 6.20: Unterschiedliche Syntax: „Function Statement“ vs. „Function Expression“

Abbildung 6.20 zeigt wie sich die Syntax bei „Function Statements“ und „Function Expressions“ unterscheidet. Expressions werden im Gegensatz zur normalen Deklaration von Funktionen sofort ausgeführt (siehe „()“ in Zeile neun) und das Objekt „anExpression“ nimmt den eingekapselten Rückgabewert der selbstausgeführten Funktion direkt an. Der Typ des Objekts nun „string“ und nicht länger „function“. Damit die Funktion von der Laufzeitumgebung als Expression gewertet wird, sollte sie in runde Klammern eingekapselt werden (siehe Zeile sieben und neun) (vgl. [Crockford 2008], S. 113).

6.8.2 „Revealing Module Pattern“

Es gibt in JS keine Klassen und für viele Objekte werden diese auch tatsächlich nicht benötigt, da Funktionalität häufig einmalig ist. Ein häufiges Pattern – in anderen Programmiersprachen – ist das Singleton-Pattern, das eingesetzt wird, damit nur ein Objekt aus einer Klasse tatsächlich instanziert werden kann. Dieses Pattern ist in JS jedoch gar nicht nötig, stattdessen sollten einfach direkt Module eingesetzt werden. Ein Modul ist im Grunde nichts anderes als ein Objekt, das Schnittstellen bereitstellt, seine interne Funktionalität, bzw. seine privaten Variablen aber verbirgt (vgl. [Crockford 2008], S. 40 und [Stefanov 2010], S. 99f.).

Eine abgewandelte Variante des „Revealing Module Patterns“ wurde bereits bei der Implementierung der Beispiel-Applikation für RequireJS eingesetzt. Hier wurde im „define“-Aufruf eine Funktion mitgegeben, die bei ihrer Ausführung das Modul initialisiert und setzt. Zur Vollständigkeit wird an dieser Stelle nun die generische Variante präsentiert, damit das Pattern besser verstanden werden kann. Die originale Variante setzt direkt auf eine selbstausführende Funktion um Privatsphäre im Modul herzustellen, im Gegensatz dazu verwendet RequireJS ein normales Funktion-Statement, da die Ausführung der Funktion erst erfolgen darf, sobald die Abhängigkeiten des Moduls vollständig erfüllt wurden.

Abbildung 6.21 auf der nächsten Seite zeigt wie das Pattern aussieht. Alle Deklarationen innerhalb der „Function Expression“ (zwischen Zeile zwei und 13) sind privat und nicht von außen sichtbar, erst durch „return“ wird die öffentliche Schnittstelle des Moduls zurückgegeben. Die Variablen „secret“ und „unlocked“ bleiben im Gültigkeitsbereich der Funktion privat verfügbar, lediglich die Methode „open“ wird öffentlich verfügbar gemacht. Auch ein Überschreiben der Methode würde keinen Zugriff auf die privaten Eigenschaften ermöglichen!

```
var myLocker = (function () {

  // everything is private in here!
  var secret = "1234",
    unlocked = false;

  function open(knownSecret) {
    if(knownSecret === secret) {
      unlocked = true;
    }
  }

  // revealing to the public takes place through the return!
  return {
    open: open
  };
}());
```

Abbildung 6.21: „Revealing Module Pattern“: Beispiel

6.8.3 „Combination Constructor/Prototype Pattern“

Gewisse Objekte müssen aus einer Schablone erstellt werden und JS bietet natürlich auch Möglichkeiten dafür, nur eben keine Klassen. Stattdessen müssen „Constructor“-Funktionen eingesetzt werden, die mit dem „new“-Schlüssel aufgerufen ein neues Objekt initialisieren. Ein populäres Pattern hierfür ist das sogenannte „Combination Constructor/Prototype Pattern“. Es sagt, dass die eigentliche „Constructor“-Funktion von den Methoden des Prototypen getrennt definiert werden soll (vgl. [Stanley 2013]).

```
function Person(name) {
  this.name = name;
}

Person.prototype = {
  constructor: Person,
  getHelloString: function () {
    return "Hello " + this.name;
  }
};

var testPerson = new Person("Mario");
testPerson.getHelloString(); // prints "Hello Mario"
```

Abbildung 6.22: „Combination Constructor/Prototype Pattern“: Beispiel

Abbildung 6.22 zeigt den Einsatz dieses Patterns mit der „Constructor“-Function

„Person“. Die Eigenschaften des zukünftigen Objekts werden dabei in der Funktion gesetzt, während die Methoden direkt dem Prototypen der Funktion angehängt werden. Zeile zwölf zeigt die Instanzierung eines derartigen Objekts.

Wie ersichtlich, wird durch dieses Pattern ein sehr klassenähnliches Verhalten – wie in der üblichen objektorientierten Programmierung – möglich. Auch Vererbungsketten, ähnlich der Sub-Klassen in anderen Programmiersprachen sind möglich. Hierfür würde sich ein weiteres bekanntes Pattern „Parasitic Combination Inheritance Pattern“ eignen. Auf dieses wird an dieser Stelle jedoch nicht eingegangen, da es in Physiogame keine Anwendung fand und tiefe Vererbungsbäume laut den „Gang of Four“ – unabhängig von der eingesetzten Programmiersprache – sowieso vermieden werden sollten (vgl. [Stanley 2013] und [Stefanov 2010], S. 4).

6.9 Zusammenfassung

Im Rahmen dieses Kapitels wurden sowohl eine Einblicke in den konzeptionellen Aufbau von Physiogame vermittelt als auch prinzipielle Best Practices für die Entwicklung von JS-Applikationen gegeben. Die Entwicklung von Applikationen mit JS mag weit komplizierter als mit anderen Programmiersprachen erscheinen. Viele Anforderungen müssen selbst oder durch externe Bibliotheken gelöst werden, bzw. es stehen für jede Anforderung generell immer mehrere Lösungswege zur Verfügung. Andererseits kann diese Entscheidungsfreiheit auch positiv angesehen werden, da sich somit eine speziell auf die Erfordernisse angepasste Anwendung schnüren lässt.

Die nachfolgende Auflistung zeigt überblicksartig welche Entscheidungen bei der Applikation Physiogame getroffen und im Rahmen der letzten Abschnitte genauer beleuchtet wurden:

- Physiogame wird in der IDE Sublime Text 2 entwickelt, die sich durch zahlreiche Plugins erweitern lässt.

- Externe Abhängigkeiten zu anderen Bibliotheken werden über zwei Paketmanagement-Tools (Bower und NPM) verwaltet.
- Zum Testen der Korrektheit der Syntax kommt JSHint zum Einsatz.
- Die Funktionalität der Applikation kann durch eine Test-Umgebung, die aus mehreren Komponenten besteht, überprüft werden.
- Durch RequireJS kann Physiogame sowohl modular aufgebaut als auch optimiert ausgeliefert werden.
- Wiederkehrende Aufgaben im Entwicklungsprozess werden mit Grunt-Tasks automatisiert.

Außerdem erörtert wurden die Fallen, Patterns und Code-Konventionen in JS, deren Verständnis als eine Voraussetzung zur Entwicklung von JS-Applikationen angesehen werden kann. Insgesamt kann daher gesagt werden, dass in JS sicherlich mehrere Hürden – als in anderen Programmiersprachen – anfänglich im Wege stehen. Diese sind zwar lösbar, erfordern jedoch viel Zeit.

Da nun ein Konzept zur Entwicklung von Physiogame steht, wird im nächsten Kapitel nun auf die tatsächliche Implementierung der Applikation Physiogame eingegangen.

7 Implementierung der Applikation Physiogame

Innerhalb dieses Kapitels wird die Implementierung der Applikation Physiogame vorgestellt. Es wird dabei auf die Methoden zur Konzeption von JS-Applikationen (aus dem vorherigen Kapitel) zurückgegriffen. Anfänglich wird im nächsten Abschnitt die Ordnerstruktur des Projekts beleuchtet. Anschließend werden im Abschnitt 7.2 auf Seite 124, die wichtigsten Module von Physiogame definiert und der Initialisierungsvorgang, der Aufbau und Zweck der Module bzw. deren Abhängigkeiten zueinander beschrieben. Darauffolgend werden die verwendeten externen Bibliotheken der Applikation im Abschnitt 7.3 auf Seite 132 präsentiert. Schlussendlich wird eine Zusammenfassung des Kapitels gegeben und auf das nächste Kapitel übergeleitet.

7.1 Ordnerstruktur

Abbildung 7.1 auf der nächsten Seite zeigt die Ordnerstruktur des Projekts auf. Der Ordner „physiogame“ dient dabei das Wurzelverzeichnis und hält sowohl Dateien (als Spickzettel abgebildet) als auch weitere Unterordner (als Akte visualisiert). Die Farbe orange steht für Dateien bzw. Ordner, die tatsächlich für die Ausführung der Applikation relevant sind. Die Ressourcen zum Testen der Applikation sind grün markiert. Die Farbe blau steht für Dateien und Ordner, die für den Build-Prozess wichtig sind, bzw. für Ordner in den die – produktiven und mit RequireJS optimierten – Applikationen (für Web und Node-webkit) exportiert werden. In den nachfolgenden Absätzen wird der Rolle bzw. Aufgabe

dieser Ordner und Dateien noch genauer beschrieben.

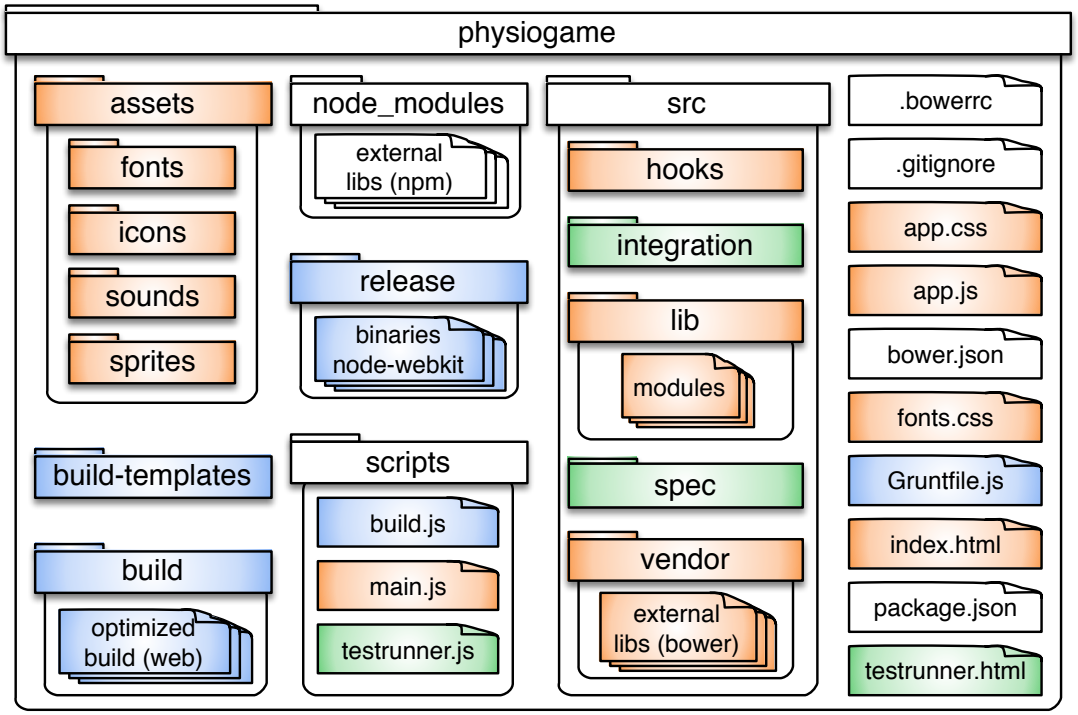

Abbildung 7.1: Physiogame Implementierung: Ordnerstruktur

Die Mediendateien – Schriftarten, Icons, Sounds, und Grafiken – von Physiogame finden sich gesammelt im Ordner „assets“. Hier finden sich sowohl die Quelldateien (zur Erstellung) als auch die produktiv einzusetzenden Dateien. Im Rahmen des Build-Tasks mit Grunt, werden übrigens nur die produktiv einzusetzenden Dateien in den „build“-Ordner kopiert. Die Medien im Ordner „assets/icons“ sind dabei nur für die Desktop-Applikation (mit Node-webkit) relevant.

Der Build-Task mit Grunt exportiert eine optimierte und auslieferbare Web-Applikation (zur Bereitstellung auf Web-Servern) in den „build“-Ordner. Zudem gibt es einen „release“-Task, der im Ordner „release“ die gepackten Desktop-Applikationen mit Node-webkit – für Windows, Mac und Linux (für 32 und 64 Bit Architekturen) erstellt. Die Tasks sind dabei im „Gruntfile.js“ im Wurzelverzeichnis definiert. Der Ordner „build-templates“ beinhaltet außerdem einige Vorlagen von Dateien, die bei der produktiven Variante von Physiogame eingesetzt werden müssen und werden ähnlich wie „assets“ in den „build“-Ordner kopiert.

Der Ordner „node_modules“ beinhaltet die (hauptsächlich) für den Entwicklungsprozess relevanten externen Bibliotheken, die mit dem Tool NPM verwaltet werden. Abhängigkeiten, wie beispielsweise zu Grunt, PhantomJS, Mocha, Chai und auch zum Paketmanager Bower (für die Web-Bibliotheken) sind in der Datei „package.json“ definiert und werden vom NPM in diesen Ordner installiert. Bibliotheken, die die Applikation Physiogame direkt benötigt und ausführt, werden mit Bower verwaltet und sind im Ordner „src/vendor“ installiert. Sie sind in der Datei „bower.json“ definiert, während die Datei „.bowerrc“ dazu dient den Installationspfad explizit auf „src/vendor“ zu setzen.

RequireJS wird in Physiogame sowohl zum Ausführen der Applikation als auch zum Testen und zur Optimierung eingesetzt. Im Ordner „scripts“ befinden sich daher die RequireJS-Konfigurationen für diese unterschiedlichen Kontexte. Die Datei „main.js“ dient als Hauptkonfiguration und wird primär zur Ausführung der Applikation eingesetzt. Die Datei „build.js“ referenziert auf die Hauptkonfiguration, setzt allerdings noch Einstellungen für den Optimierungsvorgang der gesamten Applikation und wird im Rahmen des Build-Tasks von Grunt ausgeführt. Schlussendlich hält die Datei „testrunner.js“, die zusätzlichen Konfigurationswerte für RequireJS zum Testen der Applikation. Sie wird von der Datei „testrunner.html“ eingebunden, die sowohl im Browser als auch in PhantomJS zum Starten der Unit- und Integration-Tests ausgeführt werden kann.

Im „src“-Ordner befindet sich ausschließlich Quellcode, der für die Ausführung von Physiogame bzw. zum Testen relevant ist. Der Unterordner „src/hooks“ hält dabei speziellen Quellcode, der nicht mit RequireJS eingebunden werden darf. Dies ist bei Physiogame einzig die Bibliothek Modernizr, die vor der Initialisierung von Physiogame zur Kompatibilitätsprüfung der Zielplattform eingesetzt wird und deren Optimierung mit einem eigenen Grunt-Task automatisiert wird. Die Unterordner „src/integration“ und „src/spec“ enthalten die Integration- bzw. Unit-Tests von Physiogame. Unter „src/lib“ befindet sich der eigentliche Programm-Code der Applikation. Hier befinden sich also die eigentlichen Module der Applikation (RequireJS), die – durch weitere Unterordner – in Pakete strukturiert werden. Die wichtigsten Module und die Struktur dieser, wird im

Abschnitt 7.2 beleuchtet.

Git wird als System zur Versionsverwaltung in Physiogame eingesetzt. Es ist jedoch nicht nötig alle Dateien und Ordner im Repository zu haben, sondern lediglich jene, die zur Erstellung des Projekts erforderlich sind. Die Ordner und Dateien in „build“, „node_modules“, „release“ und „src/vendor“ können ausgeschlossen werden, da sie mit NPM, Bower und den Grunt-Tasks automatisiert installiert bzw. erstellt werden. Somit befinden sich nur der tatsächliche Quellcode und die Konfigurationsdateien im Git-Repository.

Die Datei „index.html“ dient als Einsprungpunkt um die Applikation im Browser bzw. in der Desktop-Applikation (Node-webkit) zu laden. In der HTML-Datei sind das Startskript „app.js“ und die CSS-Dateien „app.css“, „fonts.css“ und andere externe CSS-Files (aus „src/vendor“) direkt eingebunden. Der Initialisierungsablauf wird im nächsten Abschnitt genauer behandelt.

7.2 Initialisierung, Module und Abhängigkeiten

Abbildung 7.2 auf der nächsten Seite visualisiert die wichtigsten Module und Pakete von Physiogame und weiters zeigt den Initialisierungsvorgang und die Abhängigkeiten zwischen den Modulen auf.

Die Farbe orange steht wiederum für die Ordner und Dateien, die für die Ausführung der Applikation wichtig sind. Die Pakete („Packages“) der Applikation, in denen die Module hausen, werden als Quader dargestellt. Es werden zwei grundlegend verschiedene Typen von Pfeilen in der Visualisierung verwendet:

- Pfeile mit unterbrochener Linie und gefüllter Spitze zeigen auf ein Zielmodul (die Abhängigkeit), das vom Ausgangsmodul benötigt wird.
- Pfeile mit durchgezogenen Strich und hohler Spitze sind (HTTP-)Requests von Ressourcen, die im Rahmen der Initialisierung benötigt werden.

Häufig verwendete Module innerhalb der Applikation sind mit der Farbe pink markiert. Die „Pfeile“ der Module, die „pinke Abhängigkeiten“ benötigen, wur-

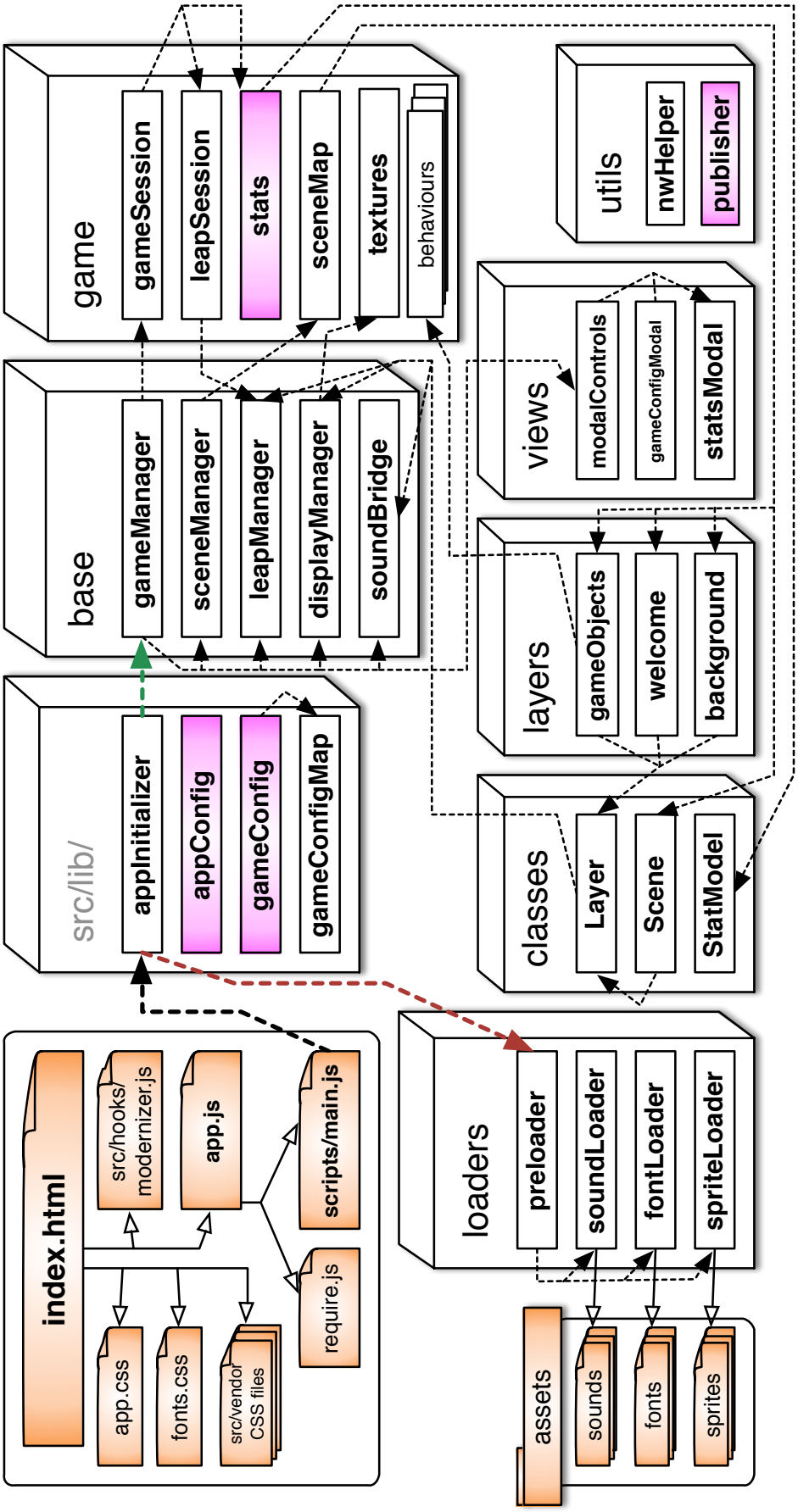

Abbildung 7.2: Physiogame Implementierung: Module und Initialisierung

den nicht visualisiert. Alle dargestellten Module sind besonders wichtig für die Applikation und wurden aus insgesamt rund 80 eigenen Modulen ausgewählt. Externe Bibliotheken (außer RequireJS und Modernizr) werden allerdings nicht abgebildet, die Vorstellung dieser erfolgt erst im Abschnitt 7.3 auf Seite 132.

Physiogame besteht aus einem namenlosen Wurzelpaket (der Order „src/lib“). Dies beherbergt neben den Modulen im „Wurzelpaket“, die anderen visualisierten sieben Sub-Pakete, wie beispielsweise „base“, „game“ oder „loaders“. Alle Module in den Paketen wurden mit der „define“-Anweisung von RequireJS definiert und verwenden – bis auf wenige Ausnahmen – die Struktur des „Revealing Module Patterns“ (siehe Abschnitt 6.8.2 auf Seite 117 im vorherigen Kapitel). In den nachfolgenden Unterabschnitten wird die Initialisierung und danach die Struktur der Module bzw. Abhängigkeiten auf Basis der vorgestellten Abbildung 7.3 auf Seite 132 genauer beschrieben. Schlussendlich findet eine Reflexion des Aufbaus statt.

7.2.1 Initialisierung

Als Einsprungpunkt dient die Datei „index.html“. Hier werden die Stylesheets (die CSS-Dateien der Applikation geladen), die Bibliothek Modernizr und das Startskript „app.js“ ausgeführt. Der „body“-Tag von „index.html“ besteht übrigens lediglich aus drei Containern: Einer für die Anzeige der Applikation (hier wird das „canvas“-Element eingefügt), einer für die Dialog-Fenster („modals“) und einer um Ladeinformationen (fürs „preloading“) anzuzeigen.

Die Datei „app.js“ wird also geladen und es erfolgt eine Kompatibilitätsprüfung. Hier wird festgestellt, ob die Ausführungsplattform beispielsweise „canvas“ und WebGL unterstützt. Falls die Applikation mit Node-webkit ausgeführt wird, werden nachfolgend zudem Einstellungen getroffen, damit es mit RequireJS nicht zu einem Namenskonflikt mit dem „require“-Statement von Node.js kommt.

Wenn die Mindestvoraussetzungen erfüllt wurden, kann die Bibliothek RequireJS durch das Einfügen eines dynamischen Skript-Tags in das „document“-

Objekt via JS geladen und das Haupt-Konfigurationsskript für RequireJS – „main.js“ aus dem Ordner „scripts“ – mitgegeben werden. Ab diesen Zeitpunkt übernimmt RequireJS die weitere Initialisierung von Physiogame.

Innerhalb des Haupt-Konfigurationsskript wurde das Modul „appInitializer“ „required“. Dieses Modul startet den Ladevorgang des „preloader“-Moduls (roter Pfeil). Das „preloader“-Modul überwacht den Ladevorgang der Submodule „soundLoader“, „fontLoader“ und „spriteLoader“. Diese laden die jeweiligen Mediendateien aus dem Ordner „assets“. Welche Dateien geladen werden sollen, ist dabei im Moduls „appConfig“ definiert. Erst nachdem der Ladevorgang abgeschlossen wurde, erhält das Modul „appInitializer“ ein Signal („event“) von „preloader“ und das Modul „gameManager“ – der Haupt-Controller der Applikation – kann dynamisch „required“ werden (grüner Pfeil).

Das Modul „gameManager“ initialisiert alle anderen Module im Paket „base“. Jedes dieser Module ist eine elementare Sub-Komponente der Applikation und für eine bestimmte Funktion der Applikation zuständig. Sobald diese Initialisierung erfolgreich durchgeführt wurde, ist die Applikation tatsächlich einsatzbereit. Die weitere Erklärung der restlichen Module und Abhängigkeiten findet nun im nächsten Abschnitt statt, da die weitere Initialisierungsreihenfolge der Module ab diesem Zeitpunkt keine Rolle mehr spielt und von RequireJS verwaltet wird.

7.2.2 Module und Abhängigkeiten

In den nachfolgenden Unterabschnitten werden die – in der vorherigen Abbildung 7.2 auf Seite 125 dargestellten – Module, nach ihrer Struktur in den Paketen („Packages“) vorgestellt und ihre Abhängigkeiten analysiert.

7.2.2.1 Wurzelpaket

Neben den Modul „appInitializer“ befinden sich im Wurzelpaket (unter „src/lib/“) noch drei wichtige Module. Alle beeinflussen die Konfiguration von Physioga-

me. Die Konfigurationswerte im Modul „appConfig“ sind dabei direkt gesetzt und können auch produktiv nicht geändert werden. Hier sind die zu ladenden Mediendateien und einige Parameter der Anzeige – wie zum Beispiel die native Auflösung des Canvas-Elements, das Seitenverhältnis und der zu verwendende Container im „index.html“-Dokument – definiert.

Das Modul „gameConfig“ stellt die Daten-Struktur und eine Event-Schnittstelle zur interaktiven Konfiguration der Applikation bereit. Es verwendet ein sogenanntes „Model“ aus der externen Bibliothek Backbone. Die Eigenschaften aller Konfigurationselemente werden außerhalb im Modul „gameConfigMap“ definiert. Ein Element hat hier beispielsweise die Eigenschaften ID, Standard-Wert, Bezeichnung, (Maß)-Einheit, Minimum- und Maximum-Wert. Die aktuellen Werte der Konfigurationsparameter finden sich dann jedoch in „gameConfig“. Über die Datenstruktur können diese weiters in einen Cache gespeichert und wieder ausgelesen werden kann.

Sowohl das Modul „appConfig“ als auch „gameConfig“, wird in Physiogame häufig als Abhängigkeit benötigt (Farbe pink in Abbildung 7.2 auf Seite 125). Die Kommunikation mit „gameConfig“ ist vor allem deswegen relevant, da nach Wertänderungen ein Event gesendet wird und andere Module somit auf interaktive Konfigurationsänderungen reagieren können. Durch diese Module werden also die – im Kapitel 4 auf Seite 50 spezifizierten – Anforderungen zu Physiogame nach einer interaktiven Konfiguration erfüllt.

7.2.2.2 Paket „base“

Das Paket „base“ beinhaltet Module – sogenannte „Controller“ – die wesentliche Teile der Applikation steuern. Das Modul „gameManager“ dient dabei als Schaltzentrale zwischen diesen „Controllern“ und kommuniziert zusätzlich mit dem Modul „gameSession“ aus dem Paket „game“ um eine Spielrunde zu starten bzw. zu stoppen. Das Modul „leapManager“ kommuniziert mit dem Leap Motion Controller und führt alle nötigen Berechnungen (pro „Frame“-Objekt) durch.

Das Modul „displayManager“ erzeugt und verwaltet die Anzeige von Physiogame (das „canvas“-Element). Es initialisiert zudem das Modul „textures“, das danach alle geladenen Bild-Mediendateien als Texturen (der Bibliothek Pixi) in einem Altas zur Verfügung stellen kann. Auf der Anzeige werden Szenen („scenes“) – die aus verschiedenen Ebenen (aus dem Paket „layers“) bestehen können – gezeichnet, die das Modul „sceneManager“ verwaltet. Die „sceneMap“ hält die IDs der verfügbaren Szenen und definiert aus welchen Ebenen diese bestehen. Das Modul „soundBridge“ dient weiters der Koordinierung aller Audio-Effekte und der Hintergrund-Musik (aus dem Szenen).

7.2.2.3 Paket „game“

Im Paket „game“ befinden sich alle Module, die für die Applikation Physiogame – für das „Spiel“ – spezifisch sind, beispielsweise welche Szenen (in „sceneMap“) und Texturen (in „textures“) zur Verfügung stehen. Weiters ist im Subpaket „behaviours“, das Verhalten der Spielobjekte (die „gameObjects“) definiert.

Besonders wichtig ist das Modul „stats“, welches eine Schnittstelle zum Hinzufügen und Ändern von Statistik-Daten bietet. Die Datenstruktur wird dabei von Modul „StatModel“ gestellt, eine „Constructor“-Funktion zur Erstellung eines neuen Statistik-Datenmodell-Objekts. Die Schnittstellen von „stats“ werden von zahlreichen anderen Modulen eingesetzt, um aktuelle Statistik-Daten zu registrieren. Das Modul ist daher in der Farbe pink, in der Abbildung 7.2 auf Seite 125 dargestellt. Das Modul stellt außerdem Funktionen zum Laden, Speichern und Exportieren von Statistik-Daten zur Verfügung.

Das Modul „gameSession“ steuert die Module „leapSession“ und „stats“. Es eröffnet und schließt Spielrunden. In „leapSession“ werden die Input-Daten vom Leap Motion Controller analysiert und Statistikdaten – wie beispielsweise die zurückgelegte Distanz und die Zeit im Interaktionsbereich – für eine Spielrunde gesammelt.

Das Paket „game“ erfüllt somit mehrere Anforderungen von Physiogame: Es stellt Funktionalität zur Erfassung von Statistik-Daten, zum Spielobjekt-Verhalten

und zum Spielablauf zur Verfügung.

7.2.2.4 Paket „loaders“

Die Module im Paket „loaders“ wurden bereits im Rahmen der Initialisierung behandelt. Die Module „soundLoader“, „fontLoader“ und „spriteLoader“ bieten Funktionalität zum Laden der Mediendateien. Dieser Vorgang wird vom Modul „preloader“ gesteuert.

7.2.2.5 Paket „classes“

Der Name des Pakets „classes“ ist etwas unglücklich gewählt. Es beinhaltet natürlich keine Klassen, sondern Module, die „Constructor“-Funktionen (daher ist der Anfangsbuchstabe groß geschrieben) zur Erstellung von Objekten definieren. Hierzu wird das „Combination Constructor/Prototype“ Pattern eingesetzt, das im Abschnitt 6.8.3 auf Seite 118 des vorherigen Kapitels vorgestellt wurde. Mit der „Constructor“-Funktion in „StatModel“ kann beispielsweise ein neues Objekt erstellt werden, das Statistik-Daten in einer Datenstruktur hält (dies wird normalerweise vom Modul „stats“ im Paket „game“ durchgeführt).

Die Module „Layer“ und „Scene“ sind maßgeblich für die Gestaltung der Anzeige verantwortlich. Sie definieren „Constructor“-Funktionen um Szenen und Ebenen zu erstellen. Ein „Layer-Objekt“ stellt automatisch Methoden zur Kommunikation mit den Modulen „soundBridge“, „displayManager“ und „leapManager“ zur Verfügung. Eine Szene verwaltet die Initialisierung und Deaktivierung aller verwalteten „Layer“-Objekte. Welche Szenen in der Applikation verwendet werden, ist in „sceneMap“ definiert. Die visuelle Gestaltung von Physiogame kann durch diese Module somit strukturiert erfolgen und gleichzeitig sind die Szenen und zugehörigen Ebenen austauschbar.

7.2.2.6 Paket „layers“

Im Paket „layers“ befinden sich die tatsächlichen Ebenen, die zur visuellen Gestaltung von Physiogame verwendet werden. Beispielsweise zeichnet die Ebene „background“ den Hintergrund und die Ebene „welcome“ den Titelbildschirm. Die Ebene „gameObjects“ beherbergt und erstellt alle interaktiven Spielobjekte, während deren Verhalten, durch die Module des Sub-Pakets „behaviours“ (innerhalb des Pakets „game“) gesteuert wird. In einigen Ebenen wird – neben der Gestaltung – aber auch die Spiel-Logik, Feedback und Interaktion (die drei Formen) umgesetzt. Hierfür können die bereitgestellten Schnittstellen von „Layer“ zu den Controllern „leapManager“, „displayManager“ und „soundBridge“ eingesetzt werden.

7.2.2.7 Paket „views“

Unter „views“ finden sich die Anzeige-Dialoge der Einstellungen und Statistiken von Physiogame. Diese werden im Gegensatz zu den anderen Anzeige-Objekten dynamisch mit HTML – durch die externe Bibliothek Handlebars – gerendert. Die aktuellen Werte der Einstellungen und Statistiken werden aus den Modulen „stats“ und „gameConfig“ ausgelesen und in eigene HTML-Vorlagen eingesetzt und ausgegeben.

7.2.2.8 Paket „utils“

Das Paket „utils“ beinhaltet Hilfs-Module für bestimmte Anwendungsfälle. Das Modul „publisher“ ist dabei besonders wichtig. Es stellt ein System zum Senden und Empfangen von Events (über die Bibliothek Backbone) zur Verfügung, das andere Module einsetzen können. Das Modul „nwHelper“ wird nur bei Node-webkit Laufzeitumgebungen initialisiert und stellt „native“ Funktionen, beispielsweise zum Ändern der Fenstergröße von Physiogame zur Verfügung.

7.2.3 Reflexion

Im Rahmen der letzten Abschnitte wurden die wichtigsten Module von Physiogame vorgestellt und deren Abhängigkeiten zueinander erläutert. Außerdem wurde der Initialisierungsvorgang der Applikation beschrieben und angesprochen, wie die festgelegten Anforderungen in den Modulen umgesetzt werden.

Eine zentrale Rolle nimmt dabei das Modul „gameManager“ im Paket „base“ ein, das die anderen Controller im Paket initialisiert und steuert. Sowohl Statistiken (durch das Modul „stats“) können erhoben werden als auch die Konfiguration (via „gameConfig“) kann jederzeit interaktiv verändert werden. Die Gestaltung wird mit Ebenen und Szenen erzeugt, die Zugriff auf bestimmte Schnittstellen der Controller-Module besitzen. Dadurch kann die Spiel-Logik, das Feedback und die Interaktion direkt in den Ebenen („Layers“) umgesetzt werden.

Im nächsten Abschnitt werden nun die verwendeten externen Bibliotheken der Applikation Physiogame vorgestellt und deren Einsatzbereich beleuchtet.

7.3 Externe Bibliotheken

Physiogame verwendet zahlreiche externe Bibliotheken um die benötigte Funktionalität in den Modulen herzustellen. Diese Pakete werden mit Bower verwaltet und durch die RequireJS-Konfigurationsdatei „main.js“ in die Applikation eingebunden. Es handelt sich dabei folglich um jene externen Abhängigkeiten, die auch tatsächlich in der produktiven Applikation – und nicht nur im Rahmen der Test-, Build- oder Verpackungsprozesse – eingesetzt werden. In den nachfolgenden Absätzen wird ein Überblick zu den wichtigsten verwendeten Bibliotheken hergestellt.

Die Bibliothek **Modernizr** wird zur Kompatibilitätsprüfung („Feature Detection“) eingesetzt. Hierdurch kann bei inkompatiblen Ausführungsumgebungen eine Fehlermeldung angezeigt werden, bevor die Applikation initialisiert wird. Die

Überprüfungen finden daher bereits in „app.js“ statt. Als Mindestvoraussetzung zum Ausführen von Physiogame muss die Ausgabe von Audio (in den Formaten „ogg“, „mp3“ oder „m4a“), die Erstellung eines Anzeige-Objekts („canvas“ bzw. „webgl“) und das temporäre Speichern („localStorage“) möglich sein.

Lo-Dash ist ein sogenannter „Werkzeug-Gürtel“ („utility-belt“) und stellt einige hilfreiche Funktionen zur Typ-Prüfung und zur funktionalen Programmierung zur Verfügung. **Lo-Dash** (oder als Alternative, die weniger performante Bibliothek Underscore) ist Voraussetzung um die Bibliothek **Backbone** einsetzen zu können. Diese bringt die Funktionalität um Daten-Strukturen („Models“) zu definieren und liefert außerdem ein Event-System, um zwischen den Modulen zu kommunizieren. Sie spielt daher in Physiogame eine äußerst wichtige Rolle. Beispielsweise verwenden sowohl „appConfig“, „gameConfig“ und „StatModel“ die „Models“ von Backbone. Die Konvertierung dieser Daten in das JSON-Format (und daraus über eine eigene Implementierung auch in das CSV-Format) werden vereinfacht. Die enthaltenen Daten (Statistiken und Konfiguration) können zudem mit dem Plugin **Backbone.localStorage** direkt im Cache der Ausführungsumgebung gespeichert werden.

Pixi wird als „2D-Renderer“ in Physiogame verwendet um die Szenen, Ebenen und Spielobjekte innerhalb des „canvas“-Elements zu zeichnen. Die Bibliothek hat eine ähnliche API wie Adobe Flash und kann außerdem GPU-beschleunigt arbeiten. Hierzu lagert sie die Berechnungen nach WebGL aus, falls dies von der Laufzeitumgebung unterstützt wird. Außerdem bietet die Bibliothek einen „Asset-Loader“, der im Modul „spriteLoader“ zum Laden der Bild-Dateien eingesetzt wird. Es wird dabei auch das Einlesen von „Spritesheets“ unterstützt, um mit Pixi Animationen umzusetzen, die aus mehreren Bildern („Sprites“) bestehen.

Zum Abspielen von Sound-Effekten wird die Bibliothek **Howler** verwendet. Sie schafft Kompatibilität zu älteren Browser-Umgebungen, vereinfacht den Ladevorgang von Audio-Dateien (im Modul „soundLoader“) und wird innerhalb des Moduls „soundBridge“ zum Starten und Stoppen aller Audio-Effekte ein-

gesetzt. Um die benötigten Schriftarten der Applikation vor Initialisierung zu laden, wird die Bibliothek **Webfontloader** (innerhalb des Moduls „fontLoader“) eingesetzt.

Die Dialoge für Einstellungen („gameConfigModal“) und Statistiken („statsModal“) werden direkt via HTML gerendert. Mit der Bibliothek **Handlebars** können HTML-Vorlagen für diese Dialoge geschrieben werden und in diese werden – bei Ausführung der Applikation – die aktuellen Konfigurations- oder Statistik-Werten automatisch eingesetzt. Die Funktionalität, damit die Dialoge auf Änderungen reagieren können, kommt dabei von den „Views“ der Bibliothek Backbone. Bei einer Veränderung der Daten in den Daten-Strukturen (Konfiguration oder Statistiken) wird die „View“ automatisch neu gerendert. Für die visuelle Gestaltung aller Dialoge kommen die Bibliotheken **Bootstrap** und **jQuery** zum Einsatz. Diese Dialoge werden also separat von Pixi gerendert.

Um den Leap Motion Controller zu steuern und Input-Daten („Frame“-Objekte) zu erhalten, wird **LeapJS** – die offizielle JS-Bibliothek von Leap Motion – verwendet. Die Kommunikation erfolgt hierbei direkt über WebSockets. Im Modul „leapManager“ werden die „Frame“-Objekte folglich analysiert und erfolgte Interaktionen an die Ebenen (einer aktuell angezeigten Szene) weitergeleitet. Input-Daten, die zur Erstellung der (Bewegungs-)Statistiken von Bedeutung sind, werden vom Modul „leapSession“ weiters durchgeführt.

Die Logging-Funktionen in JS – beispielsweise mit „console.log“ – sind eingeschränkt. Die Bibliothek **Loglevel** fügt verschiedene Wichtigkeitsstufen („levels“) hinzu und erlaubt, dass Debug-Nachrichten im produktiven Einsatz ausgeschalten werden können. Diese Bibliothek wird daher in zahlreichen Modulen der Applikation verwendet.

7.4 Zusammenfassung

Im Rahmen dieses Kapitels wurde die Implementierung der Applikation Physiogame vorgestellt. Dies umfasste Details zur Ordnerstruktur (Anwendungs-, Test-, Build- und Konfigurationsdateien), zum Ablauf der Initialisierung und zur Funktionalität der Module. Außerdem kommen einige externe Bibliotheken in Physiogame (abseits der Bibliotheken zum Build- und Testprozess) zum Einsatz, deren Zweck im vorherigen Abschnitt beleuchtet wurde.

Die Entwicklung von Physiogame mit JS kann – aus Sicht des Autors – insgesamt als äußerst komplex angesehen werden. Ohne die vorgestellten Methoden zur Konzeption derartiger JS-Applikationen aus dem Kapitel 6 auf Seite 80 wäre eine Implementierung wahrscheinlich unmöglich gewesen. Insbesondere die Bibliothek RequireJS erwies sich als unverzichtbar, um die Struktur (Pakete und Module) und Abhängigkeiten von Physiogame auszudrücken.

Im nachfolgenden Kapitel wird die umgesetzte Applikation Physiogame präsentiert.

8 Vorstellung der Applikation Physiogame

Alle definierten Anforderungen an die Applikation Physiogame konnten umgesetzt werden und im Rahmen dieses Kapitels wird nun die finale Version präsentiert. Im nächsten Abschnitt wird der Ladevorgang der Applikation (speziell im Browser) betrachtet. Vom Titelbildschirm aus kann auf alle Funktionen von Physiogame zugegriffen werden. Er wird im Abschnitt 8.2 auf Seite 138 beleuchtet. Anschließend wird der Spielablauf im Abschnitt 8.3 auf Seite 140 vorgestellt und danach noch genauer auf die Interaktion und auf das Feedback (im Speziellen bei der Interaktion mit Leap Motion) im Abschnitt 8.4 auf Seite 144 eingegangen.

Dialoge sind in Physiogame besonders wichtig, da mit ihnen die Einstellungen und Statistiken umgesetzt wurden. Sie werden daher im Abschnitt 8.5 auf Seite 145 vorgestellt und in den Unterabschnitten noch genauer beschrieben. Abschließend wird die Umsetzung zusammenfassend betrachtet.

8.1 Ladevorgang

Abbildung 8.1 auf der nächsten Seite zeigt den Ladevorgang (die Initialisierung) von Physiogame. Das Laden der kompletten Applikation dauert in der Regel ca. fünf Sekunden. Bei der erstmaligen Ausführung der Applikation im Browser hängt die Ladezeit jedoch stark von der Internet-Verbindung ab, da erst alle Mediendateien heruntergeladen werden müssen, bevor diese „gecached“ werden können. Um Feedback zu geben, wird daher eine Ladeanimation

oben in der Mitte angezeigt. Jeder einzelne „Punkt“ in der Zeile unter „Lade Assets“ steht dabei für eine erfolgreich geladene Mediendatei (Bild, Audio oder Schriftart). Nachdem alle Dateien geladen wurden, wird die Applikation initialisiert (Modul „appInitializer“ initialisiert den „gameManager“) und der Titelbildschirm von Physiogame erscheint.

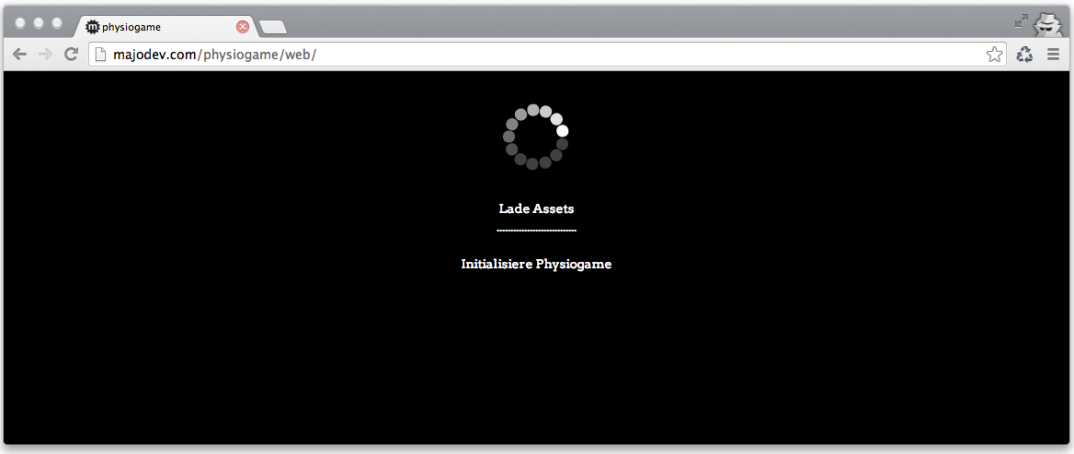

Abbildung 8.1: Physiogame: Ladevorgang (Web-Applikation via Google Chrome)

Physiogame verwendet zahlreiche neue Features in HTML5 und JS, benötigt daher zur Ausführung einen möglichst aktuellen Browser. Ältere Browser, wie beispielsweise Microsoft Internet Explorer 8.0, werden leider nicht unterstützt. In diesen Fällen schlägt die Überprüfung der Mindestvoraussetzungen – via Bibliothek Modernizr – fehl, wie in Abbildung 8.2 auf der nächsten Seite ersichtlich. In diesem Fall wurde eine ältere Version von Microsoft Internet Explorer emuliert, um das Verhalten vorzuzeigen.

Neben der Meldung, dass die Umgebung veraltet ist, wird ein Link zum Herunterladen von Google Chrome bereitgestellt. Dies kann für unerfahrene Benutzer äußerst wichtig sein, da sie dieses Problem durch die Installation eines anderen Browsers eventuell selbstständig beheben können. Außerdem wird ein Status-Report ausgegeben. Dieser enthält genauere Fehler-Informationen und zeigt auf, welche Features der aktuelle Browser nicht unterstützt. Hier unterstützt der Browser beispielsweise lediglich „LocalStorage“ (zum Speichern

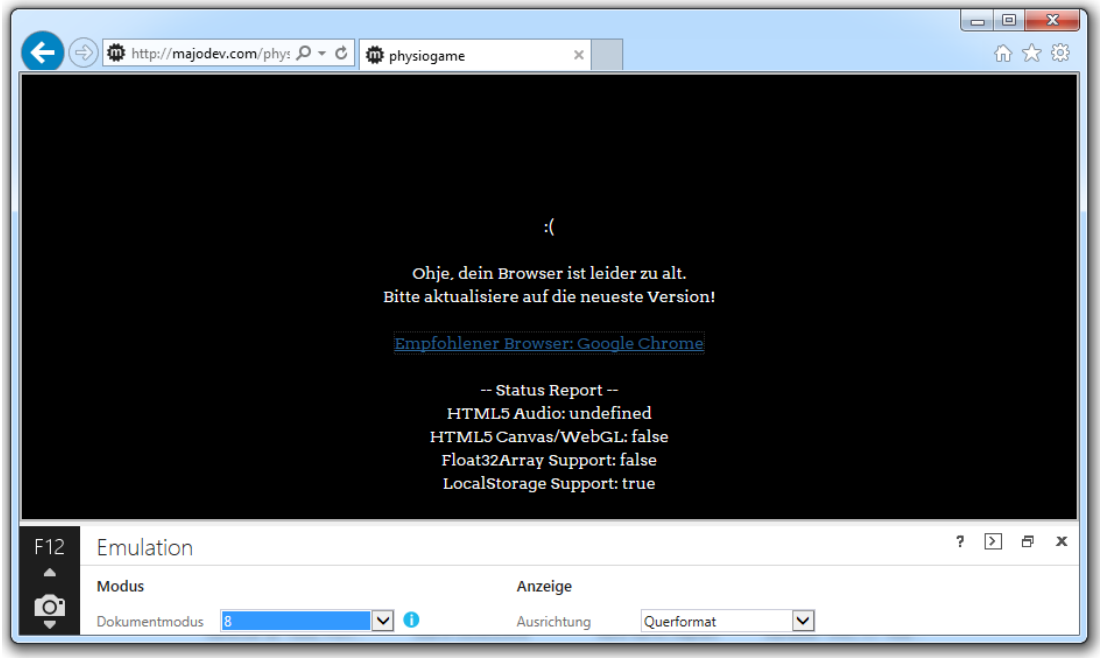

Abbildung 8.2: Physiogame: Mindestvoraussetzungen nicht bestanden (Microsoft Internet Explorer 8.0 Emulation)

der Einstellungen und Statistiken im Cache).

Da Physiogame auch als eigenständige Desktop-Applikation für das jeweilige Betriebssystem verfügbar ist, dürfte sich die Einschränkung auf möglichst moderne Browser nicht großartig für die Portabilität auswirken. Im schlechtesten Fall muss die Desktop-Applikation auf den Zielrechnern ausgeführt werden.

8.2 Titelbildschirm

Abbildung 8.3 auf der nächsten Seite zeigt den Titelbildschirm von Physiogame, nachdem die Applikation erfolgreich geladen und initialisiert wurde. In diesem Fall handelt es sich um die Desktop-Version von Physiogame. Der Titelbildschirm ist intern eine Szene (namens „welcome“), die aus verschiedenen Ebenen besteht. Der (blaue) Hintergrund, die Wolken, der Schriftzug „Physiogame“ und der Cursor (in roter Farbe) sind dabei jeweils einer eigenen Ebene definiert. Dies hat den Vorteil, dass beispielsweise die Hintergrund- und Wolken-Ebene auch in anderen Szenen eingesetzt werden kann.

Abbildung 8.3: Physiogame: Titelbildschirm (Desktop-Applikation)

Wie ersichtlich wird, stehen am Titelbildschirm sechs Schaltflächen zur Auswahl:

- Mit „Spielen“ kann eine neue Spielrunde mit den aktuell gesetzten Einstellungen (in einer anderen Szene) gestartet werden.
- „Statistiken“ öffnet ein Dialog-Fenster und zeigt alle aktuell geladenen Statistiken.
- Über „Einstellungen“ kann ein Dialog-Fenster geöffnet werden, in dem die aktuelle Konfiguration von Physiogame verändert werden kann.
- „Credits“ lädt eine Szene, in der Informationen zur Applikation angezeigt werden.
- Über die Schaltfläche „Fenster“ kann (nur in der Desktop-Version) zwischen Fenster- und Vollbild-Modus umgeschalten werden. Der Text der Schaltfläche wird dabei automatisch angepasst.
- „Beenden“ schließt die Applikation. Dies ist ebenfalls nur in der Desktop-Applikation verfügbar.

Der kreisförmige (rote) Cursor kann sowohl mit üblichen Eingabeschnittstellen (wie Maus oder Touchscreen) als auch mit den Händen – über die Positionserkennung durch den Leap Motion Controller – gesteuert werden. Er ist die primäre Schnittstelle um mit der Applikation zu interagieren und wird – bis auf die Dialoge „Einstellungen“ und „Statistiken“ – in der gesamten Applikation verwendet. Falls benötigt, kann die visuelle Gestaltung des Cursors aber geändert werden.

Durch einen Klick, einer Touch-Eingabe oder längerer Fokussierung (beim Leap Motion Controller) der „Spielen“-Schaltfläche, startet eine neue Spielrunde. Diese Interaktion könnte von Benutzern (bzw. Patienten) somit auch selbst ausgeführt werden. Es besteht daher die Möglichkeit die Applikation in den sogenannten Kiosk-Modus zu schalten. Bis auf „Spielen“ und „Credits“ werden hierdurch alle anderen Schaltflächen ausgeblendet. Dies würde eine unbeaufsichtigte Verwendung mit fixierten Einstellungen von Physiogame erlauben.

8.3 Spielablauf

Bevor die Spielrunde startet, baut sich die Spielfläche innerhalb einer (veränderbaren) Zeitspanne auf. Die Spielobjekte erscheinen und es werden Informationen zum Spielziel, zur Interaktion und ein zufällig generierter Tipp eingeblendet. Abbildung 8.4 auf der nächsten Seite zeigt diese Aufbauphase vor dem Start einer Spielrunde.

In diesem Fall ist der Spielmodus „auf Zeit“ gesetzt und eine Rundenzeit von drei Minuten definiert. Auch die Art der Interaktion wird angezeigt. In dieser Runde müssen die Luftballone beispielsweise angestoßen werden, um sie zum „Platzen“ zu bringen (zweite Art der Interaktion aus den definierten Anforderungen, siehe Abschnitt 4.2.2 auf Seite 54). Links unten wird zudem eine „zurück“-Schaltfläche angezeigt, durch die eine Spielrunde jederzeit unterbrochen werden kann, wodurch die bisherig erhobenen Daten trotzdem automatisch gespeichert werden.

Abbildung 8.4: Physiogame: Aufbauphase vor dem Start der Runde

Abbildung 8.5 zeigt eine bereits laufende Spielrunde.

Abbildung 8.5: Physiogame: Interaktion während einer laufenden Runde

Hier wurde ein Luftballon gerade zum Platzen gebracht und dieser Abschuss wird von visuellen Feedback („Boom!“) und einem Audio-Effekt begleitet. Die Präzision des Abschusses bezogen auf den Mittelpunkt des Spielobjekts (76 Prozent) wird außerdem eingeblendet. Diese Spielrunde läuft bereits seit einer

Minute und 39 Sekunden und es wurden bisher 39 Spielobjekte getroffen.

Bei der dritten Art der geforderten Interaktionsmöglichkeiten muss eine bestimmte Anzahl an Fingern über den Leap Motion Controller gezeigt werden, um das Platzen auszulösen. Abbildung 8.6 zeigt wie dies mit den sogenannten Spezialobjekten umgesetzt wurde.

Abbildung 8.6: Physiogame: Spezialobjekte

Beim aktuell fokussierten Luftballon müssen folglich vier Finger gezeigt werden. Die weiße Zahl in der Mitte des Luftballons steht dabei für die benötigte Fingeranzahl, während die (grüne) Zahl darunter, die aktuell erkannte Anzahl an Fingern repräsentiert. Der grüne Kreis um diese Zahl wächst bei richtiger Erkennung und dient als Feedback, um die nötige Zeitspanne für diese Interaktion zu vermitteln bis der Luftballon platzt. Falls die falsche Anzahl an Fingern gezeigt wird, wird diese Zahl außerdem in der Farbe rot dargestellt.

Am Ende einer Runde wird ein Abschlussbildschirm angezeigt, der eine eingeschränkte Sicht auf die erhobenen Daten bietet. Abbildung 8.7 auf der nächsten Seite zeigt diesen Bildschirm.

Wie ersichtlich, wird die Anzahl an Treffern, die (Spiel-)Zeit, Genauigkeit (Trefferpräzision) und der Bewegungsweg (aller erfassten Handbewegungen vom

Abbildung 8.7: Physiogame: Abschlussbildschirm

Leap Motion Controller) aufgelistet. In Physiogame kann der erlaubte Interaktionsraum (überhalb des Leap Motion Controllers) eingeschränkt werden. Dies ist der Bedienung dienlich, da hierdurch beispielsweise nicht der komplette mögliche horizontale Bewegungsweg durchgeführt werden muss, um den Cursor von links nach rechts zu bewegen. Daher ist die Auswertung des Bewegungswegs in der (erlaubten) Spielfläche, auch für die Statistiken relevant. In diesem Fall wurden rund 6,5 Meter auf der Spielfläche zurückgelegt, 80 Prozent des gesamten Bewegungswegs von ca. acht Metern.

Durch den Abschlussbildschirm soll die Motivation der Benutzer gesteigert werden. Er macht es möglich frühere Spielrunden (mit den gleichen Einstellungen) aufgrund der erzielten Punkte zu vergleichen. In dieser Spielrunde wurden 13 Spielobjekte getroffen, davon drei Spezialobjekte. Zur Berechnung der Punkte wird die Anzahl der Treffer mit der Trefferpräzision multipliziert und schlussendlich die Spezialpunkte (mal 100) hinzuaddiert. Dies ergibt bei dieser Runde insgesamt 929 Punkte.[4]

[4] Die Trefferpräzision wird zur Anzeige am Abschlussbildschirm immer aufgerundet. Sie beträgt im Falle der Abbildung 8.7 tatsächlich 48,384615384 Prozent. Dieser Wert muss zur Berechnung genutzt werden um 929 Punkte zu erhalten.

8.4 Interaktion und Feedback

Da das Geben von Feedback bei Applikationen in der virtuellen Rehabilitation besonders wichtig ist, werden die meisten visuellen Effekte auch von Audio-Effekten begleitet. Dies betrifft beispielsweise das Fokussieren von Schaltflächen, bzw. Spielobjekten und den Start und Abschluss einer Spielrunde. Um die Interaktion mit den Händen (via den Leap Motion Controller) durch Feedback zu unterstützen, wird visuelles Feedback in der Form von fünf verschiedenen Symbolen gegeben. Abbildung 8.8 zeigt diese gesammelt.

Abbildung 8.8: Physiogame: Visuelles Feedback bei Leap Motion

Solange keine Hand innerhalb des Interaktionsbereichs (von Leap Motion) erkannt wird, wird das Handsymbol (Symbol a) links oben am Bildschirm angezeigt. Dies wird nach einigen Sekunden zudem von einer Text-Einblendung ergänzt. Die Pfeilsymbole (im rechten Teil der Abbildung) werden eingesetzt, um anzuzeigen, dass sich eine Hand außerhalb des erlaubten (eingeschränkten) Interaktionsbereich befindet. Beispielsweise wird das Symbol b eingeblendet, um zu visualisieren, dass die Hand aktuell zu hoch positioniert ist und nach unten (zum Leap Motion Controller hin) bewegt werden muss.

Die Betätigung von Schaltflächen erfolgt durch Fokussierung automatisch. Dies gilt natürlich nur für Interaktionen mit den Leap Motion Controller, Klick- bzw. Touch-Interaktionen sind auch möglich. Abbildung 8.9 auf der nächsten Seite zeigt das Feedback bei der Fokussierung von Schaltflächen mit Leap Motion.

Abbildung 8.9: Physiogame: Betätigen von Schaltflächen via Leap Motion

Der, in grüner Farbe gefüllte, Balken im Hintergrund des Textes gibt Feedback, wie lange die Schaltfläche noch bis zur Betätigung fokussiert werden muss. Diese Zeitspanne ist einstellbar und liegt standardmäßig bei zwei Sekunden, damit die Schaltflächen nicht versehentlich ausgelöst werden.

8.5 Dialoge

Die Dialoge für Einstellungen und Statistiken werden dynamisch in HTML (durch die Bibliothek Handlebars) gerendert und über das „canvas“-Element gelegt. Abbildung 8.10 zeigt repräsentativ – am Beispiel des „Einstellungen“-Dialogs – wie sich Dialoge in das Gesamtbild der Applikation einfügen.

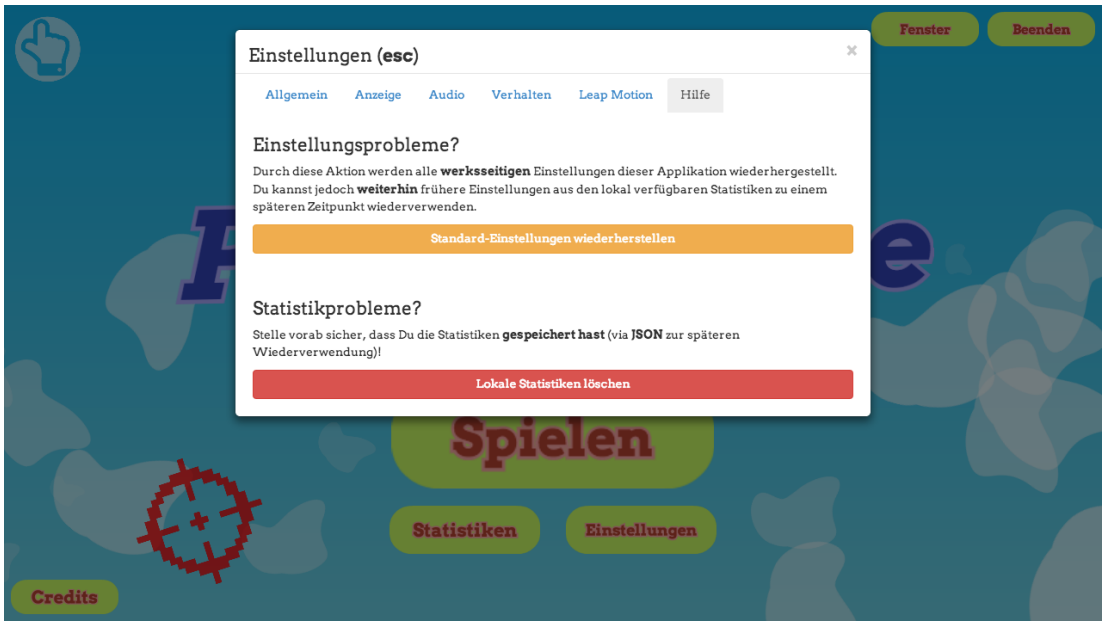

Abbildung 8.10: Physiogame: Dialoge in der Applikation Physiogame

Wie ersichtlich, wird das Spiel im Hintergrund leicht ausgeblendet, damit das Dialogfenster im Fokus steht. In Dialogen ist keine Interaktion via Leap Motion Controller möglich, lediglich Maus und Tastatur (bzw. Touchscreens) werden unterstützt. Dies ist eine sinnvolle Einschränkung, beispielsweise um unbeabsichtigte Änderungen in den Einstellungen oder Statistiken durch Patienten zu verhindern. Innerhalb des aufgezeigten Hilfe-Reiters (des „Einstellungen“-Dialogs) können übrigens die Standard-Einstellungen wiederhergestellt und die aktuell im Cache gespeicherten Daten (Statistiken) gelöscht werden.

In den nachfolgenden Unterabschnitten werden drei Arten von Dialogen (Einstellungen, Statistiken und Notifikationen) in Physiogame vorgestellt und ihre Funktion genauer betrachtet.

8.5.1 Einstellungen

Die Konfigurationsmöglichkeiten in Physiogame sind vielfältig und alle Änderungen werden sofort übernommen. Der „Einstellungen“-Dialog ist jederzeit über die „ESC“-Taste auf der Tastatur erreichbar, auch wenn sich die Applikation im Kiosk-Modus befindet und die „Einstellungen“-Schaltfläche am Titelbildschirm ausgeblendet wurde. Sollten die Einstellungen während einer Spielrunde geändert werden, wird die Runde automatisch neu gestartet. Abbildung 8.11 auf der nächsten Seite zeigt alle Einstellungsmöglichkeiten (bis auf den Reiter „Hilfe“) von Physiogame im Überblick.

Die Einstellungen sind in mehrere Reiter (Kategorien) sortiert und enthalten jeweils eine Tabelle mit den Parametern der Kategorie (auf Basis der Datenstruktur des Moduls „gameConfig“). In der ersten Spalte der Tabelle stehen dabei immer die Bezeichner der Parameter, in der zweiten Spalte kann der Wert durch Schieberegler, Schaltflächen oder Texteingaben verändert werden. In der dritten Spalte steht der aktuell verwendete Wert – in der Regel inklusive seiner Maßeinheit – und der Standardwert des Parameters.

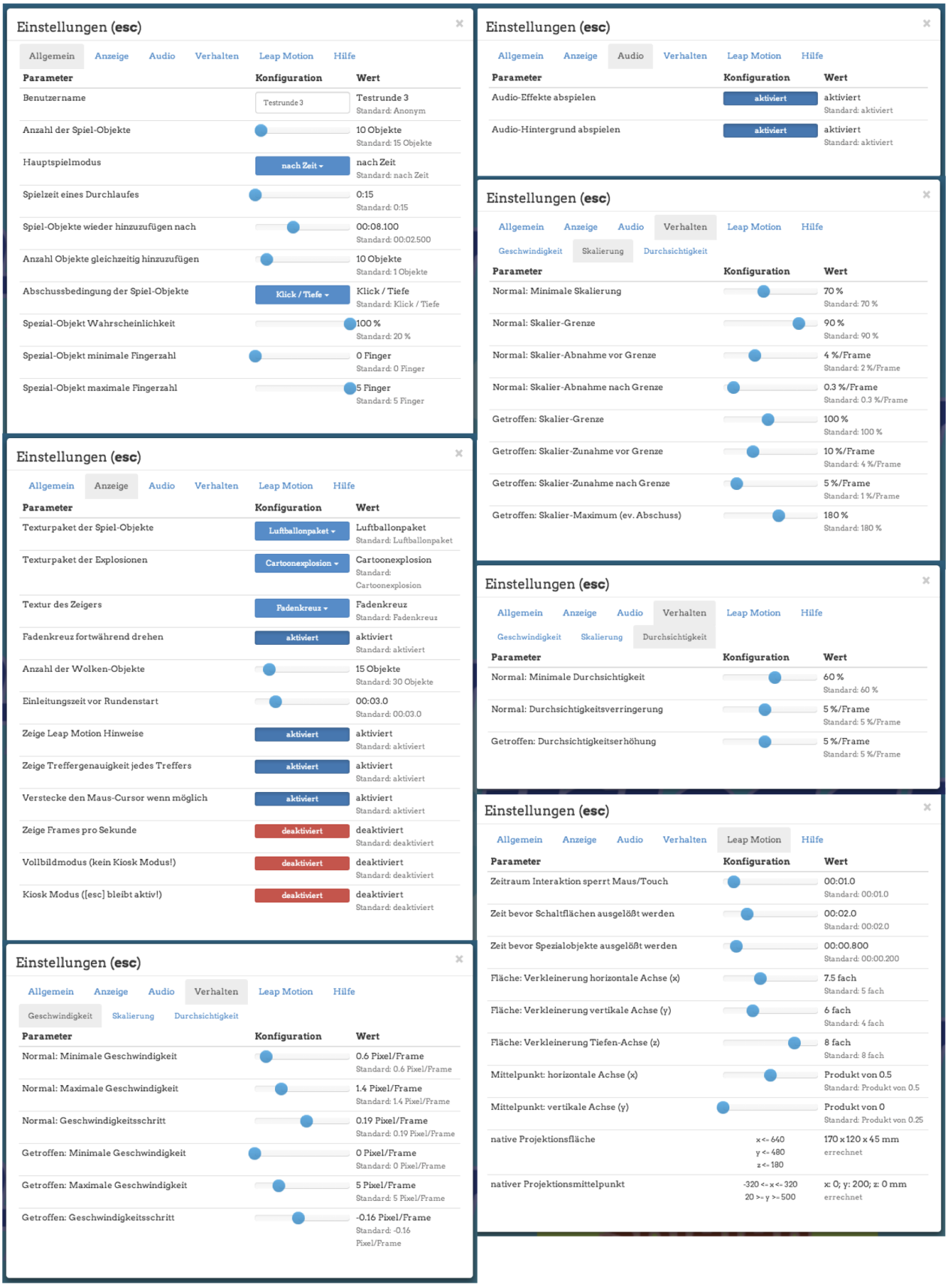

Abbildung 8.11: Physiogame: Übersicht der Reiter im Einstellungen-Dialog

Die einzelnen Kategorien aus dem „Einstellungen“-Dialog werden in den nachfolgenden Unterabschnitten nun genauer vorgestellt.

8.5.1.1 Kategorie „Allgemein“

Innerhalb dieses Reiters können die Einstellungen zum Spielmodus und zur Art der Interaktion durchgeführt werden. Das Setzen des Benutzernamens ist dabei vor allem für die Statistiken – zur späteren Wiederfindung – wichtig. Weiters können Spielparameter, wie beispielsweise die Anzahl von Spielobjekten, Zeitspanne einer Spielrunde und die Wahrscheinlichkeit, das Spezialobjekte erscheinen (zum Mischen mit normalen Spielobjekten innerhalb einer Spielrunde) gesetzt werden. Zudem ist die Einschränkung der anzuzeigenden „Fingerzahlen“ auf den Spezialobjekten möglich.

8.5.1.2 Kategorie „Anzeige“

Die visuelle Gestaltung von Physiogame soll austauschbar sein. In der Kategorie „Anzeige“ können daher die Texturen der Spielobjekte und des Cursors, bzw. die Animation des „Explosion“-Feedbacks eingestellt werden. Beispielsweise können neben Luftballons, auch Cartoon-Aliens zur visuellen Gestaltung der Spielobjekte eingesetzt werden und der Cursor kann neben der Gestaltung als „Fadenkreuz“, auch als eine virtuelle Hand angezeigt werden.

Außerdem kann in diesem Reiter die Anzahl an Hintergrund-Objekten (die Wolken) konfiguriert werden und die Aufbauzeit („Einleitungszeit“) vor Start einer Runde eingestellt werden. Auch die Feedback-Ebene zur Unterstützung der Interaktion via Leap Motion (siehe Abschnitt 8.4 auf Seite 144) und die sofortige Anzeige der Treffergenauigkeit kann ein- und ausgeblendet werden. Schlussendlich besteht innerhalb dieser Kategorie die Möglichkeit die Applikation in den Vollbild-Modus (nur bei Desktop-Applikationen) und in den Kiosk-Modus (via „ESC“-Taste können die Einstellungen trotzdem ausgerufen werden) zu schalten.

8.5.1.3 Kategorie „Audio“

Physiogame wird standardmäßig von Hintergrundmusik (unterschiedlich pro Szene) und Audio-Effekten (vorrangig für Feedback) untermalt. Innerhalb dieses Reiters kann dies jedoch nach Bedarf deaktiviert werden.

8.5.1.4 Kategorie „Verhalten“

Diese Kategorie ist in die drei Unterkategorien unterteilt, die in den nachfolgenden Paragraphen vorgestellt werden. Alle Einstellungen sind dabei für die Manipulation des Verhaltens der Spielobjekte innerhalb einer Spielrunde verantwortlich. Die Veränderung dieser Konfigurationsparameter wirkt sich primär auf den Schwierigkeitsgrad aus und erlaubt somit eine Anpassung des Spielverlaufs auf den jeweiligen Patienten.

Subkategorie „Geschwindigkeit“

Spielobjekte verhalten sich unterschiedlich, je nachdem ob sie fokussiert („getroffen“) oder nicht fokussiert sind („normal“). Innerhalb dieses Reiters kann die minimale und maximale Geschwindigkeit und der Geschwindigkeitsschritt der Spielobjekte – in der Maßeinheit „Pixel pro Frame“ – eingestellt werden. Dies erlaubt beispielsweise, dass sich fokussierte Spielobjekte schneller bewegen oder langsamer werden (bis zum Stillstand).

Subkategorie „Skalierung“

Die Größe der Spielobjekte wird über die Konfigurationsparameter im Reiter „Skalierung“ gesteuert. Wiederrum sind die Werte, ähnlich wie bei der Subkategorie „Geschwindigkeit“, für fokussierte Spielobjekte getrennt setzbar. Von besonderer Bedeutung ist dabei der Parameter des Skalier-Maximums von fokussierten Spielobjekten. Bei der ersten Art der angeforderten Interaktionen (siehe Abschnitt 4.2.2 auf Seite 54) platzen die Luftballons ab einer gewissen Skalierung. Dies wird mit den angesprochenen Parameter kontrolliert.

Subkategorie „Durchsichtigkeit“

Schlussendlich ist innerhalb von „Durchsichtigkeit“, die Sichtbarkeit (der Alpha-Wert) der Spielobjekte kontrollierbar. Der maximale Alpha-Wert liegt bei 100 Prozent. Fokussierte Spielobjekte nähern sich diesen Alpha-Wert automatisch an. Die minimale Durchsichtigkeit und die Schritte zur Erhöhung und Verringerung sind allerdings veränderbar.

8.5.1.5 Kategorie „Leap Motion“

Innerhalb dieser Kategorie können die Parameter konfiguriert werden, die maßgeblich die Hand-Interaktion via Leap Motion bestimmen. Dies betrifft beispielsweise die Zeitspannen, die bei der Interaktion mit Schaltflächen und Spezialobjekten eingesetzt werden. Weiters findet innerhalb dieses Reiters die Kalibrierung des erlaubten Interaktionsbereichs (Größe der X-, Y-, und Z-Achse) und des Mittelpunkts der horizontalen und vertikalen Achse statt.

8.5.2 Statistiken

In Physiogame werden Statistiken von Spielrunden automatisch erhoben. Der Dialog „Statistiken“ kann jederzeit eingeblendet werden (via der „s“-Taste) und zeigt standardmäßig eine Übersicht aller aufgezeichneten Statistiken. Abbildung 8.12 zeigt diesen Dialog.

Abbildung 8.12: Physiogame: Statistiken-Dialog

Wie in der Abbildung ersichtlich, besteht der Dialog aus einigen Schaltflächen und einer Tabelle mit den erhobenen Daten aus den Spielrunden. Über die Schaltflächen können die aktuell angezeigten Statistik-Daten permanent gespeichert oder exportiert werden. Gespeicherte Statistik-Daten im JSON-Format können zu einem späteren Zeitpunkt über die Laden-Schaltfläche wieder geladen werden. Dies trifft auf exportierte Daten nicht zu, da hierbei das CSV-Format verwendet wird und dies nicht mehr zu Physiogame kompatibel ist. Das Format eignet sich allerdings zur weiteren Auswertung der Daten in anderen Programmen, wie beispielsweise Microsoft Excel. Ein Überblick, wie diese exportierten Statistik-Daten aussehen, wird noch im Unterabschnitt 8.5.2.2 auf Seite 153 gegeben.

Wie ersichtlich, ist in der Tabelle aktuell nur ein Eintrag vorhanden. Es werden Basisinformationen zu den aufgezeichneten Daten der Spielrunden, wie beispielsweise Benutzername, Start-, End- und Rundenzeit, angezeigt. Die Spalte „Norm“ steht für die Anzahl der getroffenen Spielobjekten, „Spez“ für die Anzahl an Spezialobjekten. Außerdem wird die Treffergenauigkeit („Prez“) und die Anzahl der erzielten Punkte angezeigt. Diese Übersicht zeigt jedoch nur einen Ausschnitt der Daten. Eine detailliertere Ansicht kann durch einen Klick auf die Schaltfläche „Details“ betrachtet werden. Die Detailansicht eines Statistik-Eintrags wird im nachfolgenden Unterabschnitt beleuchtet.

8.5.2.1 Detailansicht

Die Detailansicht gliedert sich in die Detailpunkte „Erweiterte Statistiken“, „Leap Motion“ und „Einstellungen“ und wird in Abbildung 8.13 auf der nächsten Seite aufgezeigt. Der Schlüssel „bf8f13e4-00f9-72e6-84a9-c31317496013“ identifiziert den ausgewählten Statistik-Datensatz dabei eindeutig.

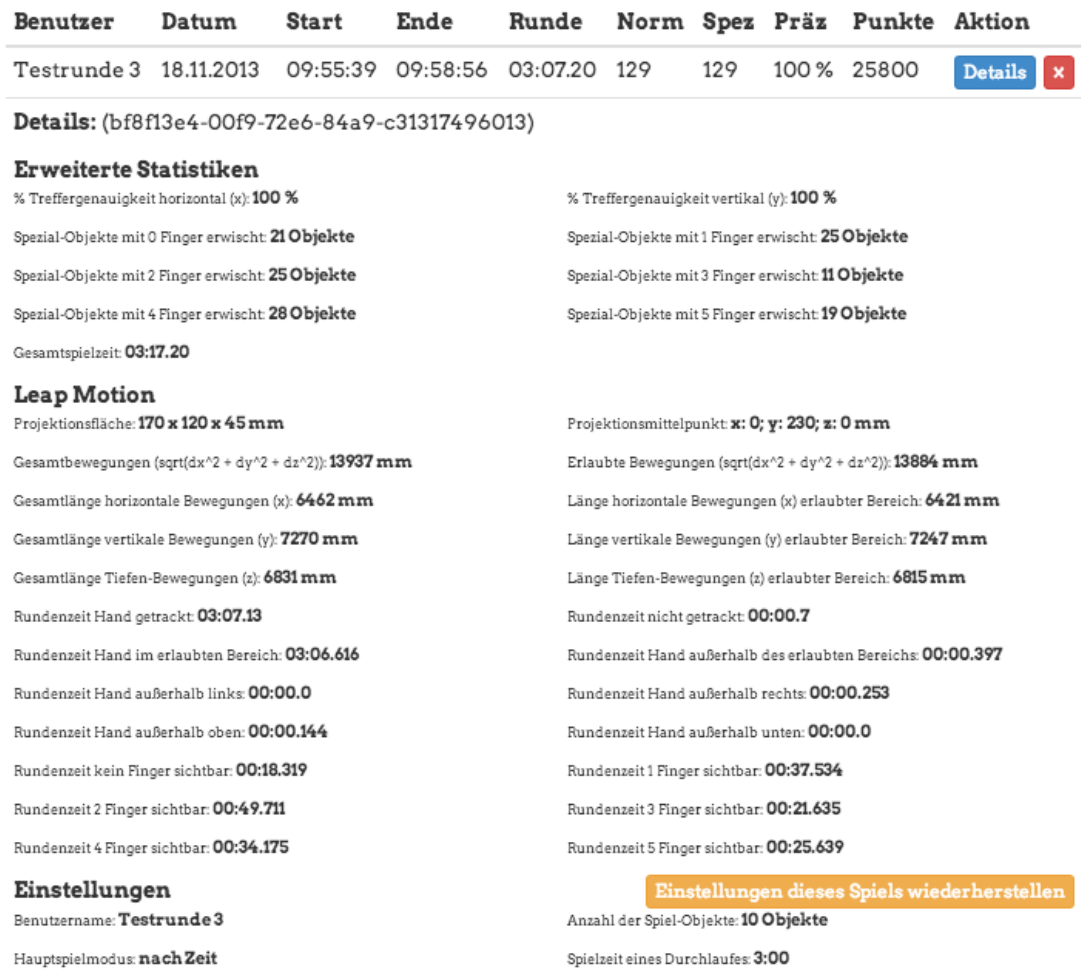

Abbildung 8.13: Physiogame: Statistiken Detailansicht

In den nachfolgenden Paragraphen werden die einzelnen Detailpunkte genauer beleuchtet.

Detailpunkt „Erweiterte Statistiken“

Innerhalb dieses Detailpunkts werden hauptsächlich detailliertere Informationen zu den aufgezeichneten Spieldaten aufgezeigt, beispielsweise welche Spezial-Objekte in welcher Anzahl getroffen wurden und mit welcher Genauigkeit die Spielobjekte horizontal und vertikal getroffen wurden.

Detailpunkt „Leap Motion“

Die Daten innerhalb dieses Detailpunkts sind vor allem zur Auswertung der Bewegungen, die durch den Leap Motion Controller erfasst wurden, relevant. Die Berechnung der Gesamtbewegung erfolgt auf Basis der getrackten Positionen (des Handmittelpunkts) aus den „Frame“-Objekten. Wie bereits aus der Besprechung des Abschlussbildschirms (siehe Abschnitt 8.3 auf Seite 140) ersichtlich wurde, werden Bewegungen in einen erlaubten (Interaktions-)Bereich und in einen Bereich außerhalb eingeteilt. Diese Einteilung kann noch detaillierter nach der Achse betrachtet werden, beispielsweise können nur die horizontalen Bewegungen im erlaubten Interaktionsraum herausgefiltert werden.

Neben den Bewegungswegen können auch die Zeitspannen der erfolgreichen Erkennung bzw. der gezeigten Finger nachvollzogen werden. Beispielsweise kann hierdurch ausgewertet werden, wie lange sich die Hand außerhalb des erlaubten Bereichs befunden hat, bzw. welche Finger am längsten gezeigt wurden.

Detailpunkt „Einstellungen“

Unter diesen Detailpunkt können alle Konfigurationswerte einer aufgezeichneten Spielrunde betrachtet werden. Die Abbildung zeigt dabei nur einen Ausschnitt (vier Elemente) aller Einstellungsparameter. Mit der Schaltfläche „Einstellungen dieses Spiels wiederherstellen“ kann eine frühere Runde übrigens wiederhergestellt werden. Somit können die Einstellungen unterschiedlicher Benutzer (identifizierbar durch den Benutzernamen) über die Statistiken verwaltet und nach Bedarf wiederhergestellt werden.

8.5.2.2 Export

Die erhobenen Statistiken in Physiogame können in das CSV-Format exportiert werden. Es werden dabei allerdings nur die wichtigsten Daten der gesetzten Einstellungen exportiert und die Bezeichner (die Spaltennamen) zusätzlich

angepasst. Tabelle 8.1 zeigt diese Spalten übersichtlich.

Bezeichnung (Spaltenname)	Erklärung
id	eindeutiger Schlüssel der Spielrunde
config_userName_string	Benutzername
startDate_date	Startdatum des Spieles (DD.MM.YY hh:mm:ss)
endDate_date	Enddatum des Spieles (DD.MM.YY hh:mm:ss)
gameTime_ms	Spielzeit in Millisekunden
playTime_ms	Rundenzeit in Millisekunden (abzüglich Einleitungszeit)
catched_count	Anzahl der erwischten Spielobjekte
specialsCatched_count	Anzahl der erwischten Spezialobjekte
specialsFingerX_count	Anzahl der erwischten Spezialobjekte mit X Fingern
points_count	Punkteanzahl
accuracy_sum_percentage	Treffergenauigkeit vertikal und horizontal gesamt in Prozent
accuracy_x_percentage	Treffergenauigkeit horizontal in Prozent
accuracy_y_percentage	Treffergenauigkeit vertikal in Prozent
config_gameMode_string	Spielmodus
config_gameObjectCondition_string	Interaktionsmodus
config_gameMaxTime_sec	maximale Spieldauer in Sekunden
config_objectsToSpawn_count	maximale Objektanzahl
leap_projectionWidth_millimeter	Projektionsweite in Millimeter
leap_projectionHeight_millimeter	Projektionshöhe in Millimeter
leap_projectionDepth_millimeter	Projektionstiefe in Millimeter
leap_projectionCenterX_millimeter	Projektionsmitte (x) in Miilimeter
leap_projectionCenterY_millimeter	Projektionsmitte (y) in Miilimeter
leap_projectionCenterZ_millimeter	Projektionsmitte (z) in Miilimeter
leap_movement_all_hyp_millimeter	Gesamtlänge aller Bewegungen im getrackten Raum (Hypotenuse pro Frame) in Millimeter
leap_movement_all_x_millimeter	Gesamtlänge der horizontalen Bewegungen (x) in Millimeter
leap_movement_all_y_millimeter	Gesamtlänge der vertikalen Bewegungen (y) in Millimeter
leap_movement_all_z_millimeter	Gesamtlänge der Tiefen-Bewegungen (z) in Millimeter
leap_movement_inside_hyp_millimeter	Länge Bewegungen im erlaubten Spielraum (Hypotenuse pro Frame) in Millimeter
leap_movement_inside_x_millimeter	Länge horizontale Bewegungen im erlaubten Spielraum (x) in Millimeter
leap_movement_inside_y_millimeter	Länge vertikale Bewegungen im erlaubten Spielraum (y) in Millimeter
leap_movement_inside_z_millimeter	Länge Tiefen-Bewegungen im erlaubten Spielraum (z) in Millimeter
leap_detected_ms	Rundenzeit Hand getrackt in Millisekunden
leap_notdetected_ms	Rundenzeit Hand nicht getrackt in Millisekunden
leap_inside_ms	Rundenzeit Hand im erlaubten Spielraum in Millisekunden
leap_outside_ms	Rundenzeit Hand außerhalb des erlaubten Spielraumes in Millisekunden
leap_outsideLeft_ms	Rundenzeit Hand außerhalb links in Millisekunden
leap_outsideRight_ms	Rundenzeit Hand außerhalb rechts in Millisekunden
leap_outsideTop_ms	Rundenzeit Hand außerhalb oben in Millisekunden
leap_outsideBottom_ms	Rundenzeit Hand außerhalb unten in Millisekunden
leap_fingerTimeX_ms	Rundenzeit X Finger sichtbar in Millisekunden

Tabelle 8.1: Übersicht der Spalten beim CSV-Export

Der Name der Bezeichner setzt sich dabei aus dem internen Namen und – in der Regel – den Datentyp des Feldes bzw. der Maßeinheit zusammen. Diese Tabelle kann als Referenz verwendet werden, um eine Auswertung der exportierten Daten in anderen Programmen vorzunehmen.

8.5.3 Notifikationen

Eine Spezialform, wie Dialoge in Physiogame außerdem noch eingesetzt werden, sind Notifikationen. Abbildung 8.14 auf der nächsten Seite zeigt eine Beispiel-Notifikation in Physiogame.

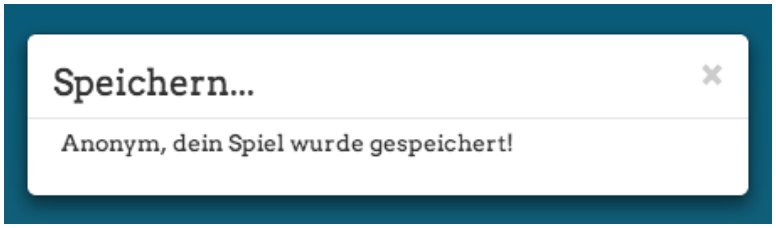

Abbildung 8.14: Physiogame: Notifikation beim Speichern

Die Dialoge werden von Audio-Feedback begleitet und automatisch ein- und ausgeblendet. Wie ersichtlich, werden sie beispielsweise beim Speichern von Spieldaten, nach Abschluss oder beim vorzeitigen Beenden einer Spielrunde, angezeigt.

8.6 Zusammenfassung

Für Physiogame wurden alle definierten Anforderungen umgesetzt und die Applikation kann nun sowohl im Browser als auch direkt als Desktop-Applikation ausgeführt werden. Der Spielverlauf von Physiogame ist durch die Veränderung der Einstellungen weitgehend frei einstellbar und zahlreiche Statistik-Daten können für eine spätere Auswertung erhoben werden. Die Usability von Physiogame – und damit auch die Einsatzfähigkeit von Leap Motion im Bereich der Bewegungstherapie – kann nun getestet werden. Der Ablauf und die Ergebnisse dieses Usability-Tests werden im nächsten Kapitel präsentiert und analysiert.

9 Durchführung eines Usability-Tests

Dieses Kapitel beleuchtet die Ziele, den Ablauf und die Ergebnisse des Usability-Tests von Physiogame. Im nachfolgenden Abschnitt wird hierzu der Ablauf des Tests erörtert. Danach erfolgt im Abschnitt 9.2 auf Seite 158 die Definition der Ziele dieses Tests. Die Erkenntnisse der drei Schritte „Vorbefragung“ (Abschnitt 9.3 auf Seite 159), „Testvorgang“ (Abschnitt 9.4 auf Seite 161) und „Nachbefragung“ (Abschnitt 9.5 auf Seite 167) werden anschließend präsentiert. Von besonderer Bedeutung sind außerdem die Meinungen der beiden Experten (eine Ergo- und eine Physiotherapeutin), die im Rahmen des Abschnitts 9.6 auf Seite 168 geschildert werden. Schlussendlich werden die Resultate im Abschnitt 9.7 auf Seite 169 präsentiert und abschießend eine Zusammenfassung des Kapitels im letzen Abschnitt gegeben.

9.1 Ablauf

Am 18. November 2013 wurde in den Therapieräumen des Kindertherapiezentrums Kids-Chance der Maria Theresia Klink in Bad Radkersburg ein Usability-Test der Applikation Physiogame durchgeführt. Er fand, inklusive Aufbau- und Aufbauzeiten, zwischen 9:00 und 12:00 Uhr statt. Es standen sechs Kinder und zwei Experten (eine Ergo- und eine Physiotherapeutin) als Probanden nacheinander zur Verfügung. Bei den teilnehmenden Kindern handelte es sich um „echte“ Patienten, die sich zu diesem Zeitpunkt in Therapie befanden. Sie waren zwischen drei und fünf Jahre alt.

Physiogame wurde von den Probanden in drei Spielrunden getestet. Es kam bei allen Spielrunden der Spielmodus „nach Zeit“ zum Einsatz, wobei alle

Spielrunden aber jederzeit vom Probanden abgebrochen werden konnte. Abgeschossene Spielobjekte wurden zum Spielfeld nach rund zehn Sekunden wieder automatisch hinzugefügt. In jeder Spielrunde kam eine andere Form der Interaktion (siehe Anforderungen aus dem Abschnitt 4.2.2 auf Seite 54) zu Einsatz:

1. In der Einführungsrunde wurde die erste Form der Interaktion mit den Probanden getestet. Es galt Luftballons zu fokussieren, bis diese platzen. Als Maximalzeit für diese Runde wurden drei Minuten festgelegt und die Anzahl an gleichzeitigen Spielobjekten auf dem Spielfeld wurde auf sechs beschränkt.
2. In der Anstrengungsrunde mussten Spielobjekte durch eine Bewegung nach vorne (Richtung Bildschirm „schlagen“) zum Platzen gebracht werden (zweite Form der Interaktion). In dieser Runde befanden sich maximal 15 Spielobjekte gleichzeitig am Spielfeld und die Maximalzeit wurde auf zwei Minuten festgelegt.
3. In der letzten Runde kamen ausschließlich Spezialobjekte zum Einsatz (Luftballons mit Zahlen) und die Probanden mussten die jeweilig angezeigte Anzahl an Fingern zeigen (dritte Form der Interaktion). Es befanden sich zehn Spezialobjekte gleichzeitig am Spielfeld und die Zeit wurde auf drei Minuten festgelegt.

Im Rahmen des Usability-Tests wurde außerdem eine Vor- und Nachbefragung der Probanden durchgeführt und Einverständniserklärungen (von einem jeweiligen Elternteil) unterschrieben, sodass die Tests per Video aufgezeichnet werden konnten. Die Videoaufzeichnung führte DI (FH) Sandra Schadenbauer durch, während der Autor dieser Diplomarbeit, die Probanden durch den Usability-Test führte.

Abbildung 9.1 auf der nächsten Seite zeigt eine Probandin während des Usability-Tests aus dem aufgezeichneten Videomaterial.

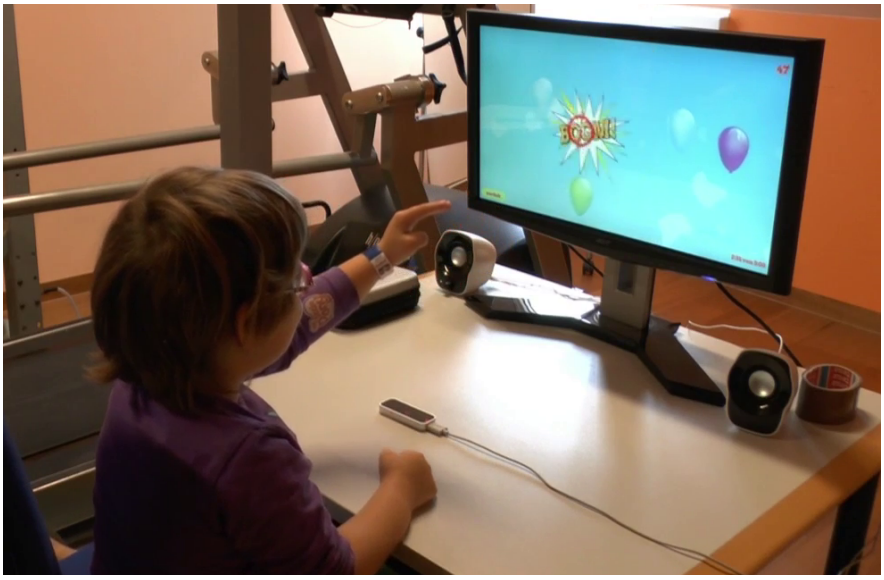

Abbildung 9.1: Probandin während des Usability-Tests

Wie ersichtlich wurde der Aufbau minimal gehalten. Alle Probanden nahmen auf einem höhenverstellbaren Sessel vor einem Schreibtisch Platz. Es wurden zwei Lautsprecher, ein Flachbildschirm und der Leap Motion Controller auf dem Schreibtisch platziert. Bei Bedarf konnte eine Ellbogenstütze – der schwarz/silberne Polster links am Tisch – in Anspruch genommen werden. Um die Probanden nicht abzulenken, wurde der Laptop des Autors auf einem kleineren Nebentisch platziert und mit den Komponenten am Schreibtisch verbunden.

9.2 Zielsetzung

Die Probanden konnten sich frei entscheiden, welche Hand sie zur Interaktion einsetzen wollen bzw. konnten diese auch während einer Spielrunde jederzeit wechseln. Es war folglich nicht das Ziel, eine tatsächliche Therapieeinheit mit Physiogame im Rahmen des Usability-Tests zu simulieren, sondern prinzipiell zu Testen, ob die Probanden mit Physiogame zurechtkommen. Aufgrund des geringen Zeitrahmens der Tests (rund 15 Minuten pro Proband) konnten keine speziellen Anpassungen des Spielablaufs (und damit des Schwierigkeitsgrads)

auf den jeweiligen Probanden vorgenommen werden. Die Einstellungen aller drei Spielrunden waren daher fix (vor-)definiert.

Im Rahmen des Usability-Tests sollte die prinzipielle Einsatzfähigkeit von Physiogame (und des Leap Motion Controllers) in einer realen Umgebung getestet werden. Der Test sollte folgende Fragen klären:

- Ist die Interaktion bzw. der Spielablauf von den Probanden zu bewältigen?
- Welche Schlussfolgerungen können aus den automatisiert erhobenen Statistik-Daten gezogen werden?
- Eignet sich Physiogame nach der Meinung der befragten Ergo- und Physiotherapeutinnen zur Bewegungstherapie?

9.3 Vorbefragung

Bevor die Probanden vorgestellt und auf ihr Handeln im Usability-Test eingegangen wird, bietet es sich an, den Erfahrungsstand der Probanden zu analysieren und auf die Erkrankungen bzw. Störungen einzugehen. Dies erfolgt in den nächsten beiden Unterabschnitten.

9.3.1 Erfahrungsstand

Abbildung 9.2 auf der nächsten Seite zeigt die Ergebnisse der Vorbefragung der Probanden.

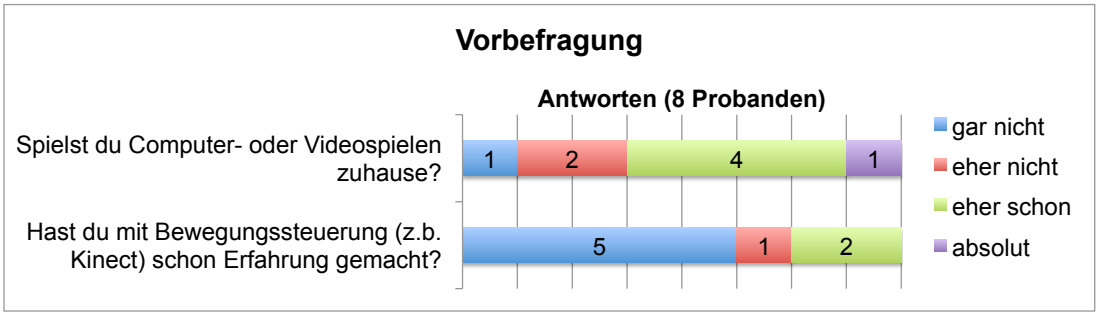

Abbildung 9.2: Vorbefragung: Erfahrungsstand

Wie ersichtlich spielen fünf von acht Probanden („eher schon“ und „absolut“) zu Hause Computer- bzw. Videospiele. Da der Spielablauf von Physiogame als simpel zu bezeichnen ist, ergaben sich daher nur wenige Probleme das Spielziel zu erklären. Hingegen hatte die Mehrheit an Probanden bisher keine Erfahrung mit natürlichen Benutzerschnittstellen, wodurch die Vermittlung der nötigen Interaktion in den jeweiligen Spielrunden der eigentliche Knackpunkt im Usability-Test war.

Es wurde übrigens auch versucht, die Regelmäßigkeit der Spiel-Aktivitäten zu bestimmen. Sie fiel bei allen beteiligten Probanden ähnlich aus, zwischen null bis zwei Stunden in der Woche und stellte für die Auswertung keinen weiteren Mehrwert dar.

9.3.2 Erkrankungen bzw. Einschränkungen

Die Probanden im Usability-Test litten vorrangig an zwei unterschiedlichen Erkrankungen bzw. Störungen. Diese werden in den nächsten beiden Unterabschnitten kurz vorgestellt.

9.3.2.1 Hemiparese

Bei der Hemiparese handelt es sich um eine halbseitige Lähmung einer Körperseite. Die Angabe der Richtung (links oder rechts) bezieht sich dabei auf

die betroffene Gehirnhälfte. Die Leitungsbahnen des Gehirns kreuzen sich jedoch später, wodurch die Lähmung tatsächlich auf der gegenüberliegenden Körperseite auftritt. Beispielsweise bedeutet „Hemiparese rechts“ daher eine Lähmung der linken Körperseite. Symptome können Kraftminderungen, Gefühlsstörungen und gesteigerte Eigenreflexe einer Körperhälfte sein, wobei der Arm bzw. die Hand häufig am stärksten betroffen ist (vgl. [Walter u. a. 2009], S. 72).

9.3.2.2 Infantile Zerebralparese

Die infantile Zerebralparese entsteht durch eine Schädigung des Funktionssystems im Gehirn, bevor sich dieses vollständig entwickelt hat. Sie äußert sich in Bewegungs- und Haltungsstörungen. Symptome sind beispielsweise die Unfähigkeit zur (Muskel-)Entspannung und Kraftstörungen (vgl. [Ferrari u. a. 1998], S. 15-17).

9.4 Testvorgang

Nach der Vorbefragung wurde die Applikation Physiogame von den Probanden in den drei definierten Spielrunden getestet. Tabelle 9.1 auf der nächsten Seite zeigt alle Probanden, die beim Usability-Test teilgenommen haben, inklusive deren Erkrankung bzw. Einschränkung. Weiters wird durch die Tabelle ersichtlich, welche Spielrunden die Probanden getestet haben (manche konnten nicht durchgeführt werden) bzw. wie lange die Tests jeweils insgesamt gedauert haben. Die zwei Experten im Bereich der Ergo- und Physiotherapie sind in der Tabelle in der Farbe grau hervorgehoben.

#	Name	Alter	Geschlecht	Erkrankung bzw. Einschränkung	Spielrunden 1	2	3	Uhrzeit | min
1	Sophia S.	3	weiblich	Hemiparese links	x	x		9:27 - 9:37 10 min
2	*Claudia Dockner*	*-*	*weiblich*	*keine (Physiotherapeutin)*	*x*	*x*	*x*	*9:49 - 9:59 10 min*
3	Daniel W.	5	männlich	Hemiparese links	x	x	x	10:05 - 10:17 12 min
4	Melanie J.	4	weiblich	Gleichgewichtsstörung	x	x	x	10:23 - 10:34 11 min
5	Niklas K.	5	männlich	Infantile Zerebralparese	x			10:50 - 11:00 10 min
6	Helena K.	5	weiblich	Hemiparese rechts	x	x	x	11:00 - 11:12 12 min
7	Laura M.	5	weiblich	Hemiparese rechts	x	x		11:12 - 11:21 9 min
8	*Melanie Tax-Haarkamm*	*-*	*weiblich*	*keine (Ergotherapeutin)*	*x*	*x*	*x*	*11:30 - 11:45 15 min*

Tabelle 9.1: Definition der Probanden (inkl. Erkrankung) des Usability-Tests

Im nachfolgenden Unterabschnitt erfolgt nun die Einzelbetrachtung der Testvorgänge, auf Basis des visualisierten Ablaufs in Tabelle 9.1. Im darauffolgenden Unterabschnitt 9.4.2 auf Seite 165 werden die Testvorgänge anschließend im Gesamten analysiert.

9.4.1 Einzelbetrachtung

In den nachfolgenden Unterabschnitten werden die Testvorgänge der einzelnen Probanden beleuchtet.

9.4.1.1 Probandin Sophia S.

Sophia S. war mit drei Jahren die jüngste Teilnehmerin des Usability-Tests. Trotz einer Hemiparese links, benutzte sie die rechte Hand während des gesamten Testvorgangs. Sie konnte die ersten beiden Spielrunden absolvieren, handelte insgesamt jedoch eher chaotisch. Das Spielziel schien für sie zwar verständlich zu sein, die Interaktionen am Spielfeld (sowohl längeres Fokussieren als auch „Schlagen“) waren allerdings nicht ohne Hilfe des Elternteils möglich.

Die dritte Spielrunde (Verbindung der Zahlen am Luftballon mit der Anzahl der zu zeigenden Fingern) wurde nicht durchführt, da Zahlen für sie (altersbedingt)

noch nicht verständlich waren. Das Spielprinzip von Physiogame – im Besonderen auch die Identifikation mit dem Cursor auf dem Spielfeld – müsste für diese Altersgruppe explizit angepasst werden.

9.4.1.2 Probandin Claudia Dockner (Physiotherapeutin)

Die Physiotherapeutin Claudia Dockner absolvierte alle drei Spielrunden ohne Schwierigkeiten.

9.4.1.3 Proband Daniel W.

Daniel W. konnte die Spielrunden durch die Mithilfe eines Elternteils absolvieren. Ähnlich wie bei Probandin Sophia S. stellte die Identifikation mit dem Cursor und das Einschätzen des Interaktionsraums die Hauptschwierigkeit dar. Auch ein Wechsel der Hand während des Spiels (Hemiparese links) konnte dies nicht ändern.

9.4.1.4 Probandin Melanie J.

Melanie J. konnte alle Spielrunden eigenständig absolvieren und startete diese über den Titelbildschirm sogar selbst. Sie leidet an einer Gleichgewichtsstörung, was sich primär auf die Präzision bzw. das Halten einer bestimmen Position (Fokussierung) hätte auswirken müssen, nach Beobachtung jedoch nicht zutraf.

Eine Schwierigkeit stellte das Spreizen der Finger bei waagrechter Haltung der Hand, im Rahmen der dritten Spielrunde, dar. Zusätzliches Feedback könnte dieses Problem eventuell noch vermindern. Die Robustheit der Erkennung der Finger durch den Leap Motion Controller ist jedoch jedenfalls eingeschränkt (toter Winkel, siehe Abschnitt 3.4 auf Seite 41).

9.4.1.5 Proband Niklas K.

Niklas K. (infantile Zerebralparese) konnte nur die erste Spielrunde durch Unterstützung seines Arms – durch den Autor – durchführen. Das Zusammenspiel seiner Bewegungen mit der Darstellung auf dem Bildschirm konnte ihm nicht vermittelt werden, wodurch er „keine Lust“ zum Spielen hatte. Laut seines Elternteils könnte er Physiogame trotzdem in der Therapie einsetzten, wenn die Interaktion länger mit ihm trainiert wird.

9.4.1.6 Probandin Helena K.

Helena K. (Hemiparese rechts) verwendete anfänglich die rechte Hand, konnte jedoch dazu animiert werden, die linke Hand ab dem mittleren Teil der ersten Spielrunde, bis zum Abschluss zu verwenden. Sie absolvierte weiters alle drei Spielrunden und startete diese selbst vom Titelbildschirm. Schwierigkeiten bereitete vor allem der eingeschränkte Interaktionsbereich in der zweiten Spielrunde (Luftballons durch Bewegung zum Bildschirm „schlagen“).

9.4.1.7 Probandin Laura M.

Laura M. (Hemiparese rechts) spielte die ersten beiden Runden ausschließlich mit der gesunden rechten Hand. Die zweite Spielrunde wurde frühzeitig abgebrochen, da die Interaktion nicht klar wurde und die Probandin nicht zum weiterspielen animiert werden konnte. Die dritte Spielrunde konnte nicht durchgeführt werden (Lesefähigkeit von Zahlen fehlte).

Ein kritischer Punkt, der durch diese Probandin aufgezeigt werden kann, ist das starke Absinken der Motivation, wenn die Interaktion bzw. das Spielprinzip nicht vermittelt werden kann.

9.4.1.8 Probandin Melanie Tax-Haarkamm (Ergotherapeutin)

Die Ergotherapeutin Melanie Tax-Haarkamm konnte alle Spielrunden – ohne beobachtete Probleme – erfolgreich absolvieren.

9.4.2 Gesamtbetrachtung

Im Nachhinein wurden für diesen Usability-Test sicherlich zu junge Probanden ausgewählt. Nichtsdestotrotz konnten zwei Kinder (Melanie J. und Helena K.) alle Spielrunden eigenständig meistern und diese auch vom Titelbildschirm aus starten. Die beiden Experten – die Ergotherapeutin Melanie Tax-Haarkamm und die Physiotherapeutin Claudia Dockner – hatten ebenfalls keine Probleme mit den Spielabläufen in Physiogame.

Es bietet sich an dieser Stelle an, die erhobenen Statistiken-Daten (durch Physiogame im Rahmen des Usability-Test) in Excel zu analysieren, um die beobachteten Probleme in der Interaktion zu diskutieren. Tabelle 9.2 auf der nächsten Seite listet einige Durchschnittswerte von aufgezeichneten Statistik-Daten während des Usability-Tests auf. Die Einrückung der Bezeichner in den linken Spalten ist dabei gewollt und drückt die Hierarchie der Prozentzahlen in den jeweiligen Spielrunden aus. Die – in der Farbe orange – eingefärbten Felder, stellen außerdem Spitzenwerte dar, die in den nachfolgenden Absätzen diskutiert werden.

Wie ersichtlich wird, wurde die dritte Spielrunde durchschnittlich am längsten gespielt und die meisten Spielobjekte wurden getroffen. Dies deckt sich auch mit den Ergebnissen der späteren Nachbefragung, wonach die dritte Spielrunde am meisten Spaß gemacht hat.

Der Leap Motion Controller erkannte zu 85 Prozent eine Hand im Rahmen aller Spielrunden. Lediglich zwischen 70 und 80 Prozent dieser erfassten Bewegungen fanden jedoch im erlaubten Interaktionsbereich (eingeschränkt durch veränderbare Parameter in den Einstellungen von Physiogame) statt. Der Großteil der Bewegungen außerhalb der Spielfläche war dabei „zu tief“ (zu weit zum

Durchschnittswerte	Spielrunde		
	1	2	3
Rundenzeit	02:17,8	01:37,0	02:40,4
Spiel-bzw. Spezialobjekte erwischt	22 Objekte	38 Objekte	48 Objekte
Hand nicht erkannt	13%	15%	15%
Hand erkannt	87%	85%	85%
Hand innerhalb erlaubter Spielfläche	79%	71%	76%
kein Finger sichtbar	30%	67%	24%
1 Finger sichtbar	35%	18%	23%
2 Finger sichtbar	15%	5%	23%
3 Finger sichtbar	6%	4%	12%
4 Finger sichtbar	7%	3%	11%
5 Finger sichtbar	7%	3%	8%
Hand außerhalb erlaubter Spielfläche	21%	29%	24%
Hand links außerhalb	15%	24%	11%
Hand rechts außerhalb	13%	11%	4%
Hand oben außerhalb	13%	20%	8%
Hand unten außerhalb	58%	45%	78%

Tabelle 9.2: Statistik-Ergebnisse: Durchschnittswerte je Spielrunde

Leap Motion Controller hin). Vor allem in der zweiten Spielrunde wurden vermehrt Bewegungen außerhalb des erlaubten Interaktionsbereichs registriert.

Ein wahrgenommenes Problem aus den Einzelbetrachtungen, dass der Interaktionsbereich als zu klein empfunden wird, könnte folglich durch Vergrößerung des erlaubten Interaktionsbereichs in den Einstellungen noch gelöst werden. Eine Verringerung der angesprochenen 15 Prozent, wo keine Hand vom Leap Motion Controller erfasst wurde, ist jedoch gegenwärtig nicht erreichbar.

Interessant ist auch die Verteilung der Fingererkennung pro Spielrunde. Während die erste Spielrunde mit der Faust bzw. mit einem einzelnen sichtbaren Finger gespielt wurde, kam bei der zweiten Spielrunde hauptsächlich die Faust als Handpose zum Einsatz. In der dritten Spielrunde waren vorrangig null, ein und zwei Finger sichtbar, es wurden folglich auch überwiegend Spezialobjekte mit diesen Zahlen erwischt.

9.5 Nachbefragung

Nach Durchführung der Testvorgänge wurde mit den Probanden eine Nachbefragung durchgeführt. Die Ergebnisse dieser Befragungen werden nun im Rahmen dieses Abschnitts analysiert. Mit dem Proband Niklas K. konnte leider keine Nachbefragung durchgeführt werden, in den Ergebnissen sind daher einige Antworten mit „keine Angabe“ vermerkt. Abbildung 9.3 visualisiert die Antworten zu eher allgemeinen Fragen, der erste Teil der Nachbefragung.

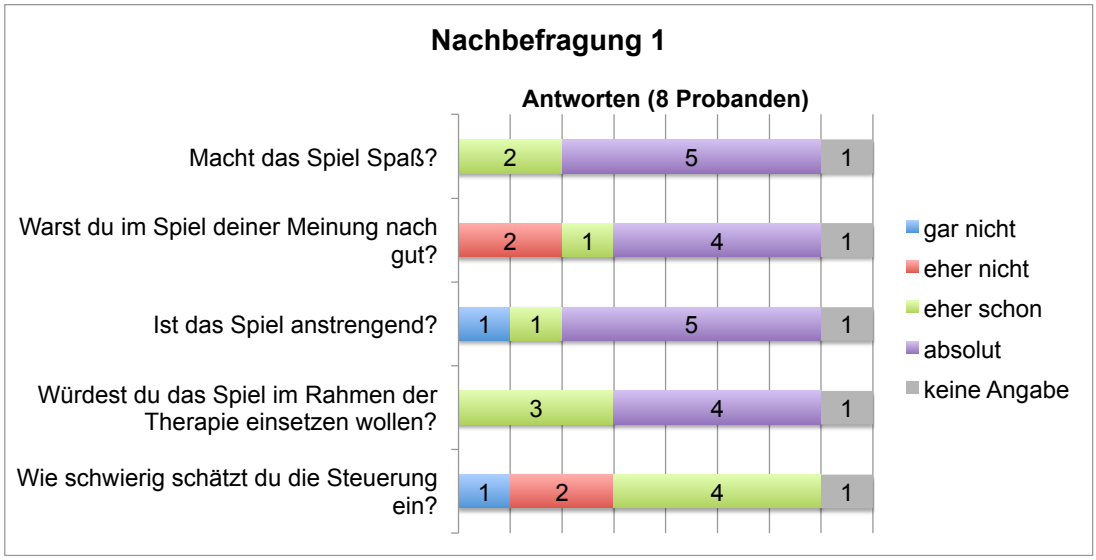

Abbildung 9.3: Nachbefragung 1: Allgemein

Wie ersichtlich machte das Spiel allen befragten Probanden („eher schon“ und „absolut“) Spaß. Dies ist ein äußerst positives Ergebnis, trotz der empfundenen Probleme mit der Interaktion. Die Probanden schätzen ihr Handeln im Spielverlauf eher positiv ein. Sie empfinden Physiogame als anstrengend, würden die Applikation aber trotzdem gerne im Rahmen ihrer Therapie einsetzen. Die Meinungen zur Steuerung gehen eher auseinander, als „absolut“ schwierig, wurde sie jedoch von keinem Probanden empfunden.

In Abbildung 9.4 auf der nächsten Seite visualisiert die Antworten zu den Fragen, die sich direkt auf die drei getesteten Spielrunden beziehen.

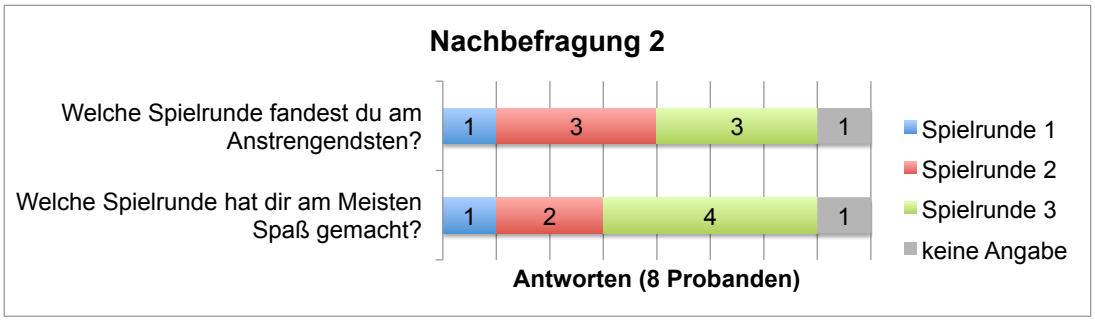

Abbildung 9.4: Nachbefragung 2: Spielrunden spezifisch

Vor allem die zweite und dritte Spielrunde wurde dabei von den Probanden als anstrengend empfunden, was sich jedoch weniger auf den empfundenen Spaß auswirkt, als erwartet. Sowohl die zweite als auch die dritte Spielrunde machten den Probanden besonders Spaß, wobei die dritte Spielrunde insgesamt präferiert wurde (falls sie von den Probanden gespielt wurde).

9.6 Expertenmeinungen

Laut beiden Experten würde sich Physiogame zur Bewegungstherapie eignen. Hierfür wären allerdings sicherlich noch erweiterte (Langzeit-)Studien nötig. Die Einstellungen von Physiogame müssten außerdem jeweils auf die Fähigkeiten der Probanden zugeschnitten werden und die Applikation müsste kontinuierlich im Rahmen eine Therapie eingesetzt werden, um verlässliche Aussagen über die positiven Effekte in der Bewegungstherapie treffen zu können.

In den nachfolgenden Unterabschnitt werden nun die Expertenmeinungen genauer beleuchtet.

9.6.1 Claudia Dockner (Physiotherapeutin)

Die Physiotherapeutin Claudia Dockern hob besonders die gesteigerte Muskelaktivität (durch die Interaktion) hervor und merkte an, dass das Spiel im Rahmen einer Therapie kontinuierlich eingesetzt werden sollte. Die dritte Spiel-

runde würde sich dabei besonders eignen, um die Hand-Augen-Koordination zu verbessern. Auch die Gestaltung und das Feedback von Physiogame hob sie positiv hervor, obwohl, laut ihr, Tier-Themen von der Zielgruppe der Kinder aktuell stark präferiert werden. Da sich die visuelle Gestaltung durch die Einstellungen verändern lässt, wäre eine Anpassung hier zukünftig möglich.

9.6.2 Melanie Tax-Haarkamm (Ergotherapeutin)

Laut der Ergotherapeutin Melanie Tax-Haarkamm ist die nötige Konzentration und Aufmerksamkeit innerhalb der Spielrunden sehr angemessen. Die dritte Interaktionsform hob sie besonders hervor, allerdings müssten die Kinder hierfür mindestens das Volksschulalter besitzen. Im Rahmen einer erweiterten Auseinandersetzung mit Physiogame, lobte sie außerdem die Exportmöglichkeiten der Statistiken von Physiogame. Die Möglichkeit zur Erhebung von Statistiken hätte bei bisherigen Produkten zur Therapie häufig gefehlt.

Die Interaktion im Rahmen der zweiten Spielrunde kommt laut ihr im Alltag nicht häufig vor und es ist daher schwierig diese Bewegung zu trainieren. Da sie zudem anstrengend ist, würde sich eine Fixierung der Hand (zur Stabilisierung) anbieten.

9.7 Resultate

Insgesamt kann die Usability von Physiogame als positiv beurteilt werden, auch wenn nicht alle Probanden alle Spielrunden durchführen konnten. Ausschlaggebend war hier das geringe Alter der Probanden und die fixierten Einstellungen, wodurch der Schwierigkeitsgrad als relativ hoch eingestuft werden kann. Weiters hatte der Großteil der Probanden bisher keine Erfahrung mit Bewegungssteuerungen gesammelt, wodurch die Interaktion sicherlich als schwer empfunden wurde. Alle Probanden hatten jedoch Spaß beim Spielen und würden Physiogame wieder im Rahmen ihrer Therapie einsetzen wollen.

Dies ist ein hervorragendes Zeichen dafür, dass Physiogame insgesamt als motivierend wahrgenommen wurde.

Mit den automatisiert erhobenen Statistik-Daten in Physiogame, ließen sich die Probleme bei der Interaktion weiter analysieren. Weitere Aussagen konnten jedoch nicht generiert werden. Die unterschiedlichen Spielzeiten und die häufige Hilfestellung durch den Autor, bzw. durch einen Elternteil beeinflusste die Punkte- und Bewegungsauswertung zu sehr. Prinzipiell dürften sich die Statistik-Daten somit eher zur Einzel-Beurteilung von Patienten, im Rahmen eines kontinuierlichen Einsatzes in der Therapie eignen.

Die Hauptschwierigkeit lag bei der durchzuführenden Interaktion. Die Ursachen hierzu sind allerdings schwer zu beurteilen. Sie könnten sowohl am kleinen Interaktionsbereich von Leap Motion als auch an den (angeforderten) Interaktionen selbst liegen. Vor allem in der zweiten Spielrunde (wie in Tabelle 9.2 auf Seite 166 ersichtlich) wurden vermehrt Bewegungen außerhalb des erlaubten Bereichs erkannt. Weiters bereitete das Spreizen der Finger und die waagrechte Haltung der Hand im Rahmen der dritten Spielrunde anfänglich Schwierigkeiten. Beide befragten Experten beurteilen Physiogame und auch die Interaktion aber äußerst positiv und können sich einen Einsatz im Rahmen der Bewegungstherapie vorstellen. Sie zeigten sich zuversichtlich, dass die beobachteten Schwierigkeiten mit der Interaktion, nach einer bestimmten Eingewöhnungszeit kein Problem mehr darstellen würden.

9.8 Zusammenfassung

Im Rahmen dieses Kapitels wurden die Ergebnisse des Usability-Test von Physiogame präsentiert. Das Spielprinzip und die Interaktion wurde dabei gut angenommen. Die Applikation dürfte sich laut den Aussagen der Experten tatsächlich zur Bewegungstherapie eignen.

Im darauffolgenden Kapitel wird nun eine Zusammenfassung über die Inhalte der Diplomarbeit gegeben und Ausblick in die Zukunft durchgeführt.

10 Zusammenfassung und Ausblick

In diesem abschließenden Kapitel werden die Inhalte der Diplomarbeit rückblickend betrachtet und ein Ausblick in die Zukunft gegeben. Das Kapitel ist daher in die Abschnitte „Zusammenfassung“ und „Ausblick“ gegliedert.

10.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit konnte die Applikation Physiogame für den Einsatz zur Bewegungstherapie entwickelt werden. Die Grundlagen zu Motion Tracking Technologien und die neuartige, natürliche Benutzerschnittstelle Leap Motion mussten hierfür untersucht werden. Die Anforderungen an die Applikation konnten danach aufgrund von Kriterien für Systeme in der virtuellen Rehabilitation und durch Feedback von Lehrenden aus den Studiengängen Ergo- und Physiotherapie der FH JOANNEUM geformt werden.

Einen großen Teil nahm die Auseinandersetzung mit der Programmiersprache JS ein. Diese besitzt viele Eigenheiten, ist aber äußerst populär und bietet mit Node.js und Node-webkit zwei äußerst interessante Einsatzgebiete. Die Fallen, Patterns und Code-Konventionen in JS wurden daraufhin näher gebracht und Methoden zur Modularisierung, Automatisierung und zum Testen von JS-Applikation erläutert.

Auf Basis der Erkenntnisse aus den JS-Teilen der Diplomarbeit konnte Physiogame implementiert werden und die Applikation und der Spielablauf vorgestellt werden. Den Abschluss bildete ein Usability-Test, der sowohl mit Experten als auch mit Patienten durchgeführt wurde. Physiogame wurde dabei äußerst positiv aufgenommen.

10.2 Ausblick

Das Interesse an Applikationen für den Einsatz in der Bewegungstherapie ist erheblich. Es existieren bereits zahlreiche Systeme zur virtuellen Rehabilitation und durch die Entwicklung von neuen innovativen Benutzerschnittstellen werden sicherlich auch zukünftig neue derartige Projekte entstehen. Beispielsweise schlägt das – im Dezember 2013 veröffentlichte – Spiel-Konzept Richard Riese von Damian Sturm, eine ähnliche Richtung zur interaktiven Therapie wie Physiogame ein und verwendet ebenso den Leap Motion Controller (vgl. [Sturm 2013]).

Physiogame wurde mittlerweile auf der Plattform Github als Open-Source-Projekt veröffentlicht. Hierdurch kann die Applikation zukünftig auch von anderen Personen weiterentwickelt werden, oder als Basis für neue Projekte dienen (ein sogenannter „Fork“). Die Internationalisierung der Applikation wäre dabei ein angebrachter nächster Entwicklungsschritt, da bislang alle Texte in Physiogame nur in der Sprache Deutsch zur Verfügung stehen. Danach wäre es angebracht die Statistik-Features weiterzuentwickeln um beispielsweise die Erstellung von interaktiven Diagrammen zu den Statistiken, direkt innerhalb der Applikation zu ermöglichen.

Die Entscheidung JS zur Entwicklung von Physiogame einzusetzen, war sicherlich die richtige. Durch Node.js und Node-webkit bietet JS zahlreiche neue Einsatzgebiete und die Vorteile von JS wurden mittlerweile auch von zahlreichen Unternehmen erkannt. Beispielsweise kündigte das Unternehmen PayPal Ende November 2013 an, alle zukünftigen Web-Applikationen mit JS und Node.js zu entwickeln, nachdem zahlreiche Vorteile – bei Performance, Entwicklungszeit, Komplexität der Applikationen – gegenüber der Programmiersprache Java (und Spring) festgestellt wurden (vgl. [Harrell 2013]). Basierend auf Node-webkit wurden mittlerweile auch kommerzielle Computerspiele, wie das Spiel Game Dev Tycoon, veröffentlicht (vgl. [Klug 2013]). Insgesamt ist daher anzunehmen, dass JS zukünftig noch eine weit größere Bedeutung, in den unterschiedlichsten Einsatzgebieten einnehmen wird.

Literaturverzeichnis

Um die Übersichtlichkeit zu wahren, werden die Vollverweise in die Abschnitte „Bücher“, „Papers und Zeitschriftenartikel“ und „Internetquellen“ gegliedert.

Bücher

[Crockford 2008] CROCKFORD, Douglas: *JavaScript: The Good Parts*. 1st. O'Reilly Media, 2008. – ISBN 9780596517748

[ECMA International 2011] ECMA INTERNATIONAL: *Standard ECMA-262 - ECMAScript Language Specification*. 5.1. June 2011

[Ferrari u. a. 1998] FERRARI, A. ; CIONI, G. ; REICH, V. ; BURGER, H. ; ROHREGGER, G.: *Infantile Zerebralparese:*. Springer Berlin Heidelberg, 1998 (Rehabilitation und Prävention). – ISBN 9783642588174

[Fogus 2013] FOGUS, Michael: *Functional JavaScript: Introducing Functional Programming with Underscore.js*. O'Reilly Media, Inc., 2013. – ISBN 1449360726, 9781449360726

[Stefanov 2010] STEFANOV, Stoyan: *JavaScript Patterns*. 1st. O'Reilly Media, 2010. – ISBN 9780596806750

[Trostler 2013] TROSTLER, M.: *Testable JavaScript*. O'Reilly Media, Incorporated, 2013 (Oreilly and Associate Series). – ISBN 9781449323394

[Walter u. a. 2009] WALTER, J. ; JOCHHEIM, K.A. ; HAUPT, W.F.: *Neurologie und Psychiatrie für Pflegeberufe*. Thieme, 2009. – ISBN 9783131679901

Papers und Zeitschriftenartikel

[Burdea 2003] BURDEA, G. C.: Virtual rehabilitation–benefits and challenges. In: *Methods of information in medicine* 42 (2003), Nr. 5, S. 519–523

[Chen 2008] CHEN, Qing: *Real-time vision-based hand tracking and gesture recognition*. Ottawa, Ont., Canada, Canada, Dissertation, 2008. – AAINR41629

[Du u. a. 2012] DU, Guanglong ; ZHANG, Ping ; MAI, Jianhua ; LI, Zeling: Markerless Kinect-Based Hand Tracking for Robot Teleoperation. In: *International Journal of Advanced Robotic Systems* 9 (2012)

[Garg u. a. 2009] GARG, Pragati ; AGGARWAL, Naveen ; SOFAT, Sanjeev: Vision Based Hand Gesture Recognition. In: *World Academy of Science, Engineering and Technology* 49 (2009), S. 972–977

[Ghosh u. a. 2010] GHOSH, Soumita ; ZHENG, Jianmin ; CHEN, Wenyu ; ZHANG, Jane ; CAI, Yiyu: Real-time 3D markerless multiple hand detection and tracking for human computer interaction applications. In: *Proceedings of the 9th ACM SIGGRAPH Conference on Virtual-Reality Continuum and its Applications in Industry*. New York, NY, USA : ACM, 2010 (VRCAI '10), S. 323–330. – ISBN 978-1-4503-0459-7

[Harshith u. a. 2010] HARSHITH, C. ; SHASTRY, Karthik R. ; RAVINDRAN, Manoj ; SRIKANTH, M. V. V. N. S. ; LAKSHMIKHANTH, Naveen: Survey on Various Gesture Recognition Techniques for Interfacing Machines Based on Ambient Intelligence. In: *CoRR* abs/1012.0084 (2010)

[Holden 2005] HOLDEN, Maureen K.: Virtual environments for motor rehabilitation: review. In: *CYBERPSYCHOLOGY AND BEHAVIOR* 8 (2005), Nr. 3, S. 187–211

[Jack u. a. 2000] JACK, David ; BOIAN, Rares F. ; MERIANS, Alma S. ; ADAMOVICH, Sergei V. ; TREMAINE, Marilyn ; RECCE, Michael ; BURDEA, Grigore C. ; POIZNER, Howard: A virtual reality-based exercise program for stroke

rehabilitation. In: TREMAINE, Marilyn (Hrsg.) ; COLE, Elliot (Hrsg.) ; MYNATT, Elizabeth D. (Hrsg.): *ASSETS*, ACM, 2000, S. 56–63

[Kiefer und Loclair 2013] KIEFER, Cedric ; LOCLAIR, Christian: Harfe spielen mit Leap & Processing. In: *weave* (2013), Jun., Nr. 3, S. 72–76

[Kim u. a. 2012] KIM, David ; HILLIGES, Otmar ; IZADI, Shahram ; BUTLER, Alex D. ; CHEN, Jiawen ; OIKONOMIDIS, Iason ; OLIVIER, Patrick: Digits: freehand 3D interactions anywhere using a wrist-worn gloveless sensor. In: *Proceedings of the 25th annual ACM symposium on User interface software and technology*. New York, NY, USA : ACM, 2012 (UIST '12), S. 167–176. – ISBN 978-1-4503-1580-7

[König 2013] KÖNIG, Peter: Berührungslos erfasst: 3D-Scansoftware für Kinect & Co. In: *c't Magazin für Computer Technik* (2013), Jun., Nr. 13, S. 118–125

[Lee 2008] LEE, Johnny C.: Hacking the Nintendo Wii Remote. In: *IEEE Pervasive Computing* 7 (2008), Juli, Nr. 3, S. 39–45. – ISSN 1536-1268

[Pavlovic u. a. 1997] PAVLOVIC, Vladimir I. ; SHARMA, Rajeev ; HUANG, Thomas S.: Visual Interpretation of Hand Gestures for Human-Computer Interaction: A Review. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 19 (1997), Juli, Nr. 7, S. 677–695. – ISSN 0162-8828

[Prisacariu und Reid 2012] PRISACARIU, Victor A. ; REID, Ian: 3D hand tracking for human computer interaction. In: *Image Vision Comput.* 30 (2012), Nr. 3, S. 236–250

[Ren u. a. 2011] REN, Zhou ; YUAN, Junsong ; ZHANG, Zhengyou: Robust hand gesture recognition based on finger-earth mover's distance with a commodity depth camera. In: *Proceedings of the 19th ACM international conference on Multimedia*. New York, NY, USA : ACM, 2011 (MM '11), S. 1093–1096. – ISBN 978-1-4503-0616-4

[Song u. a. 2008] SONG, Peng ; YU, Hang ; WINKLER, Stefan: Vision-based 3D finger interactions for mixed reality games with physics simulation.

In: *Proceedings of The 7th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*. New York, NY, USA : ACM, 2008 (VRCAI '08), S. 7:1–7:6. – ISBN 978-1-60558-335-8

[Sridhar 2013] SRIDHAR, Srinath: HandSonor: a customizable vision-based control interface for musical expression. In: *CHI '13 Extended Abstracts on Human Factors in Computing Systems*. New York, NY, USA : ACM, 2013 (CHI EA '13), S. 2755–2760. – ISBN 978-1-4503-1952-2

[Sturman und Zeltzer 1994] STURMAN, David J. ; ZELTZER, David: A Survey of Glove-based Input. In: *IEEE Comput. Graph. Appl.* 14 (1994), Januar, Nr. 1, S. 30–39. – ISSN 0272-1716

[Sugarman u. a. 2012] SUGARMAN, Heidi ; WEISEL-EICHLER, Aviva ; BURSTIN, Arie ; BROWN, Riki: Use of novel virtual reality system for the assessment and treatment of unilateral spatial neglect: A feasibility study. (2012)

[Sveistrup 2004] SVEISTRUP, Heidi: Motor rehabilitation using virtual reality. In: *Journal of NeuroEngineering and Rehabilitation* 1 (2004), Nr. 1, S. 10. – ISSN 1743-0003

[Vikram u. a. 2013] VIKRAM, Sharad ; LI, Lei ; RUSSELL, Stuart: Writing and sketching in the air, recognizing and controlling on the fly. In: *CHI '13 Extended Abstracts on Human Factors in Computing Systems*. New York, NY, USA : ACM, 2013 (CHI EA '13), S. 1179–1184. – ISBN 978-1-4503-1952-2

[Villaroman u. a. 2011] VILLAROMAN, Norman ; ROWE, Dale ; SWAN, Bret: Teaching natural user interaction using OpenNI and the Microsoft Kinect sensor. In: *Proceedings of the 2011 conference on Information technology education*. New York, NY, USA : ACM, 2011 (SIGITE '11), S. 227–232. – ISBN 978-1-4503-1017-8

[Wang und Popović 2009] WANG, Robert Y. ; POPOVIĆ, Jovan: Real-time hand-tracking with a color glove. In: *ACM Trans. Graph.* 28 (2009), Juli, Nr. 3, S. 63:1–63:8. – ISSN 0730-0301

Internetquellen

[Ackerman 2013] ACKERMAN, Dave: *Get Up And Running With Grunt.js*. 2013. – Erreichbar unter http://moduscreate.com/get-up-and-running-with-grunt-js/, besucht am 16.11.2013

[Adobe Systems Incorporated 2011] ADOBE SYSTEMS INCORPORATED: *Adobe's strategic transformation and the Flash Platform*. 2011. – Erreichbar unter http://www.adobe.com/devnet/flashplatform/articles/recent-updates.html, besucht am 02.11.2013

[Adobe Systems Incorporated 2013] ADOBE SYSTEMS INCORPORATED: *Using native extensions for Adobe AIR*. 2013. – Erreichbar unter http://help.adobe.com/en_US/air/build/WS597e5dadb9cc1e0253f7d2fc1311b491071-8000.html, besucht am 06.11.2013

[Allardice 2013] ALLARDICE, James: *An introduction to ES6 Part 1: Using ES6 Today*. 2013. – Erreichbar unter http://globaldev.co.uk/2013/09/es6-part-1/, besucht am 05.11.2013

[Ashkenas 2013] ASHKENAS, Jeremy: *CoffeeScript*. 2013. – Erreichbar unter http://coffeescript.org/, besucht am 07.11.2013

[Baldwin 2012] BALDWIN, Roberto: *Why the Leap Is the Best Gesture-Control System We have Ever Tested*. 2012. – Erreichbar unter http://www.wired.com/gadgetlab/2012/05/why-the-leap-is-the-best-gesture-control-system-weve-ever-tested/, besucht am 02.07.2013

[Bard 2013] BARD, Adam: *Top github languages for 2013 (so far)*. 2013. – Erreichbar unter http://adambard.com/blog/top-github-languages-for-2013-so-far/, besucht am 03.11.2013

[Burke 2012] BURKE, James: *ES Modules: suggestions for improvement*. 2012. – Erreichbar unter http://tagneto.blogspot.co.uk/2012/06/es-modules-suggestions-for-improvement.html, besucht am 13.11.2013

[Burke 2013a] BURKE, James: *almond readme*. 2013. – Erreichbar unter https://github.com/jrburke/almond, besucht am 14.11.2013

[Burke 2013b] BURKE, James: *RequireJS API*. 2013. – Erreichbar unter http://requirejs.org/docs/api.html, besucht am 14.11.2013

[Burke 2013c] BURKE, James: *Why AMD?* 2013. – Erreichbar unter http://requirejs.org/docs/whyamd.html, besucht am 13.11.2013

[chaijs 2013] CHAIJS: *Chai Assertion Library*. 2013. – Erreichbar unter http://chaijs.com/, besucht am 11.11.2013

[Crockford 2002] CROCKFORD, Douglas: *JSLint*. 2002. – Erreichbar unter http://www.jslint.com/, besucht am 11.11.2013

[CyberGlove Systems LLC 2010] CYBERGLOVE SYSTEMS LLC: *CyberGlove III Photos and Videos*. 2010. – Erreichbar unter http://www.cyberglovesystems.com/products/cyberglove-III/photos-video, besucht am 01.08.2013

[Dalton 2013] DALTON, John-David: *Lo-Dash Documentation*. 2013. – Erreichbar unter http://lodash.com/docs, besucht am 16.11.2013

[Donat 2013] DONAT, Jesse: *CoffeeScript's Scoping is Madness*. 2013. – Erreichbar unter https://donatstudios.com/CoffeeScript-Madness, besucht am 07.11.2013

[Farrell 2013] FARRELL, Benjamin: *C++ and Node.js: An unholy combination....but oh so right*. 2013. – Erreichbar unter http://www.benfarrell.com/2013/01/03/c-and-node-js-an-unholy-combination-but-oh-so-right/, besucht am 05.11.2013

[Fleming 2013] FLEMING, Joe: *Use Lo-Dash Instead of Underscore*. 2013. – Erreichbar unter http://joefleming.net/posts/use-lodash-instead-of-underscore/, besucht am 16.11.2013

[Golem.de 2013] GOLEM.DE: *Microsoft-Typescript: Neue Programmiersprache für Web-Apps*. 2013. – Erreichbar unter http://www.golem.de/news/microsoft-typescript-neue-programmiersprache-fuer-web-apps-1210-94860.html, besucht am 07.11.2013

[GruntJS 2013a] GRUNTJS: *grunt-contrib-watch*. 2013. – Erreichbar unter https://github.com/gruntjs/grunt-contrib-watch, besucht am 11.11.2013

[GruntJS 2013b] GRUNTJS: *Grunt: The JavaScript Task Runner*. 2013. – Erreichbar unter http://gruntjs.com/, besucht am 11.11.2013

[Habibi 2003] HABIBI, Oliver: *Real-Time Motion Tracking in digital art installations*. 2003. – Erreichbar unter http://www.b-youth.com/ddm/mo-cap%20report.pdf, besucht am 27.11.2013

[Harrell 2013] HARRELL, Jeff: *Node.js at PayPal*. 2013. – Erreichbar unter https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/, besucht am 13.12.2013

[Heise 2012a] HEISE: *Erster Meilenstein für Googles Programmiersprache Dart*. 2012. – Erreichbar unter http://www.heise.de/developer/meldung/Erster-Meilenstein-fuer-Googles-Programmiersprache-Dart-1730993.html, besucht am 07.11.2013

[Heise 2012b] HEISE: *Microsoft veröffentlicht Kinect für Windows*. 2012. – Erreichbar unter http://www.heise.de/newsticker/meldung/Microsoft-veroeffentlicht-Kinect-fuer-Windows-1426808.html, besucht am 30.11.2013

[Heise 2012c] HEISE: *Winken und Schnipsen am PC*. 2012. – Erreichbar unter http://www.heise.de/newsticker/meldung/Winken-und-Schnipsen-am-PC-1580467.html, besucht am 02.07.2013

[Heise 2013] HEISE: *Dart-Entwickler machen alles für erstes Major-Release fertig*. 2013. – Erreichbar unter http://www.heise.de/developer/meldung/Dart-Entwickler-machen-alles-fuer-erstes-Major-Release-fertig-1944343.html, besucht am 07.11.2013

[Hidayat 2013] HIDAYAT, Ariya: *PhantomJS*. 2013. – Erreichbar unter http://phantomjs.org/, besucht am 11.11.2013

[Holmquest 2012a] HOLMQUEST, Leland: *3D Sight with Kinect*. 2012.

– Erreichbar unter http://msdn.microsoft.com/en-us/magazine/jj851072.aspx, besucht am 27.11.2013

[Holmquest 2012b] HOLMQUEST, Leland: *Starting to Develop with Kinect*. 2012. – Erreichbar unter http://msdn.microsoft.com/en-us/magazine/jj159883.aspx, besucht am 27.11.2013

[Holowaychuk 2011] HOLOWAYCHUK, TJ: *mocha*. 2011. – Erreichbar unter http://visionmedia.github.io/mocha/, besucht am 11.11.2013

[Immersion Corporation 2009] IMMERSION CORPORATION: *Immersion Sells CyberGlove Division*. 2009. – Erreichbar unter http://ir.immersion.com/releasedetail.cfm?releaseid=371926, besucht am 01.08.2013

[Irish 2013] IRISH, Paul: *HTML5 Cross Browser Polyfills*. 2013. – Erreichbar unter https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills, besucht am 04.11.2013

[Johnson 2013] JOHNSON, Phil: *Wait - maybe JavaScript is the top programming language*. 2013. – Erreichbar unter http://www.itworld.com/cloud-computing/364194/wait-maybe-javascript-top-programming-language, besucht am 20.11.2013

[Joyent Inc. 2013a] JOYENT INC.: *node.js*. 2013. – Erreichbar unter http://nodejs.org/, besucht am 04.11.2013

[Joyent Inc. 2013b] JOYENT INC.: *node.js in the industry*. 2013. – Erreichbar unter http://nodejs.org/industry/, besucht am 04.11.2013

[Joyent Inc. 2013c] JOYENT INC.: *Node.js v0.10.21 Manual and Documentation: Addons*. 2013. – Erreichbar unter http://nodejs.org/api/addons.html, besucht am 05.11.2013

[Kalenda 2013] KALENDA, Florian: *Microsoft-Programmiersprache Typescript nähert sich Version 1.0*. 2013. – Erreichbar unter http://www.zdnet.de/88172986/microsoft-programmiersprache-typescript-naehert-sich-version-1-0/, besucht am 07.11.2013

[Klug 2013] KLUG, Patrick: *To our Linux players: Game Dev Tycoon for Linux and glibc*. 2013. – Erreichbar unter http://www.greenheartgames.com/2013/09/07/to-our-linux-players-game-dev-tycoon-for-linux-and-glibc/, besucht am 13.12.2013

[Kovalyov 2011] KOVALYOV, Anton: *Why I forked JSLint to JSHint*. 2011. – Erreichbar unter http://anton.kovalyov.net/p/why-jshint/, besucht am 11.11.2013

[Kovalyov 2013] KOVALYOV, Anton: *About JSHint*. 2013. – Erreichbar unter http://jshint.com/about/, besucht am 11.11.2013

[Leap Motion Inc. 2013a] LEAP MOTION INC.: *ASUS Embraces Revolutionary New Interface, Partners to Bundle Leap Motion with Select Computers*. 2013. – Erreichbar unter https://www.leapmotion.com/press_releases/asus-embraces-revolutionary-new-interface-partners-to-bundle-leap-motion-with-select-computers, besucht am 28.11.2013

[Leap Motion Inc. 2013b] LEAP MOTION INC.: *Leap Motion Architecture*. 2013. – Erreichbar unter https://developer.leapmotion.com/documentation/GetStarted/Leap_Architecture.html, besucht am 28.11.2013

[Leap Motion Inc. 2013c] LEAP MOTION INC.: *Leap Motion Documentation: Leap Overview*. 2013. – Erreichbar unter https://developer.leapmotion.com/documentation/Languages/JavaScript/Guides/Leap_Overview.html, besucht am 29.11.2013

[Leap Motion Inc. 2013d] LEAP MOTION INC.: *Leap Motion Product Info*. 2013. – Erreichbar unter https://www.leapmotion.com/product, besucht am 28.11.2013

[Leap Motion Inc. 2013e] LEAP MOTION INC.: *Leap Motion User Experience Guidelines*. 2013. – Erreichbar unter https://developer.leapmotion.com/documentation/GetStarted/Leap_User_Experience_Guidelines.html, besucht am 03.12.2013

[Leap Motion Inc. 2013f] LEAP MOTION INC.: *The unofficial Leap FAQ*.

2013. – Erreichbar unter https://forums.leapmotion.com/forum/general-discussion/general-discussion-forum/434-the-unofficial-leap-faq?420-The-unofficial-Leap-FAQ=, besucht am 28.11.2013

[Ludei Inc. 2013] LUDEI INC.: *CocoonJS*. 2013. – Erreichbar unter http://www.ludei.com/tech/cocoonjs, besucht am 04.11.2013

[Nanni 2013] NANNI, Dan: *Most popular open-source projects hosted at GitHub*. 2013. – Erreichbar unter http://xmodulo.com/2013/09/popular-open-source-projects-hosted-github.html, besucht am 04.11.2013

[Nodejitsu Inc. 2013] NODEJITSU INC.: *http-server*. 2013. – Erreichbar unter https://github.com/nodeapps/http-server, besucht am 11.11.2013

[Norgren 2013] NORGREN, Victor: *Leap Motion AS3 API*. 2013. – Erreichbar unter https://github.com/logotype/LeapMotionAS3, besucht am 28.11.2013

[Oblong Industries Inc. 2013] OBLONG INDUSTRIES INC.: *G-Speak*. 2013. – Erreichbar unter http://www.oblong.com/g-speak/, besucht am 23.07.2013

[OpenLeap Initiative 2013] OPENLEAP INITIATIVE: *OpenLeap driver*. 2013. – Erreichbar unter https://github.com/openleap/OpenLeap, besucht am 28.11.2013

[Osmani 2011] OSMANI, Addy: *Writing Modular JavaScript With AMD, CommonJS and ES Harmony*. 2011. – Erreichbar unter http://addyosmani.com/writing-modular-js/, besucht am 13.11.2013

[Ovide und Rusli 2013] OVIDE, Shira ; RUSLI, Evelyn: *Apple's Cook Hints at Wearable Devices*. 2013. – Erreichbar unter http://online.wsj.com/article/SB10001424127887323855804578512042192416604.html#, besucht am 23.07.2013

[Pierce 2012] PIERCE, David: *A look inside Leap Motion, the 3D gesture control that's like Kinect on steroids*. 2012. – Erreichbar unter http://www.theverge.com/2012/6/26/3118592/leap-motion-gesture-controls, besucht am 02.07.2013

[Rajlich 2012] RAJLICH, Nathan: *Converting a C library to gyp*. 2012. – Erreichbar unter http://n8.io/converting-a-c-library-to-gyp/, besucht am 05.11.2013

[Reed 2011] REED, Nico: *What is npm?* 2011. – Erreichbar unter http://docs.nodejitsu.com/articles/getting-started/npm/what-is-npm, besucht am 04.11.2013

[Rendle 2012] RENDLE, Mark: *The obligatory TypeScript reaction post*. 2012. – Erreichbar unter http://blog.markrendle.net/2012/10/02/the-obligatory-typescript-reaction-post/, besucht am 08.11.2013

[Rochester Institute of Technology 2011] ROCHESTER INSTITUTE OF TECHNOLOGY: *Historical Motion Tracking Systems*. 2011. – Erreichbar unter http://www.rit.edu/innovationcenter/kinectatrit/historical-motion-tracking-systems, besucht am 23.07.2013

[Sharp 2010] SHARP, Remy: *What is a Polyfill?* 2010. – Erreichbar unter http://remysharp.com/2010/10/08/what-is-a-polyfill/, besucht am 04.11.2013

[SparkFun Electronics 2013] SPARKFUN ELECTRONICS: *Leap Motion Teardown*. 2013. – Erreichbar unter https://learn.sparkfun.com/tutorials/leap-motion-teardown/all, besucht am 28.11.2013

[Stanley 2013] STANLEY, Richard: *OOP In JavaScript: What You Need To Know*. 2013. – Erreichbar unter http://javascriptissexy.com/oop-in-javascript-what-you-need-to-know/#more-1118, besucht am 20.11.2013

[Sturm 2013] STURM, Damian: *Richard Riese*. 2013. – Erreichbar unter http://www.mediendesign-ravensburg.de/portfolio/richard-riese/, besucht am 13.12.2013

[Sublime HQ Pty Ltd. 2013] SUBLIME HQ PTY LTD.: *Sublime Text*. 2013. – Erreichbar unter http://www.sublimetext.com/, besucht am 11.11.2013

[Szablewski 2013] SZABLEWSKI, Dominic: *Ejecta: A Fast, Open Source Ja-*

vaScript, Canvas and Audio Implementation for iOS. 2013. – Erreichbar unter http://impactjs.com/ejecta, besucht am 04.11.2013

[Thompson 2013] THOMPSON, Mick: *node-sphero: Control the Sphero robotic ball from Node!* 2013. – Erreichbar unter https://npmjs.org/package/node-sphero, besucht am 06.11.2013

[Tiobe Software 2013] TIOBE SOFTWARE: *TIOBE Programming Community Index for October 2013*. 2013. – Erreichbar unter http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, besucht am 03.11.2013

[Twitter Inc. 2013] TWITTER INC.: *Bower: A package manager for the web*. 2013. – Erreichbar unter http://bower.io/, besucht am 08.11.2013

[University of Southern California 2008] UNIVERSITY OF SOUTHERN CALIFORNIA: *Motion Sensors*. 2008. – Erreichbar unter http://illumin.usc.edu/165/motion-sensors/, besucht am 23.07.2013

[Waldron 2013] WALDRON, Rick: *Principles of Writing Consistent, Idiomatic JavaScript*. 2013. – Erreichbar unter https://github.com/rwaldron/idiomatic.js/, besucht am 11.11.2013

[Wang 2013a] WANG, Roger: *Node-webkit readme*. 2013. – Erreichbar unter https://github.com/rogerwang/node-webkit, besucht am 04.11.2013

[Wang 2013b] WANG, Roger: *Using Node modules*. 2013. – Erreichbar unter https://github.com/rogerwang/node-webkit/wiki/Using-Node-modules, besucht am 06.11.2013

[Watercutter 2013] WATERCUTTER, Angela: *There is a Nintendo Power Glove Documentary Coming (Yes, It Has a Wizard Reference)*. 2013. – Erreichbar unter http://www.wired.com/underwire/2013/07/power-glove-documentary/, besucht am 19.07.2013

[Wikimedia Commons 2011a] WIKIMEDIA COMMONS: *NES-Power-Glove Bild*. 2011. – Erreichbar unter http://commons.wikimedia.org/wiki/File:NES-Power-Glove.jpg, besucht am 23.07.2013

[Wikimedia Commons 2011b] WIKIMEDIA COMMONS: *Xbox-360-Kinect-Standalone Bild*. 2011. – Erreichbar unter http://commons.wikimedia.org/wiki/File:Xbox-360-Kinect-Standalone.png, besucht am 27.11.2013

[Zaytsev 2013] ZAYTSEV, Juriy: *ECMAScript 5 compatibility table*. 2013. – Erreichbar unter http://kangax.github.io/es5-compat-table/, besucht am 04.11.2013