UNIVERSITÄT PADERBORN

Die Universität der Informationsgesellschaft

Mixed Reality in the Loop

Ein iteratives, prototypenbasiertes Entwurfsvorgehen für die Entwicklung von Mixed Reality Anwendungen

Dissertation

Schriftliche Arbeit eingereicht bei der
Fakultät für Elektrotechnik, Informatik und
Mathematik der Universität Paderborn
zur Erlangung des Grades
Dr. rer. nat.

von

Jörg Stöcklein

Paderborn, den 08.06.2011

Gutachter:

Prof. Dr. Franz J. Rammig, Universität Paderborn, Germany
Prof. Dr. Volker Paelke, Institut de Geomàtica, Spain

Abstract

Mixed Reality in the Loop – Ein iteratives, prototypenbasiertes Entwurfsvorgehen für die Entwicklung von Mixed Reality Anwendungen

Mixed Reality in the Loop ist ein iteratives, prototypenbasiertes Entwurfsvorgehen für *Mixed Reality* Anwendungen. Das Vorgehen besteht aus einem iterativen Prozess, an dessen Ende immer eine testbare Designrepräsentation der Anwendung, kurz ein Prototyp, steht, der für die nächste Iteration verwendet wird. Die Iterationen werden kurz gehalten, so dass ständig eine testbare Designrepräsentation der Anwendung gewährleistet ist. Dem *Mixed Reality in the Loop*-Entwurfsvorgehen steht ein eigens dafür entwickeltes Architekturmuster zur Seite, das es erlaubt, die einzelnen Teile der Anwendung in insgesamt vier Kategorien einzuteilen, die separat und unabhängig voneinander weiterentwickelt werden können. Der zentrale Vorteil bei *Mixed Reality in the Loop* gegenüber anderen Verfahren ist die Entwicklung entlang des *Mixed Reality* Kontinuums. So bieten das Entwurfsvorgehen und der iterative Entwicklungsprozess die Möglichkeit, in einer rein virtuellen Welt mit der Implementierung der MR Anwendung zu beginnen und in den späteren Phasen schrittweise die virtuellen Teile durch ihre realen Gegenstücke zu ersetzten. Das bedeutet für die frühen Entwicklungsphasen eine Implementation in einer fest definierten virtuellen Umgebung, die komplett unter der Kontrolle des Entwicklers liegt. Um eine Einschätzung des Entwicklungsstandes der Prototypen zu erhalten wurde für jede Komponente eine eigene Metrik entworfen, die den Entwicklungsstand anhand verschiedener Parameter errechnet.

Mixed Reality in the Loop – An iterative prototype-based design method for the development of mixed reality applications

Mixed Reality in the Loop is an iterative, prototype-based design method for mixed reality applications. The design method includes an iterative process which results a testable design representation of the application, what is referred as prototype, at each iteration. This prototype is also used for further development in the next iteration. To achieve a persistent design representation of the application, the design method has short iterations. The *Mixed Reality in the Loop* design method includes an additional software architecture supporting a classification of single application parts in four categories. Each classified part can be developed separately and independently, The central advantage over other design methods is the development along the mixed reality continuum. The *Mixed Reality in the Loop* design method provides the opportunity to begin the development of a mixed reality application in a pure virtual environment and to iteratively replace the virtual parts with its real counterparts in later phases of the development process. That imply a defined virtual environment in early development phases which is completely under control of the developer. To get an estimation of the development status for the prototype a metric for each of the four components has been designed, calculating the development status with the help of various parameters.

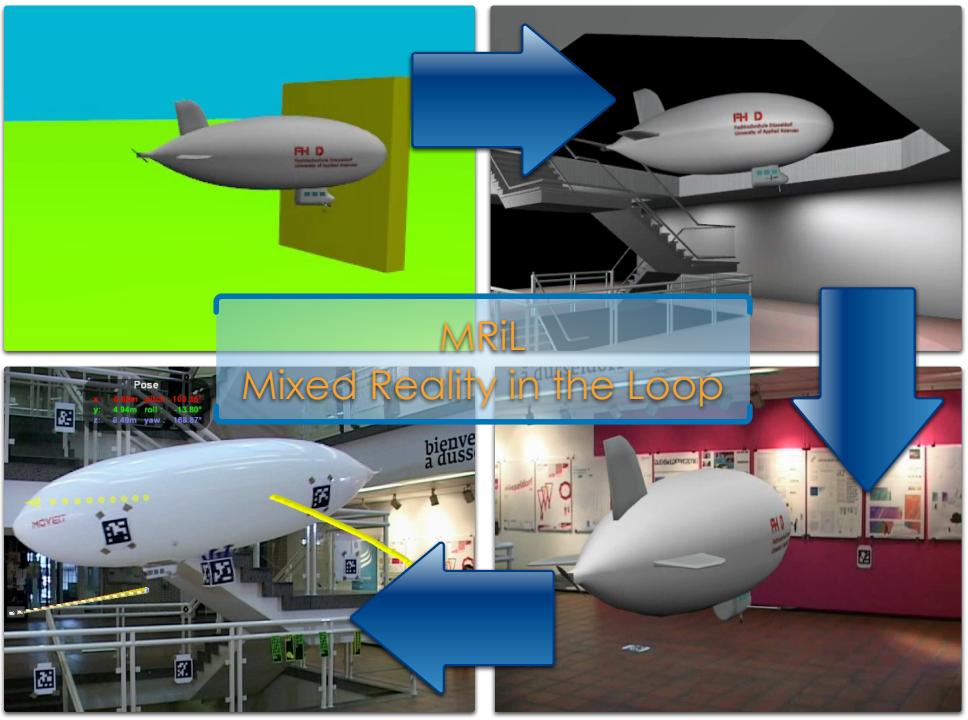
Hinweise

In dieser Arbeit wurden die Verweise auf Kapitel oder Abbildungen speziell formatiert, um die entsprechende Referenz schneller zu finden. Verweise sind folgendermaßen aufgebaut: <Kapitel>$_{<\text{Seite}>}$. <Kapitel> steht für das Kapitel, in dem die Referenz steht und die unten angehängte <Seite> gibt die jeweilige Seite an. Diese Formatierung existiert jedoch nur, sobald sich die Referenz nicht auf der aktuellen Seite befindet. Diese Formatierung erlaubt es dem Leser, schneller die Quelle des Verweises zu finden.

Inhaltsverzeichnis

KAPITEL 1

Einführung



In diesem Kapitel versuche ich mein entwickeltes „Mixed Reality in the Loop“-Entwurfsvorgehen (abgekürzt: MRiL) zu motivieren, indem ich aufzeige, dass in der Entwicklung vom *Mixed Reality* Anwendungen ein Vorgehen fehlt, das entlang des *Mixed Reality* Kontinuums aufbaut und mit Hilfe von Prototypen eine immer testbare Designrepräsentation bietet. Anschließend an die Motivation erläutere ich die Ziele, die ich mir für diese Arbeit gesetzt habe und die mit Hilfe des „Mixed Reality in the Loop“-Entwurfsvorgehens erreicht werden sollen. Daraufhin stelle ich kurz den aktuellen Stand der Arbeit vor.

Zum Ende des Kapitels folgt eine Beschreibung der Strukturierung dieser Arbeit.

1.1 Motivation

Mit der Entwicklung schneller Hardware, sowohl der CPUs als auch der Grafikeinheiten, ergab sich die Möglichkeit, reale Videobilder in Echtzeit[1] analysieren zu können. Ende der Neunziger Jahre begannen Forscher mit der Echtzeitanalyse der Videobilder, um so die Struktur der aufgenommenen Objekte zu ermitteln. Mit Hilfe kleiner schwarz-weiß Piktogramme, der sogenannten *Marker*, gelang es, Position und Orientierung eines Objektes im dreidimensionalen Raum nur unter Zuhilfenahme eines, von einer Videokamera aufgenommenen, 2D-Bildes zu ermitteln und an dieser Position ein virtuelles 3D-Objekt zu positionieren. *Augmented Reality* (AR), also die angereicherte Realität, war geboren. In den folgenden Jahren wurde das Thema *Augmented Reality* detailliert erforscht, sowohl im Theoretischen, indem der Begriff AR definiert und in Zusammenhang mit der echten Realität gesetzt wurde, als auch im Praktischen, indem Softwarelösungen angeboten wurden, die es einer breiten Masse an Entwicklern ermöglichten, selbst AR Anwendungen zu realisieren.

Anfänglich wurden die technischen Methoden, die es ermöglichten, *Augmented Reality* anzuwenden, verstärkt entwickelt und erforscht. Nachdem die ersten reinen API[2]-basierten Softwarelösungen für die breite Masse der Entwickler zur Verfügung standen, konzentrierten sich viele Forscher auf die grundlegenden Kamera-Tracking[3] Verfahren und deren Verbesserung. Die ersten Konferenzen, die sich speziell mit *Augmented Reality* und später auch mit dem erweiterten Gebiet von *Mixed Reality* (MR) auseinander setzten, wurden veranstaltet, darunter beispielsweise die IEEE ISMAR, die zum ersten Mal im Jahre 1998 stattfand [IEE98]. An den dort vorgestellten Beiträgen ist gut zu erkennen, welche Themen im Bereich *Mixed Reality* über die Jahre die Aufmerksamkeit der Forscher erhielten. Und genau hier ist zu sehen, dass viel Energie in die Erforschung der Basistechnik geflossen ist, allerdings erst sehr viel später erkannt wurde, dass sich die Entwicklung von MR Anwendung von der Entwicklung traditioneller Software in vielen Punkten unterscheidet.

[1] In dieser Arbeit ist unter dem Begriff *Echtzeit* ein weiches Echtzeitverhalten zu verstehen, das sich an der menschliche Wahrnehmung für bewegte Bilder orientiert. Da ein Mensch ab ca. 16 Bilder pro Sekunde eine flüssige Bewegung erkennt, liegen die Reaktionszeit somit bei $\leq 63ms$.

[2] API = **A**pplication **P**rogramming **I**nterface – Eine Programmierschnittstelle auf Quelltextebene.

[3] Mit *Tracking* bezeichnet man die kontinuierliche Positionsbestimmung realer Objekte im Raum. Die Positionsbestimmung kann Zwei- oder Dreidimensional erfolgen.

Es stellte sich nach einiger Zeit der Entwicklung von MR Anwendungen heraus, dass es nicht ausreicht, nur eine reine Low Level API-basierte Programmierunterstützung zu nutzen. Durch die zunehmende Komplexität der Projekte, die eine MR Unterstützung integrierten, kamen die Entwickler bald an die Grenzen ihrer Möglichkeiten strukturiert zu entwickeln. Da kein einheitliches Vorgehen für den Entwurf von AR Anwendungen existierte, mussten viele Lösungen individuell für jedes Projekt entworfen werden, testbare Designrepräsentationen der Software waren nicht vorhanden, so dass die Projekte erst kurz vor Ende der Entwicklung wirklich getestet werden konnten. Des Weiteren waren die Entwickler an die Technik gebunden, auf die sie sich zu Beginn der Entwicklung festgelegt hatten. Da im Laufe der Zeit einige konkurrierende Lösungen aufkamen, die jedoch nicht unbedingt untereinander kompatibel waren, teilweise aber Vorteile gegenüber der Konkurrenz boten, war es den Entwicklern nur schwer möglich, die Basistechnologien einfach zu wechseln.

Aus dieser Problematik heraus entstanden die ersten High Level Entwicklungsumgebungen, die es sich zur Aufgabe gemacht hatten, die MR Anwendungsentwicklung von den Basistechnologien zu trennen. Ein frühes Beispiel einer solchen Entwicklungsumgebung ist DART [MGDB04], das im Kapitel 3_{90} vorgestellt wird. Auch die von uns entwickelte Integration in die Szenegraph-Bibliothek Java3D zielte auf diese Trennung von Basistechnologie und Anwendungsentwicklung ab [GRSP02]. Die Trennung von Basistechnologie und Anwendungsprogrammierung war ein Schritt in die richtige Richtung, allerdings fehlte zu dieser Zeit komplett ein Entwurfsvorgehen, das beschreiben konnte, wie Entwickler *Mixed Reality* Anwendungen effizient programmieren können.

In den folgenden Jahren wurde auch der Bereich des Entwurfsvorgehens erforscht und es wurden Verfahren vorgestellt, die sowohl das Entwurfsvorgehen selbst als auch ein eigenes Modell zur Entwicklung anboten. Was allerdings nicht vorgestellt wurde, war ein Vorgehen einschließlich Werkzeugumgebung, das es ermöglichte, *Mixed Reality* Anwendungen mit einer immer testbaren Designrepräsentation zu realisieren. Genau hier setzt meine Arbeit an und versucht diese Lücke zu schließen.

Mit „Mixed Reality in the Loop“ steht dem Entwickler von *Mixed Reality* Anwendungen ein Entwurfsvorgehen zur Verfügung, das ihn durch die Phasen der Entwicklung der Anwendungen leitet. Mehr noch, das Vorgehen ist ein iterativer Prozess, an dessen Ende immer ein testbarer Prototyp der Anwendung steht, der für die nächste Iteration verwendet wird. Die Iterationen werden kurz gehalten, so dass ständig eine testbare Designrepräsentation der Anwendung existiert.

Dem „Mixed Reality in the Loop“-Entwurfsvorgehen steht ein eigens dafür entwickeltes Architekturmuster zur Seite, das es erlaubt, die einzelnen Teile der Anwendung in insgesamt vier Kategorien einzuteilen, die separat und unabhängig voneinander weiterentwickelt werden können.

Der zentrale Vorteil bei „Mixed Reality in the Loop“ gegenüber anderen Verfahren ist jedoch die Entwicklung entlang des *Mixed Reality* Kontinuums (siehe zur Begriffserklärung Kapitel 2.4$_{54}$). So bieten das Entwurfsvorgehen und der iterative Entwicklungsprozess die Möglichkeit, in einer rein virtuellen Welt mit der Implementierung der MR Anwendung zu beginnen und in den späteren Phasen schrittweise die virtuellen Teile durch ihre realen Gegenstücke zu ersetzten. Das bedeutet für die frühen Entwicklungsphasen eine Implementation in einer fest definierten virtuellen Umgebung, die komplett unter der Kontrolle des Entwicklers liegt. In der späteren Entwicklung, sobald die Grundfunktionalität der Anwendung ausreichend stabil läuft, können Teile dieser Umgebung dann durch reale Komponenten ersetzt werden. So ist es beispielsweise möglich, neue Algorithmen zuerst in einer sehr eingeschränkten Umgebung auf ihre Korrektheit zu überprüfen und nach erfolgreichem Abschluss dieser Tests die Algorithmen in Komponenten der realen Welt einzusetzen. Des Weiteren ist es möglich, Teile der Anwendung schon zu implementieren, obwohl die realen Komponenten entweder noch nicht existieren oder die technischen Bedingungen noch nicht geschaffen sind, mit ihnen aus der Anwendung heraus zu interagieren. Auch der Weg zurück, aus der realen Welt in die virtuelle Welt, ist mit „Mixed Reality in the Loop“ möglich. So können z. B. neue Versionen von Algorithmen zuerst in der virtuellen Welt validiert werden, bevor sie die Komponenten der realen Welt steuern.

Zusammenfassend bietet das „Mixed Reality in the Loop“-Entwurfsvorgehen einen Lösungsweg, wie schrittweise aus einer virtuellen Anwendung eine *Mixed Reality* Anwendung entstehen kann. Es bietet ein Vorgehen, um eine Transition vom *virtual Prototyping* und der finalen Anwendung erfolgreich zu realisieren.

1.2 Ziel der Arbeit

Diese Arbeit hat zum Ziel, ein werkzeuggestütztes, prototypenbasiertes, iteratives Entwurfsvorgehen für *Mixed Reality* Anwendungen zu entwickeln. Dieses Entwurfsvorgehen soll entlang des *Mixed Reality* Kontinuums führen und folgende Konzepte beinhalten:

Entwurfsvorgehen: Das Vorgehen zur Entwicklung von *Mixed Reality* Anwendungen soll auf einem iterativen, prototypenbasierten Ansatz basieren. Es soll eine Werkzeugunterstützung bieten, so dass die Entwicklung softwaretechnisch getragen wird.

Architekturmuster: Die Grundlage des Entwurfsvorgehens soll ein Architekturmuster sein, das die iterative Anwendungsentwicklung unterstützt. Es basiert auf dem bekannten MVC Architekturmuster (Kapitel 2.2.1$_{45}$) und wurde um eine Komponente erweitert, um die Besonderheit bei *Mixed Reality* Anwendungen zu unterstützten.

Akteurmodell: Zur Verfeinerung des Architekturmusters soll ein eigenes Akteurmodell verwendet werden. Dieses gewährleistet bei der Entwicklung der *Mixed Reality* Prototypen kurze Iterationen und gewährleistet somit eine ständig testbare Repräsentation der aktuellen Anwendung.

Metrik: Es soll eine Metrik entwickelt werden, die den entsprechenden Entwicklungstand des aktuellen Prototypen charakterisieren kann. Diese Metrik soll das Architekturmuster berücksichtigen und auf der Entwicklung der einzelnen Komponenten basieren.

Verglichen mit anderen aktuellen Arbeiten im Bereich des *Mixed Reality* Entwurfs hat diese Arbeit folgende Herausstellungsmerkmale:

1. Die Entwicklung der Anwendung geschieht entlang des *Mixed Reality* Kontinuums, was bedeutet, dass die Applikation aus der reinen virtuellen Welt in die reale Welt entwickelt wird. Komponenten und Objekte, die am Anfang der Entwicklung virtuell definiert werden, können im Laufe der Entwicklung zu realen Objekten übergehen. Auch der umgekehrte Fall ist möglich, dass reale Objekte wieder zu virtuellen Objekten werden. Diese Transition zwischen der realen und der virtuellen Welt ist sinnvoll, wenn am Anfang der Entwicklung die realen Objekte noch nicht zur Verfügung stehen, oder wenn im Laufe der Entwicklung auf eine spezielle Ausprägung getestet werden soll und deshalb nur die beteiligten Objekte in der realen Form vorhanden sein sollen.

2. Die Anwendung ist in kleine Komponenten, die sogenannten Akteure, die anhand des vorgestellten Architekturmusters klassifiziert und entsprechend entwickelt werden können, aufgeteilt.

3. Eine Metrik erlaubt den Entwicklungsstand bezogen auf Klassifizierung der einzelnen Akteure und so den gesamten Entwicklungsstand der *Mixed Reality* Anwendung zu ermitteln.

4. Die softwaremäßige Wiederverwendbarkeit der Akteure wird mit Hilfe des Adapter-Prinzips gelöst, so dass in den meisten Fällen die Akteure nicht neu programmiert werden müssen. Durch eine definierte Schnittstelle ist so auch der Austausch von virtuellen zu realen Akteuren möglich.

5. Ein eigenes iteratives, prototypenbasiertes Entwurfsvorgehen erlaubt kurze Iterationszyklen und eine ständig testbare Repräsentation der *Mixed Reality* Anwendung. Durch die Wiederverwendbarkeit können die Iterationszyklen noch kürzer gehalten werden.

6. Durch die Entwicklung entlang des *Mixed Reality* Kontinuums und der Klassifizierung des Architekturmusters können Komponenten während der Entwicklung unterschiedlich priorisiert werden.

7. Eine Softwareumgebung, die das komplette Entwurfsvorgehen unterstützt und so die Entwicklung einer *Mixed Reality* Anwendung von der rein konzeptionellen Ebene in die Machbarkeit überführt.

Nach der Entwicklung der konzeptionellen Grundlagen dieser Arbeit wurde das Entwurfsvorgehen zunächst an einem kleineren Beispiel getestet. Dieses entstand ohne eine spezielle Entwicklungsumgebung, so dass zuerst ein kleiner Umfang des Entwurfsvorgehens validiert werden konnte.

Während der Arbeit entstanden zwei voneinander unabhängige Entwicklungsumgebungen, die unterschiedliche Aspekte der Entwicklung fokussierten. Die erste Entwicklungsumgebung basiert auf einem proprietären 3D Autorenwerkzeug, das es dem Entwickler von Anwendungen ermöglicht, diese visuell (und nicht wie sonst üblich textuell) zu entwickeln. Dieses proprietäre Werkzeug wurde mit Hilfe von Plug-ins dahingehend erweitert, dass die Entwicklung von *Mixed Reality* Anwendungen ermöglicht wurden. Allerdings war es nicht möglich, alle Konzepte des Entwurfsvorgehens zu realisieren. Aus diesem Grund wurde ein komplett eigenes Werkzeug, das speziell für das „*Mixed Reality* in the Loop"-Entwurfsvorgehen ausgearbeitet wurde, entworfen. Hier war es möglich, alle Aspekte von MRiL zu verwenden. Mit Hilfe dieses Werkzeuges wurde dann ein komplexes Beispiel realisiert, um die Anwendbarkeit von MRiL zu demonstrieren.

1.3 Aktueller Stand der Arbeit

Aktuell ist das Entwurfsvorgehen konzeptionell vollständig (siehe Kapitel 4_{97}). Die beiden Entwicklungsumgebungen sind komplett implementiert und einsatzbereit. Da die Entwicklungsumgebungen, die auf dem proprietären 3D Autorenwerkzeug basieren, mit Hilfe von Plug-ins realisiert wurden, ist nicht jede Funktionalität für alle denkbaren Fälle implementiert. Hier müssten für konkrete Projekte entweder die vorhandenen Plug-ins angepasst oder neue Plug-ins entwickelt werden. Dies ist jedoch im Sinne des Autors, da die Entwicklung einer *Mixed Reality* Anwendung die Implementierung spezialisierter Plug-ins nicht ausschließt sondern, gerade durch die einfache Einbindung in die Entwicklungsumgebung, ermöglicht. Auch in der aus dieser Arbeit hervorgegangenen eigenen Entwicklungsumgebung ist es erwünscht, spezielle Funktionalität selbst mit Hilfe von Akteuren zu entwickeln. Die Grundstrukturen für solche Entwicklungen sind allerdings in beiden Entwicklungsumgebungen explizit vorhanden.

Bei den Beispielen, die mit Hilfe der eigenen Entwicklungsumgebung entstanden sind, ist die Implementierung zum größten Teil vollendet. Da es sich bei dem Beispiel allerdings um ein interdisziplinäres Projekt mehrerer Fachgruppen, sowohl an der Universität Paderborn als auch an der Fachhochschule Düsseldorf, handelt, konnten einige Punkte nur konzeptionell entwickelt werden (siehe dazu Kapitel 5_{159}). Leider wurde die Entwicklung einer erforderlichen Hardwarekomponente nicht fristgerecht vollendet, so dass die Prototypen, die diese Komponente erfordern, nicht vollständig implementiert werden konnten. Konzeptionell allerdings wurden alle Beispiele komplett entwickelt.

1.4 Strukturierung der Arbeit

Dem aktuellen Kapitel folgt das Kapitel über die Grundlagen. Hier werden bekannte Vorgehensmodelle (Kapitel 2.1_{12}) und Architekturmuster (Kapitel 2.2_{43}) sowie eine kurzer Überblick über Modellbildung und Simulation (Kapitel 2.3_{49}) vorgestellt und eine Einführung in *Mixed Reality* (Kapitel 2.4_{54}) gegeben. Die vorgestellten Themen sollen ein grundlegendes Wissen in den jeweiligen Bereichen vermitteln, so dass auch ein Leser, der in diesen Gebieten nicht bewandert ist, die Arbeit verstehen kann.

Es folgt das Kapitel über die aktuellen Forschungsergebnisse in dem Bereich dieser Arbeit. Dieses Kapitel unterteilt sich in die *Mixed Reality* Entwurfskonzepte (Kapitel 3.2_{60}), die Entwurfskonzepte mit Werk-

zeugumgebung (Kapitel 3.3_{66}) und den reinen Softwareumgebungen und Softwarelösungen (Kapitel 3.4_{87}). In jedem Gebiet wurden Arbeiten ausgewählt, die sich in Teilen dieser Arbeit gleichen, allerdings nicht den kompletten Umfang dieser Arbeit besitzen. Die jeweiligen Unterschiede wurden in einer Tabelle kenntlich gemacht.

Kapitel 4_{97} ist das erste der beiden Hauptkapitel. Hier wird das komplette „Mixed Reality in the Loop"-Entwurfvorgehen vorgestellt. Bei der Anforderungsanalyse (Kapitel 4.1_{98}) wird festgestellt, für welche Anwendungen sich das Vorgehen eignet. Die Vorgehensweise (Kapitel 4.2_{101}) beschreibt das Verfahren, wie Anwendungen mit dem Entwurfsvorgehen zu entwickeln sind. Die benötigten Methoden (Kapitel 4.3_{103}) werden im darauf folgenden Kapitel beschrieben. Mit Hilfe eines kleinen Beispiels (Kapitel 4.4_{128}) soll gezeigt werden, wie sich das Entwurfsvorgehen auch ohne Werkzeugunterstützung verwenden lässt. Nach dem Beispiel werden die zwei Softwareumgebungen (Kapitel 4.5_{136}) vorgestellt, die auf dem „Mixed Reality in the Loop"-Entwurfsvorgehen basieren.

Um sowohl das Entwurfsvorgehen als auch eine der Softwareumgebungen zu überprüfen, wird in Kapitel 5_{159} ein nicht triviales Beispiel entwickelt, das zehn aufeinander aufbauende Prototypen umfasst. Bei der Realisierung der einzelnen Prototypen wird das Entwurfsvorgehen und die Berechnung der Metrik komplett angewendet, so dass sich immer der Stand des aktuellen Prototypen ableiten lässt. Die einzelnen Prototypen fokussieren dabei größtenteils jeweils eine andere Ausprägung der Anwendung, beispielsweise eine Verfeinerung des Modells oder eine überarbeitete Steuerung.

Im Kapitel 6_{201} wird abschließend mein Vorgehen den aus der Literatur bekannten und in Kapitel 3_{59} vorgestellten Arbeiten gegenübergestellt und verglichen. Die einzelnen für mein Vorgehen relevanten Arbeiten werden noch einmal kurz zusammengefasst und die Unterschiede zu meinem Vorgehen aufgezeigt.

Im letzten Kapitel (Kapitel 7_{207}) fasse ich die Ergebnisse meiner Arbeit zusammen und gebe einen Ausblick auf zukünftige Forschungsschwerpunkte, die aufbauen auf dem hier vorgestellten Entwurfsvorgehen erarbeitet werden könnten.

1.5 Zusammenfassung

In diesem Kapitel habe ich motiviert, warum es sinnvoll ist, ein Entwurfsvorgehen entlang des *Mixed Reality* Kontinuums zu entwickeln. Ich habe die Ziele dieser Arbeit aufgezeigt und erläutert, in wie weit

sich das hier vorgestellt Entwurfsvorgehen von anderen Arbeiten unterscheidet. Es wurde der aktuelle Stand der Entwicklung des Vorgehens erläutert und die Strukturierung der gesamten Arbeit vorgestellt. Im nun folgenden Kapitel gehe ich auf grundlegende Verfahren in den Bereichen ein, die ich in meiner Arbeit benötige.

KAPITEL 2

Grundlagen

Dieses Kapitel behandelt die grundlegenden Methoden und Strategien, auf der meine Arbeit basiert. In Kapitel 2.1$_{12}$ werden verschiedene Vorgehensmodelle vorgestellt, die für die Erstellung von Softwareanwendungen entwickelt wurden und noch heute im Einsatz sind. Das Kapitel 2.2$_{43}$ beschreibt unterschiedliche Architekturmuster, die für verschiedene Probleme in der Softwareentwicklung geschaffen wurden. Die verschiedenen *In-the-Loop* Simulationsverfahren, die u. a. zur Entwicklung von mechatronischen Systemen verwendet werden, sind in Kapitel 2.3$_{49}$ aufgeführt und werden dort kurz erläutert. Kapitel 2.4$_{54}$ erklärt das *Reality-Virtuality* Kontinuum (RV), welches Grundlage meines Entwicklungsprozesses ist, und erläutert es an Beispielen.

Die hier vorgestellten Methoden, Strategien, Verfahren und Definitionen sind in der Literatur wohl bekannt und gelten als Standard bzw. Grundlage in der Softwareentwicklung. Die meisten Verfahren werden seit vielen Jahren in der Softwareentwicklung erfolgreich eingesetzt, gerade im Bereich Vorgehensmodelle und Architekturmuster. Teilweise sind die Methoden auch schon überholt, werden allerdings für das grundlegende Verständnis der in Kapitel 3$_{59}$ vorgestellten Stand der Forschung aufgeführt. Dieses Kapitel bietet somit einen groben Überblick über die Methoden, Strategien, Verfahren und Definitionen, die zum Verständnis meiner Arbeit dienen sollen. Aus diesem Grund beziehe ich mich in diesem Kapitel teilweise auf Artikel der freien Enzyklopädie *Wikipedia* [Wik11], referenziere jedoch immer für die einzelnen Themen die grundlegenden wissenschaftlichen Veröffentlichungen bzw. Fachbücher, so dass die einzelnen vorgestellten Themen immer wissenschaftlich belegt sind.

2.1 Vorgehensmodelle

Ein Vorgehensmodell im Allgemeinen organisiert einen Prozess der gestaltenden Produktion in verschiedene, strukturierte Phasen, denen wiederum entsprechende Methoden und Techniken der Organisation zugeordnet sind. Die Aufgabe eines Vorgehensmodells ist es, die allgemein in einem Gestaltungsprozess auftretenden Aufgabenstellungen und Aktivitäten in einer deutlich erkennbaren logischen Ordnung darzustellen.

Ein Vorgehensmodell in der Softwareentwicklung im Speziellen ist ein angepasstes Vorgehensmodell, welches bei der professionellen Anwendungsentwicklung verwendet wird. Es dient dazu, die Softwareentwicklung übersichtlicher zu gestalten und in der Komplexität beherrschbar zu machen.

Komplexe Software ist nur schwer zu erstellen und zu warten, so dass sich Softwareentwickler eines Planes zur Entwicklung von Software bedienen. Dieser Plan – das so genannte Vorgehensmodell – unterteilt den Entwicklungsprozess in überschaubare, zeitlich und inhaltlich begrenzte Phasen. Die Software wird daher Schritt für Schritt fertiggestellt. Dem eigentlichen Entwicklungsprozess stehen dabei sowohl das Projektmanagement als auch die Qualitätssicherung begleitend zur Seite.

Vorgehensmodelle spalten die einzelnen Aktivitäten auf verschiedene Phasen im Entwicklungsprozess auf. Diese werden dann, ggf. mit geringen Modifikationen, einmalig (z. B. Wasserfallmodell, siehe Kapitel 2.1.2$_{14}$) oder mehrfach (z. B. Spiralmodell, siehe Kapitel 2.1.3$_{17}$) durchlaufen. Bei mehrmaligen Durchlauf erfolgt eine iterative Verfeinerung der einzelnen Softwarekomponenten. Um die optimalen Vorgehensmodelle herrscht Uneinigkeit. In der Regel unterscheiden sie beim Entwicklungsprozess mindestens zwei große Tätigkeitsgruppen: Die, von der programmiertechnischen Realisierung unabhängige, Analyse von Geschäftsprozessen (Geschäftsprozessmodell und Datenmodell) einerseits und die EDV-technische Realisierung (Design und Programmierung) andererseits.

Vorgehensmodelle unterscheiden sich wesentlich in ihrem Detaillierungsgrad. Dabei sind z. B. der *OOTC-Approach* [IOOTC97] oder der *Rational Unified Process* [KR96] detailliert ausgearbeitete Vorgehensweisen, die den an der Entwicklung Beteiligten konkrete Arbeitsanweisungen an die Hand geben. Das V-Modell (Kapitel 2.1.4$_{19}$) nimmt diesbezüglich übrigens eine hybride Stellung ein, da es sowohl ein Prinzip (jeder Stufe der Entwicklung entspricht eine Testphase) als

auch (wie zumeist gebräuchlich) ein detailliertes Modell ist.

Agile Softwareentwicklung ist der Oberbegriff für den Einsatz von Agilität (lat. *agilis*, zu deutsch „flink, beweglich“) in der Softwareentwicklung. Die Agile Softwareentwicklung beschäftigt sich mit Methoden, die den Entwickler kreativ arbeiten und Verwaltungsaspekte zurücktreten lassen. Je nach Kontext bezieht sich der Begriff auf Teilbereiche der Softwareentwicklung – wie im Fall von *Agile Modeling* – oder auf den gesamten Softwareentwicklungsprozess – exemplarisch sei *Feature Driven Development* (Kapitel 2.1.8$_{28}$) oder *Extreme Programming* (Kapitel 2.1.10$_{31}$) angeführt. Agile Softwareentwicklung versucht mit geringem bürokratischem Aufwand und wenigen Regeln auszukommen.

2.1.1 Typen von Vorgehensmodellen

Es existieren insgesamt drei unterschiedliche Typen von Vorgehensmodellen [Wik11]:

Software-Entwicklungsprozesse: Sie dienen zur Steuerung der Softwareentwicklung von der Konzeption bis zum endgültigen Einsatz inklusive der anfallenden Änderungen einer Software. Es gibt viele verschiedene Prozesse, wobei die unten angegebenen die bekanntesten dieser Klasse sind [Wik11]:

- Wasserfallmodell
- Spiralmodell
- V-Modell

Software-Lebenszyklusmanagement: Sie erweitert die Phasen über den gesamten Lebenszyklus einer Software. Das Vorgehensmodell definiert die Anforderungen an betriebliche Prozesse (das „WAS“) und beschreibt die konkreten, EDV-technisch realisierten Prozesse (das „WIE“). Dieser Typ ist eine Mischung aus Ist-Beschreibung und normativer Vorgabe. Je nach Standardisierungsgrad werden verschiedene Entwicklungsstufen vergeben. Unternehmen können sich diese Entwicklungsstufen von externen Stellen zertifizieren lassen. Ein relevantes Beispiel hierfür ist [Wik11]:

- Norm ISO/IEC 12207

Softwareentwicklungs-Philosophie: Eine konkrete Programmierer-Philosophie bzw. ein bestimmter Ansatz, wie Software nach Ansicht

der Entwickler am besten entwickelt werden sollte. Diese Philosophien beinhalten sehr oft auch Prozesselemente und werden daher ebenfalls als Prozessmodell bezeichnet. Die unten angegebenen Modelle sind nur ein kleiner, aber für diese Arbeit relevanter, Teil:

- Norm DIN ISO 13407
- ModellgetriebeneSoftwareentwicklung (MDSD)
- Feature Driven Development (FDD)
- Rational Unified Process (RUP)
- Extreme Programming (XP)
- Scrum
- Prototyping

In den folgenden Kapiteln werden die oben genannten Vorgehensmodelle kurz vorgestellt und beschrieben. Einiger der Modelle werden ausführlicher beschrieben, da sie zum Teil Grundlage meines Entwurfsvorgehens sind. Der Vollständigkeit halber und wegen des Verständnisses stelle ich auch teilweise sehr frühe Vorgehensmodelle der Softwareentwicklung vor.

2.1.2 Wasserfallmodell

Das Wasserfallmodell ist ein lineares und nicht-iteratives Vorgehensmodell, bei dem der Entwicklungsprozess in Phasen organisiert wird. Dabei gehen die Phasenergebnisse wie bei einem Wasserfall immer als bindende Vorgaben für die nächsttiefere Phase ein [Wik11].

Jede Phase im Wasserfallmodell hat vordefinierte Start- und Endpunkte, die eindeutig definierte Ergebnisse liefern. In Meilensteinsitzungen am jeweiligen Phasenende werden die Ergebnisdokumente verabschiedet. Zu den wichtigsten Dokumenten zählen dabei das Lastenheft sowie das Pflichtenheft. In der betrieblichen Praxis gibt es viele Varianten des reinen Modells. Es ist aber das traditionell am weitesten verbreitete Vorgehensmodell [Wik11].

Das Wasserfallmodell wurde in seiner ursprünglichen Form zum ersten Mal von Dr. Winston W. Royce 1970 präsentiert [Roy87]. Man konnte aber bereits schon früher die Grundstrukturen des heutigen Wasserfallmodells in verschiedenen Publikationen der U.S. Air Force

und aus der Industrie erkennen. Es fand z. B. Einfluss bei der Entwicklung eines *Air Defense Software Systems* namens SAGE (*semi automated ground environment*) in den 1950ern [MRZ11].

Der Name „Wasserfall" kommt von der häufig gewählten grafischen Darstellung der fünf bis sechs als Kaskade angeordneten Phasen, wie in Abbildung 2.1_{16} zu sehen. Eigentlich ist das Wasserfallmodell eine Verbesserung des einfachen Phasenmodells, das Herbert D. Benington bereits 1956 vorgestellt hatte [Ben56]. In dem als *Nine Phase Stagewise Model* bekannten Ansatz wurde der Entwicklungsprozess für Software in insgesamt 9 Phasen eingeteilt. Royce's Einteilung der Phasen erfolgte so, dass jede Phase von ihrer vorhergehenden Phase abhängig ist. Somit war es möglich, den Prozess in Einzelteile zu zerlegen [MRZ11].

Dass sich das Modell ursprünglich nicht sonderlich durchsetzte lag vor allem daran, dass kein Informationsfluss (engl. *feedback*) entgegen des eigentlichen Phasenverlaufs existierte. Diese zu einem späteren Zeitpunkt eingeführte Erweiterung des Modells, die auch als Rückkopplung bezeichnet wird, ist auch die Ursache dafür, warum das von Barry Boehm in den 80er Jahren vorgestellte Modell nicht nur großes Interesse und viele Anwender fand, sondern noch heute als das Wasserfall Modell bezeichnet wird [Boe81]. Die Rückkopplung ermöglichte die Behebung aufgetretener Fehler in der nächsthöheren Phase, sofern in der aktuellen Phase Fehler erkannt wurden. Das Wasserfallmodell kann im Allgemeinen dort erfolgreich angewendet werden, wo sich Anforderungen, Leistungen und Abläufe in der Planungsphase relativ präzise beschreiben lassen [MRZ11].

Es existieren zwei Varianten des Wasserfallmodells, eine Variante mit 5 Stufen und eine erweiterte Variante mit 6 Stufen, die auch in Abbildung 2.1_{16} zu sehen ist.

Die 5-stufige Variante beinhaltet folgende Phasen [Wik11]:

1. Anforderungsanalyse und -spezifikation (engl. *Requirement analysis and specification*)
2. Systemdesign und -spezifikation (engl. *System design and specification*)
3. Programmierung und Modultests (engl. *Coding and module testing*)
4. Integrations- und Systemtest (engl. *Integration and system testing*)
5. Auslieferung, Einsatz und Wartung (engl. *Delivery, deployment and maintenance*)

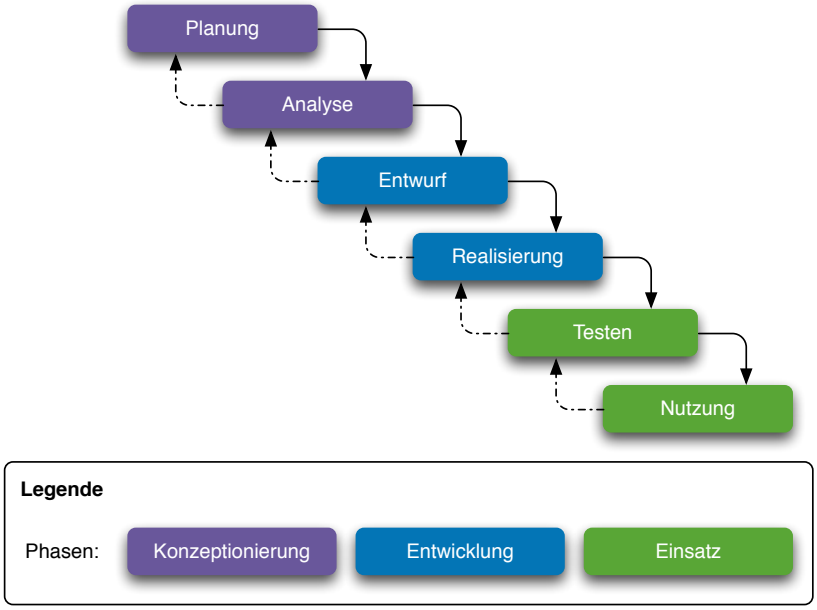


Abbildung 2.1: Ein erweitertes Wasserfallmodell mit Rückkopplung.

Die erweiterte 6-stufige Variante ist in die folgenden Phasen aufgeteilt [Wik11]:

1. Planung (mit Erstellung des Lastenhefts, Projektkalkulation, Projektplan) (engl. *Systems Engineering*)
2. Analyse (mit Erstellung des Pflichtenhefts, Produktmodell, GUI-Modell und evtl. schon Benutzerhandbuch) (engl. *Analysis*)
3. Entwurf (UML, Struktogramme) (engl. *Design*)
4. Realisierung (engl. *Coding*)
5. Testen (engl. *Testing*)
6. Nutzung und Wartung (engl. *Maintenance*)

Beim Wasserfallmodell muss jede Aktivität in der vorgegebenen Reihenfolge und in der vollen Breite vollständig durchgeführt werden, bevor eine neue Aktivität angefangen werden kann. Am Ende jeder Aktivität steht ein fertiggestelltes Dokument, d. h. das Wasserfall-Modell ist ein *dokument-getriebenes* Modell. Der Entwicklungsablauf ist rein sequenziell, d. h. jede Aktivität muss komplett beendet sein, bevor mit der nächsten Aktivität begonnen werden kann. Das Wasserfallmodell orientiert sich am so genannten Top-Down-Verfahren. Es ist einfach, verständlich und benötigt nur wenig Managementaufwand.

Eine Benutzerbeteiligung ist nur in der Anfangsphase vorgesehen, anschließend erfolgen der Entwurf und die Implementierung ohne Beteiligung des Benutzers bzw. Auftraggebers. Weitere Änderungen stellen danach Neuaufträge dar [Wik11].

Da es schwierig ist, bereits zu Projektbeginn alles endgültig und im Detail festzulegen, besteht das Risiko, dass die letztendlich fertiggestellte Software nicht den tatsächlichen Anforderungen entspricht. Um dem zu begegnen, wird oftmals ein unverhältnismäßig hoher Aufwand in der Analyse- und Konzeptionsphase betrieben. Zudem erlaubt das Wasserfallmodell nicht bzw. nur sehr eingeschränkt im Laufe des aktuellen Projekts Änderungen aufzunehmen. Die fertiggestellte Software bildet folglich nicht den aktuellen, sondern den Anforderungsstand zu Projektbeginn wieder. Da größere Softwareprojekte meist auch eine sehr lange Laufzeit haben, kann es vorkommen, dass eine neue Software bereits zum Zeitpunkt ihrer Einführung inhaltlich veraltet ist [Wik11].

2.1.3 Spiralmodell

Das Spiralmodell ist ein Vorgehensmodell in der Softwareentwicklung, das im Jahr 1988 von Barry W. Boehm [Boe88] in seinem Artikel „A Spiral Model of Software Development and Enhancement“ beschrieben wurde. Es ist ein generisches Vorgehensmodell und daher offen für bereits existierende Vorgehensmodelle. Das Management kann immer wieder eingreifen, da man sich spiralförmig voran entwickelt [Wik11]. Das Spiralmodell gehört zu den inkrementellen bzw. iterativen Vorgehensmodellen. Es ist eine Weiterentwicklung des Wasserfallmodells, in der die Phasen mehrfach spiralförmig durchlaufen werden. Es sieht also eine zyklische Wiederholung der einzelnen Phasen vor. Dabei nähert sich das Projekt langsam den Zielen an, selbst wenn sich die Ziele während des Projektfortschrittes verändern. Durch das Spiralmodell wird nach Boehm das Risiko eines Scheiterns bei großen Softwareprojekten entscheidend verringert [Bal08].

Das Spiralmodell fasst den Entwicklungsprozess in der Softwareentwicklung als iterativen Prozess auf. Jeder Zyklus ist in einzelne Quadranten unterteilt, die folgende Aktivitäten enthalten (siehe Abbildung 2.2_{18}):

1. **Festlegung der Ziele:** In dieser Teilphase werden die Ziele der laufenden Phase festgelegt, Alternativen identifiziert und Rahmenbedingungen beschrieben.
2. **Risikoanalyse:** Dieser Teil einer Iteration schätzt die Risiken ab

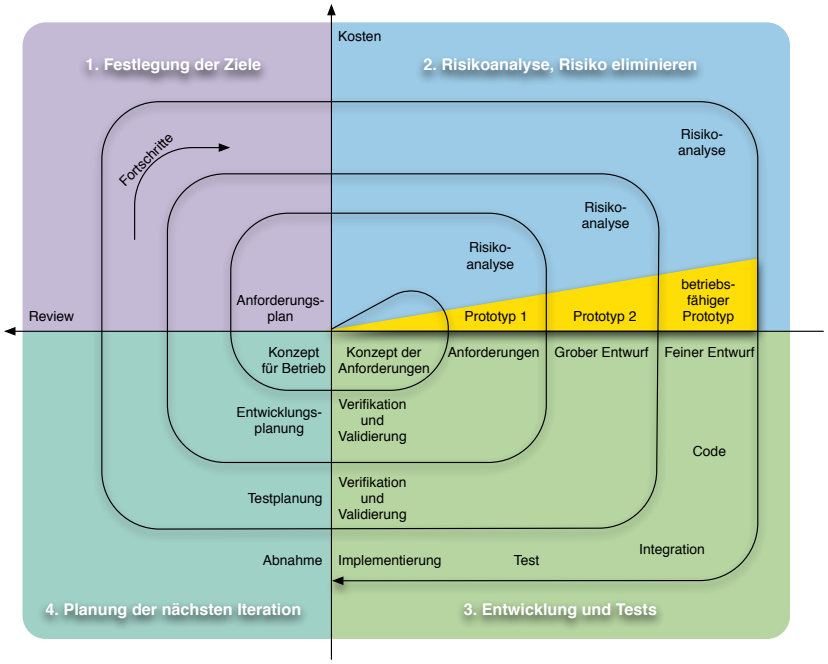


Abbildung 2.2: Spiralmodell nach Boehm.

und versucht sie zu reduzieren, z. B. durch *Prototyping*, Simulation oder Analysen. Es werden Alternativen evaluiert, sollten die Risiken zu hoch sein.

3. **Entwicklung und Tests:** Das Zwischenprodukt wird während dieser Teilphase realisiert und verifiziert.

4. **Planung der nächsten Iteration:** Soll ein Projekt weiter geführt werden, wird in dieser Teilphase die Planung der nächsten auszuführenden Schritte erarbeitet.

Ein wesentlicher Aspekt des Spiralmodells ist die Risikobetrachtung, die von anderen Vorgehensmodellen meist vernachlässigt wird. Hierbei werden zunächst alle Risiken, die im Projekt auftreten können, identifiziert und anschließend bewertet. Wird ein Risiko als zu hoch bewertet versucht werden Alternativen mit geringerem Risiko gesucht. Wird keine Alternative gefunden und das Risiko bleibt bestehen gilt das Projekt als gescheitert. Wenn hingegen keine Risiken mehr existieren, so ist das Projekt kann das Projekt erfolgreich abgeschlossen werden [Wik11].

2.1.4 V-Modell

Das V-Modell ist eine abstrakte und umfassende Methode für das Projektmanagement zur Entwicklung und Realisierung von Softwareprojekten. Dabei resultiert die Bezeichnung *V-Modell* einerseits aus dem ersten Buchstaben von „**V**orgehensmodell“, andererseits aus der V-förmigen Darstellung der Projektelemente aus Spezifikation und Zerlegung (im absteigenden Ast) und Realisierung und Integration im aufsteigenden Ast (siehe Abbildung 2.3[19]).

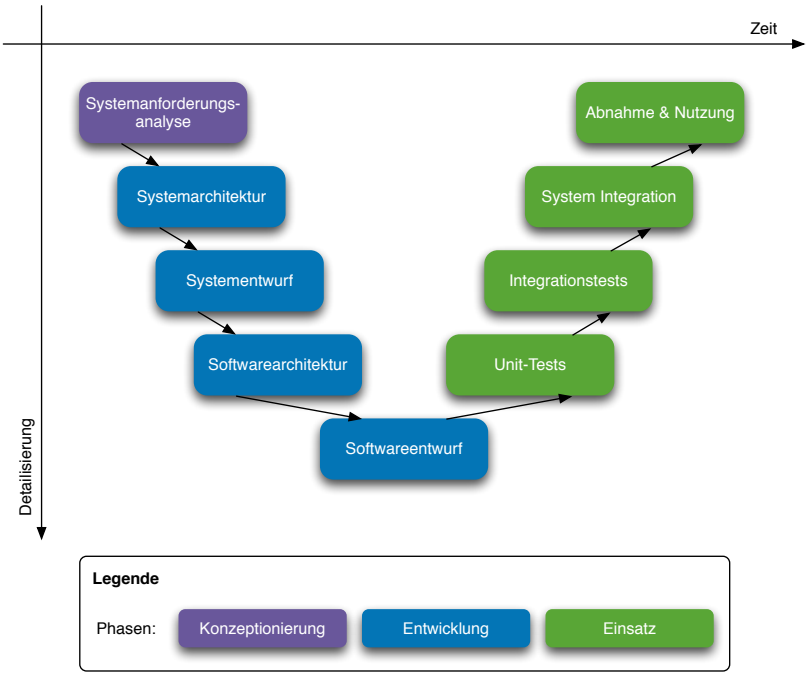


Abbildung 2.3: Phasen des V-Modells über Zeit und Detaillisierung.

Die Idee des V-förmigen Vorgehens wurde von Barry Boehm im Jahre 1979 vorgestellt [Boe79]. Das erste V-Modell wurde 1988 in Deutschland für militärische Zwecke entwickelt, ausgehend aus dem im Jahre 1986 gestarteten Projekt *SEU-WS* (Softwareentwicklungsumgebung für Waffen- und Waffeneinsatzsysteme) des Bundesverteidigungsministeriums. In dieses erste V-Modell wurden dann bis April 1990 die Erkenntnisse aus dem Projekt SEU-IS (Softwareentwicklungsumgebung für Informationssysteme) integriert und die verbesserte Version des V-Modells per Erlass vom Februar 1991 durch den Bundesminister für Verteidigung als Entwicklungsstandard für die Softwareerstellung bei der Bundeswehr festgeschrieben. Inzwischen wird das V-Modell aber auch in der Privatwirtschaft eingesetzt [Wik11].

In der Regel wird eine neue Variante des V-Modells aus der jeweils vorhergehenden Variante entwickelt, sobald ein Verbesserungsbedarf

erkannt wird. Im Gegensatz zu einem klassischen Phasenmodell werden im V-Modell lediglich Aktivitäten und Ergebnisse definiert und es wird keine strikte zeitliche Abfolge gefordert. Insbesondere fehlen die typischen Abnahmen, die ein Phasenende definieren. Dennoch ist es möglich, die Aktivitäten des V-Modells zum Beispiel auf ein Wasserfallmodell (Kapitel 2.1.2$_{14}$) oder ein Spiralmodell (Kapitel 2.1.3$_{17}$) abzubilden [Wik11].

Ein zentraler Punkt des V-Modells ist die Detailisierung. Zu Beginn wird über eine Systemanforderungsanalyse ermittelt, was die zu entwickelnde Software leisten soll. Nach Abschluss dieser Phase erfolgt die erste Detailisierung, in der die Ergebnisse der Systemanforderungsanalyse für die Entwicklung der Systemarchitektur verwendet werden (siehe auch Abbildung 2.3$_{19}$). Ist die Entwicklung an der Systemarchitektur abgeschlossen, wird in einem weiteren Detailisierungsschritt der Systementwurf erarbeitet. Weitere Detailisierungschritte folgen, namentlich die Softwarearchitektur und der Softwareentwurf, d. h. die eigentliche Realisierung der Software. Dies ist die feinste Stufe der Detailisierung und nach Vollendung der Softwarearchitektur ist die Software im Prinzip fertiggestellt. Was fehlt sind die Tests, die das korrekte Arbeitender Software prüfen. Bei den Tests wird nun die Detailisierung mit jeder Phase wieder verringert, so dass erst einzelne Module (*Units*) auf ihre Funktionalität geprüft werden, danach das Zusammenspiel der einzelnen Module miteinander (Integrationstests), folgend von den Tests der Software auf den einzelnen Arbeitsplatzrechnern (*Systemtests*). Sind alle Tests erfolgreich wird die Software abgenommen und kann genutzt werden [Wik11].

Ein Nachteil des V-Modells, der sofort auffällt, ist die geringe Einbindung des Benutzers in den Entwicklungsprozess und das Fehlen von Prototypen für Tests. Somit wird die Software erst einmal komplett entwickelt. Dies hat zur Folge, dass die Systemanforderungsanalyse sehr detailliert ausfallen muss, da logische Fehler, wurden sie nicht erkannt, sehr schwer zu beheben sind.

2.1.5 Norm ISO/IEC 12207

ISO/IEC 12207 definiert einen Rahmen für Prozesse im Lebenszyklus von Software (engl. Software *Life-Cycle Processes*) [ISO95]. In Abbildung 2.4$_{21}$ sind die verschiedenen Phasen eines Lebenszyklus von Software dargestellt. Die Norm beschreibt auf einer sehr abstrakten Ebene alle wichtigen Prozesse des Lebenszyklus einer Software, von der Ideenfindung bis hin zur Stilllegung und den Beziehungen untereinander. Die in der Norm definierten Prozesse bestehen aus Aktivitäten, die in sich wiederum aus einzelnen Aufgaben bestehen.

ISO/IEC 12207 definiert eine Prozessstruktur unter Verwendung einer allgemein akzeptierten Terminologie, sie legt sich nicht fest auf ein bestimmtes Lebenszyklusmodell oder eine bestimmte Entwicklungsmethode. Es werden keine Details bezüglich des Konzepts bei der Durchführung der Aktivitäten und Aufgaben und auch keine Vorschriften bezüglich Namen, Formaten oder Inhalten von Dokumenten vorgegeben [Wik11].

Zusätzlich beschreibt ISO/IEC 12207 wie der Standard auf eine bestimmte Organisation oder auf ein konkretes Projektvorhaben zugeschnitten werden kann [Wik11].

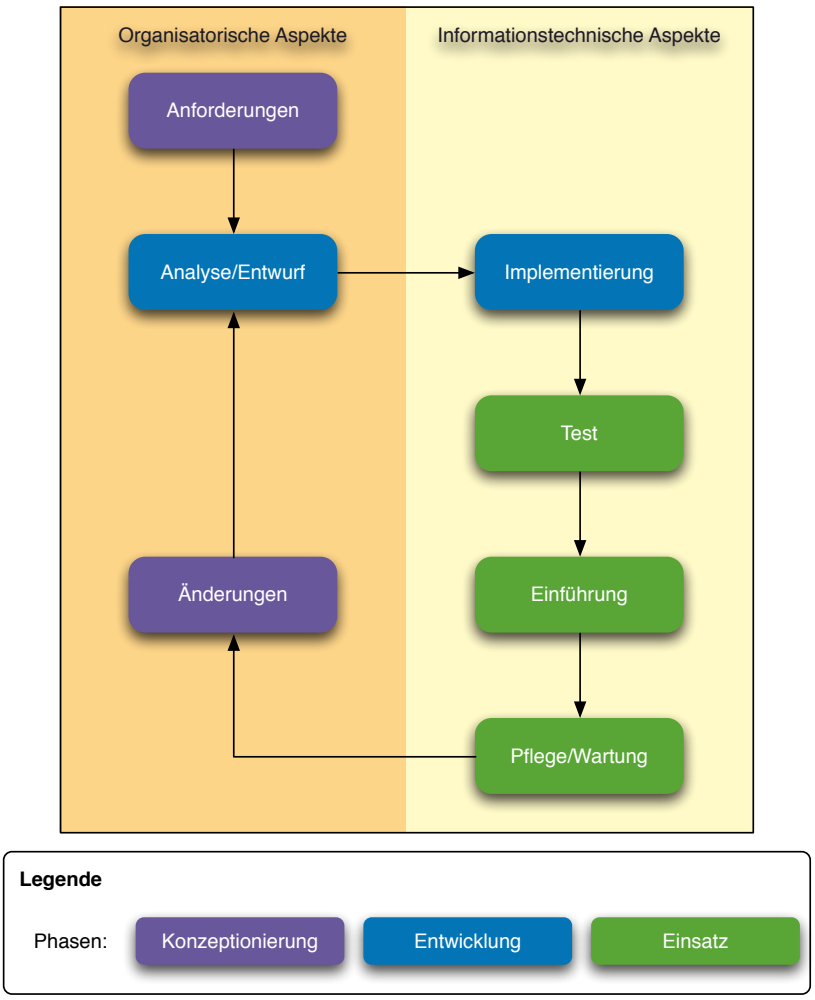


Abbildung 2.4: Phasen eines Softwarelebenszyklus.

Die Norm beschreibt folgende drei Prozesse [ISO95]:

– **Primärprozesse:** Die grundlegenden Prozesse für die Verwendung der ISO/IEC 12207. Folgende Prozesse sind dabei involviert:

Beschaffung: Stellt die Aktivitäten des Beschaffers von Software und Dienstleistungen dar

Lieferung: Beschreibt die Aktivitäten des Lieferers von Software und Dienstleistungen

Entwicklung: Aktivitäten der Entwicklung der Software

Betrieb: Zu diesen Aktivitäten zählen Systemeinführung und Systemtests sowie die Benutzerunterstützung

Wartung: Fehlerbehebung und Beseitigung von Mängel, Verbesserung des Durchsatzes, Anpassung an ein verändertes Umfeld, etc.

– **Unterstützende Prozesse:** Diese Prozesse unterstützen andere Prozesse, in dem sie spezielle Funktionalitäten zur Verfügung stellen, die nachfolgend beschreiben werden:

Dokumentation: Dokumentieren der gesamten Software über die verschiedenen Phasen hinweg

Konfigurationsmanagement: Aktivitäten für organisatorische und verhaltensmäßige Regeln auf den Produktlebenslauf der Software von seiner Entwicklung über Herstellung bis hin zur Betreuung

Qualitätssicherung: Aktivitäten zum Sicherstellen des festgelegtes Qualitätsniveaus der Software

Verifizierung: Formale Überprüfung der Prozesse

Validierung: Aktivität zur inhaltlichen Überprüfung der Prozesse

Joint Review: Aktivität zur Abstimmung zwischen dem Kunden und dem Lieferant/Entwickler

Audit: Untersuchungsverfahren zur Bewertung der Erfüllung von den Anforderungen und Richtlinien der Prozesse

Problembehebung: Aktivität zur Behebung von Problemen bei Prozessen

– **Unternehmensprozesse:** Die Unternehmensprozesse sollen spezielle Prozesse auf Unternehmensebene verwalten und verbessern:

Management: Umfasst die Steuerung von Kernprozessen im gesamten Prozess, mit dem Fokus auf Strukturierung der organisatorischen Rollen und deren Aufgaben

Infrastruktur: Aktivitäten zur Bereitstellung der notwendigen Infrastruktur wie z.B. Hardware, Software oder Werkzeuge

Optimierung: Messen, Überprüfen, Verbessern der Lebenszyklusprozesse

Schulungsmaßnahmen: Aktivität zur Schulung der Benutzer der Software

Da die Norm ISO/IEC 12207 einen Lebenszyklus einer Software beschreibt, ist diese nicht mit den anderen Vorgehensmodellen vergleichbar. Trotzdem kann man Aussagen bzgl. einiger Eigenschaften des Vorgehensmodells treffen. Die drei Prozesse, die die Norm vorschlägt, sind größtenteils nebenläufig, d. h. parallel bearbeitbar. Das ist ein großer Vorteil, da sich mehrere Teams mit verschiedenen Aufgaben gleichzeitig beschäftigen können. Über die gewählte Implementierung und das Vorgehen bei der eigentlichen Softwareentwicklung wird nichts vorgeschrieben, d. h. hier können wieder klassische Modelle (Wasserfall-Modell. Spiralmodell) benutzt werden. Die Norm ist ausgelegt für größere Projekte, denn für kleine bis mittlere Produktionen würde zuviel Mehrarbeit in die Organisation und Verwaltung fließen.

2.1.6 Norm DIN ISO 13407

Die Norm DIN EN ISO 13407 (Benutzerorientierte Gestaltung interaktiver Systeme) [ISO99] beschreibt einen prototypischen benutzerorientierten Softwareentwicklungsprozess. Sollten die Empfehlungen der Norm DIN EN ISO 13407 erfüllt werden kann ein spezieller Entwicklungsprozess als konform betrachtet werden. Die DIN EN ISO 13407 wurde im November 2000 in der deutschen Fassung als DIN-Norm veröffentlicht [Wik11].

Die Norm besteht in ihrem Aufbau sowohl aus den Beschreibungen der Planung benutzerorientierter Gestaltung, als auch aus Erläuterungen zur Entwicklung interaktiver Systeme, die sich darauf konzentrieren, benutzerfreundliche Systeme zu erschaffen. Sie beschreiben in kurzer, übersichtlicher und für eine Norm gut lesbaren Form einen iterativen Entwicklungsprozess, bei dem Nutzer- und Aufgabeneigenschaften die Entwicklung der Software bestimmen. Außerdem enthält die Norm weitere Richtlinien und Tabellen für das Berichten über benutzerorientierte Aktivitäten [Wik11].

Die Norm stellt nutzerorientierte Gestaltung als eine fachübergreifende Aktivität dar, die Wissen über menschliche Faktoren und ergonomische Kenntnisse und Techniken umfasst. Der ISO-Prozess besteht aus vier wesentlichen Phasen [ISO99]:

1. **Nutzungskontext verstehen:** Das Ergebnis dieser Phase ist eine

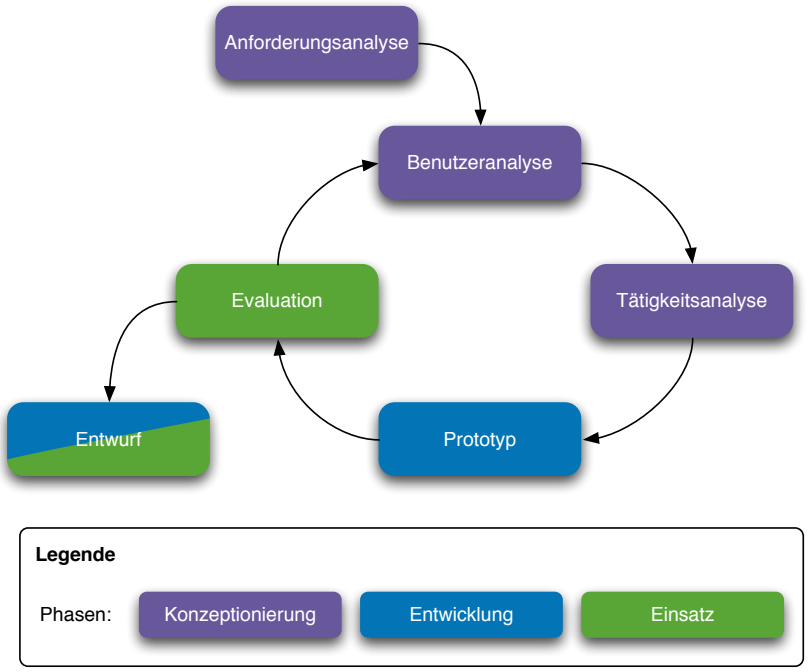


Abbildung 2.5: DIN EN ISO 13407: Der Entwicklungsprozess

Beschreibung der relevanten Benutzer, ihrer Arbeitsaufgaben und ihrer Umgebung.

2. **Anforderungen spezifizieren:** Während dieser Phase werden die Zielgrößen aus der bereits vorhandenen Dokumentation auf einer Kompromissebene abgeleitet. Dabei wird die Teilung der Systemaufgaben in solche, die von Menschen und in solche, die von der Technik durchgeführt werden, sollen bestimmt.

3. **Lösungen produzieren:** Dies kann im Sinne eines *Prototyping* oder eines anderen iterativen Prozesses erfolgen. Diese Prototypen können noch reine Papierentwürfe (*Mock-ups*) oder aber schon lauffähige Programmversionen sein.

4. **Lösungen bewerten:** Die Lösungen werden auf die Erfüllung der zuvor festgelegten Anforderungen geprüft. Dazu können Experten-Reviews, Usability-Tests, Befragungen oder auch eine Mischung daraus dienen. Die dabei entdeckten Abweichungen werden dann auf ihre Relevanz hin bewertet und sind Ausgangspunkt der nächsten Iteration des Entwicklungsprozesses.

Dieses Verfahren ist komplementär zu bestehenden Prozessmodellen des Software-Engineering und ergänzt diese. Der benutzerorientierte Gestaltungsprozess sollte der Norm zufolge bereits im frühesten Sta-

dium des Projekts beginnen und sollte dann wiederholt durchlaufen werden, bis das System die Anforderungen erfüllt [Wik11].

Die DIN EN ISO 13407 stellt einen interaktiven, benutzerzentrierten Entwurfsprozess dar, der Prototypen für die Benutzeranalyse verwendet. Zu Anfang der Entwicklung werden die Anforderungen an die Software festgelegt. Diese Anforderungen können im laufenden Prozess nicht mehr verändert oder erweitert werden, so dass diese Phase sehr ausführlich und gewissenhaft erarbeitet werden sollte. Sind die Anforderungen an die Software bekannt, beginnt die erste Iteration mit der Benutzeranalyse, bei der die relevanten Aufgaben der Benutzer identifiziert werden. Die nächste Phase analysiert die Tätigkeit, die einerseits vom Benutzer und andererseits von der Software geleistet werden soll. Es folgt die Entwicklung eines Prototypen, der von den Benutzern getestet werden kann. Die Ergebnisse dieser Tests werden in der nächsten Phase evaluiert und bewertet. Daraus folgt die nächste Iteration, oder, falls die Software den Anforderungen entspricht, der Entwurf. Bei dem Prozess werden die Prototypen in jeder Iteration neu entwickelt, es findet keine Weiterentwicklung statt.

Die DIN EN ISO 13407 wurde Anfang 2011 durch die Norm DIN EN ISO 9241-210 [ISO11] ersetzt, auf die ich allerdings in dieser Arbeit nicht weiter eingehen werde.

2.1.7 Modellgetriebene Softwareentwicklung (MDSD)

Modellgetriebene Softwareentwicklung (engl. *Model Driven Software Development*, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software bzw. kompilierbaren Quelltext erzeugen [SVEH07]. Dabei werden domänenspezifische Sprachen (engl. *Domain-Specific Languages*, DSL) zusammen mit entsprechenden Codegeneratoren und Interpretern eingesetzt [Wik11].

Bei MDSD nach Stahl et al. [SVEH07] geht es darum, sich bei der Entwicklung von Softwaresystemen möglichst nicht zu wiederholen (DRY-Prinzip – *Don't-Repeat-Yourself*). Weil allein mit den Mitteln der jeweiligen Programmiersprache nicht immer passende Abstraktionen zur Beschreibung verschiedener Sachverhalte (*Domain*) eines Softwaresystems gefunden werden können, werden unabhängig von der Zielsprache entsprechende Abstraktionen in Form von domänenspezifischen Sprachen erschaffen. Diese werden dann entweder generativ oder interpretativ auf die Zielplattform abgebildet [SVEH07].

Natürlich hat der Einsatz dieser Variante eine Auswirkung auf allen Ebenen eines Projektes, sowohl technisch und fachlich als auch im

Managementbereich. Deshalb beschreibt die MDSD nicht nur, wie man DSLs, Generatoren usw. entwickelt, sondern auch, wie man diese in (hauptsächlich agilen) Entwicklungsprozessen sinnvoll integriert [SVEH07].

Durch den erhöhten Abstraktionsgrad der DSLs sind die Problembeschreibungen wesentlich klarer, einfacher und weniger redundant festgehalten. Dies erhöht nicht nur die Entwicklungsgeschwindigkeit, sondern sorgt innerhalb eines Projektes für klar verstandene Domänenkonzepte. Das Konzept der omnipräsenten Sprache (engl. *Ubiquitous Language*) aus dem *Domain-Driven Design* wird hier auf die Konzeptebene der Softwarearchitektur angewandt [SVEH07].

Weiterhin wird die Evolution der Software durch die Trennung der technischen Abbildung und der fachlichen Modelle wesentlich vereinfacht. Auch das Testen fällt leichter, da nicht mehr jede einzelne Zeile Quelltext getestet werden muss, sondern nur noch exemplarisch die Modelle. Domänenspezifische Validierung in den Entwicklungswerkzeugen sorgt für sehr kurze *Turnarounds* [SVEH07].

Für MDSD existieren eine Vielzahl an Werkzeugen, die jeweils nur einzelne Aspekte, wie z. B. die Modellierung, oder alle Funktionalität unterstützen [Wik11].

- **Reine Modellierungswerkzeuge:** Sie dienen lediglich zur grafischen Darstellung und unterstützen keine automatischen Transformationen. Das Modell wird hier in ein Austauschformat (beispielsweise XMI[1]) exportiert und mit gesonderten Transformatoren weiterbearbeitet.

- **Reine Transformatoren:** Diese dienen ausschließlich der Transformation von Modellen und beinhalten keine grafischen Modellierungsfunktionalitäten. Die Modelle werden in einem bestimmten Austauschformat in ein internes Modellformat importiert, transformiert und danach wieder exportiert.

- **Integrierte MDD-Werkzeuge:** Diese bieten Modellierung, Modelltransformationen und Codegenerierung gebündelt in einem Werkzeug. Überflüssige Export- und Importvorgänge, Kompatibilitätsprobleme beim Datenaustausch und Rüstaufwand bzgl. Integration werden vermieden. Die Navigierbarkeit und Synchronisation zwischen fachlichem und technischem Modell und Implementierungscode wird optimal unterstützt.

[1] XMI steht für XML Metadata Interchange und ist ein Austauschformat für Metadaten in XML.

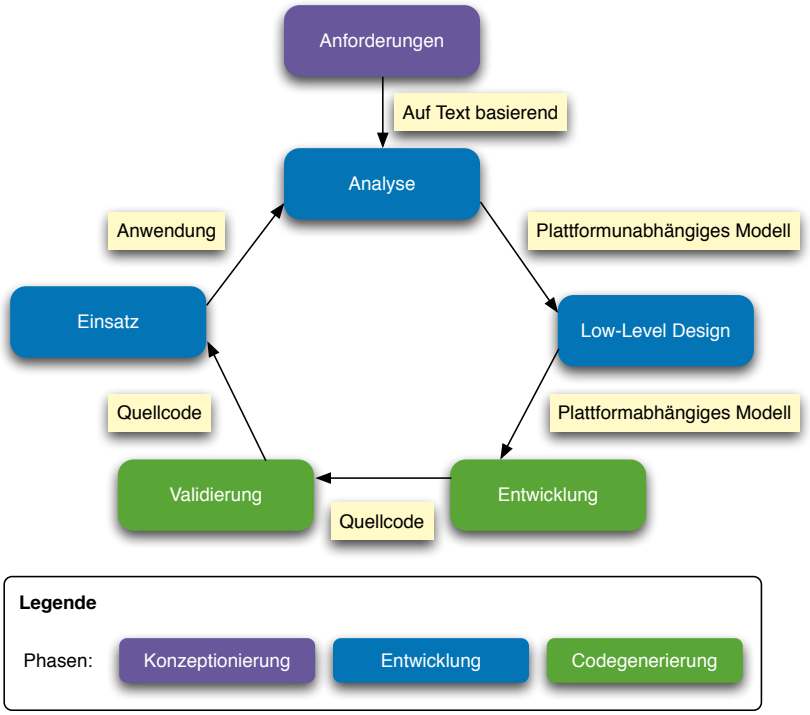


Abbildung 2.6: Modellgetriebene Softwareentwicklung.

Der Entwicklungsprozess beim MDSD kann variabel sein. Größtenteils ist es aber ein iterativer Top-Down Ansatz, der Prototypen mit berücksichtigt. Alleine schon die Vorgehensweise, wie die Modelle entwickelt werden, und dass die Codegenerierung und das Erstellen der endgültigen Applikation computergestützt verläuft, macht einen iterativen Prozess unter Benutzung von Prototypen sinnvoll. In Abbildung 2.6[27] wird ein generischer MDSD Prozess dargestellt. Die Annotationen an den Übergängen von einer Phase zur nächsten beinhalten die jeweilige Form der Applikation. Nachdem die Anforderungen benannt sind, werden sie meist in textbasierte Form (DSL) an die Entwickler weiter gegeben. Diese analysieren die Anforderungen und entwickeln ein erstes Modell, das meist plattformunabhängig ist. Ist das Modell zufriedenstellend, wird das Low-Level Design entwickelt, sodass das Modell auf der späteren (Hardware-)Plattform lauffähig ist. Ist die Anpassung, vollendet wird die Applikation mit Hilfe von Codegeneratoren in Quelltext transformiert und übersetzt. Dies geschieht automatisch und am Ende des Prozesses steht die lauffähige Applikation. Diese wird nun eingesetzt und getestet. Sollten Fehler auftreten oder die Applikation um Funktionalität erweitert werden, müssen die Änderungen erst im plattformunabhängigen Modell korrigiert bzw. zugefügt werden.

2.1.8 Feature Driven Development (FDD)

Funktionsgetriebene Softwareentwicklung (engl. *Feature Driven Development*, FDD) wurde von Jeff De Luca im Jahre 1998 als eine schlanke Methode für zeitkritische Softwareentwicklung definiert [DL98]. Seit der Zeit wurde FDD kontinuierlich weiterentwickelt. FDD stellt den *Feature*-Begriff in den Mittelpunkt der Entwicklung. Jedes *Feature* stellt einen Mehrwert für den Kunden dar. Die Entwicklung wird anhand eines Feature-Plans organisiert. Eine wichtige Rolle spielt der Chefarchitekt (engl. *Chief Architect*), der ständig den Überblick über die Gesamtarchitektur und die fachlichen Kernmodelle behält. Bei größeren Teams werden einzelne Entwicklerteams von den Chefprogrammierern (engl. *Chief Programmer*) geführt [WRL05] [PF02].

FDD definiert ein Prozess- und ein Rollenmodell, die gut mit existierenden klassischen Projektstrukturen harmonieren. Daher fällt es vielen Unternehmen leichter, FDD einzuführen als XP (siehe Kapitel 2.1.10$_{31}$) oder Scrum (siehe Kapitel 2.1.11$_{34}$). Des Weiteren ist FDD ganz im Sinne der agilen Methoden sehr kompakt und lässt sich auf wenigen Seiten komplett beschreiben [DL04].

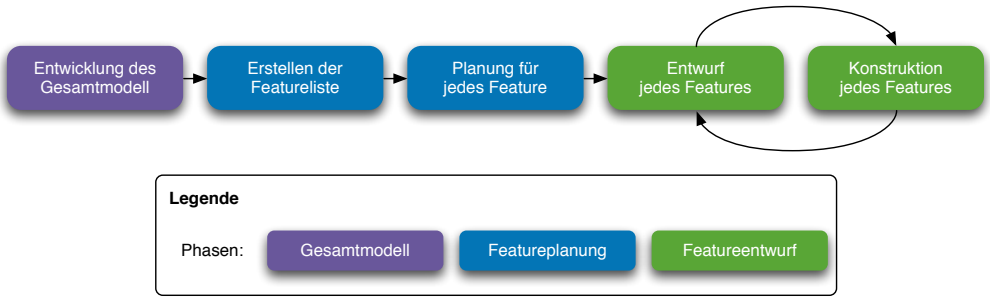


Abbildung 2.7: Feature Driven Development.

FDD-Projekte durchlaufen fünf Prozesse, wie sie in Abbildung 2.7 zu sehen sind.

Prozess 1 - Entwicklung eines Gesamtmodells: Im ersten Prozess definieren die Fachexperten und Entwickler unter Leitung des Chefarchitekten Inhalt und Umfang des zu entwickelnden Systems. In Kleingruppen werden Fachmodelle für die einzelnen Bereiche des Systems erstellt, die in der Gruppe vorgestellt, ggf. überarbeitet und schließlich integriert werden. Das Ziel dieser ersten Phase ist ein Konsens über Inhalt und Umfang des zu entwickelnden Systems sowie das fachliche Kernmodell [DL04].

Prozess 2 - Erstellung einer Feature-Liste: Die aus dem ersten Prozess festgelegten Systembereiche werden von dem jeweiligen

Chefprogrammierer in Features detaillieren. Es wird ein dreistufiges Schema verwendet: Fachgebiete (engl. *Subject Areas*) bestehen aus Geschäftstätigkeiten (engl. *Business Activities*), die durch Schritte (engl. *Steps*) ausgeführt werden. Die Schritte entsprechen den Features. Die Features werden nach dem einfachen Schema <Aktion> <Ergebnis> <Objekt> aufgeschrieben. Ein Feature darf maximal zwei Wochen zu seiner Realisierung benötigen. Das Ergebnis dieses zweiten Prozesses ist eine kategorisierte Feature-Liste, deren Kategorien auf oberster Ebene von den Fachexperten aus dem ersten Prozess stammen [DL04].

Prozess 3 - Planung jedes Feature: Projektleiter, Entwicklungsleiter und die Chefprogrammierer planen die Reihenfolge, in der Features realisiert werden sollen. Dabei richten sie sich nach den Abhängigkeiten zwischen den Features, der Auslastung des jeweiligen Programmierteams sowie der Komplexität der Features. Auf Basis des Plans werden die Fertigstellungstermine je Geschäftsaktivität festgelegt. Jede Geschäftsaktivität bekommt einen Chefprogrammierer als Besitzer zugeordnet. Außerdem werden für die bekannten Kernklassen Entwickler als Besitzer festgelegt (engl. *Class Owner List*) [DL04].

Prozess 4 - Entwurf jedes Feature: Die Chefprogrammierer weisen die anstehenden Features den Entwicklerteams auf Basis des Klassenbesitztums zu. Die Entwicklerteams erstellen ein oder mehrere Sequenzdiagramme für die Features, die Chefprogrammierer verfeinern die Klassenmodelle auf Basis der Sequenzdiagramme. Die Entwickler schreiben dann erste Klassen- und Methodenrümpfe. Schließlich werden die erstellten Ergebnisse inspiziert. Bei fachlichen Unklarheiten können die Fachexperten hinzugezogen werden [DL04].

Prozess 5 - Konstruiere jedes Feature: Die Entwickler programmieren die im vierten Prozess vorbereiteten Features. Zur Qualitätssicherung werden bei der Programmierung sowohl Komponententests als auch Code-Inspektionen eingesetzt [DL04].

FDD ist ein sehr kompaktes Vorgehensmodell, das sich im Allgemeinen schnell in Unternehmen umsetzten lässt, vorausgesetzt die zu entwickelnde Software kann auf die für die Entwicklung zentralen Features reduziert werden. Um eine schnelle Fertigstellung der einzelnen Features zu erreichen, sollten diese nicht zu komplex sein, da sonst die Implementierung viel Zeit kosten würde. Da jedoch bei FDD der Fokus nicht auf kurzen Iterationen zwischen den Softwareversionen liegt, kann der oben genannte Punkt im Allgemeinen vernachlässigt werden. Im weitesten Sinne kann mein Verfahren, das

ich in dieser Arbeit vorstelle, auch als FDD gesehen werden, allerdings wird in meinem Prozess auf die kurzen Iterationen geachtet und deshalb die Features, die implementiert werden sollen, sehr feingranular aufgeteilt.

2.1.9 Rational Unified Process

Der *Rational Unified Process* (RUP) [WIN+06] ist ein kommerzielles Produkt der Firma *Rational Software*, die seit 2002 Teil des IBM Konzerns ist. IBM entwickelt den RUP und die zugehörige Software weiter. Die neunte Version vom Jahre 2006 ist die derzeit die aktuelle Version. Der RUP benutzt die Unified Modeling Language (UML) als Notationssprache. Der RUP wurde von Philippe Kruchten [KR96] in seiner Urform erstmals 1996 vorgestellt.

Der RUP war möglich geworden, als sich die bekannten Programmierer Grady Booch, Ivar Jacobson und James Rumbaugh des Unternehmens Rational Inc. auf ein einheitliches Notationssystem einigen konnten. Als Resultat dieser Bemühungen entstand die UML. Die Standardisierung und Weiterentwicklung der Sprache wurde an die *Object Management Group* (OMG) übergeben. Mit einer gemeinsamen Sprache konnte nun eine gemeinsame objektorientierte Methode entwickelt werden. Der *Unified Process* ist dabei ein Metamodell für Vorgehensmodelle zur Softwareentwicklung und wurde parallel zur *Unified Modelling Language* von den oben genannten Personen entwickelt und veröffentlicht [Wik11].

Der *Unified Process* basiert auf mehreren Prinzipien [Wik11]:

- Anwendungsfällen
- Architektur im Zentrum der Planung
- inkrementellem und iterativen Vorgehen

Eine konkrete Implementierung des oben beschriebenen *Unified Process* ist der *Rational Unified Process*. Die erste Version des RUP aus dem Jahre 1999 [Kru99] [JBR99] führte die Vorschläge der drei oben genannten Begründer für eine einheitliche Methode zur Modellierung zusammen [Wik11].

Der *Rational Unified Process* legt grundlegende Arbeitsdisziplinen fest (siehe auch Abbildung 2.8_{31}). Die Kernarbeitsdisziplinen sind die Geschäftsprozessmodellierung (engl. *Business Modeling*), die Anforderungsanalyse (engl. *Requirements*), Analyse & Design (engl. *Analysis & Design*), die Implementierung (engl. *Implementation*) und der

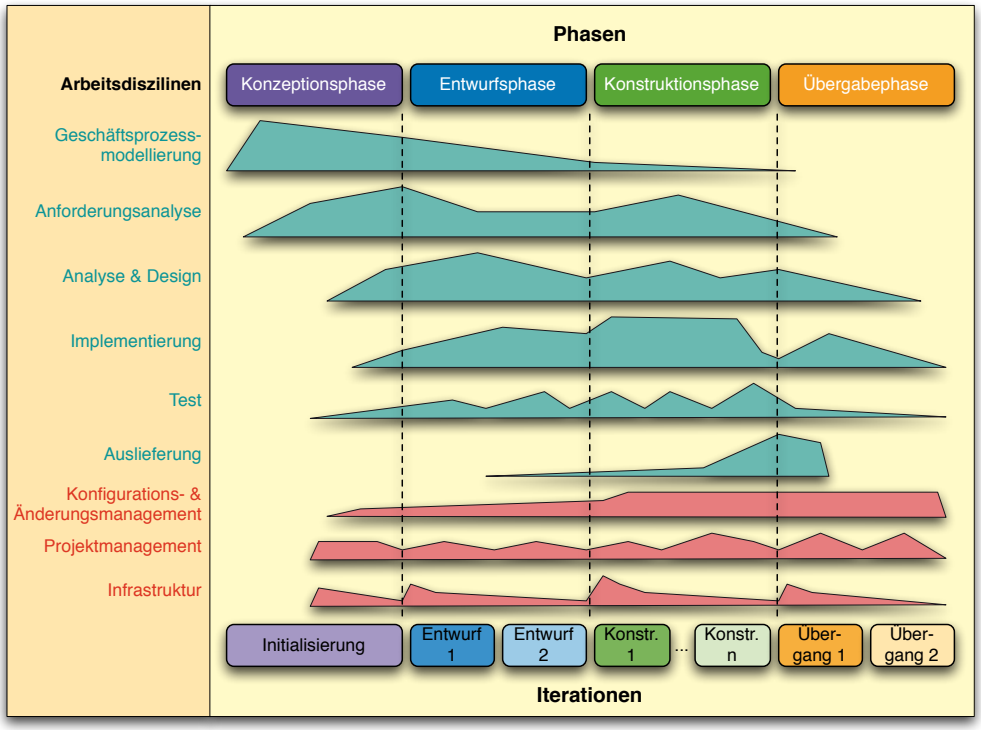


Abbildung 2.8: Rational Unified Process.

Test (engl. *Test*) und die Auslieferung (engl. *Deployment*). Zu den unterstützenden Arbeitsdiszilinen gehört das Konfigurations- & Änderungsmanagement (engl. *Configuration & Change Management*), das Projektmanagement (engl. *Project Management*) und die Infrastruktur (engl. *Environment*) [Wik11].

Orthogonal zu den Arbeitsdisziplinen gibt es im *Rational Unified Process* vier Phasen, in der jeder der Diszilinen mehr oder weniger intensiv zur Anwendung kommt. Die Phasen sind die Konzeptionsphase (engl. *Inception*), die Entwurfsphase (engl. *Elaboration*), die Konstruktionsphase (engl. *Construction*) und die Übergabephase (engl. *Transition*). Diese Phasen sind in sich in Iterationen unterteilt, so dass der *Rational Unified Process* ein iteratives Vorgehensmodell ist. Resultate der Phasen sind die so genannten Meilensteine (engl. *Milestones*) [Wik11].

2.1.10 Extreme Programming (XP)

Extreme Programming (XP), oder auch Extremprogrammierung, ist eine Methode, die das Lösen einer Programmieraufgabe in den Vordergrund der Softwareentwicklung stellt und dabei einem formalisierten Vorgehen geringere Bedeutung zumisst. Diese Vorgehensweise de-

finiert ein Vorgehensmodell, welches sich den Anforderungen des Kunden in kleinen Schritten annähert [AHo2].

Extreme Programming wurde von Kent Beck, Ward Cunningham und Ron Jeffries während ihrer Arbeit im Projekt *Comprehensive Compensation System* (C3-Projekt) bei Chrysler zur Erstellung von Software entwickelt. Die Arbeiten am *C3-Projekt* begannen 1995 und wurden 2000 nach der Übernahme durch Daimler eingestellt. Die dabei entwickelte Software wurde im Bereich der Lohnabrechnung eingesetzt [Syso6].

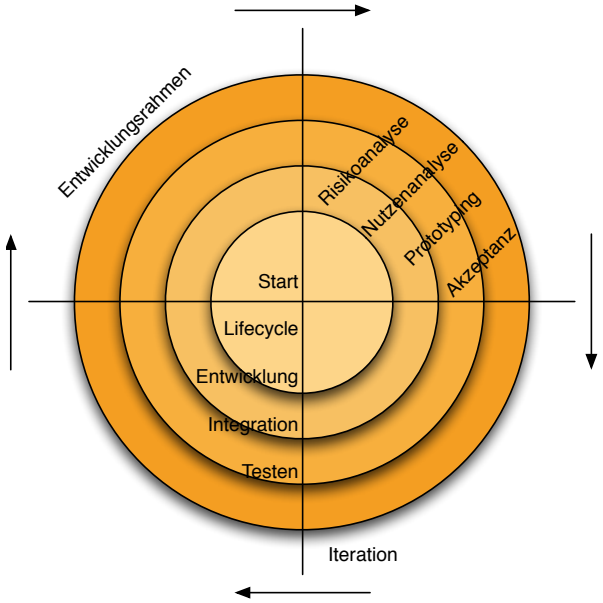


Abbildung 2.9: Lebenszyklus des *Extreme Programming*.

XP ist ein durch fortlaufende Iterationen und den Einsatz mehrerer Einzelmethoden strukturierendes Vorgehensmodell (siehe Lebenszyklus in Abbildung 2.9). Es entstand durch die Synthese verschiedener Disziplinen der Softwareentwicklung und basiert auf in der Praxis bewährten Methoden, auch *Best Practice*[2] genannt. XP folgt einem strukturierten Vorgehen und stellt die Teamarbeit, Offenheit und stete Kommunikation zwischen allen Beteiligten in den Vordergrund. Dabei ist die Kommunikation eine Grundsäule von *Extreme Programming*. Die Methode geht davon aus, dass der Kunde die Anforderungen an die zu erstellende Software zu Projektbeginn noch nicht komplett kennt und nicht hinreichend strukturieren kann beziehungsweise das mit der Realisierung betraute Entwicklerteam nicht über alle Informationen verfügt, um eine verlässliche Aufwandsschätzung über die notwendige Dauer bis zum Abschluss zu geben. Im Laufe eines Projektes ändern sich nicht selten Prioritäten und Gewichte. Zu Be-

[2]Die Bezeichnung *Best Practice* stammt aus der Betriebswirtschaft und bezeichnet eine bewährte bzw. optimale Vorgehensweise in einem Unternehmen.

ginn geforderte Funktionen der Software werden möglicherweise in einer anderen Form benötigt oder im Laufe der Zeit sogar komplett hinfällig [Wik11].

Bei einer konsequenten Ausrichtung an XP soll die zu erstellende Software schneller bereitgestellt sowie eine höhere Softwarequalität und Zufriedenheit des Kunden erreicht werden, als es mit den traditionellen Ansätzen möglich ist. Der Kunde soll ein einsatzbereites Produkt erhalten, an dessen Herstellung er aktiv teilgenommen hat. Neue Funktionalität wird permanent entwickelt, integriert und getestet. Um zu der zu entwickelnden Funktionalität zu gelangen, werden jeweils die Schritte Risikoanalyse, Nutzenanalyse, die Bereitstellung einer ersten ausführbaren Version (Prototyping) und ein Akzeptanztest durchgeführt [Beco0].

Nach Vertretern dieses Vorgehensmodells ist XP ein Risikomanagement. Es bejaht das Risiko, geht aktiv darauf ein und versucht, es zu minimieren. Dieser implizite Umgang mit dem Faktor Risiko steht im Gegensatz zu eher expliziten Vorgehensweisen, wie der Aufstellung einer Risikoliste [DeM03]. Softwareentwicklungsprojekte sind unterschiedlichen Gefahren ausgesetzt, für die XP Lösungen anbieten soll [Wik11].

Dem Kunden bietet XP, gerade durch seine kurzen Entwicklungszyklen, jederzeit die Möglichkeit, steuernd auf das Projekt einzuwirken. Dadurch soll erreicht werden, dass sich das Produkt aktuellen Anforderungen anpasst, statt überholten Anforderungen aus einer längst vergangenen Analysephase zu genügen und damit bereits bei Einführung veraltet zu sein. Zudem kann der Kunde bereits nach kurzer Zeit ein unvollständiges, aber zumindest funktionstüchtiges Produkt einsetzen. Der Kunde ist im besten Fall jederzeit auf demselben aktuellen Informationsstand bezüglich des Projektes wie das Entwicklerteam [Wik11].

Aus der Sicht der Programmierer existiert keine strikte Rollentrennung, da die Aufgabenverteilung abhängig von Situation und Fähigkeiten geschieht. Der allgemeine Wissensaustausch und die stetige Kommunikation beugen einem Wissensmonopol vor. Dies soll den Einzelnen entlasten, wohingegen der Druck auf einer Person lastet, wenn diese sich als Einzige in einem Modul auskennt [Wik11].

Dem Projekt bietet XP die Möglichkeit, Risiken zu minimieren. So sollte unter richtiger Anwendung von XP der Kunde Software erhalten, deren Umfang ihn nicht überrascht. Das Team soll ferner gegen Krankheit Einzelner nicht mehr so anfällig sein. Ein ehrlicher Umgang mit dem Kunden soll die Glaubwürdigkeit und Zufriedenheit steigern

und die Angst minimieren, die unter Umständen zwischen Kunde und Entwicklung vorherrscht [Wik11].

XP stellt aus wirtschaftswissenschaftlicher Sicht eine Form der Organisation dar, die direkt die Prozesse der Wertschöpfung[3] beschreibt. In den Wirtschaftswissenschaften werden zur Bewertung von *Extreme Programming* auch Erkenntnisse anderer Sozialwissenschaften, insbesondere der Soziologie, genutzt [Wik11].

Vereinzelt wird *Extreme Programming* als informelle und damit unverbindliche Methode bezeichnet. Das trifft jedoch weder den Ansatz noch das Ziel. Tatsächlich ist die Formalisierung der Methode des *Extreme Programming* bewusst flach und schlank gehalten. Hingegen muss ein Einvernehmen zwischen Kunden und Programmierern hinsichtlich der Verbindlichkeit der erstellten Unterlagen hergestellt werden, solange diese noch nicht durch neuere Fassungen ersetzt wurden. Weiter muss der Vorgang des Ersetzens einer Fassung einer Unterlage durch eine neuere Fassung dieser Unterlage soweit formalisiert sein, dass beide Parteien Kenntnis von dieser Ersetzung haben und diese Ersetzung annehmen [Wik11].

Extreme Programming ist die Summe einzelner, gemeinsam zur Optimierung des Nutzens eingesetzter Erfolgsmethoden. XP definiert sich selbst mit diesen Prinzipien, allerdings nicht als Patentlösung für alle Probleme. Da, wo es speziellen oder individuellen Anforderungen nicht genügt, soll es angepasst werden. Viele Prinzipien greifen verzahnt ineinander. Einzelne Praktiken sind an sich nicht neu und werden teilweise bereits lange genutzt, oder sind sogar von trivialer und doch oft unterschätzter Natur. Die Praktiken sind die greifbaren, konkreten Maßnahmen, die sich aus den Werten und den Prinzipien ableiten lassen [Wik11].

2.1.11 Scrum

Scrum (engl. das Gedränge) ist ein Vorgehensmodell, dass auf Treffen (engl. Meetings), Artefakten, Rollen, Werten und Grundüberzeugungen basiert und beim Entwickeln von Produkten im Rahmen agiler Softwareentwicklung hilfreich ist. Teammitglieder organisieren ihre Arbeit weitgehend selbst und wählen auch die eingesetzten Software-Entwicklungswerkzeuge und -Methoden. Ken Schwaber, Jeff Sutherland und Mike Beedle haben Scrum erfunden und in der Softwareentwicklung etabliert [BDS+99]. Als Methode zur Entwicklung von Soft-

[3] Der Begriff Wertschöpfung ist in einer Geldwirtschaft das Ziel produktiver Tätigkeit. Diese transformiert vorhandene Güter in Güter mit höherem Geldwert. Die allgemeine Formel lautet: $Wertschöpfung = Gesamtleistung - Vorleistungen$.

ware wird Scrum das erstmalig im Buch „Wicked Problems, Righteous Solutions“ [DS90] beschrieben. Scrum in Produktionsumgebungen wird zum ersten Mal im Artikel „The New New Product Development Game“ [Gam86] erläutert und später in der Veröffentlichung „The Knowledge Creating Company“ [NT95] weiter ausgeführt [Wik11].

Im Jahre 2003 legte Ken Schwaber ein Zertifizierungsprogramm für *Scrum Master* auf. Das Ziel, heute wie damals, ist es, die Software-Entwicklung durch das Nutzen von Scrum zu professionalisieren. Inzwischen wird das Training *Certified Scrum Master* unter der Schirmherrschaft der *Scrum Alliance* durchgeführt [Wik11].

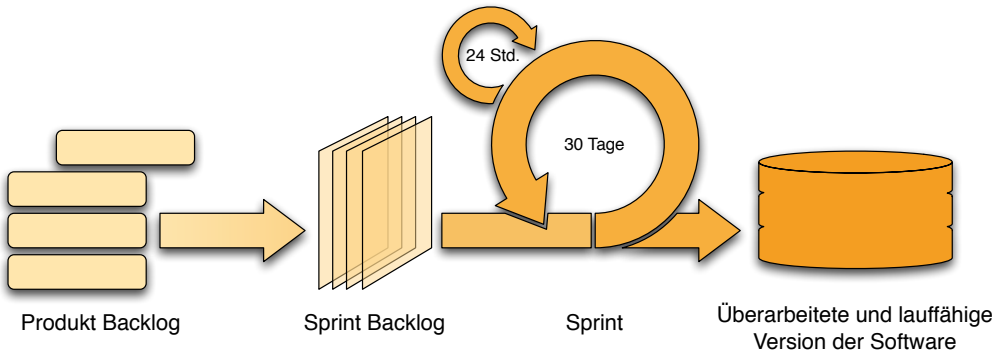


Abbildung 2.10: Der Scrum-Prozess.

Scrum erfüllt die Bedingungen der agilen Software-Entwicklung, die 2001 im Agilen Manifest u. a. von Ken Schwaber und Jeff Sutherland mit formuliert wurden [BBvB+01]:

- Individuen und Interaktionen gelten mehr als Prozesse und Tools.
- Funktionierende Programme gelten mehr als ausführliche Dokumentation.
- Die stetige Zusammenarbeit mit dem Kunden steht über Verträgen.
- Der Mut und die Offenheit für Änderungen steht über dem Befolgen eines festgelegten Plans.

Bei Scrum gibt es drei klar getrennte Rollen, die von Mitarbeitern ausgefüllt werden, die im selben Projekt zusammen arbeiten und damit auch dasselbe Ziel haben. Damit jeder für das, was er kann, zuständig und verantwortlich ist, werden die Zuständigkeiten wie folgt aufgeteilt:

Product Owner: Der *Product Owner* legt das gemeinsame Ziel fest, welches das Team zusammen mit ihm erreichen muss. Zur Definition der Ziele dienen ihm *User Stories*. Er stellt das Budget zur Umsetzung dieser *User Stories* zur Verfügung. Er setzt regelmäßig die Prioritäten der einzelnen *Product-Backlog-Elemente* (siehe unten). Dadurch legt er fest, welches die wichtigsten Features sind, aus denen das Entwicklungsteam eine Auswahl für den nächsten *Sprint* trifft [Wik11].

Team: Das Team schätzt die Aufwände der einzelnen *Backlog-Elemente* ab und beginnt mit der Implementierung der für den nächsten *Sprint* machbaren Elemente. Dazu wird vor dem Beginn des *Sprints* ein weiteres Planungstreffen durchgeführt, bei dem die am höchsten priorisierten Elemente des *Backlogs* und konkrete Aufgaben aufgeteilt werden. Das Team arbeitet selbstorganisiert im Rahmen einer *Time Box* (dem *Sprint*) und hat das Recht (und die Pflicht), selbst zu entscheiden, wie viele Elemente des *Backlogs* nach dem nächsten *Sprint* erreicht werden müssen, man spricht dabei von *commitments* [Wik11].

Scrum Master: Der *Scrum Master* hat die Aufgabe, die Prozesse der Entwicklung und Planung durchzuführen und die Aufteilung der Rollen und Rechte zu überwachen. Er hält die Transparenz während der gesamten Entwicklung aufrecht und unterstützt dabei, Verbesserungspotentiale zu erkennen und zu nutzen. Er ist keinesfalls für die Kommunikation zwischen Team und *Product Owner* verantwortlich, da diese direkt miteinander kommunizieren. Er steht dem Team zur Seite, ist aber weder *Product Owner* noch Teil des Team. Der *Scrum Master* sorgt mit allen Mitteln dafür, dass das Team produktiv ist, also die Arbeitsbedingungen stimmen und die Teammitglieder zufrieden sind. Er tritt somit für die ordnungsgemäße Durchführung und Implementierung von Scrum im Rahmen des Projektes ein [Wik11].

Bei der Rollenaufteilung wurde berücksichtigt, dass das Team sich selbst organisiert. Der *Product Owner* gibt nicht vor, welches Teammitglied wann was macht und wer mit wem zusammenarbeitet. Bei Scrum wird von der Annahme ausgegangen, dass das Team sich intuitiv selbst organisiert, und zu jeder Aufgabe dynamisch eine optimale innere Organisationsstruktur bildet, die sich relativ schnell an die sich wandelnden komplexen Aufgaben anpasst. Der *Scrum Master* hat die Pflicht, darauf zu achten, dass der *Product Owner* nicht in diesen adaptiven Selbstorganisationsprozess eingreift und das Team stört oder Verantwortlichkeiten an sich nimmt, die ihm nicht zustehen [Wik11].

Artefakte

Wie in Abbildung 2.10_{35} dargestellt besteht der Lebenszyklus von Scrum aus *Backlogs* und *Sprints*. In dem Prozess existieren insgesamt folgende *Backlogs* [Wik11]:

Product Backlog: Das *Product Backlog* enthält die Features des zu entwickelnden Produkts. Es umfasst alle Funktionen, die der Kunde wünscht, zuzüglich technischer Abhängigkeiten. Vor jedem *Sprint* werden die Elemente des *Product Backlogs* neu bewertet und priorisiert. Dabei können bestehende Elemente entfernt sowie neue hinzugefügt werden. Hoch priorisierte Features werden von den Entwicklern im Aufwand geschätzt und in den *Sprint Backlog* übernommen. Ein wesentliches Merkmal des *Backlogs* ist die Tiefe der Beschreibung von einzelnen Features. Hoch priorisierte Features werden im Gegensatz zu niedrig priorisierten sehr detailliert beschrieben. Somit wird viel Zeit für die wesentlichen Elemente und wenig für unwesentliche verwendet [Wik11].

Sprint Backlog: Das *Sprint Backlog* enthält alle Aufgaben, die notwendig sind, um das Ziel des *Sprints* zu erfüllen. Eine Aufgabe sollte dabei nicht länger als 16 Stunden dauern. Längere Aufgaben sollten in kurze Teilaufgaben zerlegt werden. Bei der Planung des *Sprint* werden nur so viele Aufgaben eingeplant, wie das Team an Kapazität aufweisen kann [Wik11].

Burndown Chart: Das *Burndown Chart* ist eine graphische, pro Tag zu erfassende Darstellung des noch zu erbringenden Restaufwands pro *Sprint*. Im Idealfall fällt die Kurve kontinuierlich (daher *Burndown*) und der Restaufwand ist somit am Ende des *Sprints* gleich Null. Am *Chart* ist anhand der Verlängerung der negativen Steigung bereits während des *Sprints* erkennbar, ob der zu Beginn geschätzte Aufwand realisierbar ist [Wik11].

Impediment Backlog: In das *Impediment Backlog* werden alle Hindernisse des Projekts eingetragen. Der *Scrum Master* ist dafür zuständig, diese Hindernisse gemeinsam mit dem Team auszuräumen [Wik11].

Zyklusmodell

Sprint: Zentrales Element des Entwicklungszyklus von Scrum ist der *Sprint*. Ein *Sprint* bezeichnet die Umsetzung einer Iteration, Scrum schlägt ca. 30 Tage als Dauer einer Iteration vor. Vor

dem *Sprint* werden die Produkt-Anforderungen des Kunden in einem *Product Backlog* gesammelt. Auch technische und administrative Aufgaben werden dort aufgenommen. Das *Product Backlog* muss nicht vollständig sein; es wird laufend fortgeführt. Die Anforderungen für den ersten *Sprint* sind meistens rasch aufgestellt. Die Anforderungen werden informell skizziert. Für einen *Sprint* wird ein *Sprint Backlog* erstellt. In diesen werden Anforderungen übernommen, die während des *Sprints* umgesetzt werden sollen. Die Entscheidung, welche Anforderungen umgesetzt werden, wird vom Kunden nach von ihm festgelegten Prioritäten getroffen. Zum *Sprint* organisiert sich das Entwicklungsteam selbst, braucht also keine detaillierten methodischen Vorschriften [Wik11].

Daily Scrum: An jedem Tag findet ein kurzes (maximal 15-minütiges) *Daily Scrum*, heißt eine Sitzung (engl. *Meeting*), statt. Das Team stellt sich gegenseitig die folgenden Fragen:

- „Bist du gestern mit dem fertig geworden, was du dir vorgenommen hast?“
- „Welche Aufgaben wirst du bis zum nächsten Meeting bearbeiten?“
- „Gibt es ein Problem, das dich blockiert?“

Die Sitzung dient dem Informationsaustausch des Teams untereinander. Hier geht es darum, dass möglichst alle alles wissen. Falls neue Hindernisse erkannt wurden, müssen diese vom *Scrum Master* bearbeitet werden. Dazu werden sie in das *Impediment Backlog* eingetragen. Größere Projekte werden durch das Einführen von *Scrum-of-Scrum Meetings*, *Product Owner Daily Scrums* und *ScrumMaster Weekly* gesteuert [Wik11].

Review: Nach einem *Sprint* wird das *Sprint*-Ergebnis einem informellen Review durch Team und Kunden unterzogen. Hierzu wird das Ergebnis des *Sprints* (also die laufende Software) vorgeführt, eventuell werden technische Eigenschaften präsentiert. Der Kunde prüft, ob das *Sprint*-Ergebnis seinen Anforderungen entspricht, eventuelle Änderungen werden im *Product Backlog* dokumentiert [Wik11].

Retrospektive: In der Retrospektive wird die zurückliegende *Sprint*-Phase betrachtet. Es handelt sich dabei nicht um *Lessons Learned*[4], sondern um einen zunächst wertfreien Rückblick auf die

[4] *Lessons Learned* ist ein Fachbegriff des Projektmanagements und bezeichnet allgemein die Auswertung von Erfahrungen in Projekten, die zuvor durchgeführt wurden.

Ereignisse des *Sprints*. Alle Teilnehmer notieren dazu die für sie wichtigen Ereignisse auf Zetteln und ordnen sie dem Zeitstrahl des *Sprints* zu. Anschließend schreiben die Teilnehmer alle Punkte auf, welche ihnen zu den Themen *Best Practice*, bzw. Verbesserungspotential einfallen. Jedes Verbesserungspotential wird priorisiert und einem Verantwortungsbereich (Team oder Organisation) zugeordnet. Alle der Organisation zugeordneten Themen werden vom *Scrum Master* aufgenommen und in das *Impediment Backlog* eingetragen. Die gesamten teambezogenen Details werden in das *Product Backlog* aufgenommen. Sollte für die Retrospektiven und deren Vorbereitung nicht genug Zeit eingeräumt werden, bleiben die Erkenntnisse und Ergebnisse oberflächlich und die Resultate nach jedem *Sprint* ähneln sich. Dann läuft man Gefahr, dass die Retrospektiven an Stellenwert verlieren oder ganz gestrichen werden, weil die Ergebnisse der Retrospektiven vorhersehbar sind [Wik11].

2.1.12 Prototyping

Prototyping, oder auch Prototypenbau, ist eine Methode der Softwareentwicklung, die schnell zu ersten Ergebnissen (den sogenannten Prototypen) führt und frühzeitiges Feedback bezüglich der Eignung eines Lösungsansatzes ermöglicht [Wik11].

Der Begriff des *Prototyping* im Bereich der Softwareentwicklung trat erstmals Anfang der achtziger Jahre in ersten Publikationen in Erscheinung. Zu dieser Zeit vollzogen sich teils drastische Wandel im Bereich der Softwareentwicklung. Man war auf der Suche nach neuen Designkonzepten und Entwicklungsstrategien um Schwächen bzw. Unzulänglichkeiten vorhandener Entwicklungsmodelle für Software zu umgehen, da insbesondere in umfangreichen Projekten zunehmend Probleme im Bereich der Anforderung bzgl. des Endprodukts auftraten [CS89] [KOSS11].

Im Jahre 1979 sollte eine weitläufig angelegte amerikanische Studie klären, worin die Gründe für ein oftmaliges Scheitern von Softwareprojekten unter Verwendung herkömmlicher Entwicklungsmodelle wären. Es zeigte sich, dass viele Softwareprojekte nicht etwa an unzureichenden Entwicklungsumgebungen oder Entwicklungswerkzeugen, sondern zunehmend am Problem mangelnder Kommunikation zwischen Auftraggebern, Benutzern und Entwicklern scheiterten. Die zur damaligen Zeit gängigen Entwicklungsmodelle setzten auf so genannte *Life Cycle Plans* auf. Das wohl bekannteste und gebräuchlichste derartige Modell war das Wasserfallmodell (Kapitel 2.1.2$_{14}$). Das Problem bei diesen Modellen war, dass die Benutzer nur zu

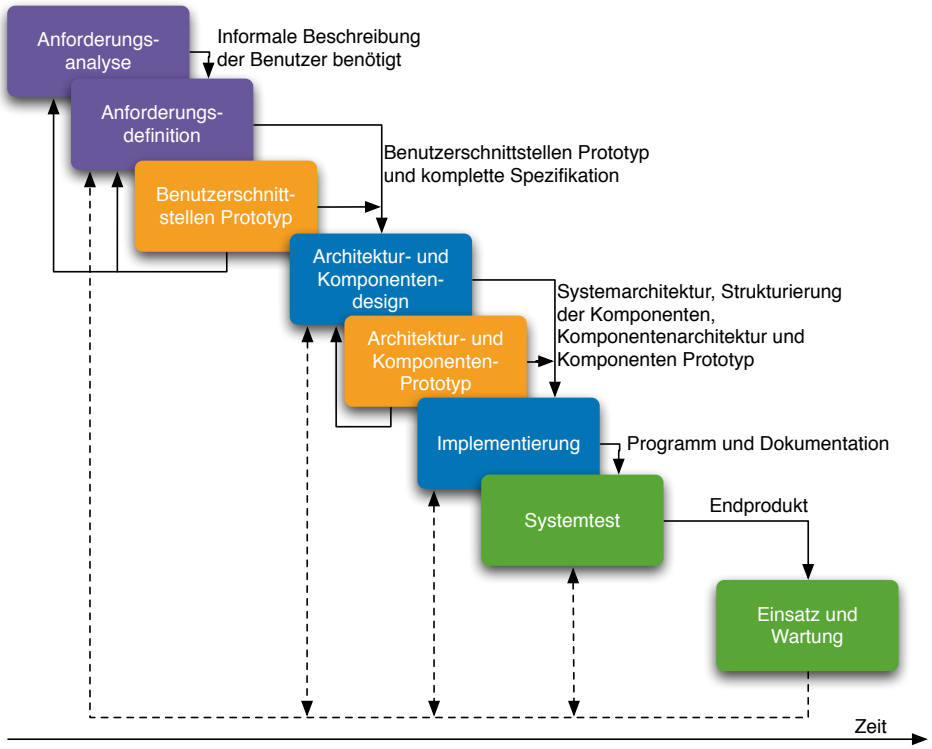


Abbildung 2.11: Entwicklungsprozess nach Pomberger [PW94].

Beginn eines Projektes im Rahmen der Aufgabenbeschreibung bzw. -spezifikation mit einbezogen wurden, jedoch vom weiteren Entwicklungsprozess konsequent ausgeschlossen waren. Dies machte zwar den Entwicklungsprozess überschaubar und kalkulierbar, resultierte allerdings in Software, die oftmals nicht den Erwartungen der Auftraggeber und Nutzer aufgrund unzureichender Aufgabenbeschreibung entsprach. Unklarheiten und Fehler, die bereits im Rahmen der Anforderungsanalyse auftraten, zogen sich somit bis ins Endprodukt durch, wobei im schlimmsten Fall die Software nahezu unbrauchbar geraten konnte [BKK92] [KOSS11].

Das *Prototyping* als Softwareentwicklungsmodell unterscheidet sich grundsätzlich nicht von traditionellen, auf dem *Life Cycle*-Prinzip basierenden Entwicklungsmodellen. Vielmehr stellt es eine Ergänzung zu herkömmlichen Modellen dar. *Prototyping* bildet ein Kernanliegen dieses Paradigmas ab, den Benutzer während des gesamten Entwicklungsfortschritts einbinden, um möglichst gute Kommunikation zwischen Entwicklern und Benutzern zu gewährleisten und eventuelle Ungenauigkeiten in der Softwarespezifikation jederzeit ausbessern zu können. Dadurch können etwaige Unklarheiten und Unzulänglichkeiten frühzeitig erkannt und auch im Laufe des Entwicklungsprozesses geklärt werden, eines der größten Mängel vorausgehender Model-

le [CS89] [KOSS11].

Das Konzept des *Prototyping* versucht relativ früh im Entwicklungsprozess funktionsfähige Prototypen von Teilfunktionen der Software zu entwickeln, welche Teilaspekte des Gesamtprojektes in ihrer Funktionsweise demonstrieren und vom späteren Nutzer getestet bzw. in Zusammenarbeit mit dem Entwickler verbessert werden können. Da Prototypen nur Basiseigenschaften von Teilaspekten des Programms beschreiben, können sie schnell und kostengünstig erstellt werden, wobei auch experimentelles Vorgehen ermöglicht wird. Die Prototypen an sich definieren zwar Teilaspekte der zu erstellenden Software, sind aber selbst als solche nicht Teil des endgültigen Produktes. Derartige Prototypen können dann sukzessive entsprechend den Nutzerbedürfnissen erweitert oder aber auch gänzlich verworfen und durch einen anderen Prototypen ersetzt werden. Durch diesen evolutionären Ansatz bei der Softwareentwicklung entsteht die Software stufenweise in Zusammenarbeit mit den Nutzern [PW94] [KOSS11].

Ansatzmethoden beim *Prototyping*

Es existieren drei verschiedene Ansätze, wie die Prototypen zu erstellen sind [KOSS11].

Throw-Away Ansatz: Bei der Verwendung eines *Throw-Away* Ansatzes wird ein nicht vollständiges, aber im Sinne der Anforderungen lauffähiges Programm beschrieben. Dies wird dann dem Benutzer zur experimentellen Auswertung übergeben. Der Prototyp wird nach der Auswertung nicht weiterverwendet, sondern verworfen. Die gewonnenen Ergebnisse werden bei der Neukonstruktion eines neuen Prototypen verwendet [KOSS11].

Inkrementeller Ansatz: Beim inkrementellen Ansatz wird zu Beginn ein stabiler Programmkern aufgebaut, dem danach schrittweise neue Funktionen oder auch neue Systemteile hinzugefügt werden. Programmteile, die dem Prototypen noch fehlen, werden durch Simulationen vervollständigt. Dieser Prototyp besteht gewissermaßen aus zwei verschiedenen Teilen, einem fest implementieren Softwareteil und einem Simulationsteil. Nachteil dieses Konzeptes ist, dass ein neu hinzugefügter Programmteil das Gesamtsystem in der Regel in einem so hohem Maße beeinflusst, so dass frühere Entwurfsentscheidungen korrigiert werden müssen [KOSS11].

Evolutionärer Ansatz: Diese Ansatzmethode legt die jeweiligen Architekturkonzepte zu keinem Zeitpunkt fest. Das ermöglicht zu

jeder Zeit das Aufnehmen neue Anforderungen. Die Architektur passt sich den Umgebungsanforderungen an. So ist der Prototyp beim Beenden der evolutionären Ansatzmethode das fertige Endprodukt. Die Idee des evolutionären Ansatzes geht bis in die 60er Jahre zurück, aber sie ist bis heute noch sehr schwer umzusetzten [KOSS11].

Arten von Prototypen

Je nach den vorliegenden Anforderungen werden unterschiedliche Prototypen benötigt. Es können vier verschiedene Arten unterschieden werden [KOSS11]:

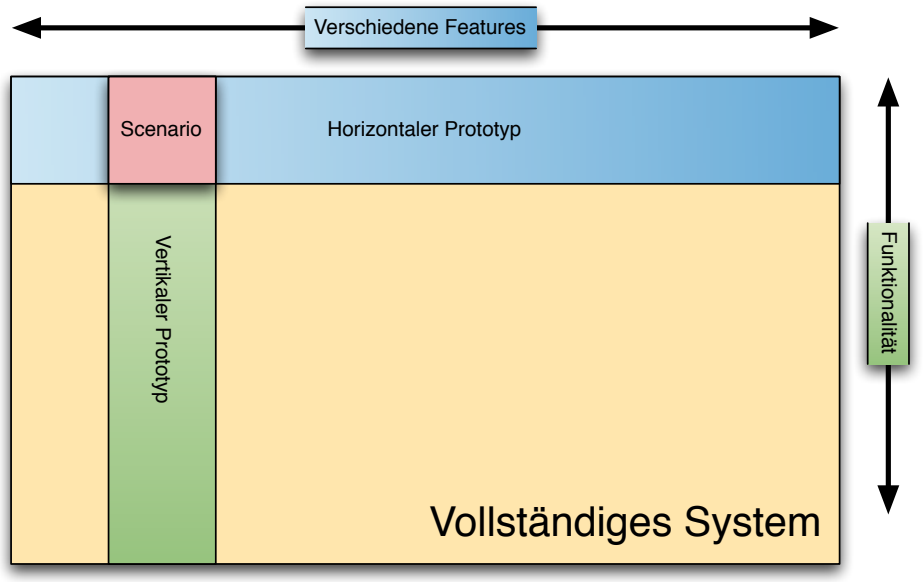


Abbildung 2.12: Horizontaler und vertikaler Prototyp.

Demonstrationsprototyp: Dieser Prototyp soll vor allem die Benutzerschnittstelle, allerdings auch die Handhabung und die prinzipiellen Einsatzmöglichkeiten des zukünftigen Endproduktes zeigen. Für Demonstrationsprototypen wird in vielen Fällen der *Throw-Away* Ansatz gewählt [KOSS11].

Funktionaler Prototyp: Bei dieser Art Prototyp werden eine oder mehrere Aspekte der Funktionalität implementiert. Dabei gibt es zwei unterschiedliche Vorgehensweisen, zum einen den horizontalen Prototyp und zum anderen den vertikalen Prototyp (siehe auch Abbildung 2.12). Der horizontale Prototyp deckt eine Vielzahl an Funktionalität, die aber nicht komplett implementiert ist.

Beispiel wäre eine Benutzerschnittstelle, die schon komplett entwickelt ist, bei der aber die Verarbeitung der Eingaben fehlt. Der vertikale Prototyp spezialisiert sich meist auf nur eine spezielle Aufgabe, dafür ist diese aber auch voll implementiert. Als Beispiel, bei einer Benutzerschnittstelle wäre diese bis auf wenige Bedienelemente leer, dafür würde allerdings die Funktionalität hinter den vorhandenen Bedienelementen schon komplett implementiert sein. Auch für den funktionalen Prototyp wird häufig der *Throw-Away* Ansatz gewählt [KOSS11].

Labormuster (Labormodell): Dieser Prototyp dient den Entwicklern intern als Bewertungsgegenstand, der Fragen der technischen Umsetzung und der Realisierbarkeit klären soll [KOSS11].

Pilotsystem: Pilotsysteme sind Weiterentwicklungen der Labormuster, die so ausgereift sind, dass sie nicht nur im Labor sondern auch schon im Anwendungsbereich selbst eingesetzt werden können. [KOSS11]

2.1.13 Vorgehensmodelle Zusammenfassung

Die in diesem Kapitel vorgestellten Vorgehensmodelle sind die klassischen Methoden zur Entwicklung von Software. Sie sind allgemeingültige Modelle, die für die Entwicklung fast jedes Softwareprojektes verwendet werden können. Sie bieten keine speziellen Lösungen für bestimmte Projekte und sind oft nicht durch Softwarewerkzeuge unterstützt. Allerdings basiert auch mein vorgestelltes Entwurfsvorgehen auf Prinzipien der hier vorgestellten Vorgehensmodelle. Begriffe wie Iteration, Prototyp und kurze Zyklen werden sich auch in meinem Entwurfsvorgehen wiederfinden.

In Kapitel 3.2_{60} gehe ich auf speziell für Mixed Reality entwickelte Vorgehensmodelle und Entwurfskonzepte ein, die den Stand der aktuellen Forschung widerspiegeln.

2.2 Architekturmuster

Im Bereich der Softwareentwicklung sind Architekturmuster (auch: Architekturstil, engl. *architectural style*) in den Arten von Mustern auf oberster Ebene einzuordnen. Im Gegensatz zu Idiomen[5] oder Ent-

[5]Idiome sind den Mustern (engl. *pattern*) zugeordnet. Buschmann definiert: „Ein Idiom ist ein programmiersprachenspezifisches Muster und damit ein Muster auf einer niedrigen Abstraktionsebene. Ein Idiom beschreibt, wie man bestimmte Aspekte von Komponenten oder Beziehungen zwischen ihnen mit den Mitteln einer bestimmten Programmiersprache implementiert.“ [BMRS98]

wurfsmustern[6] bestimmen sie nicht ein konkretes (meist kleines oder lokales) Teilproblem, sondern den Grundaufbau, also das Fundament der Anwendung. [BMRS98]

Architekturmuster lassen sich in vier verschiedene Kategorien einteilen [Wik11]:

Chaos zu Struktur (engl. Mud-to-structure): Diese speziellen Muster sollen helfen, die Vielzahl an Komponenten und Objekten eines Softwaresystems zu organisieren. Die Funktionalität des Gesamtsystems wird hierbei in kooperierende Subsysteme aufgeteilt. Diese Kategorie beinhaltet folgende Muster:

- Pipes und Filter
- Schichtenarchitektur (auch: mehrschichtige bzw. N-Tier-Architektur)
- Schwarzes Brett (engl. Blackboard)

Verteilte Systeme: Diese Kategorie unterstützt die Verwendung verteilter Ressourcen und Dienste in Netzwerken (z. B. serviceorientierte Architekturen, Orchestrierung). Die beiden Modelle *Mikrokernel* und *Pipes und Filter*) unterstützen eine Verteilung auch, aber eher zweitrangig. Folgende Muster fallen unter diese Kategorie:

- Broker bzw. Vermittler
- Client-Server

Interaktive Systeme: Interaktive Systeme sollen die Mensch-Computer-Interaktionen strukturieren und vereinfachen. In dieser Kategorie stehen folgende Muster:

- Model-View-Controller (MVC)
- Presentation-Abstraction-Control (PAC)

Adaptive Systeme: Bei diesem Muster wird die Erweiterungs- und Anpassungsfähigkeit von Softwaresystemen besonderes unterstützt. Es fallen folgende Muster unter diese Kategorie:

- Mikrokernel
- Reflexion
- Dependency Injection

[6] Entwurfsmuster (engl. design patterns) sind bewährte Lösungs-Schablonen für wiederkehrende Entwurfsprobleme in Softwarearchitektur und Softwareentwicklung. [GHJV96]

In meiner Arbeit habe ich das bekannte MVC-Modell verwendet und dahingehend erweitert, dass es in mein Entwurfsvorgehen eingepasst wurde. Deshalb werde ich hier nur die beiden Architekturmodelle MVC und PAC aus der Kategorie der Interaktiven Systeme vorstellen.

2.2.1 Model-View-Controller

Das *Model-View-Controller* (MVC, zu deutsch „Modell, Präsentation, Steuerung“) Architekturmuster dient bei der Entwicklung von Software zur Strukturierung in drei Einheiten: Dem Datenmodell (engl. *Model*), der Präsentation (engl. *View*) und Programmsteuerung (engl. *Controller*). Das Ziel dieses Musters ist ein flexibler Programmentwurf, der spätere Änderungen oder Erweiterungen erleichtert und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglicht. MVC ist in der Entwicklung von Benutzerschnittstellen (engl. *User Interfaces*, kurz *UI*) ein weit verbreitetes Muster. Es wurde im Jahre 1979 zunächst exakt für UIs in Smalltalk durch Trygve Reenskaug, der damals an Smalltalk im Xerox PARC arbeitete, beschrieben (Seeheim-Modell) [Ree03]. Mittlerweile gilt MVC aber als De-facto Standard für den Grobentwurf aller komplexen Softwaresysteme, teils mit Differenzierungen und oftmals mehreren, jeweils nach dem MVC-Muster aufgeteilten, Modulen [Wik11].

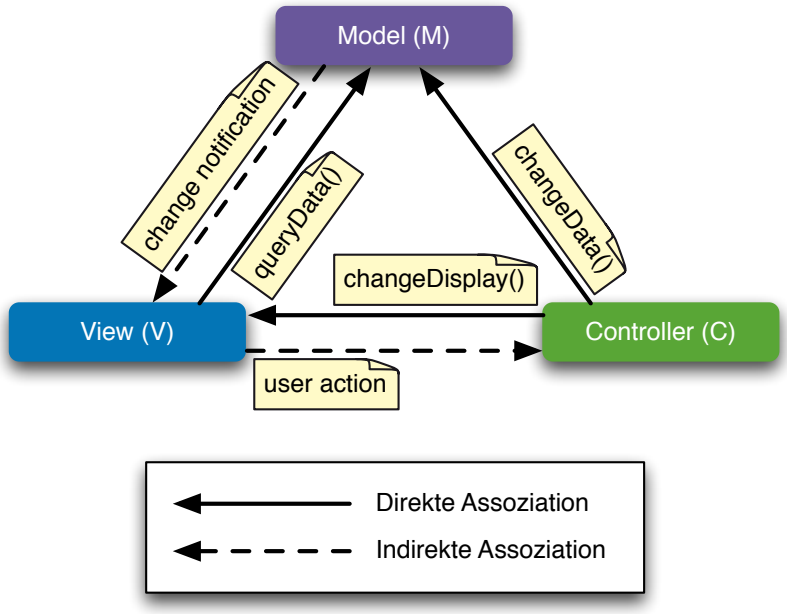


Abbildung 2.13: Model-View-Controller Architekturmuster.

In Abbildung 2.13 sind die drei Komponenten *Model*, *View* und *Controller* und deren Beziehung zueinander zu sehen. In der Abbildung

repräsentiert die durchgezogene Linie eine direkte Assoziation, die gestrichelte eine indirekte Assoziation (z. B. über einen *Observer*[7]). Die Annotation an den Pfeilen beschreibt die Aktionen bzw. Funktionen, die jeweils aufgerufen werden können. So schickt z. B. das *Model* eine Benachrichtigung an den *View*, dass sich die Daten des *Model* geändert haben (*change notification*) woraufhin der View dann die neuen Daten vom *Model* abfragt (*queryData()*) und sie dann darstellt.

Je nach Realisierung hängen die Komponenten unterschiedlich stark voneinander ab und sind für folgende Aufgaben gedacht:

Model: Es beinhaltet die darzustellenden Daten und ist vom *View* und vom *Controller* unabhängig. Die Bekanntgabe von Änderungen an relevanten Daten im Model geschieht nach dem Entwurfsmuster Beobachter (engl. *Observer*). Das Modell ist das zu beobachtende Subjekt, auch *Publisher*, genannt [Wik11].

View: Der *View* ist für die Darstellung der Daten aus dem *Model* und die Entgegennahme von Benutzerinteraktionen zuständig. Dem *View* sind sowohl sein *Controller* als auch das *Model* bekannt, dessen Daten er darstellt. Für die Weiterverarbeitung der vom Benutzer übergebenen Daten ist er aber nicht zuständig. Im Regelfall wird der *View* über Änderungen von Daten im Modell mithilfe des Entwurfsmusters Beobachter unterrichtet und kann sich daraufhin die aktualisierten Daten besorgen. Der *View* verwendet das Entwurfsmuster Kompositum[8] [Wik11].

Controller: Der *Controller* kann einen oder mehrere *Views* verwalten und nimmt von ihnen Benutzeraktionen entgegen, die er dann auswertet entsprechend agiert. Zu jedem *View* existiert ein *Model*. Es ist nicht die Aufgabe der Steuerung, Daten zu manipulieren. Der *Controller* entscheidet aufgrund der Benutzeraktion im *View*, welche Daten im *Model* geändert werden müssen. Er enthält weiterhin Mechanismen, um die Benutzerinteraktionen des *View* einzuschränken. *View* und *Controller* verwenden zusammen das Entwurfsmuster Strategie[9], wobei der *Controller* der Strategie entspricht. Der Controller kann in manchen Implementierungen

[7]Der *Observer* gehört zur Kategorie der Verhaltensmuster (engl. *Behavioural Patterns*). Es dient zur Weitergabe von Änderungen an einem Objekt an von diesem Objekt abhängige Strukturen. [GHJV96]

[8]Das Kompositum (engl. *Composite*) gehört zu der Kategorie der Strukturmuster (*Structural Patterns*). Es wird angewendet, um Teil-Ganzes-Hierarchien zu repräsentieren, indem Objekte zu Baumstrukturen zusammengefügt werden. Die Grundidee des Kompositionsmusters (*Composite-Pattern*) ist, in einer abstrakten Klasse sowohl primitive Objekte als auch ihre Behälter zu repräsentieren. Somit können sowohl einzelne Objekte, als auch ihre Kompositionen einheitlich behandelt werden. [GHJV96]

[9]Die Strategie (engl. *Strategy*)gehört zu der Kategorie der Verhaltensmuster (*Behavioural Patterns*). Das Muster definiert eine Familie austauschbarer Algorithmen.. [GHJV96]

ebenfalls zu einem Beobachter des *Model* werden, um bei Änderungen der Daten den *View* direkt zu manipulieren [Wik11].

Der Vorteil der Dekomposition nach dem *Model-View-Controller* Architekturmuster, die hier am Beispiel von Benutzerschnittstellen beschrieben wurde, ist, dass die Aspekte der Visualisierung und der Interaktion von der unterliegenden Applikation getrennt behandelt werden können. Mit dem *Model-View-Controller* Architekturmuster wird ein modulares Design ermöglicht, bei dem Änderungen einer Komponente keine Auswirkungen auf die Implementation der übrigen Komponenten haben. Ein weiterer Vorteil ist die Möglichkeit der Verwendung mehrere *Views* und *Controller* für ein *Model*.

2.2.2 Presentation-Abstraction-Control

Das Architekturmuster *Presentation-Abstraction-Control* (PAC, was ins Deutsche übersetzt bedeutet „Darstellung-Abstraktion-Steuerung“) wird zur Strukturierung von interaktiven Softwaresystemen vewrendet. Es ist eine Weiterentwicklung des in Kapitel 2.2.1[45] vorgestellten MVC Models und wurde von Prof. Joëlle Coutaz im Jahre 1987 vorgestellt [Cou87]. Mit dem PAC Muster können interaktive Systeme so entwickeln werden, dass diese aus einzelnen Teilen bestehen, die jeweils einen Teil der Aufgaben des gesamten Systems abbilden und damit eine hohe Flexibilität des Systems gewähren. PAC stellt sicher, dass die Teile zu einem funktionierenden Ganzen zusammengesetzt werden können [Wik11].

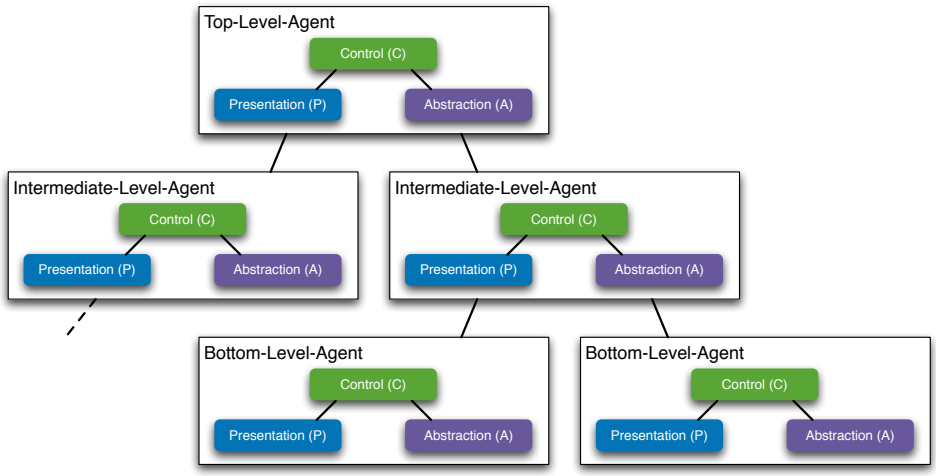


Abbildung 2.14: Aufbau von Presentation-Abstraction-Control(PAC).

PAC teilt ein System in zwei Richtungen auf: Zum einen in die drei Einheiten grafische Benutzungsschnittstelle (engl. *Presentation*), Ver-

mittlung und Kommunikation (engl. *Control*) und das Datenmodell (engl. *Abstraction*) – dies ist ähnlich dem MVC Muster – und zum anderen hierarchisch in verschiedene Elemente, die jeweils einen Teil der Aufgaben des Systems abbilden. Diese Teile werden im PAC Muster als Agenten bezeichnet und sie stellen die erste Stufe der Strukturierung während des Architekturentwurfes dar [Wik11].

Die Hierarchie von PAC ist in insgesamt drei Ebenen unterteilt, wie in Abbildung 2.14_{47} zu erkennen ist. Die oberste Ebene besteht aus einem einzigen so genannten *Top-Level-Agent*, der für das System alle globalen Aufgaben übernimmt, z. B. Datenbankzugriffe. Auf der zweiten Ebene liegen die *Intermediate-Level-Agents*, die eine Schnittstelle zwischen der untersten (*Bottom-Level*) und der obersten (*Top-Level*) Ebene bilden und mehrere *Bottom-Level-Agents* zu einer Einheit zusammenfassen. Dabei besteht die Möglichkeit, dass in dieser Ebene die Teilsysteme weiter hierarchisch aufgeteilt werden können, so dass ein Teilsystem auch aus einem oder mehreren anderen bestehen kann, was bedeutet, dass ein *Intermediate-Level-Agent* auch mehrere andere *Intermediate-Level-Agents* zusammenfassen kann. Die dritte Ebene besteht aus den *Bottom-Level-Agents*, welche die eigentlichen Funktionen des interaktiven Systems abbilden, wobei jeder seine eigene, möglichst abgeschlossene, Funktion beinhaltet und möglichst über keine Abhängigkeiten zu anderen *Bottom-Level-Agents* verfügen sollte [Wik11].

Der Architekturentwurf beginnt mit der Aufteilung der geforderten Funktionalität auf mehrere *Bottom-Level-Agents*. Anschließend wird bei dem *Top-Level-Agent* festgelegt, welche Funktionalität dieser erbringen soll. Die Hierarchie wird daraufhin mit der Festlegung der *Intermediate-Level-Agents* vervollständigt, die eine Kombination aus *Bottom-Level-Agents* darstellen und diesen den Zugriff auf den *Top-Level-Agent* vermitteln [Wik11].

Wie schon oben beschrieben wird jeder Agent in drei Komponenten aufgeteilt. Die erste Komponente entspricht der grafischen Benutzeroberfläche (*Presentation*), die die komplette Ein- und Ausgabe umfasst (anders bei dem MVC Muster, bei dem diese noch aufgeteilt wird in *View* und *Controller*). Die zweite Komponente repräsentiert die Abstraktion (*Abstraction*), die das Datenmodell des jeweiligen Agenten realisiert. Die dritte Komponente, die Vermittlung und Kommunikation (*Control*), stellt die Verbindung zwischen den beiden anderen Komponenten her und ermöglicht die Kommunikation mit anderen Agenten. Damit ist diese Komponente die zentrale Schnittstelle für die Zusammenarbeit der einzelnen Teile eines Systems im PAC-Muster [Wik11].

Es ist nicht zwingend notwendig, dass jeder Agent alle drei Komponenten beinhaltet, sondern jeder Agent bringt die Benutzerschnittstelle und das Datenmodell für seine Aufgabe mit. Es ist somit vorstellbar, dass z. B. ein *Intermediate-Level-Agent* nur in einem Fenster ihm untergeordnete Agenten zusammenfasst und anzeigt, selber aber dafür kein Datenmodell benötigt. Jeder Agent muss allerdings die Steuerungskomponente beinhalten, da ansonsten eine Kommunikation zwischen Komponenten und mit anderen Agenten nicht möglich wäre [Wik11].

2.2.3 Zusammenfassung

MVC und PAC eignen sich hervorragend, um Komponenten entsprechend ihrer Funktionalität aufzuteilen und so getrennt zu entwickeln. Für meinen Entwurfsprozess habe ich das MVE Architekturmuster als Grundlage gewählt und erweitert, da ich die Hierarchie von PAC nicht benötigte. Die Aufteilung in kleinere Teile kann in meinem Entwurfsprozess optional innerhalb der einzelnen Komponenten des erweiterten MVE Architekturmusters vorgenommen werden.

2.3 Modellbildung und Simulation kontinuierlicher Systeme

In vielen Bereichen der Entwicklung von kontinuierlichen Systemen wird heutzutage die Simulation als Werkzeug genutzt. Durch die Simulation erlangt man ein besseres Verständnis über komplexe, dynamische Systeme und sie erleichtert die Entwicklung dieser Systeme. In diesem Abschnitt beziehe ich mich nur auf die Simulation kontinuierlicher Systeme. Bei der Simulation beispielsweise diskreter Systeme (z. B. Digitalschaltungen) muss das resultierende Modell nicht notwendigerweise mit Differenzialgleichungen beschreiben werden.

Grundlage der Simulation ist ein mathematisches bzw. physikalisches Modell des kontinuierlichen Systems, welches anhand von Beobachtungen bzw. theoretischer Grundlagen entwickelt wird. Dieses Modell sollte das zu analysierende System hinreichend gut beschreiben. Die aus dem Modell resultierenden Differenzialgleichungen können durch einen Computer berechnet und gelöst werden. Modelle können, je nach Anforderung, beliebig komplex werden, meist reichen jedoch Approximationen des Systems aus, um verwertbare Daten zu erhalten. Um den Aufwand der Berechnung gering zu halten, sollte das Modell nur genau das beschreiben, was für die spätere Auswertung nötig ist [Sch04]. Die Daten der Simulation können, je nach Zielsetzung,

vielfältig genutzt werden, z. B. um das reale System oder das Modell zu verbessern.

Die Durchführung von Simulationen kann verschiedene Gründe haben, z. B.

Entwicklung: Überprüfung und Optimierung des Systems und seiner Parameter vor der eigentlichen Prototypenentwicklung.

Wiederholbarkeit: Simulationen können immer mit denselben Voraussetzungen wiederholt werden, was zu gleichen Ergebnissen führt. Daher gestaltet sich eine Fehleranalyse leicht, da Fehler reproduzierbar werden.

Beobachtbarkeit: Viele technische Systeme sind nur schwer zu beobachten. Durch die Simulation ist es möglich, die Vorgänge, die sonst nicht sichtbar wären, zu visualisieren. Dabei ist sowohl der zeitliche (z. B. Zeitlupe) als auch der optische Aspekt (z. B. Visualisierung von nicht sichtbaren Vorgängen) zu nennen.

Gefahrenvermeidung: Das Entwickeln an realen Systemen kann zu einer Gefährdung von Mensch und Maschine führen, die durch die Simulation verhindert wird.

Training: Über die Simulation kann der Benutzer lernen, das System zu bedienen, ohne sich und andere in Gefahr zu bringen oder das reale System zu beschädigen.

Kosten: Die Entwicklung von Simulatoren ist günstiger als die Herstellung realer Prototypen, gerade in den ersten Phasen der Entwicklung.

Für die Simulation gibt es zwei grundsätzlich verschiedene Arten der Durchführung, *Offline* oder in Echtzeit. Die *Offline-Simulation* generiert Daten, die nach Abschluss der Simulation analysiert werden können. Auf diesen Daten können verschiedene Visualisierungen ausgeführt werden und es ist möglich, die Simulation zu stoppen, zu verlangsamen oder rückwärts laufen zu lassen. Eine Echtzeit-Visualisierung, d. h. die Daten werden zeitlich korrekt wiedergegeben, ist möglich, aber eine Interaktion mit dem System kann in einer *Offline-Simulation* nicht erreicht werden. Sollte etwas am System verändert werden, muss die *Offline-Simulation* komplett neu ausgeführt werden.

Das andere Verfahren zur Durchführung einer Simulation ist die *Echtzeit-Simulation*. Sie erlaubt die Interaktion des Benutzers mit dem System während der Laufzeit und kann auf Änderung der Parameter

reagieren. In der Echtzeitsimulation, gerade auch wenn die Simulation mit realen Sensoren bzw. Aktuatoren gekoppelt ist, ist es nicht mehr möglich die zeitliche Abfolge zu verändern oder zu verlangsamen. Damit wird die Beobachtbarkeit der Ereignisse etwas eingeschränkt.

2.3.1 In-the-Loop Simulation

In der Elektrotechnik und im Maschinenbau beziehungsweise der Mechatronik haben sich Methoden für einen strukturierten Entwurfsprozess etabliert, die eine Simulation eines Systems und seiner Einzelkomponenten auch im Kontext der Zusammenarbeit mit realen Systemkomponenten ermöglichen. Hierbei handelt es sich häufig um Regelkreise oder Prozesssteuerungen [CMPH08].

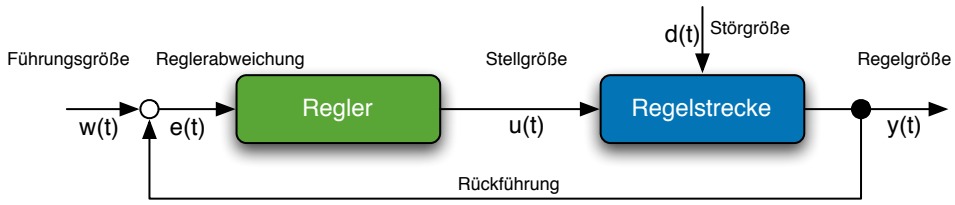


Abbildung 2.15: Aufbau eines allgemeinen Regelkreises.

Neben den hier vorgestellten mechatronischen Regelkreisen existieren auch Regelkreise für rein elektrische Systeme. Allgemein können Regelkreise für jedes komplexe System existieren, so dass die Trägerstruktur nicht notwendigerweise aus dem Bereich des Maschinenbaus stammen muss, wie in Abbildung 2.15 zu erkennen ist. Die Abbildung zeigt einen einfachen Regelkreis der aus einer Regelstrecke, einem Regler und einer Rückkopplung der Regelgröße y (dem Istwert) besteht. Dabei wird die Regelgröße y mit der Führungsgröße w (dem Sollwert) verglichen und die Regelabweichung $e = w - y$ berechnet. Die Regelabweichung wird dem Regler übergeben, der daraus die Stellgröße u gemäß der gewünschten Dynamik berechnet. Die Stellgröße u und die Störgröße d werden der Regelstrecke übergeben, die daraus die Regelgröße y bildet. Im Gegensatz zum allgemeinen Regelkreis können Mechatronischen Regelkreise wie in Abbildung 2.16[52] dargestellt werden.

Wie in Abbildung 2.16[52] zu sehen gibt es beim mechatronischen Regelkreis ein Grundsystem, dass sich in einer Umgebung befindet. Dieses bildet die Tragestruktur des gesamten Systems und ist normalerweise dem Bereich Maschinenbau zuzuordnen. Im Grundsystem befinden sich Sensoren, die Informationen der Umgebung aufnehmen oder den Zustand des Grundsystems feststellen. Die Informationen der Sensoren werden durch Hard- und Softwarekomponenten verarbeitet. Die

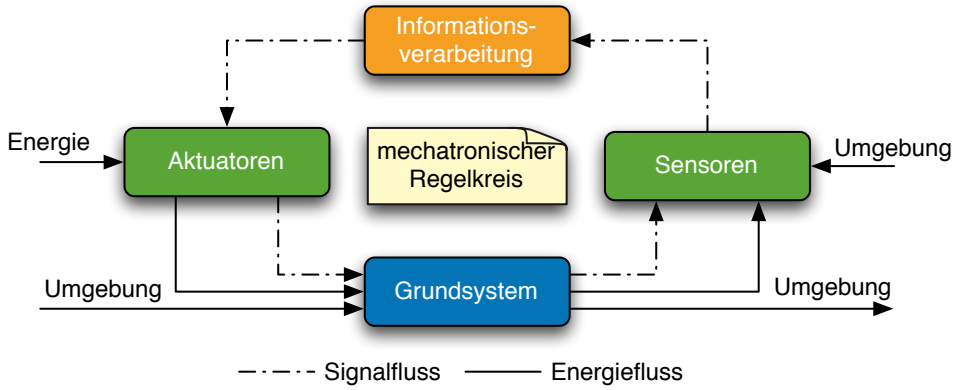


Abbildung 2.16: Aufbau eines mechatronischen Regelkreises.

Informationsverarbeitung steuert über Aktuatoren das mechanische Grundsystem.

Ein solches System kann in verschiedenen Stufen simuliert werden, bevor am Ende der Entwicklung ein fertiges System entsteht.

Model-in-the-Loop (MiL)

Die erste Phase der Simulation ist die *Model-in-the-Loop* Simulation (MiL), in der das zu simulierende Komponente mit speziellen Werkzeugen (z. B. MATLAB/Simulink oder Labview) modelliert und in das Simulationsmodell der Umgebung eingebettet wird. In dieser Phase wird die Funktionalität geprüft und ggf. das Modell angepasst. Das Modell existiert nur in den Werkzeugen und die Funktionalität wird dort simuliert.

Software-in-the-Loop (SiL)

Bei der *Software-in-the-Loop* Simulation (SiL) wird das zuvor in oberen Abschnitt beschriebene Modell nun in einen Code für eine bestimmte Plattform übersetzt. Dieser Code wird dann simuliert, d. h. er wird nicht auf der speziellen Hardwareplattform ausgeführt, sondern in einem Hardwaresimulator. Dabei sollten die simulierten Daten möglichst den Daten der MiL Simulation gleichen. Werkzeuge wie beispielsweise MATLAB/Simulink bieten eine Codegenerierung in Plattformabhängigen Quelltext an, so dass der Schritt vom Modell zum Code keine Schwierigkeiten beinhaltet.

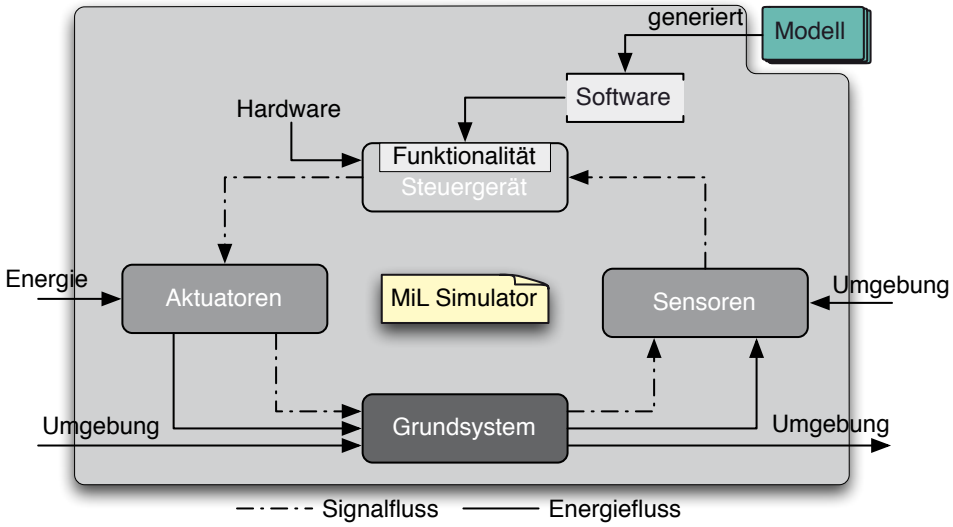


Abbildung 2.17: Model-in-the-Loop Simulation.

Hardware-in-the-Loop (HiL)

Die letzte Phase ist die *Hardware-in-the-Loop* Simulation (HiL), bei der nun der Code, der in der SiL Simulation generiert wurde, auf der entsprechenden Hardware (*Embedded System*) ausgeführt wird. Die Hardware wird mit der Simulationsumgebung gekoppelt, die die Eingaben und Ausgaben der Hardware übernimmt und auswertet.

Marin Schlager beschreibt in seinem Buch „Hardware-in-the-Loop Simulation: A Scalable, Component-based, Time-triggered Hardware-in-the-loop Simulation Framework“ [Scho8] die grundlegenden Prinzipen der Entwicklung von HiL-Simulatoren. Gerade im Bereich der sicherheitskritischen Realzeitsysteme ist es wichtig, das eine korrekte Ausführung in jeder Situation gewährleistet ist, auch bei sehr unwahrscheinlichen Situationen. Dabei sind *Hardware-in-the-Loop* (HiL) Simulationen ein gebräuchlicher Weg um die Realzeitsysteme zu validieren.

2.3.2 Zusammenfassung

Für mein Problem ergab sich, dass alle drei *In-the-Loop*-Techniken für den Design-Prozess geeignet sind und sich in das Architekturmuster perfekt eingliedern. So war es möglich, die verschiedenen Arten der Simulationen in einem Beispiel, das in Kapitel 5_{159} beschrieben wird, einzubinden. Es wurde in diesem Beispiel iterativ von der ersten bis zur dritten *In-the-Loop*-Technik zuerst das Modell, gefolgt von der Software bis hin zur Hardware eine Höhensteuerung entwickelt und

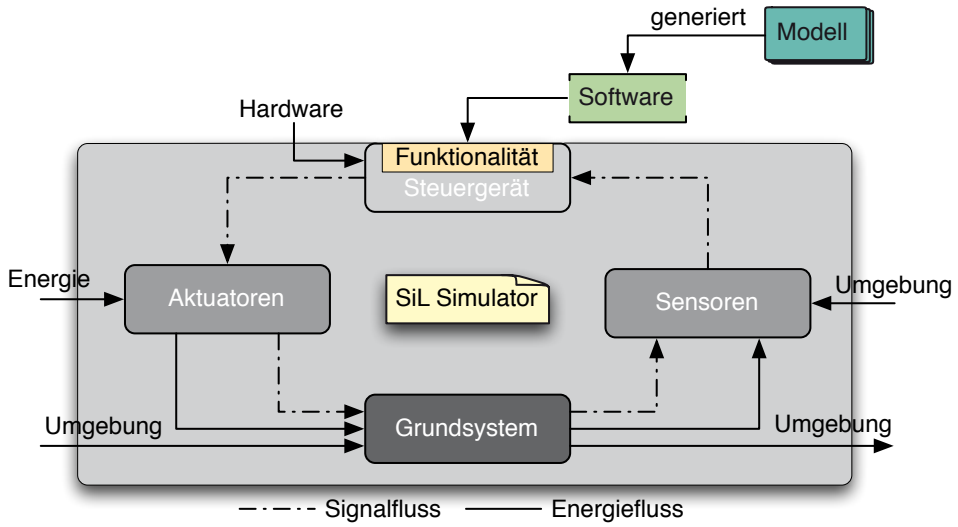


Abbildung 2.18: Software-in-the-Loop Simulation.

validiert. Dabei wurden unterschiedliche Werkzeuge verwendet, die die unterschiedlichen Stufen unterstützten.

2.4 Reality-Virtuality Kontinuum (RV)

Das *Reality-Virtuality* Kontinuum (RV)[10] wurde 1994 von Paul Milgram et. al. in der Veröffentlichung „Augmented reality: A class of displays on the reality-virtuality continuum“ [MTUK94] definiert. Es umschließt alle Mischformen von Virtualität (oder besser virtuelle Realität) und Realität, die jeweils die Grenzen des Kontinuums bilden, wie an Abbildung 2.20$_{55}$ zu sehen ist Der Raum zwischen den beiden Extremen wird „Mixed Reality“ (MR), also Vermischte Realität bzw. Gemischte Realität, genannt. Die bekanntesten Zwischenformen sind dabei die *Augmented Virtuality* (AV, zu deutsch erweiterte Virtualität) und die *Augmented Reality* (AR, zu deutsch erweiterte Realität).

2.4.1 Realität

Die Realität ist das rechte Ende des *Reality-Virtuality* Kontinuum (siehe Abbildung 2.20$_{55}$) und beschreibt die Wirklichkeit, so wie sie ein Mensch wahrnimmt. Dieses kann durch eine einfache Darstellung eines Videobildes über einen Bildschirm geschehen oder auch durch

[10] In dieser Arbeit werde ich das *Reality-Virtuality* Kontinuum auch als *Mixed Reality* Kontinuum bezeichnen, da sich die Anwendungen, die mit meinem Entwurfsvorgehen entwickelt werden, im Bereich der gemischten Realität ansiedeln.

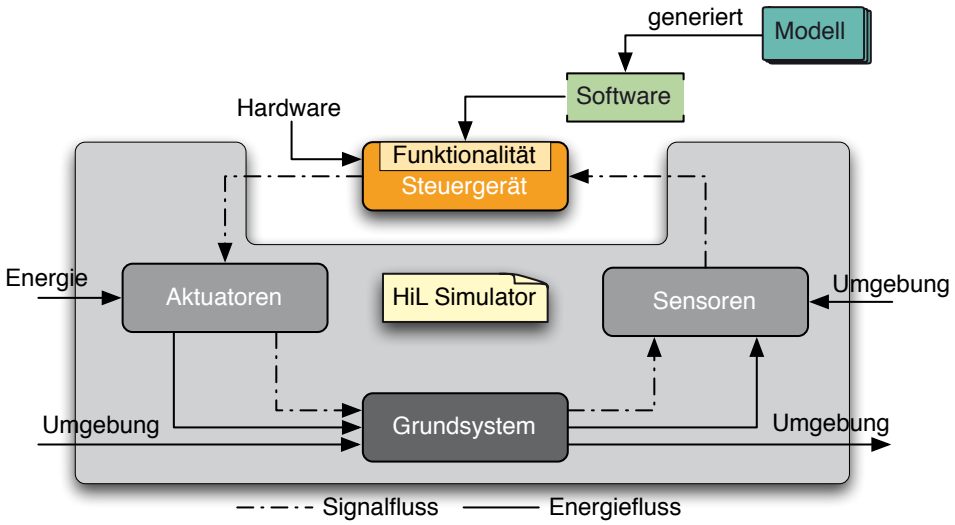


Abbildung 2.19: Hardware-in-the-Loop Simulation.

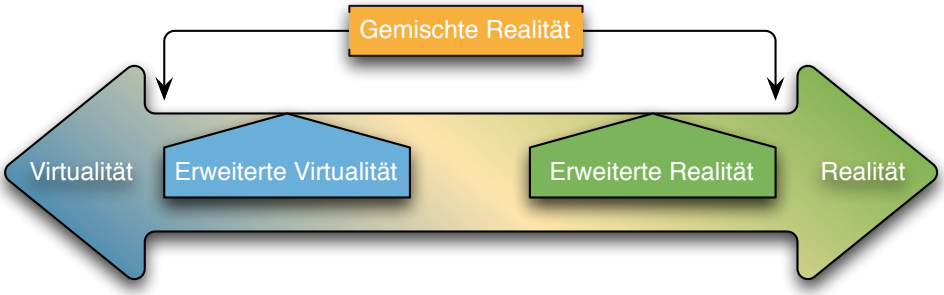


Abbildung 2.20: Reality-Virtuality Continuum von Milgram [MTUK94].

die direkte Betrachtung ohne zusätzliche elektronische Hilfsmittel. Die Realität benötigt keine Positionsbestimmung, da hier nur rein reale Objekte dargestellt werden. Auch aufwendige Visualisierungen fallen hier weg. Software, die rein auf der Realität basiert, ist z.B. eine Videokamera, die einfach die Realität als Abfolge von Einzelbildern auf dem Computer speichert.

2.4.2 Virtuelle Realität

Als virtuelle Realität (engl. *Virtual Reality*, kurz VR), wird die Darstellung und gleichzeitige Wahrnehmung der Wirklichkeit und ihrer physikalischen Eigenschaften in einer in Echtzeit computergenerierten, interaktiven virtuellen Umgebung bezeichnet. Es wird versucht die Wirklichkeit so gut wie möglich abzubilden und die physikalischen Eigenschaften zu simulieren. Beispiele sind Flug- oder Fahrzeugsimu-

latoren, die versuchen die Gravitationskräfte nachzuempfinden. Aber auch einfache virtuelle Umgebungen, die nicht die physikalischen Eigenschaften zu simulieren versuchen, werden zur virtuellen Realität gezählt. Hierzu zählen auch Großraumprojektionen und *CAVEs*[11]. Für virtuelle Systeme werden meist leistungsstarke Rechner und Grafiksysteme benötigt, die in der Lage sind die virtuelle Umgebung in Echtheit darzustellen. Für *CAVEs* werden zusätzlich Trackingsysteme[12] benötigt um die korrekte Projektion für den Benutzer bestimmen zu können.

2.4.3 Augmented Virtuality

Als erweiterte Virtualität (engl. *Augmented Virtuality*, kurz AV) werden Systeme angesehen, die größtenteils aus einer VR-Umgebung bestehen und Teile der Realität in die VR-Umgebung einbeziehen. Beispiele wären ein reales Video, was in der virtuellen Umgebung dargestellt wird, oder reale Audioquellen wie z. B. eine Türklingel, die an die virtuelle Umgebung weitergeleitet wird. Leistungsstarke Rechner und Grafiksysteme sind für die Verwendung von AV notwendig, da die Umgebung zum größten Teil virtuell ist. Tracking kann in Einzelfällen nötig sein, um z. B. das reale Bild mit der virtuellen Umgebung zu synchronisieren.

2.4.4 Augmented Reality

Bei der erweiterte Realität (engl. *Augmented Reality*, kurz AR) steht im Gegensatz zur virtuellen Realität, bei der der Benutzer komplett in einer virtuellen Welt eintaucht, die Darstellung zusätzlicher Informationen im Vordergrund. Für die visuelle Modalität führt dies zu wesentlich härteren Anforderungen an die Positionsbestimmung (*Tracking*) und Kalibrierung. Unter einem AR-System (kurz ARS) versteht man das System der technischen Bestandteile, die nötig sind, um eine *Augmented-Reality* Anwendung aufzubauen, die da wären: Kamera, Trackinggeräte, Unterstützungssoftware usw.

Die Literatur verwendet meist die Definition der erweiterten Realität von Azuma [Azu97]:

- Die virtuelle Realität und die Realität sind miteinander kombiniert (teilweise überlagert).

[11] CAVE steht für „Cave Automatic Virtual Environment“, zu deutsch „Höhle mit automatisierter, virtueller Umwelt“.

[12] Mit *Tracking* bezeichnet man die kontinuierliche Positionsbestimmung realer Objekte im Raum. Die Positionsbestimmung kann Zwei- oder Dreidimensional erfolgen.

- Die Interaktivität erfolgt in Echtzeit.
- Reale und virtuelle Objekte haben einen dreidimensional Bezug zueinander.

Diese Definition beschränkt sich leider nur auf die technischen Merkmale, die allerdings nur ein Teilaspekt der erweiterten Realität sind. AR wird in anderen Arbeiten (beispielsweise bei „Cybertechnologien als Werkzeug im Bauwesen“ [EB08]) als eine Ausweitung der Sinneswahrnehmung des Menschen durch Sensoren von Umgebungseigenschaften definiert, die der Mensch selbst nicht wahrnehmen kann. Beispiele für diese Definition sind Radar, Infrarot, Distanzbilder, etc., die die Normale Sichtweise des Benutzers erweitern können.

2.5 Zusammenfassung

In diesem Kapitel wurden zu Anfang die verschiedenen Vorgehensmodelle und Architekturmuster vorgestellt, die Grundlage dieser Arbeit sind. Die herausragenden Aspekte der einzelnen Modelle und Muster wurden in den jeweiligen Unterkapiteln kurz herausgestellt und erläutert. Weiterhin wurde der Aufbau der Modelle und Muster dargestellt und durch Abbildungen verdeutlicht. Die wichtigsten Quellen wurden für weiterführende Nachforschungen angegeben.

Im Kapitel „Modellbildung und Simulation“ wurden die *In-The-Loop*-Simulationen vorgestellt, die Teil meines Entwurfsvorgehens sind. Es wurden die einzelnen Simulationstechniken in einzelnen Kapitel vorgestellt und an Grafiken deren Vorgehen verdeutlicht.

Als eine weitere Grundlage dieser Arbeit wurde das von Milgram eingeführte *Reality-Virtuality* Kontinuum in diesem Kapitel vorgestellt und beschrieben. Dabei wurde der Begriff *Mixed Reality* eingeführt und die einzelnen Zwischenformen Realität, virtuelle Realität, erweiterte Virtualität und angereicherte Realität wurden in einzelnen Abschnitten kurz erklärt.

KAPITEL 3

Stand der Forschung

In diesem Kapitel wird der aktuelle Stand der Forschung in den Gebieten Mixed Reality Entwurfskonzepte und Software Frameworks vorgestellt, auf denen der MRiL-Entwurfsprozess basiert bzw. die er zu verbessern versucht. Dabei werden aktuelle Arbeiten in den Gebieten vorgestellt und kurz umrissen. Diese Arbeiten sind als Grundlage zum einen des Entwurfsprozesses selbst und zum anderen des MiReAS Software Framework zu sehen.

3.1 Übersicht

Abbildung 3.1: Anwendungen basierend auf dem ARToolKit.

Die Technik, Augmented Reality bzw. Mixed Reality Konzepte softwaremäßig zu verwenden, ist schon einige Jahre alt und wurde gerade durch das ARToolKit [KB99] für viele Entwickler erstmals einsetzbar. Dabei wurde mit dem ARToolKit eine Software-Umgebung bereitgestellt, die es dem Entwickler ermöglichte, einfach und unkompliziert eigene Inhalte in AR zu realisieren (linkes Bild in Abbildung 3.1).

allerdings benötigte die Verwendung des ARToolKits Programmiererfahrung in C und speziell in OpenGL. Da bei den ersten Versionen des ARToolKits die Grafik, die Videobildaufzeichung und das Tracking softwaretechnisch verbunden waren, war es nicht möglich, diese Komponenten zu trennen und Teile in anderen Bibliotheken zu verwenden. Ein früher Versuch sich zumindest von der Programmiersprache C zu trennen, wurde in [GRSP02] vorgestellt, bei dem das ARToolKit mit Hilfe der JNI[1] in Java eingebunden wurde (mittleres und rechtes Bild in Abbildung 3.1). Des Weiteren wurde die OpenGL Grafikschnittstelle gekapselt und in die damals aktuelle Szenegraphbibliothek Java3D eingebunden [GSS04b]. So war es möglich auf einem höheren Level AR Applikationen zu entwickeln [GSS04c] [GSS04a]. Da die Entwicklung von Java3D eingestellt wurde, war auch die Anbindung an das ARToolKit obsolet.

Um komplexere Anwendungen im Bereich Mixed Reality zu entwickeln benötigt es jedoch mehr als nur eine Softwareumgebung. Wichtig ist auch ein Entwurfsvorgehen, um die neuen Herausforderungen, die bei der Entwicklung von bzw. mit Mixed Reality entstehen, zu bewältigen. In diesem Bereich sind auch schon einige Arbeiten entstanden. Das „Mixed Reality in the Loop“- Entwurfsvorgehen (MRiL) basiert auf den Grundlagen der iterativen Entwicklung von Software, die bereits im Kapitel 1_1 vorgestellt wurden, dem Virtual Prototyping vorgestellt von Rix [RHT95] und Krassi [Kra08] und dem Prinzip der „Hardware in the Loop“ (HiL) Simulationen, das bei Schlager [Sch08] veröffentlicht wurde.

Sowohl das Konzept als auch die Implementierung des MRiL-Entwurfsvorgehens ist im Allgemeinen nahe verwandt mit existierenden Arbeiten im Bereich der Entwicklung von Benutzerschnittstellen , speziell mit Arbeiten aus dem Bereich der „Mixed Reality User Interfaces“, wie beispielsweise den Arbeiten von Cuppens [CRC06], Ishii [Ish08] und De Boeck [DBVRC07].

Nachfolgend werden die aktuellen Arbeiten in diesem Gebiet kurz vorgestellt und umrissen.

3.2 Mixed Reality Entwurfskonzepte

In den letzten Jahren haben sich eine Reihe namhafter Expertengruppen mit der Entwicklung neuartiger Software-Entwurfskonzepte für und mit Mixed Reality befasst. Grund dafür ist die steigende

[1] **J**ava **N**ative **I**nterface, eine Schnittstelle in Java, um plattformabhängigen C/C++-Code in Java einzubinden.

Komplexität dieser Anwendungen und das Fehlen passender Entwurfskonzepte gerade im Bereich Mixed Reality.

Virtual Prototyping - Virtual environments and the product design process

In seinem Buch „Virtual Prototyping - Virtual environments and the product design process“ [RHT95] fasste Rix et al. den damals neuen Begriff des „Virtual Prototyping“ durch verschiedene Arbeiten mehrerer Wissenschaftlern zusammen. Rix erklärt den virtuellen Prototypen als „einen bedeutenden Zwischenschritt zum Endprodukt. Anhand der gegebenen Design Information [· · ·] wird es möglich sein, einen Prototypen mit dem Computer zu generieren, der sowohl für realistische Präsentationen als auch zur Interaktion mit dem Produkt schon in frühen Entwicklungsphasen geeignet ist“[2]. Rix erwartete „durch die Entwicklung dieser Technologie und die Integration in den Produkt-Entwicklungsprozess bedeutende Vorteile im industriellen Einsatz“[3]. Dabei waren die Hauptargumente die Zeit- und Kostenersparnis und die Steigerung der Qualität eines Produkts. Aus diesem Grund wurden zwei Workshops im Herbst 1994 abgehalten, die zum Ziel hatten, die damaligen Konzepte und Methoden von „Virtual Prototyping“ zusammenzutragen. Zu diesem Zeitpunkt existierte allerdings noch kein Entwurfsvorgehen für jegliche Art von virtuellem Prototyping.

VP - Virtual environments and the product design process			
Autoren	Rix, Haas, José	Jahr	1995
Bereich	Grundlagen, Theorie		
Beschreibung	Vorstellung des Begriffs Virtual prototyping		
Merkmale:	+ Definition des Begriffs		
	+ Anwendungsbeispiele		
	- Nur VR		
	- Kein konkretes Entwurfsvorgehen		
	- Kein Modell		

[2] Aus „Virtual Prototyping - Virtual environments and the product design process“ [RHT95], Seite viii (eigene Übersetzung)

[3] Aus „Virtual Prototyping - Virtual environments and the product design process“ [RHT95], Seite viii (eigene Übersetzung)

Dynamic Virtual Prototyping for Control Engineering

Boris Krassi erläutert in seinem Buch „Dynamic Virtual Prototyping for Control Engineering“ [Krao8], dass „virtuelle Prototypen, oder digitale Mockups, die Basis der digitalen Entwicklung sind. Virtual Prototyping ist seit Jahren ein nützliches Werkzeug im Bereich der Control System Analyse und nach und nach wächst das Interesse an der direkten Entwicklung von Kontrollsystemen auf Basis von virtuellen Prototypen. Würde man die Lücke zwischen dem Control Design und dem Virtual Prototyping schließen, könnte man die Redundanz von Modellen vermeiden, Designfehler minimieren, Anpassungen von Produktänderungen erleichtern, die Zusammenarbeit zwischen Produkt- und Lifecycle-Prozessen verbessern und die Zeit bis zur Produkteinführung verkürzen. Erschwert wird dies jedoch durch die Heterogenität, Komplexität, Inkompatibilität und Unvollständigkeit der Modelle, vergleicht man die virtuellen Prototypen und Modelle, die für die Entwicklung von Kontrollsystemen benötigt werden“[4]. Krassi stellt in seinem Buch nun Konzepte, Methoden und Werkzeuge vor, um das Control System Development im dynamischen virtuellen Prototyping – einer Unterklasse des virtuellen Prototyping – zu integrieren. Das Buch zeigt, dass virtuelle Prototypen auch in Bereichen eingesetzt werden können, die normalerweise sehr präzise Modelle für die Entwicklung benötigen. Das vorliegende Buch konzentriert sich allerdings nur auf den Bereich Control Engeneering und auch die Überführung der virtuellen Prototypen zu realen Prototypen wird vernachlässigt.

Dynamic Virtual Prototyping for Control Engineering			
Autor	Krassi	Jahr	2008
Bereich	Grundlagen, Anwendungen		
Beschreibung	Verknüpfung zweier Gebiete		
Merkmale:	+ Konzepte, Methoden, Werkzeuge		
	+ Anwendungsbeispiele		
	- Nur VR		
	- Nur der Bereich Control Engeneering		
	- Nur dynamisches Prototyping		

[4] Aus „Dynamic Virtual Prototyping for Control Engineering“ [Krao8], Vorwort (eigene Übersetzung)

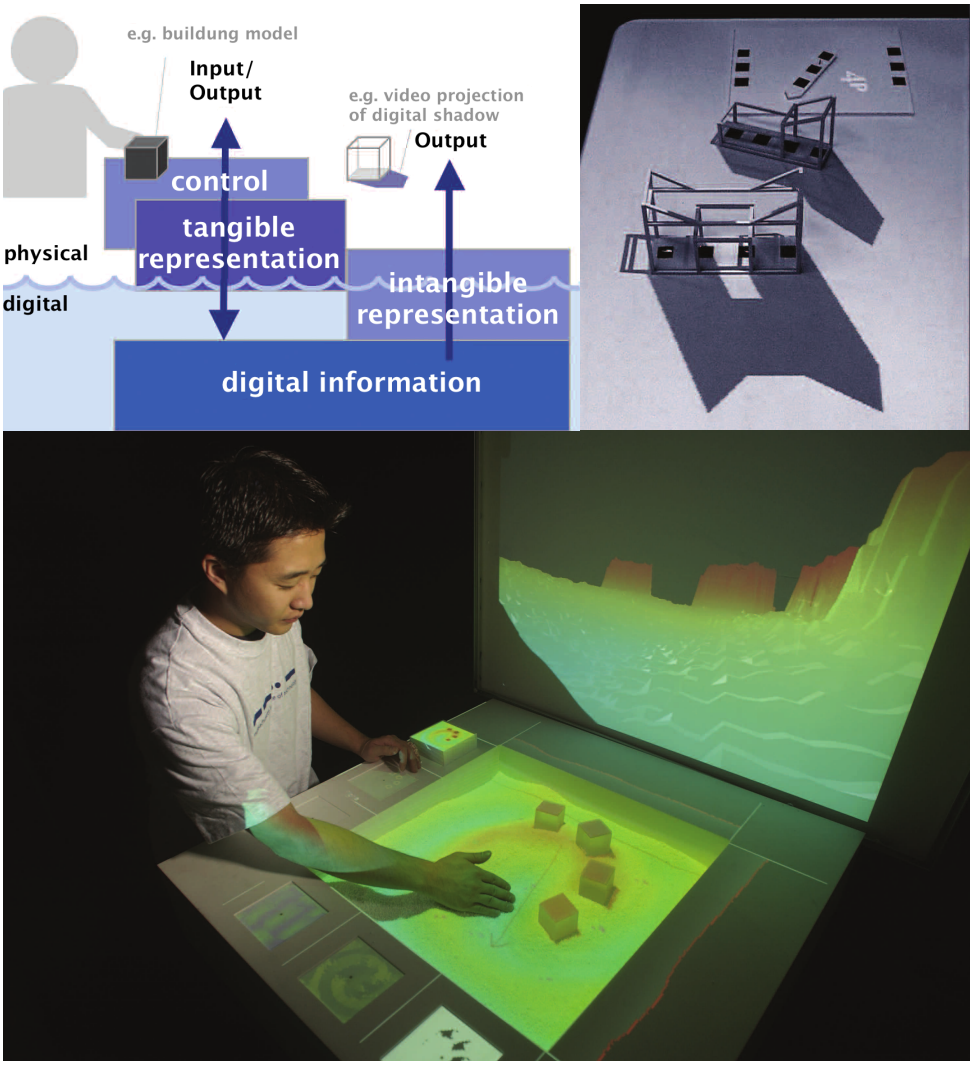


Abbildung 3.2: TUI von Ishii. [Isho8]

The tangible user interface and its evolution

Auch Ishii hat schon in seinen Arbeiten, die er in „The tangible user interface and its evolution“ [Isho8] aus dem Jahre 2008 zusammenfasst, versucht, das Prinzip der grafischen Benutzerschnittstelle zu verändern. Dies ist ähnlich dem in dieser Arbeit vorgestellten Vorgehen, das MVC Architekturmuster zu erweitert. In seiner Arbeit beschrieb er die Entwicklung der sogenannten *Tangible User Interfaces* (TUI), was zu Deutsch bedeutet: fühlbare bzw. greifbare Benutzerschnittstellen.

„Die *Tangible Media Group* des MIT Media Laboratory stellte schon Mitte der 90er Jahre von der GUI zu TUIs um. Dabei repräsentieren TUIs einen neuen Ansatz der Vision von Mark Weiser [Wei91] über

ubiquitous computing (was soviel bedeutet wie die Allgegenwärtigkeit der rechnergestützten Informationsverarbeitung), indem digitale Technologie in die Strukturen der physikalischen Umgebung eingewoben wird und so die Technologie unsichtbar erscheint. Anstatt Pixel zu Benutzerschnittstellen zu verschmelzen, benutzen TUIs eine physikalische Form, die sich nahtlos in die physikalische Umgebung des Benutzers einpasst. Ziel der TUIs ist das haptische Können bei der Interaktion auszunutzen, ein Ansatz der sich signifikant von den GUIs unterscheidet. Dabei bleibt die Hauptidee bestehen: Verleihe physikalischen Gegenständen digitale Informationen [IU97], benutze sie als Repräsentation und kontrolliere damit die digitalen Gegenstücke. TUIs machen somit digitale Informationen direkt durch unsere Hände manipulierbar und sind mit Hilfe unserer peripheren Sinne über ihre physikalische Verkörperung wahrnehmbar, siehe Abbildung 3.2 oben links“[5].

Mit Hilfe der Definition von TUIs entwickelte Ishii 1999 die erste Generation von TUIs, die sogenannte *Urban Planning Workbench* oder kurz *Urp* [UI99].

„*Urp* verwendet maßstabsgetreue physikalische Modelle architektonischer Bauwerke, um damit eine Simulation der Schatten, der Lichtreflexionen sowie der Windbewegung und der Verkehrsbelastung einer Stadt zu konfigurieren und zu kontrollieren, dargestellt in Abbildung 3.2_{63} oben rechts“[6]. Dabei sind die physikalischen Modelle der Bauwerke die greifbaren (tangible) Repräsentanten der digitalen Modelle. „Um die Position oder Orientierung eines Gebäudes zu ändern muss der Benutzer nur das physikalische Modell bewegen anstatt mit einer Maus die graphische Repräsentation am Bildschirm auszuwählen und zu verschieben. Die physikalische Form der Modellgebäude in *Urp* und die Informationen, die mit der Position und Orientierung auf der Arbeitsfläche verknüpft sind, repräsentieren und kontrollieren somit den Status der Simulation“[7].

Urp hatte allerdings das Problem, dass alle Modelle, sowohl physikalisch als auch digital, vordefiniert sein mussten. Der Benutzer konnte während der Arbeit mit *Urp* die Formen nicht ändern. Mit diesem Hintergrund entstand die zweite Generation der TUIs von Ishii., SandScape [IRP+04].

„Die Entstehung neuer Abtast- (sensing) und Anzeigetechnologien machte es möglich, die Entwicklung von dynamischen Formen in TUIs zu integrieren. Vorgeschlagen wurde die Richtung hin zu digi-

[5] Aus „The tangible user interface and its evolution“ [Isho8], Seite 34 (eigene Übersetzung)
[6] Aus „The tangible user interface and its evolution“ [Isho8], Seite 34 (eigene Übersetzung)
[7] Aus „The tangible user interface and its evolution“ [Isho8], Seite 35 (eigene Übersetzung)

talen/physikalischen Materialien, die nahtlos Fühlen (sensing) und Anzeigen miteinander verbinden. Anstatt vordefinierte diskrete Objekte mit statischen Formen zu benutzen, entwickelte die *Tangible Media Group* einen neuen Typ der organischen TUIs, der ein kontinuierliches fühlbares Material (ähnlich Ton oder Sand) verwendete. Als Beispiele wurden der *Illuminating Clay* [PRI02] und *SandScape* [IRP+04] (siehe Abbildung 3.2$_{63}$ unten) entwickelt. Mit der Entwicklung von flexiblem Materialien, die vollflexible Sensoren und Anzeigen integrieren, zeigt die Kategorie der organischen TUIs ein großes Potenzial um Ideen in fühlbarer Form auszudrücken“[8].

Allgemein versucht Ishii eine physikalische/reale Repräsentation digitaler Daten zu erzeugen, mit dem der Benutzer interagieren kann, die jedoch gleichzeitig wieder die digitalen Daten ändern. Dieselbe Vorgehensweise ist auch bei vielen Mixed Reality Anwendungen zu finden: Es wird versucht Informationen, aus der realen Welt zu extrahieren und diese dann digital zu verarbeiten. Dabei ist es wichtig, dass das Feedback zum Benutzer mit der physikalischen/realen Welt konform geht, so dass eine gewisse Verschmelzung der realen und der digitalen Welt entsteht. Ishii setzt hier mehr den Fokus auf die Benutzerschnittstellen und stellt einige Prototypen von TUIs vor, geht allerdings nicht auf die allgemeine Entwicklung ein. In dieser Arbeit steht jedoch die Entwicklung solcher Applikationen im Vordergrund und es wird versucht, eine Vorgehensweise bei der Entwicklung von MR Anwendungen zu finden.

The tangible user interface and its evolution			
Autor	Ishii	Jahr	1999
Bereich	Grundlagen, Anwendungsbeispiele		
Beschreibung	Beispiele und Methoden für TUIs		
Merkmale:	+ Konzepte, Methoden, Werkzeuge		
	+ Anwendungsbeispiele		
	+ Mixed Reality		
	- Beschränkt auf Benutzerschnittstellen		
	- Kein Entwurfsvorgehen		
	- Kein Prototyping-Ansatz		

[8] Aus „The tangible user interface and its evolution“ [Isho8], Seite 35 (eigene Übersetzung)

Weitere Entwurfskontzepte

Weitere interessante Arbeiten im Themengebiet des modellbasierter Entwurf und der grafische Programmierung sowie die Prinzipien und Leitlinien der Mensch-Maschine-Interaktion für Mixed Reality Systeme, deren konkrete Vorstellung jedoch den Rahmen dieser Arbeit überschreiten würde, finden sich u. a. bei den Präsentationen des MRUI07-Workshop, der auf der VR 2007 stattgefunden hat. [DFL06] Eine weitere sehr gute Quelle für vertiefende Informationen ist der jährlich stattfindende SEARIS Workshop. [SEA08]

3.3 Entwurfskonzepte mit Werkzeugumgebung

Im vorangegangenen Kapitel wurden reine Konzepte (teilweise mit konkreten Implementierungen) vorgestellt. In diesem Kapitel beinhalten die vorgestellten Konzepte gleichzeitig noch eine Werkzeugumgebung zur Entwicklung eigener Projekte im Bereich Mixed Reality. Teilweise bieten die Arbeiten auch automatische Generatoren, die aus den gegebenen Konzepten ausführbare Programme generieren. Ich möchte hier nur die wichtigsten Arbeiten in diesem Bereich vorstellen, die auch einen Bezug zu meiner Arbeit haben.

A model-based design process for interactive virtual environments

Ein spezielles Entwurfsvorgehen stellt Cuppens bereits 2006 in seiner Arbeit „A model-based design process for interactive virtual environments“ [CRC06] vor. Es ist ein modellbasierter Entwurfsprozess für virtuelle Umgebungen, im Speziellen die Benutzerschnittstellen innerhalb dieser virtuellen Umgebungen. Cuppens führt an, dass „die Entwicklung der Benutzerschnittstellen in diesen virtuellen Umgebungen kein unkomplizierter Prozess und somit für Nicht-Programmierer nicht einfach verständlich und anwendbar ist. Das Papier stellt einen modellbasierter Entwurfsprozess für genau diese im hohen Maße interaktiven Anwendungen vor, um die Diskrepanz zwischen Designer und Programmierer zu minimieren. Der Prozess basiert sowohl auf den Anforderungen eines modellbasierter Entwurfsprozesses für Benutzerschnittstellen und Werkzeugen und Toolkits für virtuelle Umgebungen“[9].

Bei dem Entwurfsprozess, der in Abbildung 3.3$_{67}$ visuell dargestellt

[9] Aus „A model-based design process for interactive virtual environments“ [CRC06], Seite 225 (eigene Übersetzung)

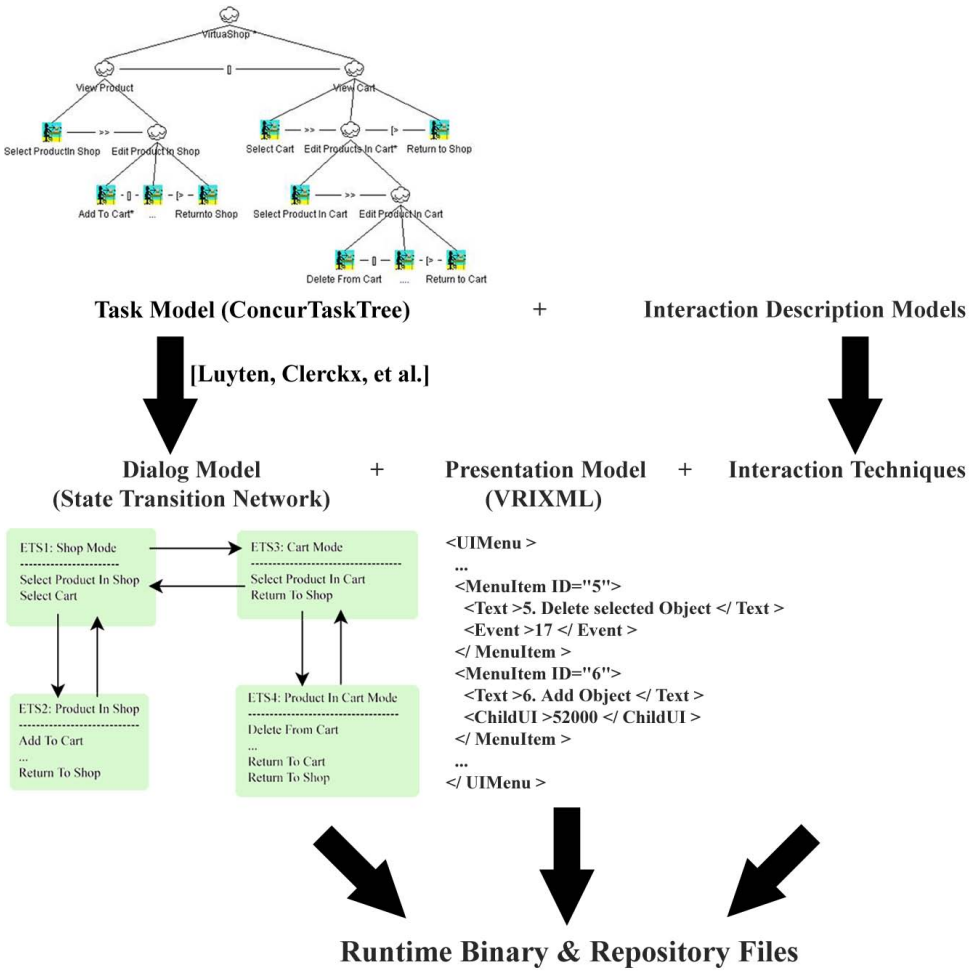


Abbildung 3.3: Entwurfsprozess von Cuppens [CRC06].

ist, wird zu Beginn ein Aufgabenmodell (*Task Modell*) mit Hilfe der *ConcurTaskTree* (CTT) Notation [Pat99] erstellt, und über das *Interaction Description Model* (IDM) erweitert. „Nachdem ein Task Model einmal entworfen wurde, kann der Algorithmus, der von Luyten et al. [LCCV03] beschrieben wurde, dazu benutzt werden, um das Dialog Model automatisch von der CTT zu extrahieren. Das Dialog Model basiert auf den Enabled Task Sets (ETS) [Pat99], die vom Task Modell der Applikation abgeleitet werden. [···] Die resultierenden ETSs können auf verschiedene Zustände der Anwendung abgebildet werden, die alle Interaktionsaufgaben des jeweiligen Zustands beinhalten. [···] Nachdem das Dialog Model extrahiert wurde, können die unterschiedlichen Zustände der Anwendung mit den Interaktionstechniken verbunden werden, die von den IDMs definiert wurden. Des Weiteren muss der Entwickler nun das Dialog Model mit dem Presentation Model zusammenfügen. [···] Nach Beendigung des gesamten Entwurfsprozesses werden die spezifizierten Modelle für eine automatisierte Generierung einer lauffähigen Version der virtuellen

Umgebungs-Anwendung verwendet. "[10]

Das von Cuppens vorgestellte Entwurfsvorgehen basiert auf einem linearen Ansatz und sieht keine Iterationen und dementsprechend keine Entwicklung mehrerer Prototpyen vor. Weiterhin bezieht es den Aspekt der Mixed Reality nicht mit ein, der Prozess ist ausschließlich für reine virtuelle Umgebungen entworfen worden.

A model-based design process for interactive virtual environments	
Autoren	Cuppens, Raymaekers, Coninx Jahr 2006
Bereich	Entwurfsvorgehen
Beschreibung	Spezielles Entwurfsvorgehen in VR
Merkmale:	+ Konzepte, Methoden, Werkzeuge
	+ Anwendungsbeispiele
	+ Automatische Generierung
	- Beschränkt auf Benutzerschnittstellen
	- Nur VR
	- Nicht iterativ

Mixed Reality: A model of Mixed Interaction

Eine weitere Arbeit im Bereich Entwurf, die sich speziell auf die Interaktion in Mixed Reality bezieht, ist „Mixed Reality: A model of Mixed Interaction" von Coutrix und Nigay aus dem Jahre 2006 [CN06]. „Bei Mixed Reality Systemen wird versucht, die physikalische und die elektronische (digitale) Umgebung nahtlos zu verknüpfen. Obschon Mixed Reality Systeme sich stetig weiter verbreiten, existiert noch kein klares Verständnis über das Interaktionsparadigma. Um dieses Problem zu zu lösen, wird in dem Artikel ein neues Interaktionsmodell [BL04] mit dem Namen *Mixed Interaction Model* vorgestellt"[11]. Dabei ist es „das Ziel des Interaktionsmodells dem Designer ein Framework anzubieten, die ihn durch der Erschaffung von interaktiven Systemen leiten"[12]. Ein Interaktionsmodell kann dabei entlang der folgenden drei Dimensionen charakterisiert werden [BL04]:

Darstellung/Klassifizierung: Entspricht dem Potential, eine aussagekräftige Auswahl existierender Schnittstellen zu beschreiben

[10] Aus „A model-based design process for interactive virtual environments" [CRC06], Seite 231 (eigene Übersetzung)

[11] Aus „Mixed Reality: A model of Mixed Interaction" [CN06], Seite 43 (eigene Übersetzung)

[12] Aus „Mixed Reality: A model of Mixed Interaction" [CN06], Seite 43 (eigene Übersetzung)

und zu klassifizieren.

Erzeugung: Entspricht dem Potential, dem Designer dabei zu helfen, neue Designs zu entwerfen.

Komparativität: Entspricht dem Potential, mehrere Designalternativen zu beurteilen.

„Das *Mixed Interaction Model* setzt den Schwerpunkt auf die Verknüpfung der physikalischen und digitalen Welt und die Interaktion des Benutzers mit der dadurch entstandenen Mixed Reality Umgebung. [· · ·] Das Hauptkonzept des *Mixed Interaction Model* ist dabei das *mixed object*“[13].

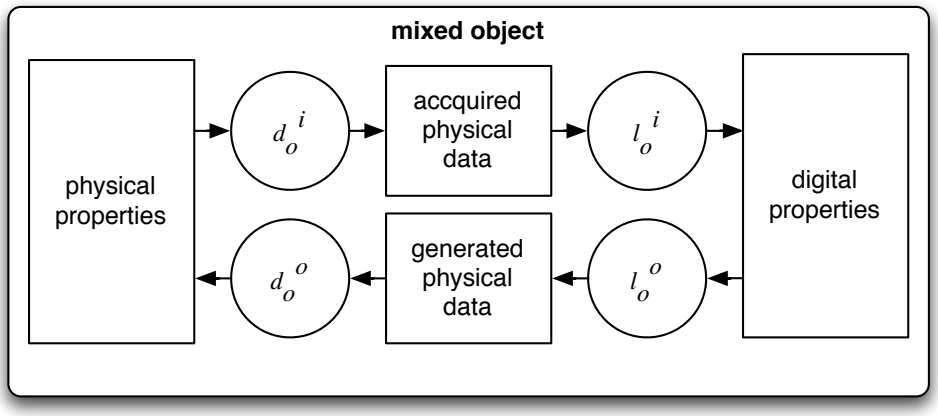


Abbildung 3.4: Mixed Object [CNo6]

„Ein reales Objekt besteht aus einer Menge verschiedener physikalischer Eigenschaften. Gleiches gilt für ein digitales Objekt, welches aus einer Menge verschiedener digitaler Eigenschaften besteht. Ein *mixed object* ist nun eine Kombination beider Mengen: Eine Menge physikalischer Eigenschaften, die mit einer Menge digitaler Eigenschaften verknüpft sind. Um die Verknüpfung der beiden Mengen zu beschreiben, werden zwei Arten (d, l) berücksichtigt. Einerseits die Verknüpfungen zwischen physikalischen und digitalen Eigenschaften eines Objektes, die *linking modalities* genannt werden, andererseits die Verknüpfungen zur Interaktion des Benutzers mit der Mixed Reality Umgebung, die *interaction modalities* heißen. Aus Sicht des Systems können also zwei Arten von Verknüpfung für ein *mixed object* identifiziert werden, wie in Abbildung 3.4 zu sehen ist“[14]:

- Die Eingabeverknüfungen $(d_o{}^i, l_o{}^i)$ sind zuständig für:

[13] Aus „Mixed Reality: A model of Mixed Interaction“ [CNo6], Seite 43 f. (eigene Übersetzung)
[14] Aus „Mixed Reality: A model of Mixed Interaction“ [CNo6], Seite 44 (eigene Übersetzung)

1. Beschaffung einer Untermenge von physikalischen Eigenschaften mit Hilfe des Devices ${d_o}^i$ (object input device).
2. Interpretieren der empfangenen physikalischen Daten bezüglich der digitalen Eigenschaften mit Hilfe der Sprache ${l_o}^i$ (object input language).

- Die Ausgabeverknüfungen $({d_o}^o, {l_o}^o)$ sind zuständig für:

1. Generierung von Daten auf Basis der Menge der digitalen Eigenschaften mit Hilfe der Sprache ${l_o}^o$ (object output language),
2. Übersetzten der generierten physikalischen Daten in erkennbare physikalische Eigenschaften mit Hilfe des Devices ${d_o}^o$ (object output device).

„Ein *mixed object* kann nun (1) auf einer Eingabeverknüfung, (2) auf einer Ausgabeverknüfung oder (3) auf einer Ein- und Ausgabeverknüfung basieren“[15].

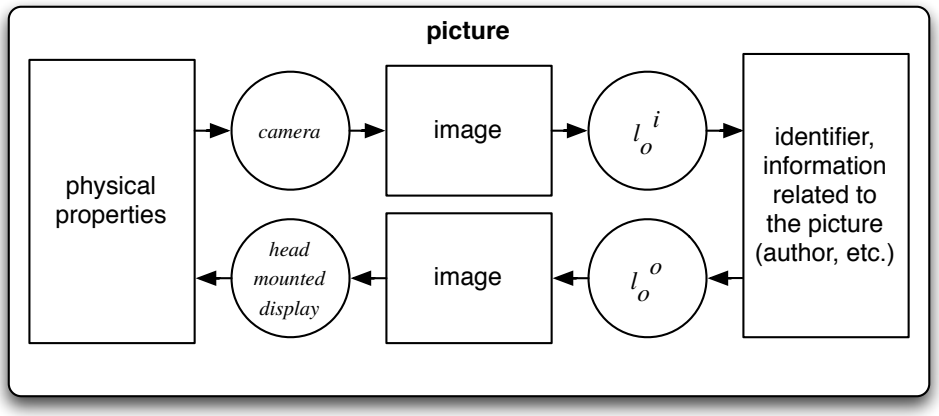


Abbildung 3.5: Ein Bild in NaviCam [RN95]

„In Abbildung 3.5 wir das Beispiel des NaviCam Systems von Rekimoto [RN95] betrachtet und ein *augmented picture* als *mixed object* modelliert. Dabei zeichnet die Kamera die physikalischen Eigenschaften dieses Objektes auf. Das Foto wird dann in ein Identifizierungsmerkmal des erkannten Bildes übersetzt. Die zu diesem Bild zugehörigen Informationen werden nachfolgend auf einem Head-Mounted Display (HMD) angezeigt. Die Verknüpfungen bei diesem Beispiel sind elementar, allerdings können die Ein- und Ausgabeverknüfung auch zusammengesetzt sein. Die Zusammensetzung der Ein- und Ausgabeverknüfungen wurde basierend auf dem CARE (Complementarity,

[15] Aus „Mixed Reality: A model of Mixed Interaction“ [CN06], Seite 44 (eigene Übersetzung)

Assignment, Redundancy and Equivalence) Framework charakterisiert [NC97] [VN00]“[16].

„Zusammenfassend kann also ein *mixed object* anhand seiner Ein- und Ausgabeverknüfung charakterisiert werden, wobei die Verknüpfungen ihrerseits entweder nicht vorhanden, elementar oder zusammengesetzt sind“[17].

„Eine *Mixed Interaction* bedingt ein *mixed object*. [· · ·] Um eine *Mixed Interaction* zu modellieren wurde das *Instrumental Interaction model* [BL04] durch die Definition des *mixed object* und die Definition der Art der Interaktion, die als eine Kopplung eines Devices d mit einer Sprache l beschreiben ist, erweitert“[18].

In der Arbeit wurde mit Hilfe der *mixed object* versucht, die reale bzw. physikalische Umgebung zu erfassen und sofwaretechnisch abzubilden. Die *mixed object* sind eine Möglichkeit, die realen Eigenschaften der Umgebung zu kapseln bzw. digitale Ereignisse der Umgebung zur Verfügung zu stellen. In meiner Arbeit gehe ich allerdings einen anderen Weg, da ich die Umgebung als gegeben und unveränderlich (bzgl. der programmierten Anwendung) sehe. Durch eine Erweiterung des MVC Architekturmusters kann dieses erreicht werden.

Mixed Reality: A model of Mixed Interaction			
Autoren	Coutrix, Nigay	Jahr	2006
Bereich	Entwurfsvorgehen		
Beschreibung	Interaktion in Mixed Reality		
Merkmale:	+ Modell, Methoden, Werkzeuge		
	+ Anwendungsbeispiele		
	+ Mixed Reality		
	- Beschränkt auf Interaktionen		
	- Nicht iterativ		

High-level modeling of multimodal interaction techniques using NiMMiT

Ein weiteres Projekt im Bereich Entwurf von VR/MR Interaktionstechniken ist NiMMiT[19]. In der Arbeit von von De Boeck et al. mit dem

[16] Aus „Mixed Reality: A model of Mixed Interaction“ [CN06], Seite 44 (eigene Übersetzung)
[17] Aus „Mixed Reality: A model of Mixed Interaction“ [CN06], Seite 44 (eigene Übersetzung)
[18] Aus „Mixed Reality: A model of Mixed Interaction“ [CN06], Seite 44 (eigene Übersetzung)
[19] NiMMiT steht für **N**otat**i**on for **M**odeling **M**ultimodal **i**nteraction **T**echniques.

Titel „High-level modeling of multimodal interaction techniques using NiMMiT“ [DBVRC07] wird eine grafische Notation für multimodale VR Interaktionstechniken vorgestellt, die auf der Statechart-Notation von David Harel [Har87] basiert.

„NiMMiT erlaubt es dem Designer multimodaler Interaktionstechniken schnell Alternativen zu testen oder sehr einfach existierende Lösungen zu adaptieren je nach Evaluation von Benutzertests, was den Entwicklungszyklus signifikant verkürzt. Die automatische Ausführung der Interaktionstechniken wird von NiMMiT unterstützt indem die Diagramm-Repräsentation interpretiert wird. Des Weiteren wird durch die High-Level Beschreibung die Wiederverwendung einzelner Lösungen vereinfacht“[20].

Für De Boeck sollte eine Notation, die eine Interaktionstechnik beschreiben will, folgenden Anforderungen entsprechen [DBVRC07]:

- Ereignisgetrieben (event driven)
- Zustandsgetrieben (state driven)
- Datengetrieben (data driven)
- Unterstützung einer Kapselung für die hierarchische Wiederverwendbarkeit

Die Notation von NiMMiT basiert auf allen oben angeführten Anforderungen, so dass sich eine Interaktionstechnik folgendermaßen beschreiben lässt:

„NiMMiTs Notation ist sowohl ereignis- als auch zustandsgetrieben, so dass ein Diagramm grundsätzlich der Gestalt von Statecharts entspricht. Eine Interaktionstechnik wird immer mit einem Startzustand initialisiert. Ein Zustand reagiert auf eine beschränkte Menge von Ereignissen (Events), beispielsweise Spracheingabe, Zeigerbewegung oder einen Click auf einen Button. Wird ein Event erkannt wird eine *Task chain* ausgeführt, zu sehen in Abbildung 3.6_{73} a). [···] Die Ausführung einer *Task chain* ist strikt linear, was bedeutet, dass der nächste Task einer *Task chain* dann und nur dann ausgeführt wird, wenn der vorherige Task erfolgreich beendet wurde. Abbildung 3.6_{73} b) zeigt eine *Task chain* mit zwei Tasks. [···] Ein Ausgangsport eines vorangegangenen Tasks ist typischerweise mit dem Eingangsport des nachfolgenden Task verbunden. Diese Eingangsports können sowohl optional als auch obligatorisch sein. Sollte ein obligatorischer

[20] Aus „High-level modeling of multimodal interaction techniques using NiMMiT“ [DBVRC07], Seite 2 (eigene Übersetzung)

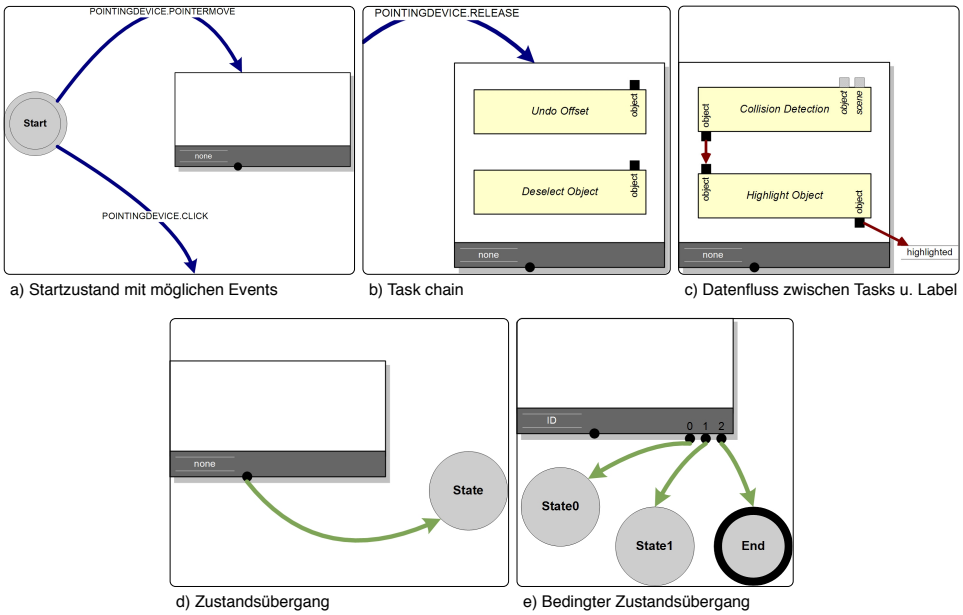


Abbildung 3.6: Grafische Repräsentation von NiMMiT [DBVRC07].

Eingangsport eines Tasks einen nicht zulässigen Wert erhalten, wird die Ausführung der kompletten *Task chain* abgebrochen. Um Daten zwischen Tasks unterschiedlicher *Task chains* miteinander nutzen oder um Daten für spätere Verwendung speichern zu können, wurden High-Level Variablen in Form von *Labels* eingeführt, wie in Abbildung 3.6$_{73}$ c) gezeigt wird. Nach der erfolgreichen Ausführung einer *Task chain* findet ein Zustandsübergang statt, zu sehen in 3.6$_{73}$ d). Hier kann nun in einen neuen Zustand gewechselt oder wieder zurück in denselben Zustand gesprungen werden (in einer Schleife). In einem neuen Zustand könnte die beschriebene Interaktionstechnik nun auf eine andere Menge von Ereignissen reagieren. Einer *Task chain* können mehrere verschiedene Zustandsübergänge assoziiert werden: Der Wert des *Label* einer *Task chain* legt fest welcher Zustandsübergang ausgeführt wird. Abbildung 3.6$_{73}$ e) zeigt eine *Task chain* mit dem *Label* 'ID' und drei mögliche Zustandsübergänge"[21].

Mit Hilfe der grafischen Notation kann der Designer schnell Interaktionstechniken realisieren. Damit diese allerdings auch getestet werden können, bedarf es einer schnellen Umsetzung der Diagramme in ausführbaren Programmcode. Der NiMMiT Editor bietet diese Funktionalität an. Da NiMMiT für die Erstellung virtueller Umgebung entwickelt wurde, ist es mit dem Editor möglich, die Interaktionstechniken über ein XML Austauschformat zu exportieren und dieses in die für NiMMiT entwickelte virtuelle Umgebung zu laden. Diese führt

[21] Aus „High-level modeling of multimodal interaction techniques using NiMMiT" [DBVRC07], Seite 3 f. (eigene Übersetzung)

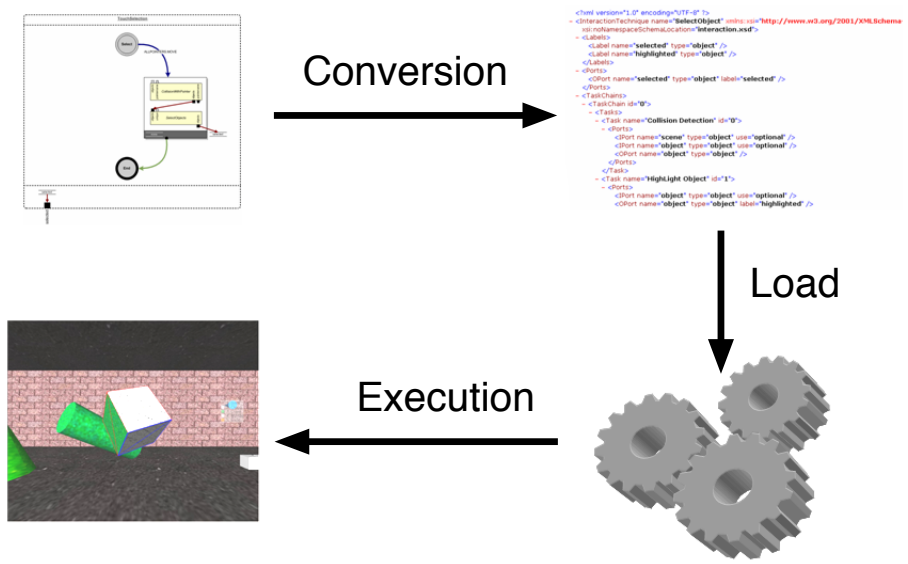


Abbildung 3.7: Die NiMMiT Toolchain [DBVRC07].

dann die Interaktionstechniken aus. Der Ablauf wird in Abbildung 3.7[74] noch einmal verdeutlicht. Nachdem die Interaktionstechnik im Editor entwickelt wurde, wird sie in XML umgewandelt und in die virtuelle Umgebung geladen, die sie dann ausführt.

NiMMiT ist für die Entwicklung von Interaktionstechniken gut geeignet: Gerade mit der grafischen Repräsentation lassen sich schnell neue Techniken entwickeln. Jedoch basiert NiMMiT auf einer rein virtuellen Umgebung, die statisch definiert und somit schwer erweiterbar ist. Da sich mit NiMMiT nur Interaktionstechniken schnell evaluieren lassen, ist es für einen kompletten Entwurfsprozess nicht geeignet.

High-level modeling of multim. interaction techniques			
Autoren	De Boeck, Coninx et al.	Jahr	2007
Bereich	Entwurfsvorgehen		
Beschreibung	Multimodale Interaktionstechniken in VR		
Merkmale:	+ Konzept, Modell, Werkzeuge		
	+ Anwendungsbeispiele		
	+ Automatische Generierung		
	+ Grafische Repräsentation		
	- Beschränkt auf Interaktionstechniken		
	- Nur VR		
	- Nicht iterativ		

A Design-Oriented Information-Flow Refinement of the ASUR Interaction Model

Eine weiteres Modell einschließlich grafischer Notation wurde in der Veröffentlichung „A Design-Oriented Information-Flow Refinement of the ASUR Interaction Model“ von Emmanuel Dubois und Philip Gray aus dem Jahre 2008 vorgestellt [DG08]. Basierend auf den beiden Arbeiten „ASUR++: A Design Notation for Mobile Mixed Systems“ [DGN02] aus dem Jahre 2002 und „Requirements and Impacts of Model driven engineering on Mixed Systems Design“ [DCD05] aus dem Jahre 2005 beschreibt die Arbeit ein Modell und eine Modellierungstechnik zum Erfassen der Schwerpunkte der Benutzerinteraktion während der Anforderungsanalyse in einer frühen Phase der Entwicklung von Mixed Reality Systemen.

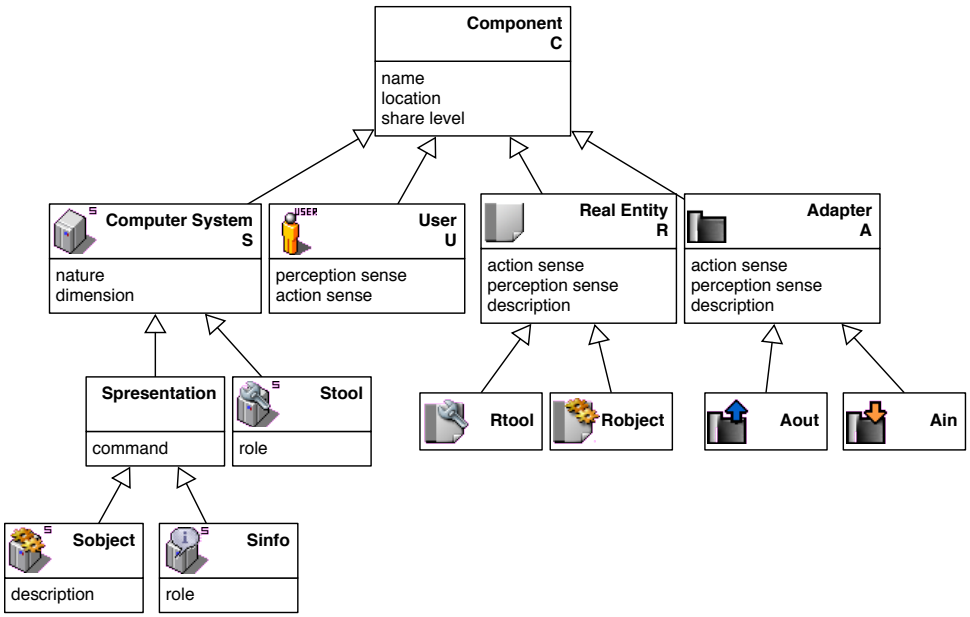


Abbildung 3.8: Komponenten bei *ASUR* [DCD05].

„*ASUR* ist ein auf grafischer Darstellung basiertes Modell zur Beschreibung von Benutzer-System Interaktionen in Mixed Reality Systemen. *ASUR* soll bei der Beurteilung helfen, physikalische und digitale Welten so zu verbinden, dass benutzerfreundliche Resultate erzielt werden. Zusätzlich wir es in Verbindung mit der traditionellen Benutzer-System Aufgabenbeschreibung verwendet, um Objekte zu identifizieren, die bei der Interaktion beteiligt sind und zwischen den Grenzen der beiden Welten liegen. Aus Sicht der Benutzerinteraktion hilft das Modell die Resultate der Anforderungsanalyse zu beschreiben und die globale Entwurfsphase von Mixed Reality Systemen zu unterstützen. AUSR unterstützt die Beschreibung von digitalen und

physikalischen Entitäten, die ein Mixed Reality System ausmachen, u. a. Adapter (**Ain**, **Aout**), die die Kluft zwischen der digitalen und physikalischen Welt überbrücken, digitale Werkzeuge (**Stool**) bzw. Konzepte (**Sinfo**, **Sobj**), ein oder mehrere Benutzer (**U**) und reale Objekte, die als Werkzeuge (**Rtool**) involviert sind oder den Fokus der Aufgabe darstellen (**Robj**). Weiterhin drücken gerichtete Beziehungen (Linien mit Pfeilen) den physikalischen und/oder digitalen Fluss von Informationen und die Verbindung zwischen Komponenten aus. Um diese Elemente besser spezifizieren zu können, d. h. AUSR Komponenten und Beziehungen, wurden eine Anzahl an Charakteristika identifiziert“[22]. Die unterschiedlichen Komponenten incl. ihrer Bezeichnungen, die in *ASUR* existieren, sind in Abbildung 3.8 zu sehen.

Die *ASUR* Komponenten (**Compontent**) können somit einem der folgenden vier Typen angehören [DCD05]:

Computer System (S): Die Komponente repräsentiert das Computersystem und alle digitalen Entitäten, die an einer Interaktion beteiligt sind. Ähnlich den realen Objekten kann das Computersystem auch in zwei Teile aufgeteilt werden. Zum einen in die digitalen Entitäten, die entweder das Verhalten oder das Erscheinungsbild anderer digitaler Entitäten verändern (**Stool**), zum anderen die digitalen Entitäten, die ein Ziel einer Interaktion und somit Ziel einer Veränderung darstellen. Dabei wird zwischen Objekten, die nur Feedback-Informationen liefen (**Sinfo**) und anderen Objekten (**Sobject**) unterschieden.

User (U): Der Benutzer repräsentiert den Anwender des Systems. *ASUR* unterstützt sowohl Einzel- als auch Mehrbenutzeranwendungen.

Real Entity (R): Die realen Entitäten repräsentieren reale Objekte. Das können sowohl physikalische Werkzeuge (**Rtool**) sein, die vom Benutzer verwendet werden können, um eine Funktion auszuüben, oder physikalische Objekte (**Robject**), auf die sich eine Funktion bezieht.

Adapter (A): Adapter schlagen eine Brücke zwischen der digitalen und der physikalischen Welt. Dabei unterscheidet *ASUR* zwischen Eingabeadapter (**Ain**), die Daten von der physikalischen Welt in die digitale Welt übertragen (z. B. eine Kamera)

[22] Aus „A Design-Oriented Information-Flow Refinement of the ASUR Interaction Model“ [DG08], Seite 467 (eigene Übersetzung)

und Ausgabeadaptern (**Aout**), die numerische Daten an die physikalische Welt liefern (z. B. ein Bildschirm).

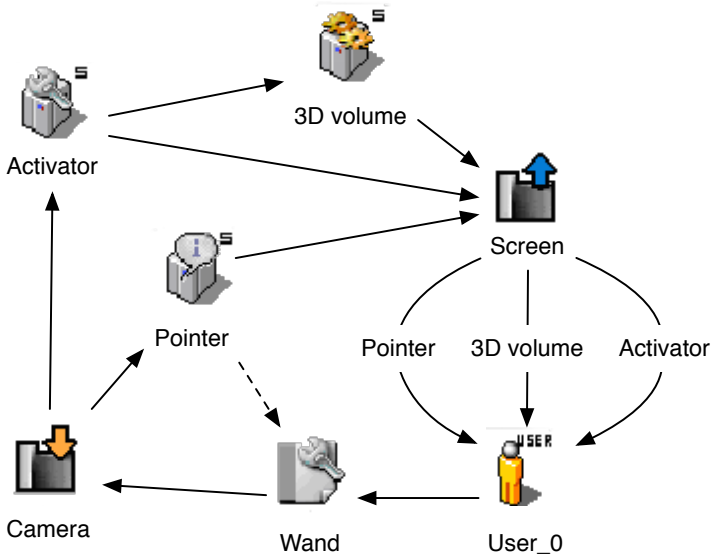


Abbildung 3.9: Beispiel eines *ASUR* Diagramms [DG08].

In Abbildung 3.9 „wird die Interaktion eines Benutzers und einer digitalen 3D Umgebung mit Hilfe eines „magischen Zauberstabes“ gezeigt. Der Benutzer *User_0* verwendet und bewegt einen physikalischen „Zauberstab“ *Wand*, der mit Hilfe einer Kamera *Camera* (**Ain**) verfolgt wird. Die Kamera sendet die Position des Zauberstabes an einen digitalen *Aktivator* (**Stool**), der vielleicht noch auf andere digitale Entitäten wirkt. Die Kamera sendet des Weiteren die Position an das Zeigerobjekt *Pointer* (**Sinfo**). Das Zeigerobjekt ist eigentlich eine Repräsentation der Spitze des physikalischen Zauberstabes (angedeutet durch den gestrichelten Pfeil); diese Repräsentation ist gerade für die Bereitstellung eines Interaktion-Feedbacks zweckmäßig. Sobald die Funktionalität aktiviert ist, werden Daten wie beispielsweise der Rotationswinkel des Zauberstabes an den 3D-Raum *3D volume* (**Sobj**) transferiert. Letztendlich wird der 3D Raum, der Aktivator und das Zeigerobjekt auf dem Bildschirm *Screen* (**Aout**) dargestellt. Eine detailliertere Beschreibung dieses Beispiels mit allen modellierten Charakteristika ist in [DCD05] zu finden“[23].

„*ASUR* ist eine reine Modell-Notation: während *ASUR* eine hervorragende Orientierungshilfe für die Entwickler beschreibt, ist das entwickelte Modell nicht ausführbar und muss per Hand in den entspre-

[23] Aus „A Design-Oriented Information-Flow Refinement of the ASUR Interaction Model“ [DG08], Seite 467 (eigene Übersetzung)

chenden Quelltext konvertiert werden. *ASUR* bietet keine explizite Unterstützung von Kollaboration“[24].

Mit *ASUR* ist es sehr einfach möglich, eine Mixed Reality Applikation auf Basis eines Modells zu entwickeln. Mit Hilfe der vorhandenen Komponenten, die *ASUR* zur Verfügung stellt, ist eine Abdeckung der digitalen und physikalischen Objekte gegeben. *ASUR* unterstützt allerdings nicht eine Entwicklung entlang des Mixed Reality Kontinuums. Objekte, die entweder in der digitalen oder physikalischen Welt verankert wurden, verbleiben während der gesamten Entwicklung darin. Will man die Objekte ändern, so bedarf es einer kompletten Umstrukturierung der beteiligten Objekte. *ASUR* bietet weiterhin nicht die Möglichkeit, aus dem entwickelten Modell eine lauffähige Applikation zu generieren, wie es beispielsweise bei NiMMiT (siehe Abschnitt „High-level modeling of multimodal interaction techniques using NiMMiT“ auf Seite 71) möglich ist.

Information-Flow Refinement of the ASUR Interaction Model		
Autoren	Dubois, Gray	Jahr 2008
Bereich	Modell und Notation	
Beschreibung	Modellierung von Interaktionstechniken in MR	
Merkmale:	+ Konzept, Modell, Werkzeuge	
	+ Anwendungsbeispiele	
	+ Grafische Notation	
	+ Mixed Reality	
	- Keine Entwicklung entlang d. MR Kontinuums	
	- Keine Automatische Generierung	
	- Nicht iterativ	

The Engineering of Mixed Reality Systems

In dem aus dem Jahre 2010 stammenden Buch „The Engineering of Mixed Reality Systems“ [DGN10], herausgegeben von Emmanuel Dubois et al., wird speziell auf die Entwicklung von Mixed Reality Systemen eingegangen. Es ist eine Zusammenfassung vieler Artikel bekannter Wissenschaftler aus den drei großen Bereichen Interactiondesign, Software Design und Implementierung sowie Anwendungen von Mixed Reality. „Human Computer Interaction (HCI – zu Deutsch: Mensch-Maschine-Interaktion) ist nicht mehr auf die Interaktion des Benutzers mit dem Computer über die Tastatur und Bildschirm beschränkt: zur

[24] Aus „The Engineering of Mixed Reality Systems“ [DGN10], Seite 298 (eigene Übersetzung)

Zeit ist es eine der herausfordernden Aufgaben von interaktiven Systemen, die Integration von physikalischen und digitalen Aspekten auf benutzbare Konzepte zu verknüpfen. Die Herausforderung bei der Entwicklung solcher Mixed Reality (MR) Systeme liegt in der fließenden und harmonischen Fusion der physikalischen und der digitalen Welten. Beispiele solcher Systeme beinhalten Tangible User Interfaces (TUIs – zu deutsch: Greifbare Benutzerschittstellen), Augmented Reality, Augmented Virtuality und Embodied Interfaces[25]. Alleine die Vielfältigkeit der Begriffe in diesem Bereich hebt das wachsende Interesse an MR Systemen hervor und die hieraus resultierende dynamische und herausfordernde Domäne“[26].

Im Allgemeinen sind die Arbeiten aus dem ersten Teil (Interactiondesign) und dem zweiten Teil (Software Design und Implementierung) des Buches sind hervorzuheben. Insbesondere sind „An Integrating Framework for Mixed Systems“ von Céline Coutrix und Laurence Nigay und „Fiia: A Model-Based Approach to Engineering Collaborative Augmented Reality“ von Christopher Wolfe et al. hier erwähnenswert uns sollen im Folgenden näher vorgestellt werden.

Im Artikel „An Integrating Framework for Mixed Systems“ wird erläutert, dass „in dem sehr dynamischen Mixed Reality Bereich ein Vergleich der vorhandenen Mixed (Reality) Systems und die daraus folgende Designspace Exploration sehr schwer realisierbar sind. Um dieses Entwurfsproblem zu lösen, stellt der Artikel eine einheitliche Betrachtungsweise auf *Mixed Systems* vor, indem auf sogenannte *mixed objects* [27], die bei der Interaktion involviert sind, der Fokus gelegt wird. Der vorgestellte integrierte Framework besteht aus zwei sich gegenseitig ergänzend Aspekten der *mixed objects*: Es definiert sowohl die inhärenten als auch die extrinsischen Charakteristika eines Objekts unter Berücksichtigung der seiner Rolle bei der Interaktion. Solche Charakteristika eines Objekts sind für den feingranularen Vergleich existierender Mixed Systems nützlich. Dabei wird die taxonomische Mächtigkeit dieser Charakteristika an aus der Literatur bekannten Mixed Systems diskutiert. Die generative Mächtigkeit wird anhand eines von den Autoren entwickelten Systems namens *Roam* erklärt“[28].

In Abbildung 3.10_{80} ist das Schema der Charakteristika von *mixed*

[25] Schnittstellen, die durch Objekte der realen Welt definiert sind. „Embodiment“ meint nicht nur die physische Verkörperung von Objekten, sondern bezieht auch andere Aspekte der realen Welt wie Sprache und soziale Faktoren mit ein. Häufig wird auch der Oberbegriff „Ubiquitous Computing“ verwendet.

[26] Aus „The Engineering of Mixed Reality Systems“ [DGN10], Seite 1 (eigene Übersetzung)

[27] *mixed object* sind hybride physikalisch-digitale Objekte, die die physikalische und digitale Welt überspannen. Sie wurden schon im Kapitel 3_{68}:„Mixed Reality: A model of Mixed Interaction“ [CN06] vorgestellt.

[28] Aus „The Engineering of Mixed Reality Systems“ [DGN10], Seite 9 (eigene Übersetzung)

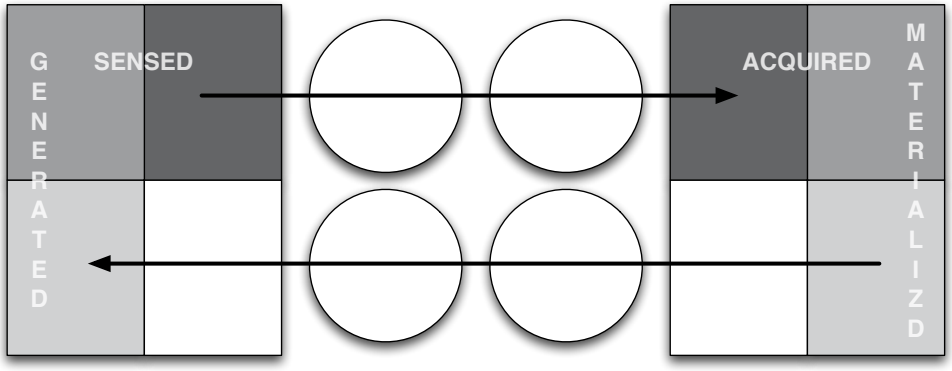


Abbildung 3.10: Charakterisierung von *mixed objects* [DGN10].

objects zu sehen. „Die physikalischen Eigenschaften werden über zwei orthogonale (sensed/generated)-Achsen definiert, korrespondierend zu den Eingabe-Ausgabe Achsen aus Bricks [FIB95]. [···] Unter Berücksichtigung der Besonderheit, dass digitale Eigenschaften symmetrisch zu den physikalischen Eigenschaften sein sollen, können digitale Eigenschaften über zwei orthogonale (acquired/materialized)-Achsen definiert werden. Eine digitale Eigenschaft kann über jede Art von Verbindung erlangt (engl. acquired) und/oder materialisiert werden. Diese Menge an Charakteristika ist unabhängig vom Typ der Verbindung.“[29].

Als Beispiel für die Verwendung von *mixed objects* wird der Digitale Tisch genutzt, zu sehen in Abbildung 3.11_{81}. „Der Benutzer verwendet und bewegt den Radiergummi – das *mixed tool*. Diese Aktion, basierend auf den physikalischen Eigenschaften des Objekts, wird von der Eingabeverbindung (hier: eine Kamera und ein Computer-Vision Algorithmus) erkannt, um dann die entsprechenden digitalen Eigenschaften <location> und <recognized movements> zu aktualisieren. Die Änderungen in den digitalen Eigenschaften des *mixed tools* werden danach von der *interaction language* zu einer elementaren Aufgabe (*elementary task*) interpretiert: Die (x, y)-Position wird in die elementare Aufgabe “Lösche Zeichnung an Position (x, y) auf dem Tisch“ überführt. Die elementare Aufgabe wird dann auf das Task Objekt angewendet und die digitalen Eigenschaften der *mixed drawing* werden infolgedessen modifiziert. Die *mixed drawings* zeigt ihre interne digitale Veränderung über die Aktualisierung der Anzeige ihrer Ausgabeverbindung – dem Feedback für den Benutzer“[30].

Zusammenfassend kann über die Arbeit von Céline Coutrix und Laurence Nigay gesagt werden, dass sie mit diesem Artikel eine neue

[29] Aus „The Engineering of Mixed Reality Systems“ [DGN10], Seite 16 ff. (eigene Übersetzung)
[30] Aus „The Engineering of Mixed Reality Systems“ [DGN10], Seite 20 f. (eigene Übersetzung)

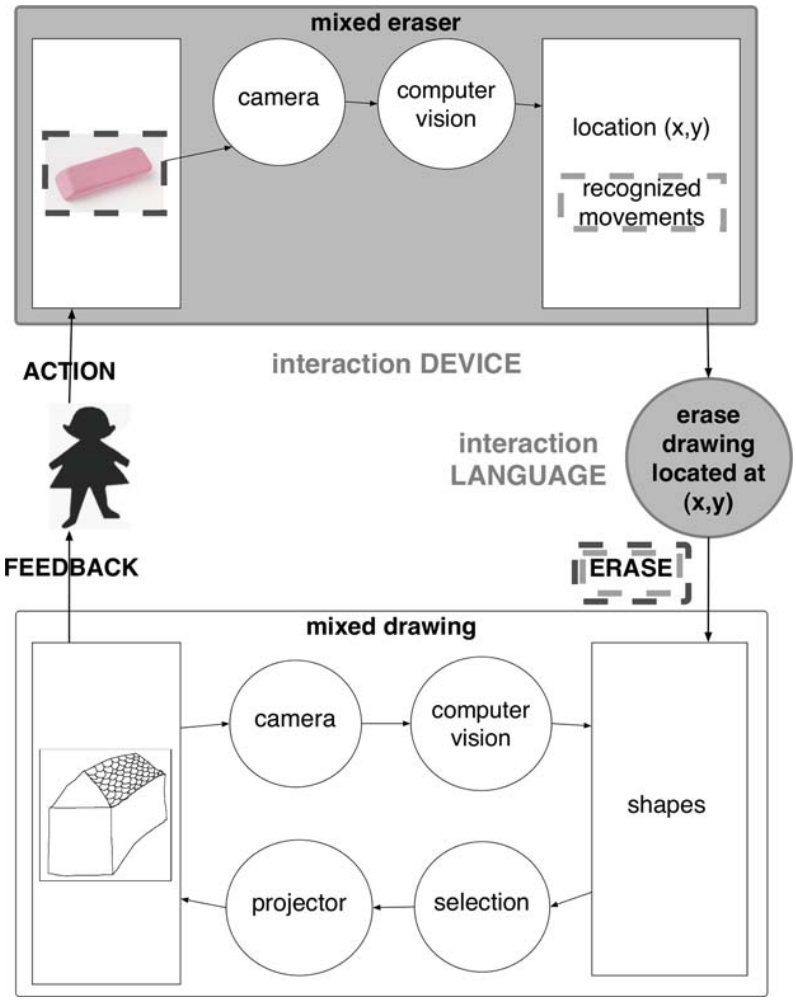


Abbildung 3.11: Interaktion: Benutzer und *mixed objects* [DGN10].

Sichtweise auf die Entwicklung von Interaktionen in Mixed Reality Systemen mit Hilfe der *mixed objects* ermöglicht haben. „Sie haben die inhärenten als auch die extrinsischen Charakteristika der *mixed objects* vorgestellt, bei dem das Objekt entweder als Werkzeug oder aber eine Aufgabe gesehen werden kann. Indem ähnliche existierende Systeme mit Hilfe der vorgestellten Charakteristika klassifiziert werden konnten, wurde damit ihre taxonomische Mächtigkeit demonstriert“[31].

Als weitere hervorzuhebende Arbeit in diesem Buch ist der Artikel „Fiia: A Model-Based Approach to Engineering Collaborative Augmented Reality“ von Christopher Wolfe et al. zu nennen. Er beschreibt die visuelle Notation *Fiia* zu Formulierung der Entwicklung von kollaborativen AR Anwendungen. Diese visuelle Notation basiert auf der „Actor & Adaptor“-Metapher, die ich für meine Arbeit auch verwendet habe.

[31] Aus „The Engineering of Mixed Reality Systems“ [DGN10], Seite 29 (eigene Übersetzung)

„AR Anwendungen benötigen häufig eine Kollaboration von Personen oder Personengruppen. Obwohl eine Reihe an Werkzeugen existieren, die die Entwicklung solcher AR Systeme unterstützen (z. B. das AR-ToolKit oder das Groupkit), bleibt jedoch eine große Kluft zwischen der Spezifikation und der Implementierung dieser Systeme “[32]. Diese Kluft versucht der Autor mit Hilfe von *Fiia* zu schließen.

„*Fiia.Net* ist ein Werkzeug (basierend auf der visuellen Notation *Fiia*), welches die Entwicklung von kollaborativen AR Applikationen vereinfachen soll. Mit Hilfe der *Fiia modeling language* kann der Entwickler die Struktur seiner Anwendung spezifizieren, um so die Details der Vernetzung zu abstrahieren und die Spezifikation der Adapter zwischen der physikalischen und realen Welt auf einer hohen Ebene zu definieren. Das *Fiia.Net* Laufzeitsystem bildet dieses konzeptionelle Modell auf eine Laufzeitumgebung ab“[33].

„Die *Fiia* Entwurfsnotation ist ein Architekturmuster oder besser eine visuelle Notation, mit der kollaborative AR Applikationen softwaretechnisch entwickelt werden können. Viele andere Architekturmuster haben das Ziel, die Schwierigkeiten bei der Programmierung entweder der Groupware (also kollaborativen Anwendungen) oder der AR Anwendungen zu minimieren. Es ist zur Zeit kein Architekturmuster bekannt, das beide Gebiete unterstützt“[34].

„Architekturmuster legen Regeln fest, die es Entwicklern ermöglichen, die Groupware-Systeme in ihre Komponenten aufzuteilen, so dass der „Teile-und-Hersche“ Ansatz (engl.: *devide and conquer*) der Softwareentwicklung angewendet werden kann. Beispiele für dieses Vorgehen sind das *Clover-Modell* [LN02] und *PAC** [CCN97], die dem Entwickler Ratschläge geben, wie die Benutzerschnittstelle bzw. die Applikation unter Berücksichtigung der Gruppenaufgaben *Produktion*, *Kommunikation* und *Koordination* am geeignetsten aufgeteilt werden kann. Beide Architekturen sind rein konzeptionell, was bedeutet, dass sie sich nicht mit den Problemen befassen, wie die Vorschläge auf einem verteilten System implementiert werden sollen. Die Entwickler stehen also der schwierigen Aufgabe gegenüber, die Vorschläge in ausführbaren Code zu transformieren. Phillips liefert eine detaillierte Zusammenfassung der konzeptionellen Architekturmuster für Groupware-Anwendungen [Phi99]“[35].

„Aus den Architekturmustern für die Entwicklung von AR Anwendungen ist *ASUR* [DG08] erwähnenswert. Ähnlich wie bei *Fiia* erlaubt

[32] Aus „The Engineering of Mixed Reality Systems“ [DGN10], Seite 293 (eigene Übersetzung)
[33] Aus „The Engineering of Mixed Reality Systems“ [DGN10], Seite 293 (eigene Übersetzung)
[34] Aus „The Engineering of Mixed Reality Systems“ [DGN10], Seite 298 (eigene Übersetzung)
[35] Aus „The Engineering of Mixed Reality Systems“ [DGN10], Seite 298 (eigene Übersetzung)

ASUR die Modellierung der Applikation mit Hilfe von Szenarios, die aus Komponenten und Verbindungen zwischen diesen bestehen. "[36]. *ASUR* wurde schon in diesem Kapitel auf Seite 75 vorgestellt.

„Es existiert eine Vielzahl an Werkzeugen für die Entwicklung von Groupware, allerdings nur wenige für die Entwicklung von AR Applikationen. Ein Werkzeug für beide Arten ist den Verfassern des Artikels nicht bekannt"[37].

„Die meisten Werkzeuge zur Entwicklung von Groupware (wie beispielsweise Groupkit [RG96], ALV [HBR+94] oder Clock [UG99]) bedingen, dass alle Benutzer mit dem System in gleicher Weise interagieren, so dass es nicht möglich ist, eine heterogene Rollenverteilung der Benutzer und der Systeme zu realisieren. Als einziger Ansatz in diesem Bereich unterstützt *Networking shared dictionary* [MGRB06] die Verwendung von heterogenen Klienten"[38].

„Mehrere Groupware-Tooklits unterstützen dynamische Adaption, die zur Transition zwischen Szenen genutzt werden kann. Schmalsteig et al. liefert einen Ansatz zur Migration von Klienten in VR Anwendungen [SH02]. Realisiert wird dies über eine geteilte Szenegraph Datenstruktur mit Hilfe von Replikation. Der Code und der Szenegraph können so mit Hilfe einer Device-Anpassung auf neue Klienten migriert werden"[39].

„Die meisten Werkzeuge, die für die Entwicklung von AR Anwendungen gedacht sind, widmen sich speziell dem Problem der Ermittlung der Kameraposition und Orientierung und der Ermittlung der Position von physikalischen Objekten. [· · ·] Augmented Reality Werkzeuge sind typischerweise Bibliotheken, die in einer großen Auswahl von Programmiersprachen eingebunden und benutzt werden können, z. B. das ARToolKit [KB99] oder ARTag [Fia05]. Beide verwenden spezielle Bildmuster (Pattern, Tags), die an die reale Umgebung angebracht werden. Diese werden dann programmiertechnisch mit Objekten der virtuellen Welt verknüpft. Die Position und Orientierung der Kamera kann dann über die Analyse der Tags im aufgezeichneten Bild errechnet werden. GoblinXNA verfolgt einen ähnlichen Ansatz mit dem Unterschied, dass es in die XNA Entwicklungsumgebung von Microsoft integriert ist [OLWF07]. Weitere Probleme, mit denen sich Augmented Reality Werkzeuge befassen, sind beispielsweise Lokalisierung, Gestenerkennung und Grafik [Fis02]. [· · ·] *Fiia* kann nun als Generalisierung der vorgestellten Ansätze gesehen werden, indem es

[36] Aus „The Engineering of Mixed Reality Systems" [DGN10], Seite 298 (eigene Übersetzung)
[37] Aus „The Engineering of Mixed Reality Systems" [DGN10], Seite 298 (eigene Übersetzung)
[38] Aus „The Engineering of Mixed Reality Systems" [DGN10], Seite 298 (eigene Übersetzung)
[39] Aus „The Engineering of Mixed Reality Systems" [DGN10], Seite 299 (eigene Übersetzung)

weit mehr Flexibilität liefert, zum einen auf der konzeptionellen und verteilten Architektur-Ebene, zum anderen in der Bereitstellung der notwendigen Infrastruktur für die Realisierung von AR Anwendungen“[40].

„*Fiia* ist eine Entwurfsnotation für kollaborative AR Anwendungen. Mit Hilfe des Werkzeugs *Fiia.Net* werden diese Entwürfe zu ausführbaren Anwendungen realisiert. *Fiia* ist weiter ein modellbasierter Ansatz der es erlaubt, ein abstraktes High-Level Modell eines Systems als verteile Anwendung mit physikalischen und virtuellen Objekten automatisch zu erzeugen. Verglichen mit früheren Ansätzen enthält *Fiia* drei prinzipielle Fortschritte“[41]:

High-Level Notation: *Fiia* verwendet eine High-Level Notation zur Modellierung sowohl der Groupware als auch der Augmented Reality einschließlich der Funktionalität von gemeinsamer Datennutzung (Data Sharing) und der virtuellen/physikalischen Adapter.

Szenario-basierter Entwurf: *Fiia* nutzt einen Szenario-basierten Entwurf und eine Szenario-basierte Implementation.

Einfacher Modell-Code Übergang: *Fiia* bietet einen einfachen Übergang vom abstrakten Modell zur ausführbaren Anwendung an.

Abbildung 3.12$_{85}$ zeigt ein Beispiel eines *Fiia* Diagramms (hier die von Wolfe entwickelte Anwendung mit Namen *Raptor*). Der Entwickler (*Designer*) interagiert bei dieser Anwendung mit dem *Editor*, der es erlaubt, die Szene zu manipulieren, das Terrain zu gestalten, Spielelemente einzufügen und diesen dann Verhalten zuzuordnen. Elemente werden in Form eines Szenegraphen gespeichert. Dieser Szenegraph ist in der Komponente *Scene* gespeichert. Der Editor ist eine *Actor*-Komponente (), was bedeutet, dass diese Komponente in der Lage ist, Aktionen zu initiieren. Die *Scene* ist eine *Store*-Komponente (), das heißt ein passiver Datenspeicher. [DGN10]

Der Entwickler benutzt den *Editor* mit Hilfe eines Multitouch-Tisches, der in Abbildung 3.12$_{85}$ durch das *Table Surface* dargestellt ist. Das *Table Surface* bieten Eingabe- und Ausgabemöglichkeiten für den realen Multitouch-Tisch (benutzt wird ein Microsoft Surface [Mic11]). Interagiert wird mit dem *Table Surface* über einen bidirektionalen Informationsfluss. Das wird im Diagramm durch die beiden *Stream*-Verbindungen (⟶≫) angedeutet (die Bidirektionalität wird durch

[40] Aus „The Engineering of Mixed Reality Systems“ [DGN10], Seite 299 (eigene Übersetzung)
[41] Aus „The Engineering of Mixed Reality Systems.“ [DGN10], Seite 299 (eigene Übersetzung)

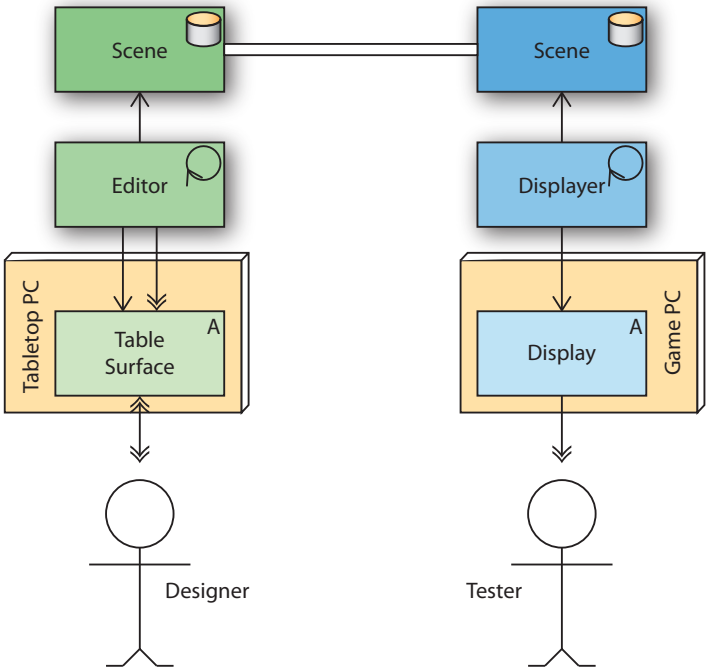


Abbildung 3.12: *Fiia* Modell einer Anwendung [DGN10].

die Doppelpfeile an beiden Enden der Linie dargestellt). Der *Editor* erfragt den aktuellen Status des Multitouch-Tisches mit Hilfe der *Call*-Verbindung (⟶) und aktualisiert die Anzeige des Tisches über die *Stream*-Verbindung. Allgemein repräsentieren *Streams* einen asynchronen Datenfluss, der für die Kommunikation diskreter Ereignisse oder kontinuierlicher Daten, beispielsweise Audio oder Video, verwendet werden kann. Im Gegensatz dazu steht die *Call*-Verbindung, die den traditionellen synchronen Aufruf von Methoden repräsentiert. [DGN10]

Gleichzeitig, während der Entwickler die Szene im *Editor* bearbeitet, können Testbenutzer (*Tester*) die Anwendung auf dem *Display* anschauen. Der *Displayer-Aktor* aktualisiert die Darstellung entsprechend den Änderungen des Entwicklers. Beide Versionen der *Scene*, sowohl die der Entwickler als auch die von den Testbenutzern, werden über eine *Sychronisationsverbindung* (⟷) konsistent gehalten. Eine *Synchronisationsverbindung* sorgt dafür, dass zwei oder mehr Datenspeicher konsistent bleiben, so dass Änderungen automatisch an alle *Stores* propagiert werden. [DGN10]

Die Diagramme von *Fiia* werden als verteiltes System implementiert. In diesem Beispiel werden zwei Computer verwendet; Ein Computer, der den Multitouch-Tisch ansteuert (*Tabletop PC*) und vom Entwickler benutzt wird, und ein Computer, der die Darstellung der Szene für

die Testbenutzer aufbereitet (*Game PC*). *Fiia* Diagramme spezifizieren jedoch nicht die Details des verteilten Systems. Sie abstrahieren die wichtigen Aufgaben wie beispielsweise die Aufteilung der Komponenten, den verwendeten Algorithmus zur Datenübertragung zwischen Knoten und die Konsistenzverwaltung für die Datensynchronisation. Das *Fiia.Net*-Werkzeug kann bei der späteren Umwandlung in ausführbare Anwendungen diese abstrakten Aufgaben automatisch bestimmen. Das erlaubt es dem Entwickler, sich auf die Funktionalität seiner Anwendung zu konzentrieren und nicht die Details der Implementation zu betrachten. [DGN10]

Fiia bietet über eigene Notation eine gute und einfache Möglichkeit, verteilte Mixed Reality Anwendungen konzeptionell zu realisieren und mit Hilfe des mitgelieferten *Fiia.Net*-Werkzeuges in eine ausführbare Anwendung zu überführen. *Fiia* bietet leider nicht die Möglichkeit, eine Anwendung entlang des Mixed Reality Kontinuums zu entwickeln. So ist es schwierig, Komponenten, die als virtuelles Objekt definiert sind, in real existierende Komponenten zu überführen. *Fiia* bietet hier weder einen Automatismus noch ein Vorgehen an. So müsste für jeden Prototypen das komplette Modell überarbeitet werden, um eine entsprechende Entwicklung entlang des Mixed Reality Kontinuums zu realisieren. Das ist allerdings nicht wünschenswert. Die Groupware-Eigenschaften und die automatische Generierung ausführbarer Applikationen sind jedoch bei *Fiia* hervorzuheben.

Fiia: Model-Based Approach to Engineering Collaborative AR	
Autoren	Wolfe, Smith, Phillips, Graham Jahr 2010
Bereich	Modell und Notation
Beschreibung	Entwicklung von kollaborativen MR-Anwend.
Merkmale:	+ Notation, Modell, Werkzeuge
	+ Anwendungsbeispiele
	+ Grafische Notation
	+ Automatische Generierung
	+ Groupware-Eigenschaften
	- Keine Entwicklung entlang d. MR Kontinuums
	- Nicht iterativ

Weitere Entwurfskonzepte

Eine große Anzahl an weiteren Entwurfskonzepten wurde schon im vorherigen Abschnitt ab Seite 78 erwähnt. Auch hier sei des Weiteren

der jährlich stattfindende SEARIS Workshop für weitere, tiefergehende Informationen zu diesem Themenfeld erwähnt [SEA08].

3.4 Softwareumgebungen und -lösungen

Die im letzten Kapitel vorgestellten Arbeiten stellen neben einem konzeptionellen Entwurf gleichzeitig eine Software zur (automatischen) Erzeugung von ausführbaren Programmen zur Verfügung. Die mit Hilfe der verschiedenen, im letzten Kapitel vorgestellten Konzepte entwickelten Anwendungen bieten somit nur eine sehr abstrakte Sicht. Das bedeutet, dass die Entwickler eine Sicht auf ihre Anwendungen nicht auf quelltextbasis erhalten, sondern auf Modellebene die Anwendungen entwickeln.

Eine Grundvoraussetzung für einen praktikablen Entwurfsprozess ist die Bereitstellung von Werkzeugen, die eine rasche Entwicklung von Mixed Reality Komponenten unterstützten. Hier existiert eine Vielzahl von Arbeiten in diesem Bereich, angefangen von API[42]-basierenden Low-Level Ansätzen über Software-Frameworks bis hin zu komplexen High-Level Autoren-Werkzeugen. Ich beschränke mich hier auf die Arbeiten, die mein System beeinflusst haben.

Im folgenden Kapitel möchte ich gerne einige dieser Softwareumgebungen und -lösungen kurz vorstellen, die es erlauben, eigene Anwendungen auf eine etwas konkreteren Art zu entwickeln. Vorgestellt werden verschiedene Softwareumgebungen, die es dem Entwickler ermöglichen, Virtual und Mixed Reality Anwendungen zu entwerfen.

Frühe API-basierte VR Frameworks

Ein sehr frühes API-basiertes Framework, das ausschließlich für VR Anwendungen verwendet werden kann, ist VRJuggler [BJH+01]. Es ist ein objektorientierts C++ VR-System, kann plattformübergreifend verwendet werden und steht unter einer Open Source Lizenz. VRJuggler erlaubt eine beliebige Kombinationen von Eingabegeräten, Grafik-APIs und Plattformkonfigurationen. Eine VR-API mit einem Fokus auf Simulationen ist Delta3D [DMJ05]. Dieses Framework basiert auf der häufig im wissenschaftlichen Bereich eingesetzten Szenengraph-Bibliothek OpenSceneGraph [Ope10] und vereint eine erhebliche Anzahl an Open Source Bibliotheken beispielsweise für Charakteranimation, Physik oder Künstliche Intelligenz und bietet daher eine enorme

[42] API = **A**pplication **P**rogramming **I**nterface – Eine Programmierschnittstelle auf Quelltextebene.

Funktionsvielfalt. Delta3D kann zusammen mit der Skriptsprache Python benutzt werden und liefert einen umfangreichen Szenen-Editor namens *STAGE*.

Autorenbasiertes MR Framework AMIRE

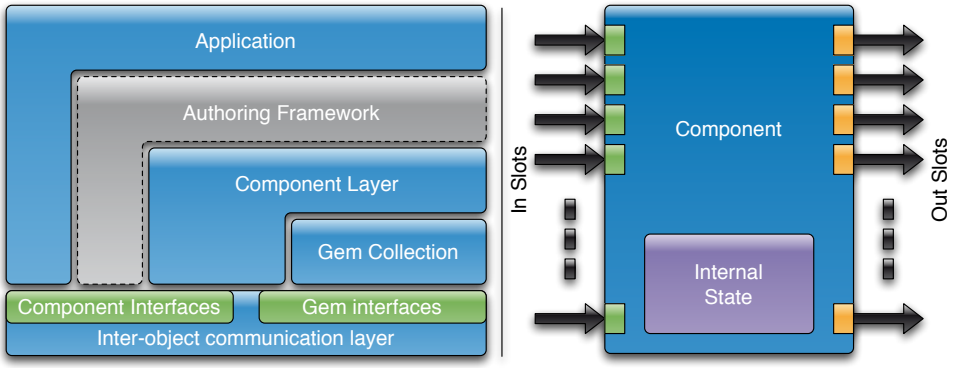


Abbildung 3.13: AMIRE Framework und Komponente [DGHP02].

Einer der ersten Versuche von den reinen APIs hin zu einer autorenbasierten Entwicklung zu gelangen, war das AMIRE[43] Werkzeug, das als Teil des europäischen IST Projektes entwickelt wurde [GHP+02]. AMIRE stellt ein Autorenframework zur Verfügung (siehe Abbildung 3.13), auf den die Anwendungen basieren. Die Basis dieses Frameworks sind die sogenannten *Gems*. *Gems* sind eine Sammlung von Techniken und Algorithmen, die für Programmierer gedacht sind. Die Grundgedanke bei diesen *Gems* ist, dass Entwickler ihre Ideen, Algorithmen und Werkzeuge mit anderen Entwicklern (und den Entwicklern von AMIRE) teilen und so die Ressourcen wiederverwendet werden können. Durch eine große Beteiligung von vielen Programmierern entwickelt sich so eine große Basis an innovativen Lösungen für eine Vielzahl von Problemen im Mixed Reality Bereich. Ein Problem bei der Bereitstellung der Gems ist allerdings, dass viele Ideen und Algorithmen nicht für die Wiederverwendung programmiert wurden bzw. die Schnittstellen sehr unterschiedlich sind. Auch ist es schwierig, allgemeine Prozesse für Mixed Reality Anwendungen von Lösungen Dritter zu finden und wieder zu verwenden. Die Lösung, die AMIRE vorschlägt, ist, etablierte Lösungen für einzelne Aufgaben in einer *MR Gem Collection* zu sammeln [GHP+02].

Die *Mixed Reality Komponenten* (oder einfach *Komponenten*) sind die grundlegenden Elemente im Entwicklungsprozess.Dabei repräsentieren *Komponenten* konkrete Lösungen für domänenspezifische Proble-

[43] AMIRE steht für „**A**uthoring **MI**xed **RE**ality“.

me und kombinieren bzw. erweitern typischerweise die *Gems* in Richtung High-Level Funktionalität von MR Anwendungen. *Komponenten* sind als domänenspezifische Elemente definiert. Abstrakt betrachtet teilen sich *Komponenten* in geometrische Modelle und Verhalten auf, wobei Verhalten sowohl die Animation als auch die Simulation eines spezifischen Verhaltens beinhalten kann. Einige wünschenswerte Ausprägungen der *Komponenten* sind folgende: strukturiert, wiederverwendbar in unterschiedlichen Versionen, wiederverwendbar in unterschiedlichen Anwendungen, erweiterbar, flexibel [GHP+02].

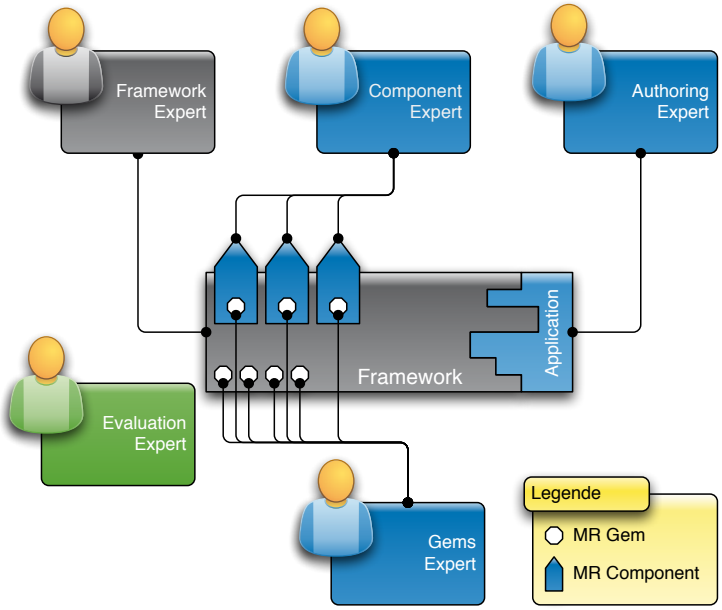


Abbildung 3.14: Experten bei einer AMIRE-Entwicklung [DGHP02].

Das *MR Framework* von AMIRE dient schließlich als Bindeglied zwischen den *Gems* und den *Komponenten*. Weiterhin bietet es eine High-Level API und eine Schnittstelle für die Komponenten an. Das *MR Framework* beinhaltet sowohl ein Laufzeit-Framework als auch ein Autoren-Framework. Dies stellt bei der Anwendungsentwicklung sicher, dass die domänenspezifischen Experten an den dafür vorgesehen Baustellen arbeiten können, wie in Abbildung 3.14 zu sehen ist [GHP+02]. Bei einer Entwicklung einer Anwendung sind fünf domänenspezifische Experten beteiligt: Der *Gem Expert* entwickelt neue benötigten *Gems* bzw. findet schon existierende aus der *Gem Collection*. Der *Component Expert* entwickelt die Komponenten mit Hilfe der vorhandenen *Gems*. Der *Framework Expert* integriert die Komponenten schließlich in das Framework, so dass der *Authoring Expert* sie in der Anwendung verwenden kann. Eine überprüfende Rolle hat der *Evaluation Expert*, der das komplette Projekt evaluiert und ggf. bei den entsprechenden Experten Korrekturen vorschlägt. Jeder dieser

Experten muss somit nur Teilaufgaben übernehmen, und zwar genau in dem Gebiet, in dem seine Kompetenzen liegen.

Konzeptionell leistet AMIRE gute Arbeit im Bereich Mixed Reality Anwendungsentwicklung. Ein Problem ist jedoch, dass normalerweise Lösungen sehr schwer auf die *Gems*-Ebene reduziert werden können, da sie meist in einem komplexeren Zusammenhang entwickelt wurden. Des Weiteren ist es schwierig, eine einheitliche, gut verständliche Schnittstelle für alle zur Verfügung gestellten Gems zu entwerfen, so dass bei der Entwicklung einer Applikation meist viel Arbeit in die eigentlichen Bausteine, den *Gems* und den *Components* fließt und so die eigentliche Entwicklung verkompliziert. Für diese Arbeit fehlte auch das Entwurfsvorgehen, das bei AMIRE nicht konkret entwickelt wurde.

Autorenbasiertes MR Framework AMIRE			
Autoren	Europäisches IST Projekt	Jahr	2002
Bereich	Framework		
Beschreibung	Mixed Reality Framework		
Merkmale:	+ Iterativ		
	+ Komponentenbasiert		
	+ Anwendungsbeispiele		
	- Kein konkretes Entwurfsvorgehen		
	- Keine Entwicklung entlang d. MR Kontinuums		

The Designer's AR Toolkit: DART

Im Mixed Reality Bereich ist *DART*[44] eines der ersten erfolgreichen Autoren Werkzeuge, die Mixed Reality Erweiterung in die Anwendung Director, einer kommerziellen Autorenlösung von Adobe, integriert [MGDB04]. MacIntyre stellte *DART* im Jahre 2004 vor. Da es sich um eine Erweiterung einer proprietären Software handelt, musste sich MacIntyre an das von Director vorgegebene konzeptionelle Modell halten, das aus der Theater-Metapher basiert. In Director existieren Akteure, die sichtbare und interaktive Objekte repräsentieren, eine Bühne (*stage*), auf dem die Akteure platziert werden, und Kameras, die die Akteure auf der Bühne darstellen. *DART* erweiterte diese Metapher konsequent indem es spezielle Akteure, sogenannte *DART-Aktors* und Kameras lieferte, die für Mixed Reality Anwendungen benötigt wurden. In Abbildung 3.15$_{91}$ ist die Integration von *DART* in Director

[44] Abk. für „The Designer's AR Toolkit", das Augmented Reality Werkzeug für Designer.

zu sehen.

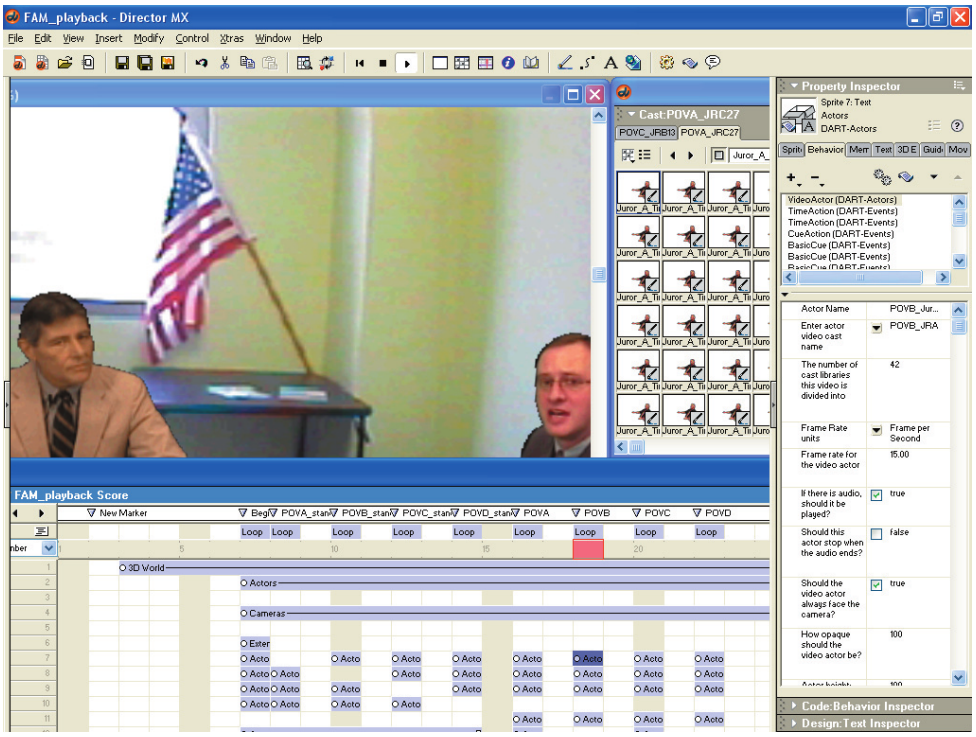


Abbildung 3.15: *DART* von MacIntyre [MGDB04].

„*DART* will besonders die frühen Entwurfsaktivitäten unterstützten, speziell die schnelle Umsetzung von Storyboards zu Prototypen, so dass der experimentelle Teil des Entwurfs sehr früh und sehr oft getestet werden kann. *DART* erlaubt es dem Entwickler komplexe Beziehungen zwischen der physikalischen und der virtuellen Welt zu spezifizieren und unterstützt 3D Animatic[45]-Akteure (informeller, skizzenbasierter Inhalt) zusätzlich zu den besser aufbereiteten Inhalten. Die Entwickler können Video und Sensordaten synchronisiert aufzeichnen und abspielen, was es erlaubt, unabhängig von der echten Kamera zu arbeiten und spezifische Teile der Anwendung effizient zu testen“[46].

DART ging denselben Weg, den ich am Anfang meiner Arbeit gegangen bin, indem ein proprietäres Werkzeug, das schon eine große Verbreitung bei Anwendern hatte, mit Komponenten aus dem Bereich Mixed Reality erweitert wurde. Dadurch musste nur ein kleiner Teil der Software entwickelt werden und man konnte auf schon bestehende und gut funktionierende Strukturen zurück greifen. Diese festen Strukturen können sich allerdings auch zum Nachteil entwickeln,

[45] Eine Animatic, auch Story Reel genannt, ist ein gefilmtes Storyboard.[Wik11]

[46] Aus „DART: a toolkit for rapid design exploration of augmented reality experiences“ [MGDB04], Seite 197 (eigene Übersetzung)

gerade in meinem Fall, wie man in Kapitel 4.5.1$_{136}$ nachlesen kann. *DART* hatte das Problem, dass sich viele Entwickler von Director abgewendet haben und stattdessen auf Flash umgestiegen sind, so dass die Entwicklung von *DART* eingestellt wurde.

DART: toolkit for rapid design exploration of AR experiences	
Autoren	MacIntyre, Gandy, Dow, Bolter Jahr 2004
Bereich	Framework
Beschreibung	Mixed Reality Framework in Director
Merkmale:	+ Theatermetapher
	+ Professionelle Entwicklungsumgebung
	+ Anwendungsbeispiele
	- Kein konkretes Entwurfsvorgehen
	- Keine Entwicklung entlang d. MR Kontinuums
	- Nicht iterativ

Studierstube

Ein weiteres bekanntes System im Bereich Mixed Reality ist das *Studierstube* System [SFH+02] von Dieter Schmalstieg, entwickelt an der Technischen Universität Wien und weiterentwickelt an der Technischen Universität Graz. *Studierstube* basiert auf der Szenegraph-API Coin3D, einem Open Inventor Clone der norwegischen Firma Kongsberg Oil & Gas Technologies [Tec11], und einer Middleware für die I/O-Abstraktion namens OpenTracker [RS01], welche einen flexiblen Einsatz von Tracking-Geräten erlaubt. *Studierstube* erlaubt eine einfache Entwicklung von Mehrbenutzer AR-Anwendungen, sowohl auf klassischen Computern als auch auf mobilen Endgeräten mit der speziellen allerdings kommerziellen Version *Studierstube ES*. Die Verwendung von mobilen Endgeräten wird auch im Bereich Mixed Reality immer interessanter, da es die heutige Leistung erlaubt, die komplette Verarbeitung lokal auf dem mobilen Gerät auszuführen. Noch vor einigen Jahren was das nicht möglich, und mobile AR Anwendungen mussten mit Hilfe von Bildübertragung und Bildberechnung auf dem PC gelöst werden, wie das Beispiel der AR-Enigma aus dem Jahre 2002 [PSG+02] oder die testbare Design-Repräsentation für mobiles AR Authoring [GPR+02] zeigt.

Studierstube ES läuft auf mehreren (mobilen) Plattformen (z. B. Windows, Windows CE, Symbian, Andriod und iOS[47]). Alle relevanten

[47] iOS Geräte sind nicht offiziell unterstützt, es existiert jedoch eine Entwicklerversion.

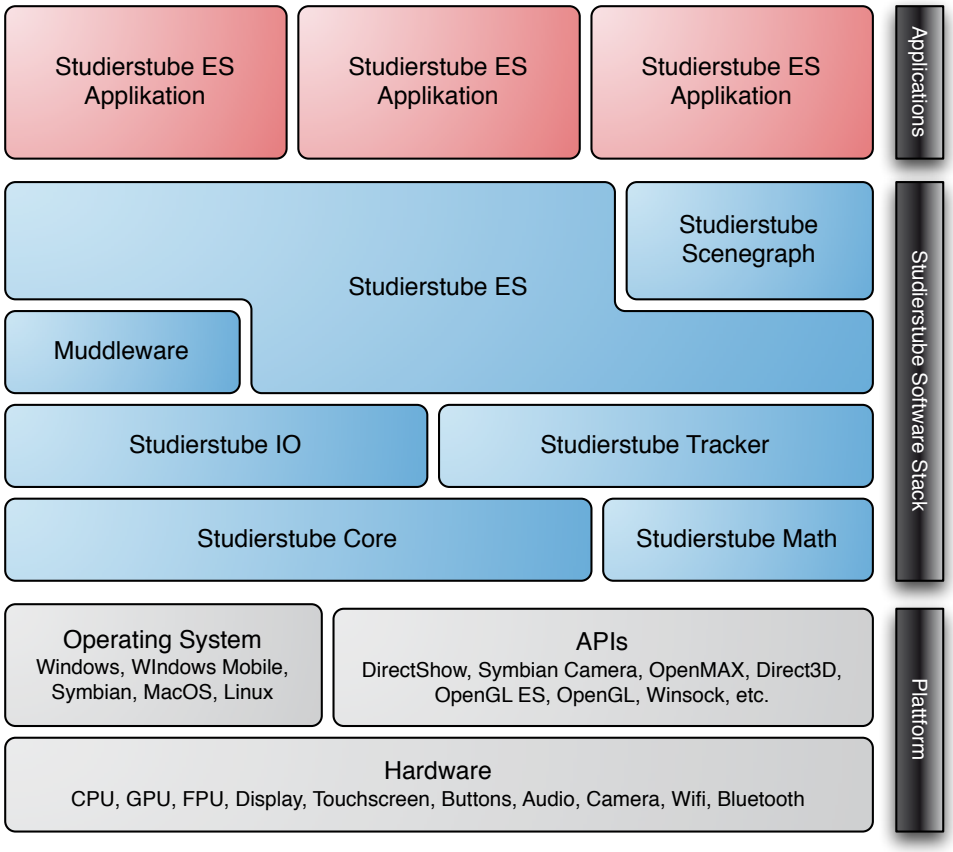


Abbildung 3.16: Aufbau von Studierstube ES [SW07].

Details der Entwicklung von AR Anwendungen werden von *Studierstube ES* berücksichtigt: Grafikausgabe, Videoverarbeitung, Tracking, Multimedia, Speicherverwaltung und Synchronisation von Mehrbenutzer Mehrbenutzern (siehe Abbildung 3.16_{93}). *Studierstube ES* bietet eine große Zahl von Schnittstellen (*Muddleware*[48], *Studierstube Scenegraph*, *Studierstube Tracker, etc.* und, da es von Grund auf für mobile Endgeräte programmiert wurde und nicht auf existierenden Lösungen basiert, eine sehr hohe Performance. Damit ist die Entwicklung von komplexen AR Anwendungen auf mobilen Endgeräten für den kommerziellen Markt bzw. im akademischen Rahmen sehr schnell realisierbar. Da *Studierstube ES* speziell auf mobile Endgeräte optimiert wurde, werden 3D Beschleunigung der eingesetzten Grafikprozessoren, Fließkommazahl- und Fixpunktzahlarithmetik sowie die Verwendung der verbauten Kameramodule unterstützt, soweit diese auf der mobilen Plattform vorhanden sind. Das ermöglicht mobile AR Anwendungen, die vor wenigen Jahren nur auf Desktop-PC möglich waren. Durch die Integration vom *Studierstube ES* in die Softwareum-

[48]Muddleware ist eine Netzwerklösung für mobile Endgeräte und stammt von denselben Entwicklern wie *Studierstube ES*.

gebung der einzelnen mobilen Plattformen ist die Einbindung in eigene Projekte sehr einfach. Da es sich, wie bereits erwähnt, um eine kommerzielle Lösung handelt, ist die Benutzung von *Studierstube ES* leider nur Entwicklern vorenthalten, die eine Lizenz besitzen.

Studierstube			
Autoren	Schmalstieg et al.	Jahr	2002
Bereich	Framework		
Beschreibung	Mixed Reality Framework		
Merkmale:	+ Komplexes Framework		
	+ Mobile Plattformen		
	+ Anwendungsbeispiele		
	- Kein konkretes Entwurfsvorgehen		
	- Entwicklung nicht entlang d. MR Kontinuums		
	- Nicht iterativ		

Weitere High-Level Werkzeuge der letzten Jahre

Broll beschreibt in seiner Arbeit, die 2008 auf der 3DUI-Konferenz vorstellt wurde, ein visuelles Autorenwerkzeug für 3D Interaktionstechniken [BHB08]. Das Konzept „Interactive Bits“ ist ein komponentenbasierter Ansatz zur visuellen Spezifikation von Mixed Reality Interaktionstechniken, Objektverhalten oder vollständiger Mixed Reality Prototypen. Das Werkzeug kombiniert synchronen Kontroll- und Datenfluss mit asynchronen Events und Netzwerkkommunitation. Spezifiziert wird es über eine XML-basierte Beschreibung, die die Objekte, die Komponenten und den Kontroll- bzw. Datenfluss definiert.

Sandor et al. beschreiben in ihrer Arbeit „Immersive mixed-reality configuration of hybrid user interfaces“ [SOBF05] ein Mixed Reality System, welches Anwendern erlaubt, eine Mixed Reality Applikation zur Laufzeit zu konfigurieren und eine Vielzahl von Anzeige- und Interaktionsgeräten beliebig zu kombinieren. Dabei wird eine Datenfluss-orientierte Visualisierung durch verbindende Linien zwischen grafischen 2D-Symbolen verwendet. Diese Methode habe ich auch in meinem ersten Ansatz verwendet, indem ein proprietäres 3D Autorensystem als Grundlage für die Entwicklung von Mixed Reality Applikationen verwendet wurde. Ein Unterschied allerdings ist, dass die Software, die ich verwendet habe, eine Kontrollfluss-orientierte visuelle Programmierung verwendete. Näheres kann in Kapitel 4.5.1$_{136}$ nachgelesen werden.

Envir3D ist ein Modellierungswerkzeug, mit dem 3D Inhalte visuell spezifiziert werden. Dabei steht immer ein abstraktes Modell der Benutzungsschnittstelle zur Evaluierung und Verfeinerungen zur Verfügung. Dieses wird zur Erzeugung einer VRML-Darstellung verwendet [VCBT04].

Ein Beispiel für ein High-Level Authoring-Werkzeug, das allerdings nicht für Mixed Reality Anwendungen gedacht war, ist 3DVIA Virtools [Das09]. In Kapitel 4.5.1$_{136}$ wird 3DVIA Virtools und die von mir entwickelte Mixed Reality Erweiterung näher vorgestellt.

Es existieren auch in diesem Bereich noch viele andere gute Werkzeuge. Eine gute Quelle ist hier wieder der jährlich stattfindende SEARIS Workshop, der eine vertiefende Quelle für diese Entwicklungen bietet. [SEA08] Die für meine Arbeit wichtigsten Arbeiten habe ich jedoch vorgestellt.

3.5 Zusammenfassung

In diesem Kapitel wurden aktuelle Forschungsergebnisse in den für diese Arbeit relevanten Bereichen „Mixed Reality Entwurfskonzepte“, „Entwurfskonzepte mit Werkzeugumgebung“ und „Softwareumgebungen und -lösungen“ vorgestellt und bewertet. Es wurden die für diese Arbeit wichtigsten Beiträge in den jeweiligen Gebieten kurz beschrieben, die Merkmale herausgestellt und deren Vor- und Nachteile angeführt. Bei den früheren Arbeiten galt das Hauptaugenmerk vorwiegend einer einzelnen Ausprägung, wie beispielsweise der Vorstellung eines neuen Frameworks oder einer neuen Methode der Modellierung, während bei späteren Arbeiten die Zusammenführung mehrerer Ausprägungen im Vordergrund stand. Waren es zu Beginn nur einfache API-basierte Frameworks, die es dem Benutzer ermöglichten, Mixed Reality Anwendungen zu realisieren, so entwickelten sich später daraus komplette Werkzeuge, die es ermöglichten, auf abstrakte Weise seine Anwendung zu entwerfen. Zusammenfassend kann man sagen, dass alle Arbeiten in ihrem speziellen Gebiet ihre Vorteile haben.

Was allerdings noch nicht in der Literatur versucht wurde, ist, eine Anwendung entlang des Mixed Reality Kontinuums zu entwickeln. So muss man bei allen Arbeiten die realen und virtuellen Objekte schon zu Beginn der Entwicklung festlegen, eine spätere Transformation ist nicht vorgesehen. Dabei ist es gerade bei der Entwicklung von Mixed Reality Applikationen sinnvoll, reale Komponenten zu Beginn virtuell zu modellieren, sei es, weil die realen Objekte noch nicht

existieren oder weil die technischen Grundlagen zur Erkennung der realen Objekte noch nicht existiert. Allein bei DART [MGDB04] ist es möglich, die Entwicklung auf zuvor aufgezeichneten Daten (sowohl Video als auch Position und Orientierung) fortzusetzen. So kann auch bei Abwesenheit der benötigten Hardware zur Erkennung von realen Objekten bzw. der realen Objekte selbst (sei es, weil die Objekte noch nicht existieren oder sich an einem anderen Ort befinden) die Entwicklung fortgesetzt werden. DART jedoch beschränkt sich nur auf Video bzw. Tracking-Informationen, eine Transition von virtuellen zu realen Objekten findet hier nicht statt. In meinem Ansatz ist die Transition von virtuellen zu realen Objekten (und zurück) möglich, um so eine während der Entwicklung größtmögliche Unterstützung zu erhalten.

Viele der späteren hier vorgestellten Arbeiten stellen eine eigene Vorgehensweise bzw. ein eigenes Modell vor. Das ist durchaus sinnvoll, da die Entwicklung von Mixed Reality Anwendungen mit üblichen Entwicklungswerkzeugen und -Abläufen nicht vollständig abgedeckt wird. Eine Unterscheidung zwischen den realen, physikalischen Objekten und den virtuellen Objekten wird bei allen Arbeiten im Bereich Mixed Reality gemacht, sei es *Fiia*, *ASUR* oder Studierstube. Mein Ansatz geht indes noch einen Schritt weiter und kategorisiert die vorhandenen Objekte einer Anwendung in vier spezielle Arten, die dem MVC-Ansatz (siehe Seite 2.2.1$_{45}$) angelehnt ist. Diese Aufteilung erlaubt eine feingranularere Entwicklung einzelner Komponenten. Des Weiteren wird damit auch die Entwicklung entlang des Mixed Reality Kontinuums unterstützt, was bedeutet, dass Objekte bzw. Komponenten zuerst virtuell entworfen und später durch reale, physikalische Komponenten bzw. Objekte ersetzt werden. Dieser Schritt lässt sich auch umkehren, so dass reale Objekte wieder durch ihre virtuellen Gegenstücke ersetzt werden können. Das ermöglicht bei der Entwicklung eine gezielte Fokussierung auf eine spezielle Ausprägung einer Anwendung, indem nicht benötige Funktionalität auf ein Minimum reduziert wird.

Im Allgemeinen ist in der Literatur zu erkennen, dass die Bedeutung von speziellen Vorgehensweisen und Modelle für Mixed Reality Anwendungen mit steigender Leistungsfähigkeit, sowohl der klassischen PCs als auch der mobilen Endgeräte, zunimmt. Da sich die Entwicklung von Mixed Reality Anwendung von der Entwicklung klassischer Anwendungen stark unterschiedet und sich noch keine allgemein gültige Vorgehensweise kristallisiert hat, ist ein erhöhter Forschungsbedarf in diesem Gebiet vorhanden. Das sieht man auch speziell an den vielen, eigenständigen Konferenzen und Workshops im Bereich Mixed Reality.

KAPITEL 4

Mixed Reality in the Loop

Das folgende Kapitel beschreibt das eigentliche „Mixed Reality in the Loop“-Entwurfsvorgehen (MRiL). Den Anfang macht die Anforderungsanalyse, in der die Anforderungen an die zu entwickelnde Applikation beschrieben und erläutert werden. Nachdem die Anforderungen klar definiert sind, wird die Vorgehensweise des MRiL beschrieben. Da MRiL neue bzw. angepasste Methoden benötigt, werden im Kapitel 4.3$_{103}$ diese vorgestellt. Beginnend mit dem MVCE-Modell wird die Grundstruktur des Entwurfsvorgehens erklärt. Darauf aufbauend wird das Akteurmodell beschrieben, welches benötigt wird um die Entwicklung zu strukturiert. In Kapitel 4.3.4$_{124}$ wird das das eigentliche Entwurfsvorgehen beschrieben, welches auf einem iterativen Vorgehen basiert.

Um das Vorgehen besser verständlich zu machen, wird in Kapitel 4.4$_{128}$ ein kleines Beispiel, eine Entwicklung nach MRiL, beschrieben. Dieses Beispiel ist klein gehalten uns soll nur die Besonderheiten von MRiL aufzeigen.

Damit MRiL sinnvoll angewendet werden kann, wird eine Software- und Werkzeugumgebung benötigt, die diesen Prozess unterstützt. Es wurden insgesamt zwei exemplarische Softwareumgebungen im Rahmen dieser Arbeit entwickelt. Bei der ersten Umgebung lag der Fokus auf der einfachen und visuellen Erstellung von Prototypen und basiert auf einer proprietären Softwarelösung. Sie unterstützt den Prozess aber nicht vollständig, gerade im Bezug auf die Akteure gab es dort Defizite. Die zweite Umgebung basiert auf einer Open Source Grafikbibliothek und versucht alle Aspekte der MRiL Vorgehens abzubilden. Dabei wurde auf die einfache visuelle Programmierung verzichtet und auf textuelle imperative Programmierung zurückgegriffen. Für

Experten, die einer Programmiersprache mächtig sind, ist die letzte Umgebung zu bevorzugen. Anwendern, die aus z. B. dem Bereich Design stammen, wird die erste Softwarelösung mehr zusagen.

Eine Reihe an eigenen Veröffentlichungen zum Thema MRiL, u.a. „Mixed reality in the loop – design process for interactive mechatronical systems“ [SGP10], „MVCE – A Design Pattern to Guide the Development of Next Generation User Interfaces“ [SGP+09], „Mixed Reality Design of Control Strategies“ [GSBP09], „Modellbasierter Entwurf von Mixed Reality-Interaktionstechniken für ein Indoor-Zeppelin“ [GPS+09], „A Design Method for Next Generation User Interfaces inspired by the Mixed Reality Continuum“ [SGPP09], „MiReAS: a mixed reality software framework for iterative prototyping of control strategies for an indoor airship“ [SPGP10] und „Iteratives Mixed-Reality-Prototyping und virtuelle Studiopräsentation einer Steuerung für ein Indoor-Lufschiff“ [PSHG10], wurde auf verschiedenen nationalen und internationalen Konferenzen vorgestellt und überwiegend positiv bewertet.

4.1 Anforderungsanalyse

Um das „Mixed Reality in the Loop“-Entwurfsvorgehen erfolgreich einsetzten zu können werden bestimmte Voraussetzungen an die zu entwickelnde Applikation gestellt. Da der zugrundeliegene Ansatz des Entwurfs auf einem prototypenbasierten Vorgehen (siehe Kapitel 2.1.12$_{39}$ basiert, sollten während der Entwicklung der Software mehrere verschiedene Prototypen vorgesehen sein. Sollte die Entwicklung keine Prototypen erfordern, wäre MRiL nicht das richtige Entwurfsvorgehen. Die Entwicklung der Prototypen ist noch mit der Bedingung der schrittweisen Verfeinerung verknüpft. Nicht jeder Prototyp sollte eine komplett andere Funktionalität bieten, sondern die Prototypen sollten auf einander aufbauen und nach und nach mehr Funktionalität erhalten. Sollte die Entwicklung der Applikation fordern, dass mehrere Prototypen mit komplett unterschiedlicher Funktionalität entwickelt werden, ist die Entwicklung mit MRiL nicht optimal aber durchaus möglich.

Selbst wenn eine Entwicklung einer Applikation die Herstellung von Prototypen beinhaltet, muss MRiL noch nicht der passende Ansatz sein. Es sollte mindestens eines der folgenden vier Kriterien für die Prototypenentwicklung zutreffen, um MRiL erfolgreich anwenden zu können:

Darstellung: Damit ist sowohl die Verfeinerung der rein virtuellen

Visualisierung als auch die Änderung der Darstellung von rein virtuell über gemischte Realität hin zur reinen Realität gemeint.

Steuerung: Die Steuerstrategien der zu entwickelnden Prototypen wird verfeinert. Dabei sollte die Entwicklung von einfachen, Tastatur- oder Maus-basierten Steuerungen hin zu komplexen, neuartigen Steuerstrategien verfeinert werden.

Modell: Das Modell der Software wird verfeinert. Es ist möglich, von einem rein virtuellen Modell zu einem realen Modell zu migrieren.

Umgebung: Die Umgebung hat Einfuss auf die Applikation. Auch hier, wie bei der Darstellung, kann mit einer rein virtuellen Umgebung begonnen werden, die dann nach und nach realer wird. Die Umgebung wird nie zu 100% erfasst, sondern Teile der Umgebung werden über Sensoren erkannt und der Applikation zur Verfügung gestellt.

Der Ansatz ist nicht für jede Art von Applikation sinnvoll. Es sollte mindestens einer der im Folgenden aufgeführten Punkte zutreffen, damit die Verwendung von MRiL erfolgreich angewendet werden kann:

Mixed Reality Applikation: Bei der Entwicklung von Mixed Reality Applikationen, im speziellen Augmented Reality Applikationen, kann das MRiL-Entwurfsvorgehen erfolgreich eingesetzt werden. Mixed Reality Applikationen zeichnen sich durch die Einbeziehung der realen Umgebung in die Software aus. Teile der realen Umgebung können dabei durch verschiedene Sensoren erkannt und der Software zur Verfügung gestellt werden. Beispiele hierfür sind Ultraschall Entfernungssensoren oder Videotracker (z. B. Studierstube, ART+) zur Erkennung von Position und Orientierung.

Mixed Reality Benutzerschnittstellen: Eine gesonderte Klasse von Applikationen sind Mixed Reality Benutzerschnittstellen (*MR User Interfaces*, auch *Next Generation User Interfaces* (NGUI) genannt), die neue Benutzerschnittstellen für Mixed Reality Applikationen implementieren. Durch die Verwendung des MRiL-Entwurfsvorgehens können die neuartigen Benutzerschnittstellen in frühen Phasen auch ohne die spezielle Hardware, die normalerweise benötigt wird, getestet werden.

Mechatronische Systeme: Software für die Steuerung von mechatronischen Systemen ist sehr gut mit dem MRiL-Entwurfsvorgehen

zu realisieren. Dabei wird mehr die Entwicklung der Software als die Einbettung in Hardwarecontroller (Entwicklung des eingebetteten Systems) in den Vordergrund gestellt. Es ist aber auch möglich, die Software später auf speziellen für die Anwendung Controllern auszuführen, dabei wird hier auf das Prinzip von MiL, SiL und HiL (Siehe Kapitel 2.3.1$_{51}$) zurückgegriffen.

Für bestimmte Klassen von Applikationen ist die Verwendung von des MRiL-Entwurfsvorgehens nicht sinnvoll, da die klassischen Entwurfsprozesse dort besser geeignet sind:

Klassische Applikationen: Damit ist die Klasse von Anwendungen gemeint, die als Standardsoftware bezeichnet werden kann. Dazu zählen Officeanwendungen, Datenbanken, etc. Diese Programme basieren größtenteils auf den Eingabemöglichkeiten, die von einem Betriebssystem zur Verfügung gestellt werden, und verwenden die normale fensterbasierte Ausgabe.

WIMP[1]: Anwendungen, die auf einer grafischen Benutzeroberfläche basieren, die mit Hilfe einer Maus bedient werden können sind für die Entwicklung mit MRiL nicht geeignet. Beispiele für solche Anwendungen sind dialogbasierte Programme, die über die Maus und die Tastatur gesteuert werden.

Webbasierte Inhalte[2]: Webanwendungen sollten vorzugsweise mit klassischen Ansätzen der Softwareentwicklung realisiert werden, da die MRiL hier keine Vorteile sondern eher Nachteile bringen würde.

Eingebettete Systeme: Diese Art von Software benötigt meist spezielle Werkzeuge, die das Programm auf die passende Plattform kompilieren. Daher ist hier auch einer der klassischen Ansätze für die Entwicklung sinnvoll.

Damit sind die Anwendungen, die mit Hilfe des MRiL-Entwurfsvorgehen entwickelt werden können, klar eingegrenzt. Vorgestellt wurden sowohl Anwendungen, die von MRiL unterstützt werden, also auch Software, die nicht unterstützt wird und sinnvollerweise einem anderen Entwurfvorgehen folgen sollte. Die nachfolgende Tabelle zeigt nochmals im Überblick, welche Applikationen mit Hilfe von MRiL entwickelt werden können:

[1] WIMP steht für **W**indows **I**cons **M**enus **P**ointer.

[2] Als webbasierte Anwendungen, auch Webanwendung oder Webapplikation genannt, wird Software bezeichnet, die auf einem Webserver ausgeführt wird und beim Anwender mit Hilfe eines Webbrowsers angezeigt wird.

Applikation	Geeignet für MRiL		
	Ja	Teilweise	Nein
Mixed Reality Applikation	×		
MR Benutzerschnittstellen	×		
Mechatronische Systeme	×	×	
Eingebettete Systeme		×	×
Webbasierte Inhalte			×
Klassische Applikationen			×

Zusammenfassend zeigt die unten angegebene Tabelle die Kriterien, nach denen das MRiL-Entwurfsvorgehen für eine Anwendungsentwicklung verwendet werden kann:

Kriterien	Relevanz		
	Wichtig	Optional	Unwichtig
AR / MR	×		
Prototyping	×		
Komponentenbasiert	×		
Akteurbasiert		×	
Plattformabhängig			×
Programmiersprache			×

Damit sind die Anforderungen an eine Applikation bestimmt. Im folgenden Kapitel wird die Vorgehensweise des MRiL-Entwurfsvorgehen beschrieben.

4.2 Vorgehensweise

Das in dieser Arbeit entwickelte MRiL Entwurfsvorgehen basiert auf mehreren, speziell angepassten Vorgehensmodellen und dem Model-View-Controller Architekturmuster (Siehe Kapitel 2.2.1$_{45}$), welches für MRiL eine Erweiterung erfahren hat. Um eine Applikation nach dem MRiL-Entwurfsvorgehen zu entwickeln, muss folgendermaßen vorgegangen werden:

1. Beschreibung in schriftlicher Form (Optional)
Um die spätere Applikation in die vier Komponenten unterteilen zu können ist es sinnvoll die Funktionsweise der Anwendung schriftlich aufzuzeichnen. In dieser Form können dann die Identifizierungen des nächsten Schritts leichter realisiert werden.

Diese schriftliche Zusammenfassung der Anwendung sollte so genau wie möglich sein, es sind jedoch normalerweise zu Beginn einer Entwicklung noch nicht komplett alle Aspekte bekannt, so dass die erste Fassung der Beschreibung eher ungenau und oberflächlich sein wird. Zu Beginn reicht diese Beschreibung allerdings aus, um mit der Identifizierung beginnen zu können.

2. Identifizierung einzelner Elemente

Die einzelnen Elemente der Applikation sollten zu Beginn der Entwicklung identifiziert werden. Da es sich beim MRiL-Entwurfsvorgehen um einen iterativen Prozess handelt, kann diese Identifizierung erst recht grob und ungenau sein. In späteren Iterationen können die Elemente weiter verfeinert werden. Bei der Identifizierung sollte allerdings schon darauf geachtet werden, dass die Elemente der Applikation nicht in mehr als eine Komponente der MVCE Architekturmusters fallen.

3. Kategorisierung identifizierter Elemente

Nach der Identifizierung der einzelnen Elemente einer Applikation müssen diese nun in die MVCE-Komponenten kategorisiert werden. Nach der Kategorisierung können schon die Schnittstellen zwischen den Elementen definiert werden, die ja über das MVCE Architekturmuster vorgegeben sind.

4. Unterteilung in Akteure. (Optional)

Die Unterteilung der identifizierten Elemente in Akteure ist ein optionaler Schritt, der jedoch sinnvoll ist, wenn die Softwareumgebung dies unterstützt. Vorteil ist, dass die einzelnen Akteure einzeln verfeinert werden können. Ohne Akteure müsste jeweils eine gesamte MVCE-Komponente verfeinert werden. Je nach Komplexität dieser Komponente könnte eine Verfeinerung längere Zeit in Anspruch nehmen. Bei einer geringen Komplexität, wie sie normalerweise in frühen Prototyp-Stadien vorzufinden ist, ist der Aufwand der Verfeinerung ohne Akteure jedoch nicht viel aufwändiger, so dass am Anfang der Entwicklung auf die Unterteilung in Akteure verzichtet werden kann. Sollte die Softwareumgebung allerdings die Unterteilung in Akteure unterstützen (siehe $4.5.2_{148}$), sollte sie auch durchführt werden.

5. Erster Prototypen mit Platzhalter

Nach Identifikation und Unterteilung kann das Grundgerüst der Elemente bzw. Akteure in einer Softwareumgebung implementiert werden. Eine Implementation der definierten Schnittstellen der Komponenten bzw. Akteure ist denkbar, eine Funktionalität muss allerdings zu diesem Zeitpunkt noch nicht vorhanden sein. Der erste Prototyp wird in den meisten Fällen nur das

Programmgerüst mit den Schnittstellen und Verbindungen zwischen den Komponenten bzw. Akteuren abbilden.

6. Iterationen zur Verfeinerung
Der Prototyp kann nun durch das iterative Entwurfsvorgehen weiter verfeinert werden. Je nach festgelegtem Schwerpunkt bei der Entwicklung können alle Komponenten gleichzeitig oder aber einzelne Komponenten verfeinert implementiert werden. Durch dieses Vorgehen besteht z. B. die Möglichkeit, zuerst die Komponenten des Controllers zu verfeinern, um sie schon in frühen Prototypen zu testen, andere Komponenten jedoch in einem sehr frühen Entwicklungsstadium beizubehalten. Es ist auch möglich, von einer verfeinerten Komponente zurück zu einer früheren Implementation zu gehen, wenn z. B. andere Komponenten isoliert betrachtet werden sollen. Beispiele für die verschiedenen Arten der Vorgehensweise bei der Verfeinerung zeigt Kapitel 5159

7. Fertige Applikation
Nachdem jede Komponente zu ihrer gewünschten Form verfeinert ist kann der Prototyp als fertige Applikation bezeichnet werden. Soll diese Applikation zu einem späteren Zeitpunkt weiter entwickelt werden, stehen ihr die gesamten Prototypen der vergangenen Iterationen zur Verfügung. D. h. die Entwickler können für eine Komponente wieder auf ein sehr frühes Stadium der Entwicklung zugreifen um beispielsweise für andere Komponenten eine kontrollierte Umgebung zu erhalten. Es ist also Möglich auch nach der Fertigstellung der Applikation wieder in nachfolgenden Iterationen weiter zu entwickeln.

Damit die hier vorgestellte Vorgehensweise realisierbar ist, mussten Methoden zur Unterstützung des Verfahrens entwickelt werden. In nun folgenden Kapitel werden diese Methoden vorgestellt und beschrieben.

4.3 Entwickelte Methoden

Zur Realisierung des MRiL-Entwurfsvorgehen mussten mehrere Methoden entwickelt werden. Diese Methoden basieren meist auf schon in der Literatur bekannten Verfahren, wurden aber für das MRiL-Entwurfsvorgehen angepasst oder erweitert. Im einzelnen sind das:

MVCE: Das aus dem Entwurf von Benutzerschnittstellen bekannte Model-View-Controller Architekturmuster wurde um die Kom-

ponente „Environment“ erweitert und ergibt nun das MVCE Architekturmuster. Es ist die Grundlage des MRiL-Entwurfsvorgehens, welches voraussetzt, einzelne Teile der Anwendung zu einer der vier Komponenten zuzuordnen. Die klassifizierten Teile der Anwendung können dann unabhängig von den anderen MVCE Komponenten entwickelt und verfeinert werden. Die Aufteilung in die einzelnen Komponenten ermöglicht darüber hinaus die gemeinsame Visualisierung des Entwicklungstand jeder einzelnen Komponente gemeinsam in einem Kiviatgraph.

Die MRiL Metrik: Um eine Einschätzung über den Entwicklungstand der Applikation zu erhalten muss müssen für die einzelnen Komponenten des MVCE Architekturmusters eigene Metriken entwickelt werden. Die Metriken sind essenziell für eine sinnvolle Visualisierung des Entwicklungstand im Kiviatgraphen.

Akteurmodell: Zur Verfeinerung der MVCE-Komponenten wurde das Akteurmodell entwickelt, das die Möglichkeit bietet, die jeweiligen Komponenten feingranularer zu unterteilen. Die Akteuere können einzeln und unabhängig voneinander entwickelt und über sogenannte Adapter erweitert werden. Das Akteurmodell ist für die spätere Implementation des MRiL-Entwurfsvorgehen nicht zwingend erforderlich. Sollte die verwendete Entwicklungsumgebung das Akteurmodell unterstützten, sollte die Aufteilung allerdings durchgeführt werden.

Das Entwurfsvorgehen: Das vorgestellte Entwurfsvorgehen basiert auf einem iterativen Prototyping Prozess, der in jedem Schritt Teile der Applikation verfeinert bzw. weiter entwickelt. Nach jeder Iteration steht ein neuer Prototyp zur Verfügung, der für Tests verwendet werden kann. Bei der Verfeinerung ist es unerheblich, ob alle Komponenten gleichzeitig oder nur eine spezielle Komponente verfeinert werden.

Die einzelnen oben aufgezählten Methoden werden im einzelnen in den nachfolgenden Kapiteln ausführlich beschrieben.

4.3.1 MVCE - Model-View-Controller-Environment

Das Model-View-Controller Architekturmuster ist unter anderem in der Benutzerschnittstellen-Entwicklung so erfolgreich, da es die einzelnen Aufgaben in drei unterschiedliche und voneinander getrennte Komponenten unterteilt. Daher können die interaktiven und visuellen Aspekte einer Benutzerschnittstelle getrennt von der eigentlichen Applikation bearbeitet werden. Das Modell (**M**odel) repräsentiert dabei

die Daten der Applikation und kapselt diese. Die Darstellung (**V**iew) kapselt die visuellen Elemente wie z. B. die Schaltflächen, Textboxen oder Visualisierungen. Die Steuerung (**C**ontroller)[3] implementiert die Interaktionsdetails zwischen der Applikation und dem Benutzer, beispielsweise Mausklicks oder Tastatureingaben. Diese leitet er dann weiter an das Modell welches dann die Änderungen der jeweiligen Interaktion ausführt. Das Modell wiederum benachrichtigt die Darstellung, das sich Daten geändert haben, so dass sich die Benutzerschnittstelle passend ändern kann. MVC ermöglicht also modulares Design, indem die einzelnen Komponenten nicht voneinander abhängig sind. Es ermöglicht weiterhin die Benutzung mehrerer Darstellungen und unterschiedlicher Controller innerhalb derselben Applikation für dasselbe Modell. Diese Eigenschaften sind gerade im MRiL-Entwurfsvorgehen wünschenswert, da so modular und komponentenweise implementiert werden kann.

Ein zentrales Merkmal des MRiL-Entwurfsvorgehen ist die Integration der realen Umgebung in die digitale Anwendung. Die Applikation benötigt Informationen über Objekte oder Koordinaten der realen Welt, dessen Geometrie und Verhalten aber nicht unter der Kontrolle der Applikation steht. Reale Objekte können unter dem Einfuss von realen Manipulationen oder externen Kräften stehen. Daher muss es für die Applikation eine Möglichkeit geben, die Veränderungen der realen Objekte zu erkennen. In der Praxis besteht solch ein *Real World Model* einer Mixed Reality Applikation aus der Kombination aus statischen Informationen, z. B. Geometriedaten, die als fest gelten, und dynamischen Informationen, z. B. die Position und Orientierung des Nutzers bzw. eines zentralen Objektes. Diese dynamischen Daten können über spezielle Sensoren während der Laufzeit ermittelt werden. Sensordaten könnten als Controller-Events im MVC Modell gehandhabt werden, allerdings würde dies zu einem sehr unübersichtlichen Modell der Applikation führen.

Um daher Mixed Reality tatsächlich in einer digitalen Anwendung zu berücksichtigen, wurde das Model-View-Controller Architekturmuster [Ree03] (siehe auch $2.2.1_{45}$) um eine Dimension „Umgebung“ (**E**nvironment) erweitert, so dass die Sonderstellung der Umgebung bei Mixed Reality Anwendungen abgedeckt wird. Diese Erweiterung ist in Abbildung 4.1_{106} zu sehen. Ein Problem bei MR-Applikationen ist, dass die Umgebung nicht zu 100 Prozent erfasst werden kann bzw. muss. Durch verschiedene Sensoren und Techniken können Informationen wie Position und Orientierung von Objekten, bestimmte technische oder physikalische Eigenschaften oder Zustandsänderun-

[3] Steuerung wird im weiteren Text durch das englischen Wort „Controller“ ersetzt, da sich dieser Begriff in diesem Zusammenhang in der Literatur durchgesetzt hat.

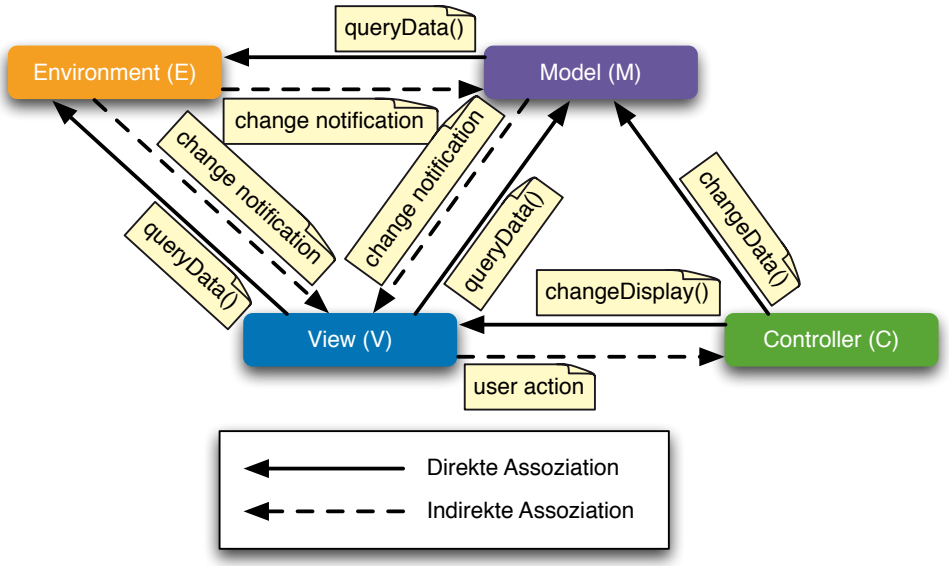


Abbildung 4.1: MVCE Architekturmuster

gen erfasst werden. Es ist aber fast unmöglich und auch nicht sinnvoll, alle Informationen der Umgebung zu erhalten. Für eine performante MR-Applikation sollten nur Informationen der Umgebung abgefragt werden, die wirklich benötigt werden. Beispielsweise sollte eine Applikation, die die Höhe eines Objektes der Umgebung erfordert, diese auch möglichst direkt geliefert bekommen und sie nicht über komplizierte Algorithmen umständlich berechnen müssen. Letzteres würde für die Berechnung der realen Höhe viel Ressourcen und Rechenzeit benötigen. Besser wäre es, diese Information mit Hilfe z. B. eines Höhensensors zu realisieren, der an dem abzufragenden Objekt angebracht ist.

In Abbildung 4.1 wird die Integration der Komponente „Umgebung“ dargestellt. Sie ist ähnlich dem Modell an die Darstellung gekoppelt und tauscht ihrerseits Daten mit dem Modell aus. Ein entscheidender Unterschied ist, dass der Controller keinen Einfluss auf die Umgebung hat. Das weist auf die Sonderstellung gegenüber dem Modell hin. Der Controller ist nicht in der Lage, Daten der Umgebung zu verändern. Auch das Abfragen der Daten geht über den Umweg des Modells, da dies die Daten der Umgebung empfängt und passend aufbereitet. Die Darstellung und das Modell benutzten dieselben Methoden zur Interaktion mit der Umgebung, auch werden beide von der Umgebung benachrichtigt, sollte sich an den Daten etwas geändert haben. Die Darstellung wird versuchen, die geänderten Daten dann zu visualisieren. Das Modell kann die Daten der Umgebung weiter verarbeiten und ggf. auch die Darstellung über Veränderungen am Modell informieren. Über die Darstellung, die die Benutzeraktionen

an den Controller weiterleitet, ist es nun auch möglich, Änderungen der Umgebung als Benutzerinteraktion zu interpretieren. Dafür sendet die Umgebung ein Notifikation an die Darstellung, dass sich Daten geändert haben. Die Darstellung kann diese Änderung als Eingabe des Benutzers interpretieren und eine Meldung einer Benutzerinteraktion an den Controller senden. So sind Mixed Reality Benutzerschnittstellen mit Hilfe von MVCE einfach realisierbar und strukturiert zu implementieren.

Es folgt nun eine detaillierte Übersicht der einzelnen Komponenten von MVCE, ihren Eigenschaften und der Verwendung:

Model: Das Modell kapselt die eigentlichen Daten der Applikation. Das Modell ist im allgemeinen eine passive Komponente, d. h. es kann sich nicht selbstständig ändern. Änderungen werden nur über den Controller realisiert. Es ist indes möglich , dass sich Änderungen auf mehrere Teile des Modells auswirken, die intern im Modell berechnet werden können. Als Beispiel könnte der Controller die Position eines Modells ändern und das Modell könnte zusätzlich aus der gegebenen Position die Farbe der Geometrie über eine interne Funktion neu bestimmten.

View: In der Darstellung wird das Modell komplett oder auch nur teilweise visualisiert. In MVCE kann es eine oder mehrere unterschiedliche Darstellungen vom selben Modell geben, die spezifische Teile des Modells darstellen. Als Beispiel könnten das zwei Ansichten sein, die eine visualisiert die virtuelle 3D Umgebung, eine andere nur die Kollisionsmodelle der Physikbibliothek. Die Darstellung erhält Notifikationen bei Änderung der Umgebung und des Modells um seine Darstellung anpassen zu können. Die Informationen muss die Darstellung allerdings selbst bei der Umgebung bzw. beim Modell erfragen. Da weder Umgebung noch Modell wissen, was die jeweilige Darstellung für Daten benötigt, ist es sinnvoller, dass sich die Darstellung selbst um die Beschaffung der Daten kümmert, als wenn Umgebung und Modell die Änderungen an jede Darstellung schicken. So wird das Datenvolumen zwischen den Komponenten kleinstmöglich gehalten.

Controller: Der Controller ist für die Verarbeitung der Benutzereingaben zuständig. Die Eingaben können sowohl über den View mitgeteilt als auch intern erzeugt werden. Eine Interaktion mir einer grafischen Benutzerschnittstelle wäre ein Beispiel für den ersten Fall, eine Bewegung mit einer Wiimote[4] ein Beispiel für

[4]Die Wiimote ist ein Gamecontroller der Spielkonsole Wii von Nintendo, der auch mit Standard-

den zweiten Fall. Auch Komponenten, die keine Benutzereingabe erfordern, jedoch zu gegebenen Zeitschritten das Modell ändern, werden im Controller gekapselt. Beispielsweise würde eine Physiksimulation, die auf den Daten des Modells arbeitet, als Controller angesehen.

Environment: Die Umgebung ist die Komponente in MVCE, in der die Applikation wenig Einfluss ausüben kann. In dieser Komponente wird nicht die komplette Umwelt abgebildet, sondern nur ein kleiner Teil, der für die Applikation sinnvoll und wichtig ist. Dieser Teil der Umgebung wird meist über Sensoren erfasst und der Darstellung bzw. dem Modell zur Verfügung gestellt. Es gilt, dass alle Komponenten, die mit der Umwelt interagieren, in der Umgebungs-Komponente von MVCE realisiert werden. Als Beispiel kann ein Objekt, welches über einen visuellen Tracker erfasst wird, seine Position ändern. Der Tracker, der die Verbindung zur Umgebung darstellt, teilt diese Änderung dem Modell mit, das seinerseits die Daten aktualisiert. Keine der anderen Komponenten hat Einfluss auf die Umgebung und kann diese nicht manipulieren. Es ist eine reine „Read-Only"-Komponente, die der Darstellung und dem Modell Daten zur Verfügung stellt. Welche Daten das sind, entscheiden die Sensoren, die die Umgebung analysieren, z. B. Position und Orientierung eines Objektes, Höhe und Entfernung aber auch Geschwindigkeit und Lage im Raum. Letzteres kann gut zur Interaktion von realen Objekten mit der Applikation genutzt werden und wird oft im Bereich „Mixed Reality User Interfaces" eingesetzt.

Durch diese Aufteilung der Komponenten lässt sich die Applikation sehr modular entwickeln. Ein weiterer Vorteil der Einteilung in die vier Komponenten MVCE ist, dass die Software anhand des Enwicklungsstaus der einzelnen Komponenten analysiert werden kann. In Abbildung 4.2$_{109}$ ist ein Kiviatgraph zu sehen, der den Enwicklungsstatus einer Applikation repräsentiert. Hierbei werden die einzelnen Komponenten von der Mitte aus zu den Rändern abgetragen, je nach ihrem aktuellen Status. Im Kiviatgraphen entspricht die Mitte einer sehr geringen Komplexität bzw. Realismus. Die ersten Prototypen einer Applikation werden somit in der Mitte des Kiviatgraphen angesiedelt sein. Je weiter die Komponenten entwickelt werden und komplexer bzw. realer werden, um so weiter entfernt sich die Komponenten aus der Mitte. Das heißt, je weiter eine Komponente zum Rand abgetragen werden kann, desto komplexer bzw. realer ist sie.

Rechnern über Bluetooth angeschlossen werden kann. Über die Wiimote können Beschleunigungen und die Lage des Gamecontrollers im Raum ermittelt werden. So können z. B. Gesten des Benutzers erkannt und als Interaktion genutzt werden.

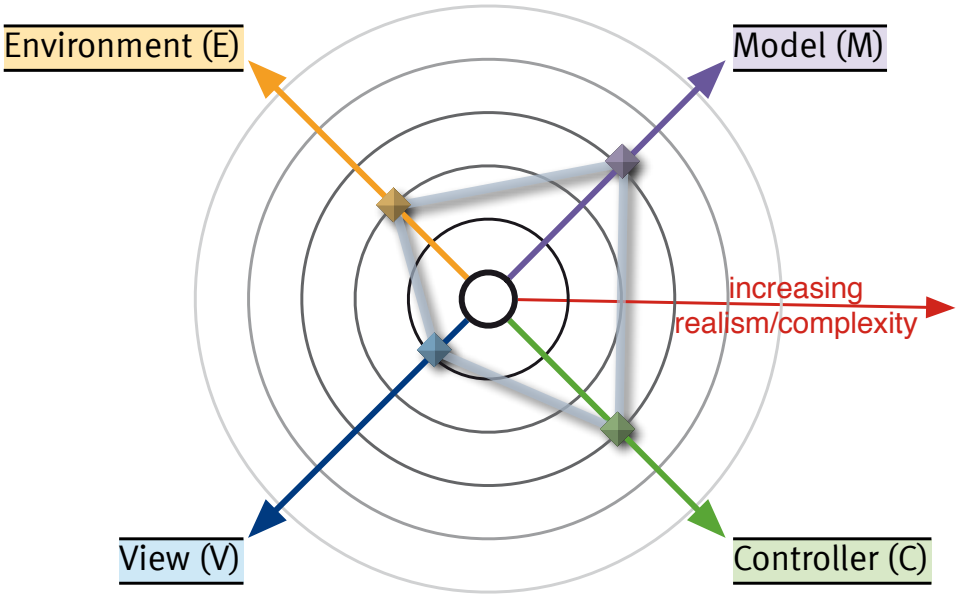


Abbildung 4.2: Kiviatgraph zu Analyse des Entwicklungsstatus.

Der Kiviatgraph ermöglicht somit eine genauere Analyse des Entwicklungsstandes der MVCE-Komponenten. Bei der Entwicklung von Mixed Reality Applikationen kann es öfter sinnvoll sein, von einer komplexen zurück auf eine etwas einfachere Implementation einer einzelnen Komponente zurück zu gehen, um beispielsweise eine andere komplexere Komponente sicher testen zu können bzw. Seiteneffekte auszuschließen. Über den Kiviatgraphen kann man diese beiden Prototypen der Applikation gut unterscheiden, da sich die einzelnen aufgetragenen Komponenten ändern, anders als hätte man nur einen Wert, der die Entwicklung beschreibt, wie das bei einer normalen Softwareentwicklung der Fall ist (z. B. über eine Versionsnummerierung). Theoretisch könnte der Kiviatgraph auch noch feingranularer aufgezeichnet werden, beispielsweise über alle Akteure (siehe Kapitel $4.3.3_{120}$) in der Applikation. Diese Information wäre in einigen Situationen sinnvoll, im Allgemeinen reicht allerdings der allgemeine Stand der MVCE-Komponenten, da auch diese Informationen einfacher zu lesen sind.

Für die Bewertung des Entwicklungsstatus der einzelnen Komponenten war es allerdings notwendig, eine geeignete Metrik zu definieren. Diese MRiL-Metrik wird im folgenden Kapitel beschrieben.

4.3.2 Die MRiL-Metrik

Bei der Bewertung des Entwicklungsstatus der Applikation kann der in Kapitel $4.3.1_{104}$ vorgestellte Kiviatgraph genutzt werden, Um eine

definierte Aussage treffen zu können, wird für jede Achse des Kiviatgraphen eine eigene Metrik benötigt, die aussagt, in wie weit die Applikation in der entsprechenden Dimension entwickelt ist. Detailliert werden folgende Metriken benötigt:

Modell-Metrik: Ein Maß, welches angibt, in wie weit das Modell der Applikation entwickelt ist.

View-Metrik: Metrik für die visuelle Komponente einer Applikation.

Controller-Metrik: Eine Einheit zur Beschreibung der Entwicklung des Controllers.

Environment-Metrik: Metrik für die Einordnung der Umgebung in die Applikation.

Für jedes dieser Maße wird eine eigene Metrik benötigt, da sich die Methoden zur Bestimmung für jede Dimension im Kiviatgraphen zum Teil grundlegend unterscheiden. Angelehnt sind die Metriken an Arbeiten aus dem Bereich der Wiederverwendbarkeit von Software-Komponenten, u. a. die Arbeiten von Boxall et al. [BA04] und Washizaki et al. [WYF03]. Dabei werden Daten, die schon aus der finalen Quelle stammen, höher gewertet als simulierte Daten. Daten, die nur für bestimmte Prototypen benutzt werden und in der finalen Applikation nicht vorhanden sind (hier virtuelle Daten genannt), werden komplett unberücksichtigt, da diese Daten eigentlich nicht wiederverwendet werden können.

Modell-Metrik

Um die Metrik für das Modell zu bestimmen, benötigt man eine Definition des Modells der finalen Applikation. In diesem Kontext kann das Modell folgendermaßen definiert werden:

Definition 4.3.2.1 – Modell

Ein Modell ist die Summe aller Daten σ_n, die es der Applikation zur Verfügung stellt bzw. die die Applikation auslesen und/oder verändern kann. Die Daten können dabei virtuell, simuliert oder real sein (ϵ_i), was aussagt, ob die Daten aus der endgültigen Quelle stammen (real) oder noch in irgendeiner Form simuliert werden (simuliert). Sollten die Daten im späteren Modell nicht existieren, so sind sie virtuell.

$$M := \sum_{i=0}^{n} \epsilon_i \tag{4.1}$$

n : Anzahl der Daten von σ_n

$$\epsilon_i = \begin{cases} 0 & : \ \sigma_i \text{ ist ein virtuelles Datum} \\ 0,5 & : \ \sigma_i \text{ ist ein simuliertes Datum} \\ 1 & : \ \sigma_i \text{ ist ein reales Datum} \end{cases}$$

Das Modell wird hier auf einer objektorientierten Ebene definiert und beinhaltet nicht Parameter wie beispielsweise die Anzahl der Zeilen im Quelltext. Da das Modell abstrakt angesehen werden soll, muss auch die Definition und die daraus resultierende Metrik abstrakt gehalten werden. Dies ist mit der Definition 4.1 des Modells erreicht worden. Es werden hier nur die Ein- bzw. Ausgaben eines Modells betrachtet, also die Schnittstellen zur Applikation. Des Weiteren wird dabei berücksichtigt, ob die Daten, die zur Verfügung gestellt werden, schon den endgültigen Daten entsprechen.

Um nun eine passende Metrik für das Modell zu definieren, betrachtet man das Modell der finalen Applikation und vergleicht es mit dem zur Zeit vorhandenen Modell. Die Modell-Metrik lässt sich daher folgendermaßen definieren:

Definition 4.3.2.2 – Modell-Metrik

Die Model-Metrik Γ_M ist Quotient vom vorliegenden Modell zum finalen Modell.

$$\Gamma_M := \frac{M_{proto}}{M_{final}} \quad , 0 \leq \Gamma_M \leq 1 \tag{4.2}$$

M_{proto} : Modell des aktuellen Prototypen

M_{final} : finales Modell

Mit der Definition des Modells durch 4.1 wird sicher gestellt, dass Prototypen nie einen größeren Wert erhalten als das finale Modell. Somit liegt der skalare Wert der Modell-Metrik Γ_M immer zwischen 0 und 1. Dieser Wert kann auf der Modell-Achse des Kiviatgraphen abtragen werden, wobei 0 dem Punkt auf dem innersten und 1 dem Punkt auf dem äußersten Kreis des Kiviatgraphen entspricht, wie es an Abbildung 4.3$_{112}$ dargestellt ist.

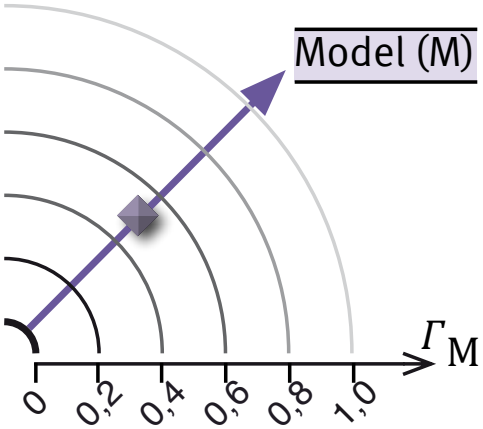


Abbildung 4.3: Modell-Metrik dargestellt im Kiviatgraphen.

Sollte sich das finale Modell ändern, da es z. B. in der weiteren Entwicklung erweitert wird, müssen die Werte für Γ_M neu berechnet werden, da sonst ein Vergleich mit älteren Versionen des Modells nicht möglich ist.

Ein andere Möglichkeit bei der Weiterentwicklung und Erweiterungen von Prototypen ist die Version des Prototypen mit in die Modell-Metrik Γ_M einfließen zu lassen.

Definition 4.3.2.3 – Modell-Metrik (Weiterentwicklung)

Die Modell-Metrik Γ_{M_n} ist Quotient einer Weiterentwicklung des Modells $(n-1)$ zum finalen weiterentwickelten Modell n.

$$\Gamma_{M_n} := (n-1) + \frac{M_{proto}}{M_{final_n}} \quad , n-1 \leq \Gamma_M \leq n, \quad n \in \mathbb{N}^+ \tag{4.3}$$

M_{proto} : aktuelles Modell auf Basis vom Modell (n-1)

M_{final_n} : finales weiterentwickeltes Modell n

Der Wert der Modell-Metrik Model-Metrik Γ_{M_n} liegt nun zwischen dem Vorgängermodell $(n-1)$ und der finalen Weiterentwicklung n. Das bedeutet für die grafische Repräsentation mit Hilfe des Kiviat-

graphen, dass ein neuer Bereich (von $(n-1)$ - n) hinzugefügt werden muss, wie in Abbildung 4.4[113] zu sehen ist.

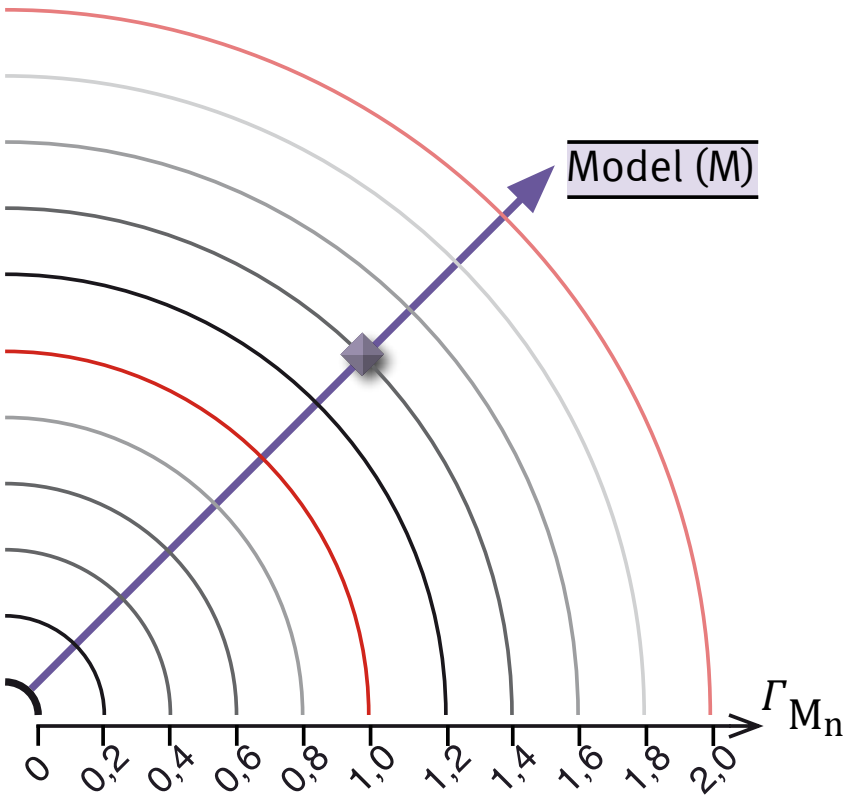


Abbildung 4.4: Kiviatgraph eines weiterentwickelten Modells.

Problematisch ist bei mehreren Weiterentwicklungen, dass der Kiviatgraph immer größer und daher auch unübersichtlicher wird. Um dem vorzubeugen, kann auch nur die Entwicklung des aktuellen Prototypen verwendet werden. Damit können jedoch nur die Entwicklungen der aktuellen Entwicklungsstufe miteinander verglichen werden.

View-Metrik

Ähnlich der Modell-Metrik kann auch die View-Metrik definiert werden. Zuvor muss jedoch der View definiert werden, um darauf danach eine Metrik anwenden zu können. Dabei muss der View der finalen Applikation bekannt sein, da die Klassifizierung der Objekte davon abhängt.

Definition 4.3.2.4 – View

Der View ist die Summe aller Objekte ω_n, die von der Applikation visualisiert werden. Diese Objekte können dabei bezüglich ihrer Existenz in folgende Klassen aufgeteilt werden: temporär, virtuell oder real (ϕ_i). Objekte, die genau so in der finalen Applikation vorhanden sind, bezeichnet man als real. Objekte, die zwar in der finalen Applikation vorhanden sind, allerdings in irgendeiner Weise anders dargestellt werden, bezeichnet man als virtuell. Sollten Objekte nur in Zwischenversionen und nicht in der finalen Applikation vorhanden sein, so nennt man diese temporär.

$$V := \sum_{i=0}^{n} \phi_i \tag{4.4}$$

n : Anzahl der Objekte von ω_n

$$\phi_i = \begin{cases} 0 & : \ \omega_i \text{ ist ein temporäres Objekt} \\ 0,5 & : \ \omega_i \text{ ist ein virtuelles Objekt} \\ 1 & : \ \omega_i \text{ ist ein reales Objekt} \end{cases}$$

Auch bei dieser Definition des Views wird ein abstraktes Maß zugrunde gelegt. Dabei kann sich die Granularität der einzelnen Objekte sehr stark unterscheiden, je nachdem, in welchen Fokus sie stehen. Dabei ist indes zu beachten, dass sich die Granlarität während der Entwicklung zur Finalen Version nicht ändern darf, da sonst keine eindeutige Metrik berechnet werden kann. Aus der Definition des Views ergibt sich folgende Metrik:

Definition 4.3.2.5 – View-Metrik

Die View-Metrik Θ_V ist Quotient des vorliegenden Views zum finalen View.

$$\Theta_V := \frac{V_{proto}}{V_{final}} \quad , 0 \leq \Theta_V \leq 1 \tag{4.5}$$

V_{proto} : View des aktuellen Prototypen

V_{final} : finaler View

Genau wie bei der Definition des Modells wird auch bei der Definition des Views in 4.4 darauf geachtet, dass keine Zwischenversion einer Applikation einen Wert $V \geq 1$ bezüglich der finalen Applikation bekommen kann. Dieser Wert kann dann im Kiviatgraphen abgetragen

werden, genau wie es auch beim Modell in Abbildung 4.3$_{112}$ zu sehen ist.

Für Weiterentwicklungen der finalen Version der Applikation kann die View-Metrik genauso erweitert werden, wie die Modell-Metrik:

Definition 4.3.2.6 – View-Metrik (Weiterentwicklung)
Die View-Metrik Θ_{V_n} ist Quotient einer Weiterentwicklung des Views $(n-1)$ zum finalen weiterentwickelten View n.

$$\Theta_{V_n} := (n-1) + \frac{V_{proto}}{V_{final_n}} \quad , n-1 \leq \Theta_V \leq n, \quad n \in \mathbb{N}^+ \tag{4.6}$$

V_{proto} : aktueller View auf Basis vom View (n-1)

V_{final_n} : finaler weiterentwickelter View n

Θ_{V_n} liegt nun zwischen $(n-1)$ und n und kann genau wie der Wert der Modell-Metrik für Weiterentwicklungen auf dem Kiviatgraphen abgetragen werden.

Controller-Metrik

Der Controller kann sich von einem Prototyp zu anderen sehr stark unterscheiden, da genau hier die unterschiedlichsten Bedienungskonzepte implementiert und getestet werden. Daher kann der Controller nicht ähnlich dem Modell oder dem View definiert werden. Die implementierten Controller-Strategien können nicht nach ihrer Komplexität gemessen werden, da dies keine Aussage über die Qualität der Strategie aussagt. Somit muss der Controller eine Metrik erhalten, die mehr die Benutzbarkeit widerspiegelt.

Für den Controller ist es daher sinnvoll, sich an der Usability-Metrik nach Jakob Nielsen [Nie93] [Nie94] anzulehnen. Bei der Usability-Metrik nach Nielsen sind die Erfolgsrate, die Ausführungszeit und die Fehlerrate die wichtigsten Kenngrößen zur Einordnung des Controllers. Optional kann die Zufriedenheit und die Erlernbarkeit zur Metrik hinzugezogen werden. Gerade bei neu entworfenen Benutzerschnittstellen, bei denen noch nicht abzusehen ist, in wie weit sie vom Anwender verstanden werden und effizient verwendbar sind, ist eine Erhebung der optionalen Kenngrößen vorteilhaft, um so die Akzeptanz der Benutzer zu erfahren.

Im speziellen sollten folgende Kriterien die Metrik bestimmen:

Obligatorisch

Erfolgsrate: Ist es dem Benutzer gelungen, eine gegebene Aufgabe überhaupt zu lösen?

Ausführungszeit: Wie lange hat der Benutzer benötigt, um eine gegebene Aufgabe zu bewältigen?

Fehlerrate: Wie viele Fehler hat der Benutzer bei der Bewältigung der Aufgabe begangen?

Optional

Zufriedenheit: Wie zufrieden ist der Benutzer mit dem Controller?

Erlernbarkeit: Wie viel Zeit hat der Benutzer benötigt, um sich die Steuerung anzueignen?

Die ersten drei Kriterien können mit Hilfe entsprechend vorbereiteter Tests objektiv gemessen werden. Im Gegensatz dazu ist die Zufriedenheit eine subjektive Größe, die je nach Vorlieben des Benutzers extrem unterschiedlich ausfallen und nicht objektiv bestimmt werden kann. Dementsprechend sollte die Zufriedenheit des Benutzers optional berücksichtigt und eine geringere Gewichtung als die anderen Kenngrößen erfahren. Allerdings kann bei Tests mit vielen Benutzern eine Tendenz der Zufriedenheit abgeleitet werden, die besonders bei neuen Benutzerschnittstellen vorteilhaft ist. Die Erlernbarkeit des Controllers sollte auch nur optional behandelt werden, da auch diese Bestimmung subjektiv ist, da jeder Benutzer sowohl verschiede Vorkenntnisse als auch unterschiedliches Lernverhalt besitzt. Bei sehr komplizierten Steuerungen sollte allerdings auf diese Messung nicht verzichtet werden, da auch hier Tendenzen sichtbar sind.

Definition 4.3.2.7 – Controller

Der Controller ist für die Steuerung der Applikation zuständig. Da sich die unterschiedlichen Arten der Controller jedes Prototyps essenziell unterscheiden können, existiert keine mathematische Definition eines Controllers wie es bei dem View oder beim Modell der Fall ist. Die Metrik wird über Benutzertests ermittelt.

Für die Controller-Metrik können die aus den Usability-Tests ermittelten Werte verwendet werden, um einen eindeutigen Wert zu bekommen, so dass Controller über die Controller-Metrik miteinander verglichen werden können:

Definition 4.3.2.8 – Controller-Metrik

Die Contoller-Metrik Ψ_C ist die Summe der Ergebnisse der unterschiedlichen Benutzertests geteilt durch die Anzahl der durchgeführten Benutzertests.

$$\Psi_C := \sum_{i=0}^{n} \frac{\alpha_{C_i}}{n} \quad , 0 \leq \Psi_C \leq 1, \quad n \in \mathbb{N}^+ \tag{4.7}$$

n : Anzahl der unterschiedlichen Usability-Tests

α_{C_i} : Ergebnis des Usability Tests C_i

Dabei ist α_{C_i} folgendermaßen definiert:

$$\alpha_{C_i} := 1 - \frac{1}{m} \sum_{j=1}^{m} \zeta_j \tag{4.8}$$

$$\text{mit } \zeta_i := \frac{\xi_i - min_{k=1}^{m}(\xi_k)}{max_{k=1}^{m}(\xi_k) - min_{k=1}^{m}(\xi_k)}, \quad m \in \mathbb{N}^+ \tag{4.9}$$

ξ_i : nicht normierter Messwert i des Usability-Test

ζ_i : normierter Messwert i des Usability-Test, $0 \leq \zeta_i \leq 1$,

m : Anzahl der ermittelten Werte in einem Usability-Test

Die Controller-Metrik Ψ_C aus 4.7 liefert einen Wert zwischen 0 und 1, der, wie auch die Werte der anderen Metriken, auf dem Kiviatgraphen dargestellt werden können. Dabei werden die Daten aus den Benutzertests normiert und der arithmetische Mittelwert mit Hilfe der Gleichung 4.8 bestimmt. Die Normierung geschieht über die Formel 4.9, die das Minimum und das Maximum der Messwerte bestimmt und so den aktuellen Messwert in den Bereich von $[0..1]$ legt.

Bei der Angabe zur Zufriedenheit des Benutzers muss darauf geachtet werden, wie die Daten, die der Benutzer macht, gewertet werden. Würde das deutsche Schulnotensystem als Grundlage genommen werden, bei dem eine 1.0 eine sehr hohe Zufriedenheit und eine 6.0 eine ungenügende Zufriedenheit des Benutzers ausdrückt, stimmt die Berechnung mit Hilfe der Formel 4.8. Bei Bewertungssystemen, bei denen ein höherer Wert eine größere Zufriedenheit darstellt, muss die Formel allerdings entsprechend umgestellt werden:

Definition 4.3.2.9 – Controller-Metrik mit inverser Gewichtung

Bei Bewertungssystemen, bei denen ein höherer Wert eine bessere Leistung ausdrückt, muss die Formel 4.8 zur Berechnung des arithmetischen Mittels geändert werden.

$$\alpha_{C_i} := \frac{1}{m} \sum_{j=1}^{m} \zeta_j \tag{4.10}$$

Alle anderen Formeln und Definitionen können beibehalten werden.

Über die Gleichung 4.10 können nun auch Bewertungsysteme verwendet werden, die mit höheren Werten auch eine bessere Leistung bzw. Zufriedenheit ausdrücken. Diese Änderung ist wichtig, damit die Darstellung im Kiviatgraphen analog zu den anderen Metriken ist. So würde ein Wert, der größer als ein anderer ist, immer ein besseres Ergebniss darstellen und im Kiviatgraphen weiter außen dargestellt werden.

Environment-Metrik

Die Environment-Metrik kann größtenteils analog zur View- bzw. Modell-Metrik definiert werden. Dabei kann die Umgebung (Environment) folgendermaßen definiert werden:

Definition 4.3.2.10 – Environment

Das Environment ist die Summe aller Daten η_n, die der Applikation aus der realen Umgebung bereitgestellt werden. Diese Daten können dabei bezüglich ihrer Herkunft als real oder simuliert klassifiziert werden (δ_i). Daten, die aus der realen Umgebung mit Hilfe von Sensoren ausgelesen werden, werden real genannt. Alle anderen Daten können als simuliert angesehen werden.

$$E := \sum_{i=0}^{n} \delta_i \tag{4.11}$$

n : Anzahl der Daten von η_n

$$\delta_i = \begin{cases} 0,5 & : \quad \eta_i \text{ ist ein simuliertes Datum} \\ 1 & : \quad \eta_i \text{ ist ein reales Datum} \end{cases}$$

Bei der Definition des Environments werden die Daten, die der Applikation aus der realen Umgebung geliefert werden, berücksichtig. Je

nach Entwicklungsstand können diese Daten in jeglicher Art simuliert sein oder aus der realen Umgebung stammen. In frühen Entwicklungsphasen werden diese Daten meist simuliert. Für die Applikation ist die Art der Daten, also ob sie simuliert oder real sind, transparent. Simuliert kann in diesem Zusammenhang auch bedeuten, das immer nur ein fester Wert zurück geschickt wird, es wird hier also nicht die Qualität der Simulation mitbewertet. Das ist gewollt, denn es wird hier nicht die Qualität der Simulation der Umgebung betrachtet, sondern der Anteil an realen Daten, die der Applikation bereitgestellt werden.

Aus der Definition des Environments kann nun die Environment-Metrik analog zu der Modell- bzw. View-Metrik definiert werden:

Definition 4.3.2.11 – Environment-Metrik

Die Environment-Metrik Ω_E ist Quotient vom vorliegenden Environment zum finalen Environment.

$$\Omega_E := \frac{E_{proto}}{E_{final}} \quad , 0 \leq \Omega_E \leq 1 \tag{4.12}$$

E_{proto} : Environment des aktuellen Prototypen

E_{final} : finales Environment

Die Definition 4.12 stellt sicher, das die Environment-Metrik Ω_E immer einen Wert zwischen $[0..1]$ erhält. So kann Ω_E auf der entsprechenden Achse des Kiviatgraphen abgetragen werden, wobei auch hier ein größerer Wert einer höheren Entwicklungsstufe entspricht.

Ebenso lässt sich für die Weiterentwicklung eines Prototypen eine Definition der Environment-Metrik definieren:

Definition 4.3.2.12 – Environment-Metrik (Weiterentwicklung)

Die Environment-Metrik Ω_{E_n} ist Quotient einer Weiterentwicklung des Environments $(n-1)$ zum finalen weiterentwickelten Environments n.

$$\Omega_{E_n} := (n-1) + \frac{E_{proto}}{E_{final_n}} \quad , n-1 \leq \Omega_E \leq n, \quad n \in \mathbb{N}^+ \tag{4.13}$$

E_{proto} : aktuelles Environment auf Basis von Environment (n-1)

E_{final_n} : finales weiterentwickeltes Environment n

Ω_{E_n} liegt nun zwischen den Werten $(n-1)$ und n und kann genau wie der Wert der Modell- bzw. View-Metrik für Weiterentwicklungen auf dem Kiviatgraphen abgetragen werden.

Zusammenfassung

Die vier hier vorgestellten Metriken können im MRiL-Entwurfsvorgehen dazu verwendet werden, einen erstellten Prototypen bezüglich seines Entwicklungsgrades zu klassifizieren. Dies geschieht, indem die Metriken ermittelt werden und auf den entsprechenden Achsen des Kiviatgraphen abgetragen werden. Da sich das MRiL-Entwurfsvorgehen sowohl für die schnelle Entwicklung von Prototypen als auch für die Evaluierung von neuen Eingabemethoden eignet, muss für jede dieser beiden Prioritäten der Aufwand zur Bestimmung der einzelnen Metriken mitberücksichtigt werden. Insbesondere die Ermittlung der Controller-Metrik kann viel Zeit und Resourcen an Benutzern kosten, wenn ausführliche Tests an einer großen Gruppe an Anwendern ausgeführt werden sollen. Die Tests und deren Auswertung kann sehr viel Zeit kosten. Deshalb stehen die beiden Ziele „schnelle Prototypenentwicklung“ und „ausführliche Evaluierung neuer Benutzerschnittstellen“ entgegengesetzt zueinander. Für die schnelle Prototypenentwicklung sind keine bzw. nur in einem geringen Maße ausgeführte Benutzertests für die Bestimmung der Controller-Metrik sinnvoll. Hier könnte schon ein Test mit einem der Entwickler reichen. Anders ist es bei der Entwicklung von neuen Benutzerschnittstellen. Hier sollten ausführliche Tests zumindest für die final entwickelten Strategien durchgeführt und ausführlich analysiert werden. So ist es möglich, eine sehr genaue Bestimmung der Controller-Metrik zu erhalten, die dann mit anderen Entwicklungen in diesem Bereich verglichen werden kann.

4.3.3 Das Akteurmodell

Im vorherigen Kapitel wurde das MVCE Architekturmodell vorgestellt, das es erlaubt, die zu entwickelnde Applikation in vier unterschiedliche Komponenten zu unterteilen und diese dann getrennt voneinander zu entwickeln. Um bei der Entwicklung eine noch feinere Granularität der Elemente zu erhalten, basiert das MRiL-Entwurfsvorgehen auf sogenannten Akteuren (engl. Actors), die wiederum unabhängig voneinander entwickelt werden können. Die Verwendung von Akteuren sind beim MRiL-Entwurfsvorgehen optional, d. h. diese Unterteilung ist bei der Entwicklung nicht zwingend erforderlich, allerdings wirkt sie sich positiv auf die Implementierung aus. Da die

Akteure untereinander über festgelegte Ports miteinander kommunizieren, ist eine getrennte Entwicklung der einzelnen Akteure nach Festlegung der Schnittstellen leicht möglich.

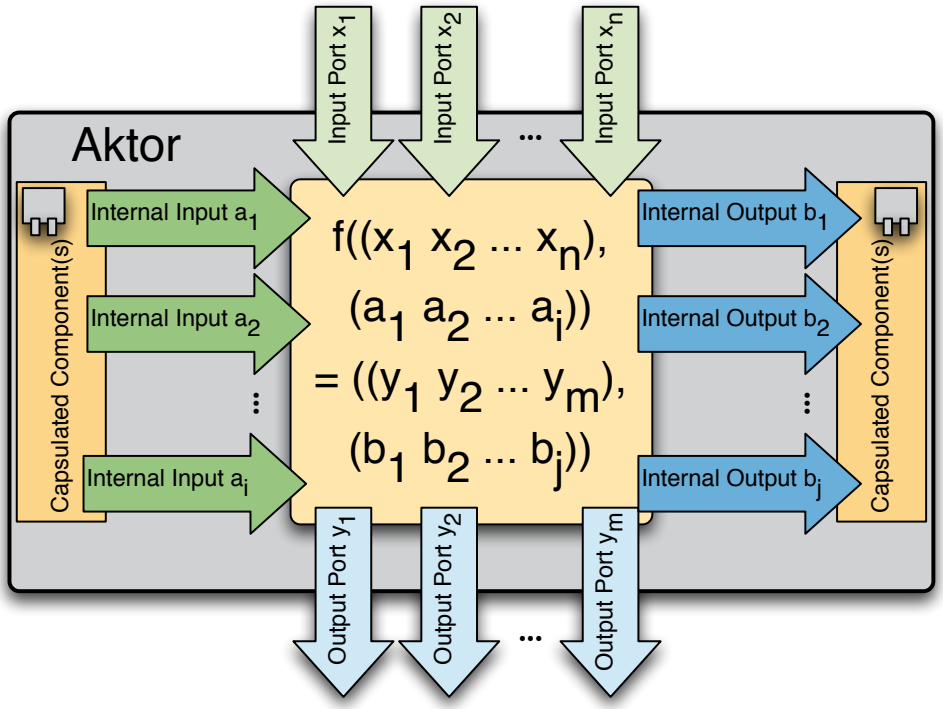


Abbildung 4.5: Abstrakter Aufbau eines Akteurs.

In Abbildung 4.5 ist Aufbau eines Akteurs abstrakt dargestellt. Ein Akteur besteht aus einer Anzahl vom Eingangs-Ports, die skalare Werte, Vektoren oder Matrizen als Wert annehmen können. Der Akteur benutzt diese Werte zur Erzeugung seiner Ausgaben. Dies kann über einfache Transformationen bis hin zu komplizierten Funktionen reichen. Ein Akteur kann des Weiteren noch interne Eingabe-Ports besitzen, die mit verarbeitet werden. Diese internen Eingabe-Ports stammen aus gekapselten Komponenten, beispielsweise dedizierter Hardware oder interne Softwarekomponenten, die über diesen Akteur abgefragt werden kann. Dabei kann ein Akteur sowohl eine einzelne als auch mehrere Komponenten in sich kapseln, die sowohl Ausgaben produzieren als auch Eingaben annehmen. Gerade Akteure, die zur Komponente Controller gehören, besitzen solche internen gekapselten Komponenten, da sie häufig spezielle Hardware kapseln müssen. Die Ausgaben werden einerseits über die Ausgabe-Ports zur Verfügung gestellt und können andererseits über interne Ausgabe-Ports ausgegeben werden. Intern werden die Ausgaben auch wieder an die gekapselten Komponenten übertragen, die beispielsweise eine dedizierte Hardware darstellt. Die internen Ausgabe-Ports sind, genau wie die internen Eingabe-Ports, wichtig für die Kommunikation zwischen

dem Akteur und den gekapselten Komponenten. Die Akteure können nun untereinander verbunden werden, so dass ein Datenflussnetzwerk entsteht. Über eine entsprechende „Verdrahtung“ der einzelnen Akteure entsteht so die Applikationslogik einer Applikation. Dabei wird jeder Akteur einer bestimmten MVCE-Komponente zugeordnet. Es ist zu beachten, dass ein Akteur keinen zwei Komponenten zugeordnet werden darf, damit die Aufteilung in Modell, Darstellung, Umgebung und Controller nicht verletzt wird.

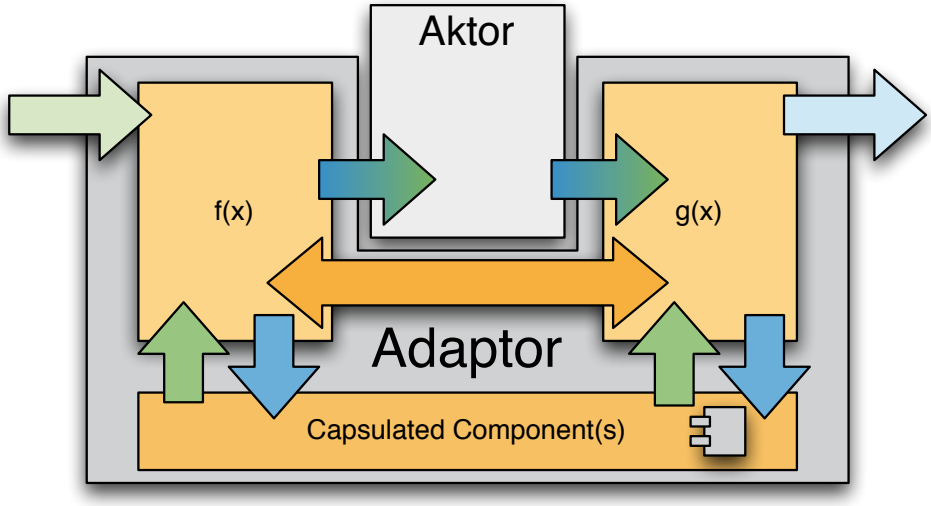


Abbildung 4.6: Abstrakter Aufbau eines Eingabe-Ausgabe-Adapters.

Um die Entwicklung der Akteure zu vereinfachen wurde das Prinzip des Adapters eingeführt. Adapter erweitern die Funktionalität eines Akteurs, indem Sie über ihre interne Logik passende Werte an die Eingabe- bzw. Ausgabeports der Akteure liefern bzw. entgegennehmen. Adapter haben ihrerseits eine Menge an Eingabe- und Ausgabeports, so dass sie für das Datenflussnetzwerk wie normale Akteure wirken. Weiterhin können Adapter auch Komponenten kapseln, die intern Ausgaben erzeugen und Eingaben erwarten. Dabei kann ein Adapter sowohl keine Komponente kapseln, so dass keine internen Ein- und Ausgabe-Ports existieren und ein Adapter auch als Protokollkonverter gesehen werden kann. Es können jedoch auch ein oder mehrere Komponenten in einem Adapter gekapselt werden, wie es auch beim Akteur der Fall ist. Damit sind Erweiterungen von Akteuren denkbar, die ihrerseits keine Komponenten kapseln, allerdings durch einen Adapter spezielle Komponenten zur Verfügung gestellt werden.

Es gibt insgesamt drei unterschiedliche Arten von Adaptern: den Eingabe-Adapter, der nur die Eingaben des jeweiligen Akteurs kapselt und durch eigene Eingabe-Ports ersetzt, den Ausgabe-Adapter, der

die Ausgaben des Akteurs kapselt und den Eingabe-Ausgabe-Adapter, der sowohl die Ein- als auch die Ausgaben des Akteurs kapselt. In Abbildung 4.6 ist der Aufbau eines Eingabe-Ausgabe-Adapters zu sehen. Er nimmt Daten aus dem Datenflussnetzwerk an und berechnet die Eingaben für den gekapselten Akteur über die Funktion $f(x)$. Die Ausgaben, die der Akteur zur Verfügung stellt, wandelt der Adapter anschließend mit der Funktion $g(x)$ um und gibt diese an das Datenflussnetzwerk weiter. Adapter können, genau wie Akteure, interne Ein- und Ausgaben verarbeiten, um so z. B. zusätzliche Hardware anzusprechen. Des Weiteren hat der Eingabe-Ausgabe Adapter die Möglichkeit, Daten zwischen den beiden Funktionen $f(x)$ und $g(x)$ auszutauschen. So hat z. B. die Funktion $g(x)$ die Möglichkeit, auf die Eingabedaten des Adapters zuzugreifen.

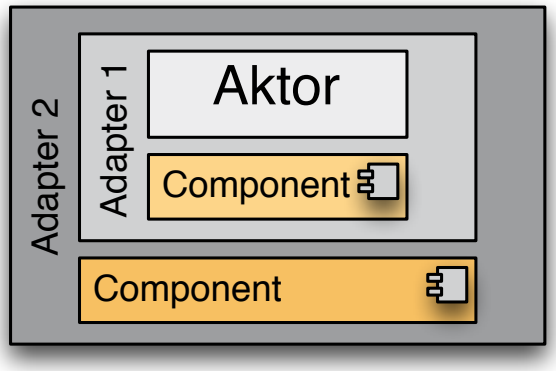


Abbildung 4.7: Schachtelung von Adapern.

Mit dem Prinzip des Adapters lassen sich sehr elegant Spezialisierungen bzw. Verfeinerungen von Akteuren realisieren. So müssen die Akteure nicht komplett neu programmiert werden, um auf neue Daten reagieren zu können. Es reicht aus, einen Adapter vor den Akteur zu schalten, der die passende Transformation der Daten vornimmt. Über Adapter ist es auch möglich, inkompatible Ports in einem Datenflussnetzwerk miteinander zu verbinden. Ein Adapter übernimmt in einem solchen Fall die Transformation der Daten in das für den Akteur geforderte Format. So ist ein Austausch bestimmter Softwarebibliotheken (beispielsweise unterschiedliche Bibliotheken zur Berechnung von physikalischen Effekten) einfach möglich, da das Datenflussnetzwerk zum größten Teil unverändert bleiben kann.

Ein über einen Adapter erweiterter Akteur kann von außen betrachtet wiederum als einzelner Akteur betrachtet werden. So ergibt sich die Möglichkeit, mehrere Adapter zu verschachteln, wie in Abbildung 4.7 dargestellt ist. So ist es möglich, einen Akteur, der zu Anfang nur Grundfunktionalität anbot, schrittweise über iterative Kapselung von

Adaptern allmählich Funktionalität zuzufügen. Dies entspricht dem Prinzip der Vererbung in der objektorientierten Programmierung, bei der auch einem Objekt mit jeder weiteren Vererbung mehr Funktionalität zugefügt wird. Diese Möglichkeit der iterativen Verfeinerung bzw. Erweiterung eines Akteurs finden wir im Grundprinzip vom MRiL wieder, da dieser Prozess auf einem iterativen Prozess aufsetzt. Somit können in einem Iterationsschritt die Verfeinerungen der Akteure über die Verschachtelung von Adaptern gelöst werden. Über dieses Prinzip lassen sich sehr kleine Iterationszyklen erreichen, so dass die Erstellung von Prototypen zu Testzwecken schnell und mit wenig Aufwand realisiert werden kann.

4.3.4 Das Entwurfsvorgehen

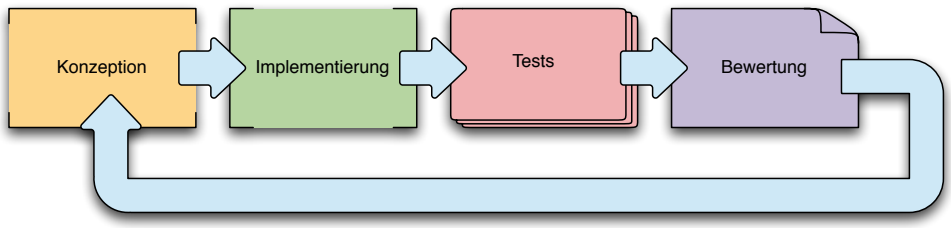


Abbildung 4.8: Abstrakte Übersicht des iterativen Prototyping Prozess.

Wie schon in Kapitel 4.2_{101} kurz erwähnt, ist die Grundlage des MRiL-Entwurfsvorgehen ein iterativer Prototyping Prozess. Im Wesentlichen basiert dieser Prozess auf dem von Pomberger vorgestellten Prototyping Entwicklungsprozess [PW94], der in Kapitel $2.1.12_{39}$ vorstellt wurde. Es wurden allerdings einzelne Phasen anders definiert. Die Grundlage der kurzen Iterationen stammt aus dem Scrum Vorgehensmodell [BDS+99], das in Kapitel $2.1.11_{34}$ vorgestellt wurde. Abbildung 4.8 zeigt die abstrakte Übersicht des gesamten Prozesses. Der Prozess ist in vier Phasen unterteilt, die aus der Konzeptionierung, der Implementierung, der Testphase und der Bewertungsphase bestehen. Diese Phasen werden in mehreren Iterationen durchlaufen.

In der detaillierteren Ansicht des Prozesses in Abbildung 4.9_{125} ist der konkrete Aufbau des Prozesses zu erkennen. Die einzelnen abstrakten Phasen sind hier in die konkreten Phasen eingebettet worden. So wird die Konzeptionierung in der Initialisierungsphase des Prozesses bearbeitet. Die Implementierung geschieht in der Verfeinerungsphase, die Tests werden in der Prototypphase durchgeführt. Die Bewertung der Tests werden in der gleichnamigen Bewertungsphase ermittelt. Im Einzelnen haben die vier Phasen folgende Aufgaben:

Initialisierung: In der Initialisierungsphase wird der Prototyp auf

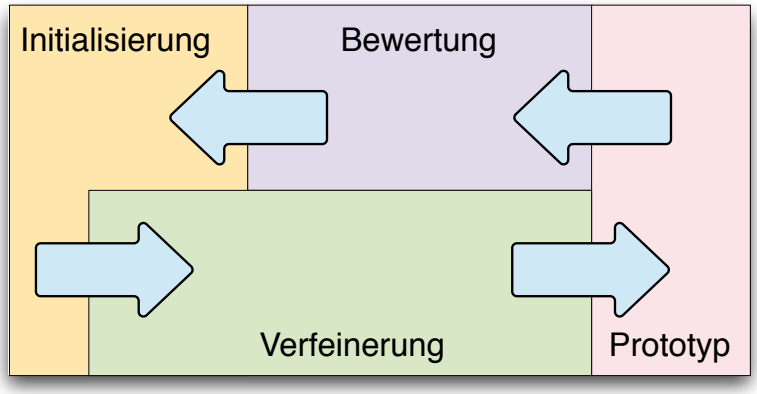


Abbildung 4.9: Konkreter Aufbau des iterativen Prototyping Prozess.

den nächsten Verfeinerungsschritt vorbereitet. Zu Beginn der Entwicklung, wenn noch kein Prototyp aus einer früheren Iteration vorliegt, wird in dieser Phase das grundlegende Verhalten des ersten Prototypen festgelegt.

Verfeinerung: Am existierenden Prototypen werden die Verfeinerungen vorgenommen. Verfeinerung kann bedeuten, dass eine Komponente komplexer wird, dass sie aufgeteilt wird bzw. dass mehrere Teiler einer Komponente verschmolzen werden. Dasselbe gilt selbstverständlich auch für Akteure, falls die Applikation in solche aufgeteilt wurde. Da zu Beginn der Entwicklung noch kein Prototyp aus einer vorherigen Phase vorhanden ist, wird hier das grundlegende Verhalten, wie es in der Initialisierung festgelegt wurde, implementiert. Meist wird in dem ersten Prototypen der Funktionsumfang und die Komplexität sehr einfach gehalten.

Prototyp: In dieser Phase sind alle vorangegangenen Verfeinerungen abgeschlossen und ein neuer Prototyp steht zur Verfügung. Mit diesem Prototypen können nun Benutzer- und Useablility-Tests durchgeführt und analysiert werden. Der entstandene Prototyp gilt als Basis für weitere Iterationen.

Bewertung: In dieser Phase werden die Ziele des nächsten Prototypen anhand der Bewertung der zuvor durchgeführten Tests festgelegt und die daraus folgenden Verfeinerungen definiert. Sollten sich vorhandene Schnittstellen ändern oder neue hinzu kommen, werden sie hier definiert.

Nach diesem groben Überblick werden nun die einzelnen Phasen detailliert beschrieben.

Die Initialisierungsphase

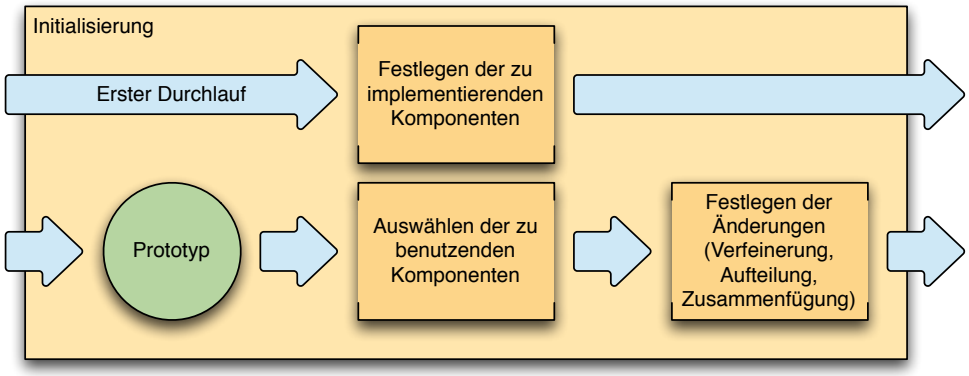


Abbildung 4.10: Detailansicht der Initialisierungsphase.

Die Initialisierungsphase wird zur Vorbereitung der Verfeinerungsphase benötigt. Hier wird festgelegt, welche Komponenten verfeinert werden sollen und auf welchen zuvor schon implementierten Komponenten aufgebaut wird. Zu Beginn der Entwicklung einer Applikation wird in dieser Phase das Verhalten des ersten Prototyps festgelegt, welches dann in der Verfeinerungsphase implementiert wird. Diese Phase sollte nach Möglichkeit nicht viel Zeit verbrauchen. Das kann man erreichen, wenn nur wenig während der Verfeinerungsphase verändert wird. Es sollte jedoch nicht zu wenig verändert werden, da sich dann die Prototypen nicht weit genug unterscheiden und so keine sinnvollen Schlüsse aus der Entwicklung gezogen werden können.

In dieser Phase wird konzeptionell an der Applikation entwickelt, Implementierungen sind nicht vorgesehen. Es werden die Vorraussetzungen für die nächste Verfeinerungssphase geschaffen. Im ersten Durchlauf wird die Grundfunktionalität des ersten Prototypen festgelegt. Da hier noch nicht auf einen aus einer vorherigen Phase stammenden Prototypen zurückgegriffen werden kann und auch nicht auf schon vorhandene implementierte Komponenten, sollten der Funktionsumfang und die Komplexität recht einfach gehalten werden, um die Implementierungsphase zeitlich kurz zu halten.

Die Verfeinerungssphase

In der Verfeinerungsphase werden die konzeptionellen Änderungen und Verfeinerungen, die in der Initialisierungsphase entwickelt wurden, implementiert. Es werden zuerst die Komponenten in die Applikation eingebunden, die für den jeweiligen Schritt benötigt werden.

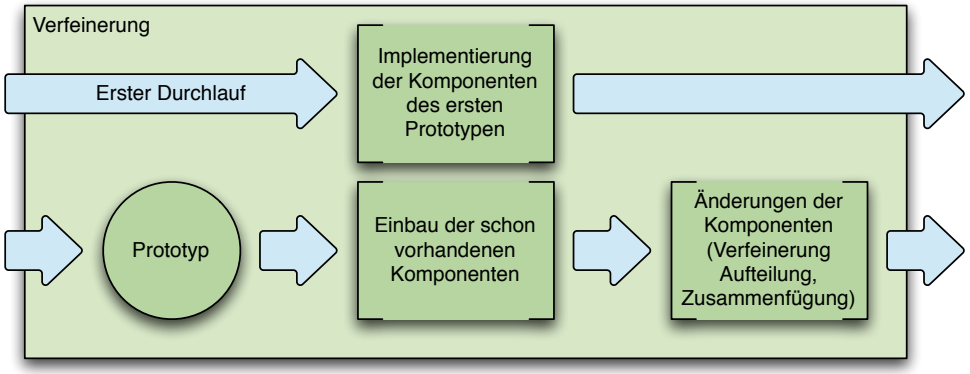


Abbildung 4.11: Detailansicht der Verfeinerungsphase.

Diese werden dann je nach Konzept erweitert, verfeinert, aufgeteilt oder zusammengefügt.

Im ersten Durchlauf des Prozesses werden hier die Teile für den ersten Prototypen implementiert. Normalerweise sollten zu diesem Zeitpunkt noch keine Verfeinerungen der einzelnen Komponenten entwickelt werden, da Ziel des ersten Durchlaufs die Fertigstellung eines rudimentären Prototypen ist.

Diese Phase benötigt die meiste Zeit im Entwurfsvorgehen, kann jedoch von mehreren Entwicklern gleichzeitig bearbeitet werden (vorausgesetzt es wird an mehr als einer Stelle entwickelt).

Die Prototypphase

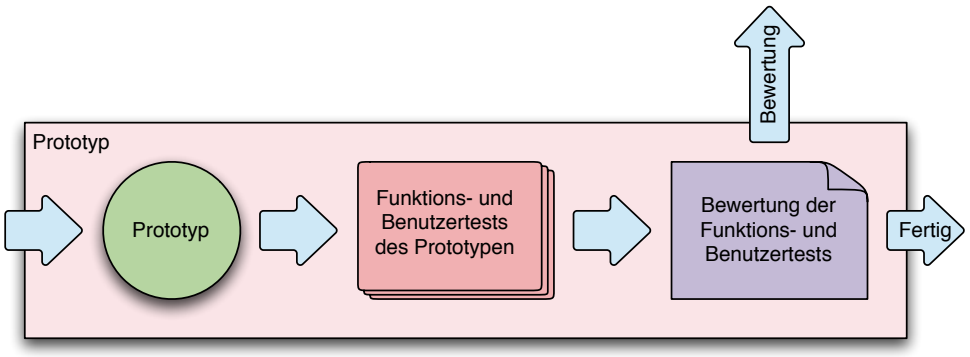


Abbildung 4.12: Detailansicht der Prototypphase.

Die Prototypphase wird dazu verwendet, den Prototypen, der in der Verfeinerungsphase entwickelt wurde, zu testen. In dieser Phase können Benutzertests durchgeführt werden und die neuen bzw. verfeinerten Komponenten auf ihre Tauglichkeit getestet werden. Kom-

ponenten, die in der Verfeinerungsphase weiter entwickelt wurden, sollten in der Prototypphase genauestes untersucht werden, sowohl auf ihre funktionale Korrektheit als auch auf die Benutzbarkeit. Hierzu lassen sich sehr gut Benutzertests verwenden, die dann über die Qualität der Applikation Aufschluss geben. Die Ergebnisse dieser Tests können in der Bewertungsphase ausgewertet und für den nachfolgenden Durchlauf in die Entwicklung mit eingebracht werden.

Sollte der Prototyp sein Endstadium erreicht haben, kann in der Prototypphase die Endkontrolle der Funktionalität und Benutzbarkeit ausführlich getestet werden. Nach erfolgreichen Tests ist die Applikation fertig und kann verwendet werden. Sollten spätere Änderungen gewünscht werden, können einfach weitere Iterationen verwendet werden, um diese Änderungswünsche zu realisieren.

Die Bewertungsphase

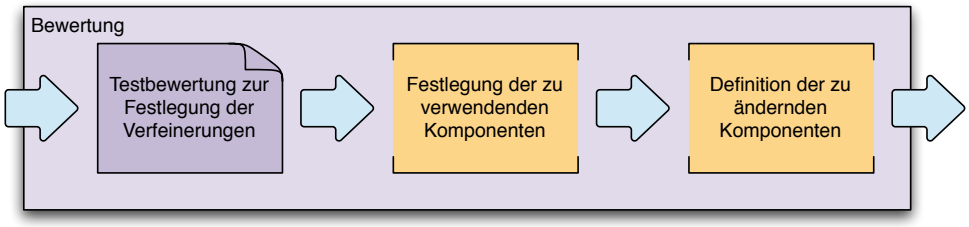


Abbildung 4.13: Detailansicht der Bewertungsphase.

In der Bewertungsphase werden die Ergebnisse der Prototypphase ausgewertet und die Ziele des nächsten Prototypen festgelegt. Es wird in dieser Phase entschieden, welche Komponenten geändert werden sollen und ob auf bereits vorhandene, allerdings weniger verfeinerte Komponenten bei der nächsten Iteration zurückgegriffen werden soll.

Nachdem der Entwurfsprozess erläutert wurde, ist es sinnvoll, diesen an einem kleinen Beispiel exemplarisch durchzuführen. Nachfolgend stelle ich ein solches Beispiel vor und beschreibe, wie hier MRiL eingesetzt wurde.

4.4 Erläuterung des Entwurfsvorgehens an einem Beispiel

Die zuvor vorgestellten Methoden wurden an einem kleineren Beispiel angewendet, um deren Anwendbarkeit zu testen. Es wurde

hier noch keine spezielle Softwareumgebung verwendet sondern das Beispiel wurde in reinem Java implementiert. Realisiert wurde das Beispiel im Rahmen eine Bachelorarbeit, die zum Ziel hatte, mehrere wissensbasierte Verfahren zur Wegeplanung zu entwerfen und zu vergleichen [Bol10]. Um das MRiL-Entwurfsvorgehen auf diese Aufgabe anwenden zu können, wurde diese Applikation auf einem Multitouch-Tisch (siehe Abbildung 4.14$_{129}$) implementiert und eine neue Bedienung hinzugefügt [SBK+10].

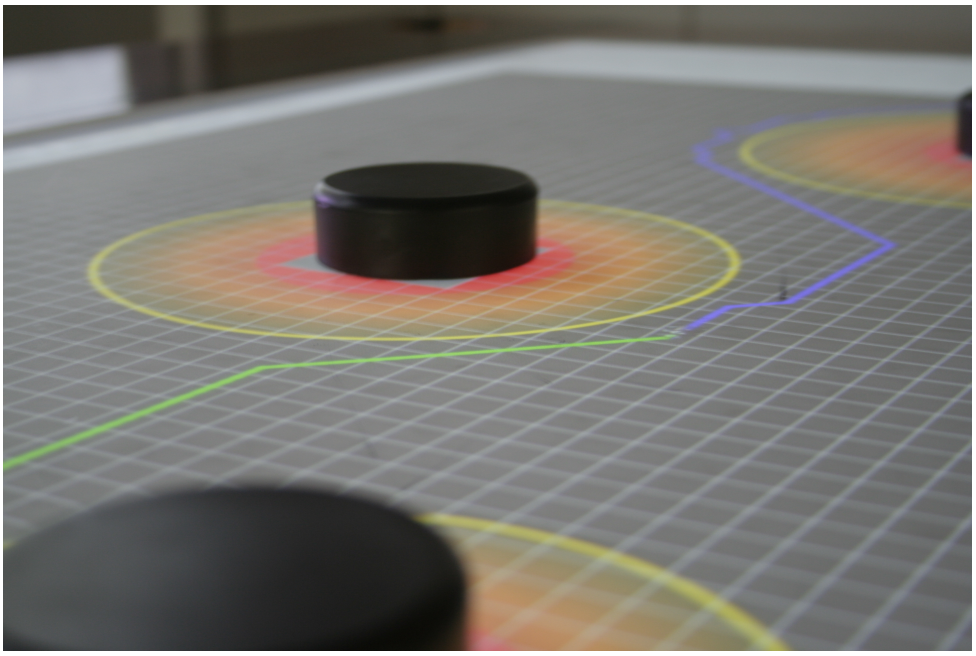


Abbildung 4.14: Beispielapplikation auf einem Multitouch-Tisch.

4.4.1 Überblick des Beispiels

Als Aufgabe für die Bachelorarbeit wurde der Entwurf und der Vergleich wissensbasierter Verfahren zur Wegeplanung gestellt. Die Idee war hierbei, verschiedene Arten von Wegeplanungsalgorithmen mit dem Hintergrund zu testen, dass autonome Roboter einen kurzen, nicht gefährlichen Weg in einer gegebenen Werkshalle finden sollten. Dabei sollte zum einen die Wegstrecke und zum anderen die Gefahren der Wegstrecke berücksichtigt werden. So wurden stationäre Fertigungsanlagen definiert, die einen gewissen Raum immer für sich beanspruchten, allerdings teilweise in freie Gebiete hineinragen konnten. Die autonomen Roboter können diesen Raum für ihre Wegeplanung nutzen, müssen allerdings warten, falls die Fertigungsanlagen diesen Raum zeitweise belegen. Daher kann es zwar einen sehr kurzen Weg durch eine Werkshalle geben, jedoch durch die Wartezeiten der autonomen Roboter muss dies nicht der schnellste Weg

sein.

Die Arbeit sollte auf einem schon vorhandenen Software-Framework der Hochschule Harz aufgebaut werden [SGR08]. Das Framework bot eine schon vorhandene Realisierung einer 3D Darstellung samt Management und Import verschiedener 3D Modelle und Texturen aus entsprechenden Dateien an. Eine Wegeplanung war grundsätzlich implementiert, sollte allerdings im Rahmen der Arbeit neu entwickelt werden. In der Bachelorarbeit sollte zunächst der Framework um passende Schnittstellen erweitert und im zweiten Schritt verschiedene Wegeplanungen implementiert werden. Konzeptionell wurde die Applikation soweit entwickelt, dass sie augmentiert in der realen Umgebung benutzt werden sollte, wobei die Hindernisse, die von der Wegeplanung umgangen werden sollten, über reale Objekte, sogenannte „Tangibles", realisiert wurden. Implementiert wurde allerdings nur soweit, dass die Applikation auf einem Multitouch-Tisch lauffähig war.

Das Konzept, dass die Grundlage der Anwendung bildete, wurde mit Hilfe des MRiL-Entwurfsvorgehens realisiert. Dabei wurden folgende Teile der Applikation den entsprechenden MVCE-Komponenten zugeteilt:

Model: Jeder implementierte Algorithmus zur Wegeplanung wird der Modell-Komponente zugeordnet.

View: Sowohl eine 2D- als auch eine 3D-Ansicht der Szene wurde dem View zugeordnet.

Controller: Der Controller beinhaltet alle Interaktionsmethoden, angefangen von einfachen Klicks mit der Maus bis zu späteren Interaktionen mit *Tangibles*.

Environment: Die reale Umwelt, die der Applikation zur Verfügung gestellt wurde. Anfangs wurde kein Teil der realen Umgebung erkannt, später wurden die *Tangibles* in der realen Umgebung zur Positionierung verwendet. In der AR-Version sollten noch weitere Aspekte der Umgebung verwendet werden.

Auf eine weitere Einteilung in Akteure wurde in diesem Beispiel verzichtet, da dies vom eingesetztem Software-Framework nicht unterstützt wurde.

4.4.2 Realisierung des Beispiels

Zu Beginn der Entwicklung wurde das vorhandene Software-Framework auf die bevorstehende Aufgabe vorbereitet und soweit konfiguriert, dass es effizient eingesetzt werden konnte. Nachdem die Teile der Applikation, wie oben beschrieben, in ihre MVCE-Komponenten eingeteilt wurden, begann die Implementierung des ersten Prototypen. Dieser sollte nicht sehr komplex sein und nur sicherstellen, dass Framework und das Zusammenspiel der einzelnen Komponenten funktioniert. Da die 3D Darstellung schon vom Framework bereitgestellt wurde, wurde schon zu Anfang eine sehr detaillierte Darstellung gewählt, wie in Abbildung 4.15$_{131}$ zu sehen ist. Dabei wurde der autonome Roboter (1) sowie die Fertigungsanlagen (5) durch Platzhalter visualisiert, die im Framework vorhanden waren. Die Grundfläche der Werkshalle wurde in der 3D Darstellung über eine Textur (2) definiert, die die Ausmaße repräsentiert. Der Start- und Endpunkt (3 und 4) wurden zuerst fest gewählt, sollte jedoch in einem späteren Prototypen frei wählbar sein.

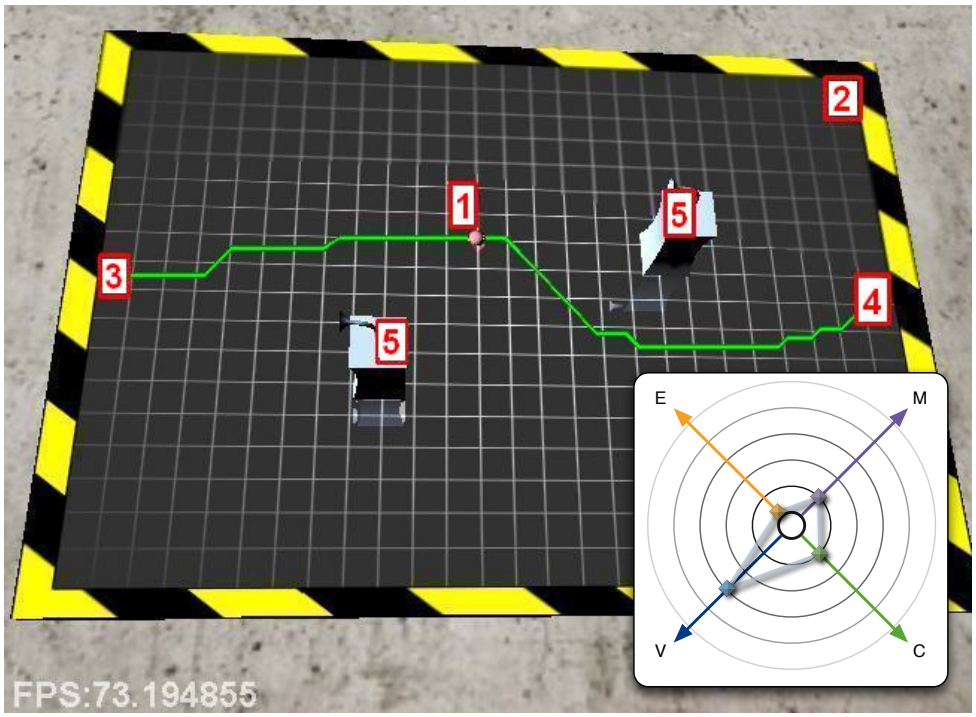


Abbildung 4.15: 3D Darstellung des ersten Prototypen.

Als Controller des ersten Prototypen kam eine einfache 2D-GUI zum Einsatz (siehe Abbildung 4.16$_{132}$), die es ermöglichte, die Wegeplanung zu starten, stoppen und zu pausieren. Dies wurde über Kreise im oberen Bildteil von Abbildung 4.16$_{132}$ realisiert. Die Fertigungsanlagen wurden über Maus-Interaktionen gesetzt, in der die Art,

Anzahl, Position und Orientierung der einzelnen Fertigungsanlagen gesetzt werden konnten. Da für spätere Prototypen eine AR-Version geplant war, die auf reale Objekte reagieren sollte, wurde schon in diesem Prototyp die Interaktion mit dem Benutzer über das TUIO-Protokoll[5] [KBBC05] realisiert, so dass die 2D GUI die entsprechenden TUIO-Befehle simulierte. Da die Anbindung an das TUIO-Protokoll vom Framework angeboten wurde, war die die Implementation einer TUIO-basierten GUI ohne großen Aufwand zu realisieren.

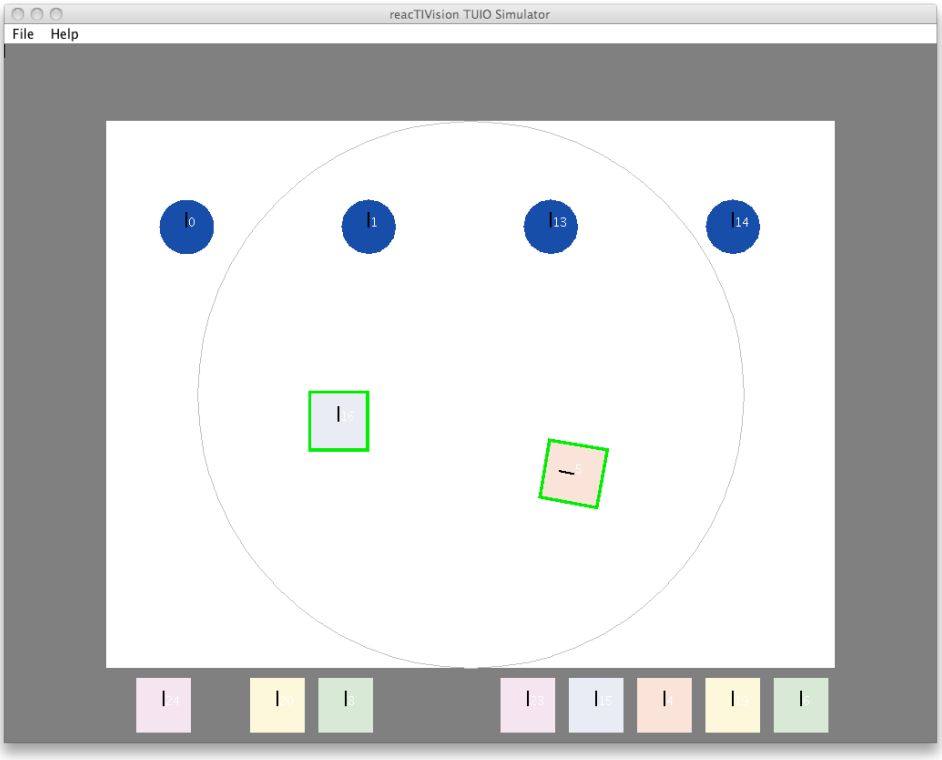


Abbildung 4.16: Rudimentäre GUI zur Steuerung der Prototypen.

Für die Wegeplanung des ersten Prototypen kam der in der Literatur bekannte A^*-Algorithmus [Stooo] [Rabooa] [Raboob] zum Einsatz. Dieser gilt als robuster Algorithmus, der ohne Ausnahme einen Weg findet, solle ein Pfad vom Start- zum Endpunkt existieren. Der A^*-Algorithmus sollte als Referenz für die folgenden Wegeplanungsalgorithmen gelten, um diese vergleichen zu können. Im Rahmen der Bachelorarbeit sollten verschiedene Wegeplanungsalgorithmen, die aus dem Bereich Organic Computing stammen, miteinander bzgl. Robustheit, Weglänge und Effizienz verglichen werden.

Am Kiviargraph in Abbildung 4.15_{131} wurde der Entwicklungsstand

[5] TUIO steht für **T**angible **U**ser **I**nterface **O**bject. Das TUIO-Protokoll erlaubt die Übertragung (über Netzwerk) einer abstrakten Beschreibung von interaktiven Oberflächen (und auch über Kamera getracke Objekte), die beispielsweise den Zustand oder die Position eines Objektes enthalten.

des ersten Prototypen dargestellt. Es ist zu erkennen, dass die Umgebung in diesem Prototypen nicht berücksichtigt wurde, allerdings die Darstellung schon ein hohes Niveau besitzt. Der Controller, der im ersten Prototypen durch die 2D GUI realisiert wurde, ist noch sehr rudimentär genau wie das das Modell, das durch den zugrundeliegenden Algorithmus zur Wegeplanung definiert wird. Nach Fertigstellung des ersten Prototypen gab es noch viel Potential zur Verfeinerung.

Um die Aufgabenstellung der Bachelorarbeit schnellst möglich zu realisieren, wurde in der Bewertungsphase für den zweiten Prototypen festgelegt, einen komplexeren Algorithmus, den Ant Colony Optimization-Algorithmus (ACO) [DBS06], zu implementieren und eine abstrakte 2D Darstellung des Szenarios zu integrieren, die es erlaubt, die gefundenen Wege der Algorithmen besser zu visualisieren. Da im ersten Prototypen die Simulation an die 3D Darstellung gekoppelt war, musste hier noch eine Entkopplung der Darstellung und der Simulation erfolgen. Das war erforderlich, da der ACO ein heuristischer, biologisch-inspirierter Algorithmus ist und erst nach einer gewissen Anzahl an Durchläufen eine geeignete Lösung findet.

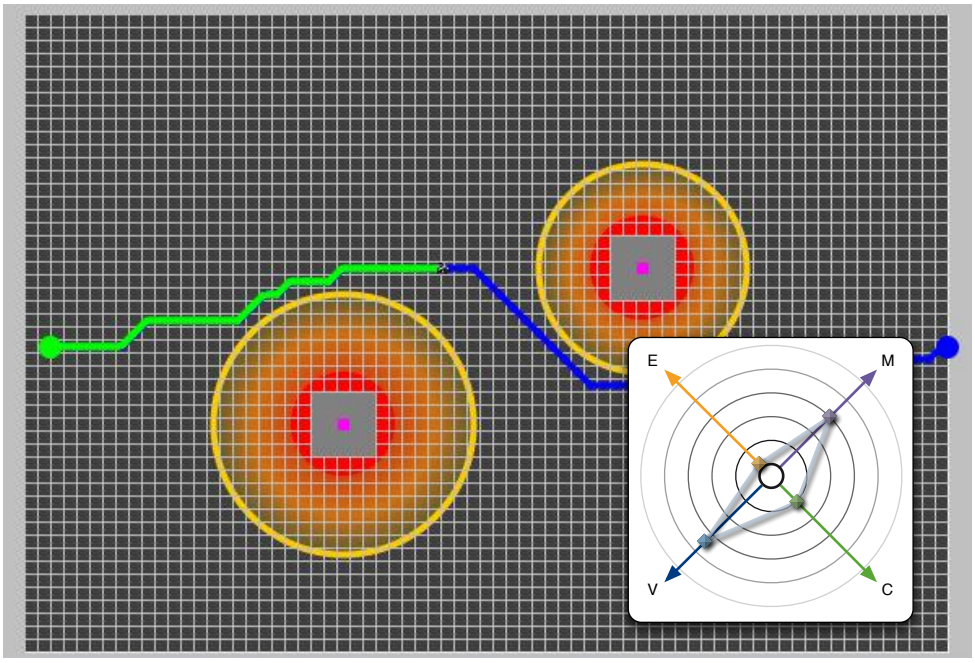


Abbildung 4.17: 2D Ansicht des zweiten Prototypen.

In Abbildung 4.17 ist die 2D-Ansicht des zweiten Prototypen zu sehen. Die Fertigungsanlagen werden durch ein graues Quadrat dargestellt. Die Kreise um diese Quadrate sind die Gefahrenbereiche, in denen Teile der Fertigungsanlage temporär reinragen können. Der Weg des autonomen Roboters ist in zwei Teile geteilt. Der erste Teil ist der zurückgelegte Weg (hier grün dargestellt), der zweite Teil ist der

geplante Weg, den der Roboter noch zurücklegen muss. Am Kiviatgraphen erkennt man, dass das Modell des Prototypen verfeinert wurde, da hier der ACO-Algorithmus gewählt werden kann. Die Darstellung ist etwas komplexer geworden, da die abstrakte 2D Darstellung zur schon vorhandenen 3D Darstellung hinzugekommen ist. Sowohl der Controller als auch die Umgebung haben keine Verfeinerung erfahren und sind deshalb unverändert gegenüber dem ersten Prototypen.

Nach Fertigstellung des zweiten Prototypen waren die Tests erfolgreich und es war möglich, die beiden implementierten Algorithmen miteinander zu vergleichen. In der nachfolgenden Iteration sollte nun ein letzter Algorithmus, der Particle Swarm Optimization-Algorithmus (PSO) [KE95] [LQH06], implementiert werden. Weiterhin ergab sich durch eine Zusammenarbeit mit einer Projektgruppe die Möglichkeit, die Benutzerschnittstelle auf einen Multitouch-Tisch zu realisieren [SBK+10]. Da das Framework schon eine Anbindung an das TUIO-Protokoll hatte, war die Umsetzung nicht sehr kompliziert, da auch die Tracking-Software des Mulitouch-Tisches auf dem TUIO-Protokoll basierte.

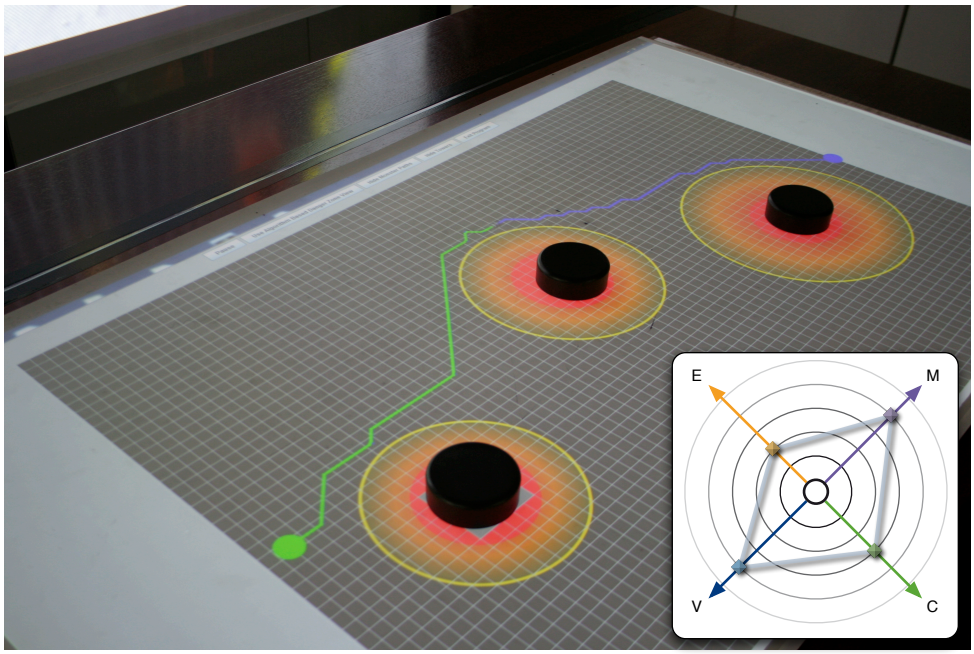


Abbildung 4.18: Multitouch-Oberfläche des dritten Prototypen.

In der Abbildung 4.18 ist die Benutzerschnittstelle auf dem Multitouch-Tisch zu sehen. Die Fertigungseinheiten können über *Tangibles*. die hier als schwarze Zylinder zu erkennen sind, auf dem Tisch positioniert werden. Weiterhin wurde die Möglichkeit geschaffen, bestimmte Parameter der implementierten Algorithmen über den Multitouch-Tisch zu verändern. Eine 3D-Darstellung der Szene wurde über ein

L-Shape realisiert, das hinter dem Multitouch-Tisch platziert wurde. So war eine gleichzeitige Darstellung der abstrakten 2D Ansicht und der realistischen 3D Darstellung möglich. Am Kiviatgraphen ist zu erkennen, dass durch die Implementierung des PSO-Algorithmus und die Möglichkeit der Veränderung der Parameter der Algorithmen das Modell weiter verfeinert wurde. Auch die Ansicht wurde verfeinert, da die vormals einfachen Modelle durch realistische 3D-Modelle ersetzt wurden. Die Verfeinerung des Controllers ergab sich durch die Interaktion über en Multitouch-Tisch und der Benutzung von „Tangibles“ als reale Repräsentanten der Fertigungsanlagen. Die Verwendung der „Tangibles“ hat auch Einfluss auf die Verfeinerung der Umgebung, da nun Teile der realen Umgebung erkannt und verarbeitet werden.

Der letzte Prototyp wurde aus zeitlichen Gründen nur theoretisch vorbereitet. Anstelle des im vorangegangenen Prototypen verwendeten Multitouch-Tisches einschließlich der 3D-Darstellung über das L-Shape sollte eine Augmented Reality Anwendung entwickelt werden. Das Tracking sollte auch über „Tangibles“ realisiert werden, die jedoch nun im 3D Raum getrackt werden sollten. Auf der Position dieser „Tangibles“ sollte dann eine 3D-Darstellung der Fertigungsanlagen „augmentiert“ werden. Durch diesen Schritt würden die 3D-Darstellung und die realen „Tangibles“ zu einer Einheit zusammengefügt werden. Alle vorhandenen Informationen, die derzeit auf der 2D-Benutzerschnittstelle des Mutlitouch-Tisches zu sehen sind, sollten in die reale Umgebung gezeichnet werden. Die AR-Applikation hätte dementsprechend die Darstellung und die Umgebung im Kiviatgraphen verfeinert. Modell und Controller wären gleich geblieben, da sich an der grundsätzlichen Benutzung der Applikation wenig geändert hätte.

4.4.3 Fazit des Beispiels

An diesem kleinen Beispiel wurde das MRiL-Entwurfsvorgehen getestet. Es kam keine spezielle Softwareumgebung, die das Vorgehen unterstützt, zum Einsatz. Deshalb konnten nur die generellen Aspekte angewendet werden. Allerdings unterstützte das Framework teilweise die Verarbeitung von Tracking-Informationen, was die Realisierung von MR-Interaktionen erleichterte. Die Einteilung in die MVCE-Komponenten war hilfreich, da sie die Entwicklung erleichtert und beschleunigt hat. Des Weiteren war die Erstellung von Prototypen mit Hilfe der MVCE-Einteilung schnell zu realisieren.

Da das hier eingesetzte Framework nicht auf das MRiL-Entwurfsvorgehen angepasst war, konnten viele Vorteile, wie z. B. der Einteilung in Akteure und die Verfeinerung durch Adapter, nicht verwendet werden.

Grundsätzlich war die Verwendung von MRiL aber vorteilhaft, da die schnelle Anpassung an innovative Benutzerschnittstellen sehr einfach möglich war.

Es ist zu sehen, dass MRiL grundsätzlich auch ohne spezielle Werkzeugunterstützung anwendbar ist. Um allerdings eine optimale Unterstützung des MRiL Entwurfsvorgehens zu erhalten, sollte dieser mit Hilfe spezieller Entwicklungswerkzeuge angewendet werden. Da es allerdings keine Werkzeuge dieser Art gab, wurden zwei Lösungen implementiert, die im Folgenden beschrieben werden.

4.5 Die Softwareumgebung

Um MRiL in der Entwicklung einsetzten zu können, benötigt man eine Softwareumgebung, die das Vorgehen unterstützt. Dabei ist es wichtig, dass die grundsätzlichen Prinzipen von MRiL in den Werkzeugen unterstützt werden. Um zu Beginn der Arbeit die Ansätze von MRiL anwenden zu können, wurde auf eine proprietäre 3D Entwicklungsumgebung zurückgegriffen, die durch unterschiedliche eigene Erweiterungen auf das Entwurfsvorgehen angepasst wurde. Da sich im Laufe der Arbeit aber herausstellte, dass nicht alle Aspekte des Entwurfsvorgehens mit dieser Softwarelösung abgebildet werden konnten, habe ich mich entschieden, eine komplett eigene Entwicklungsumgebung im Rahmen einer Masterarbeit an der FH Düsseldorf entwickeln zu lassen [Pog09]. Hier wurde das MRiL-Entwurfsvorgehen komplett in Software abgebildet, so dass es keine Unterschiede zwischen dem Konzept und der späteren Implementierung auftraten.

Im Folgenden Kapitel 4.5.1 stelle ich 3DVIA Virtools und die Erweiterungen, die die Entwicklung von Mixed Reality Anwendungen ermöglichen, kurz vor und beschreibe dann in Kapitel 4.5.2$_{148}$ die eigens für MRiL entwickelte Softwareumgebung MiReAS.

4.5.1 Erweiterungen des proprietären Autorensystems 3DVIA Virtools

Um das MRiL-Entwurfsvorgehen zu Beginn dieser Arbeit schnell evaluieren zu können und Anwendern mit wenig Programmiererfahrung ein Werkzeug an die Hand zu geben, mit dem sie schnell Applikationen entwerfen können, wurde auf eine proprietäre Entwicklungsumgebung zurückgegriffen und diese mit Hilfe von Plugins erweitert. Ich habe mich für das Autorensystem 3DVIA Virtools (vormals Virtools) von Dassault Systems in der Version 4.0 entschie-

den [Das09]. Hier war einiges an Erfahrung schon vorhanden, so dass die Entwicklung eigener Komponenten für dieses System in kurzer Zeit möglich war. Diese Umgebung wurde in einigen Veröffentlichen und Demonstratoren verwendet, beispielsweise „Entwicklung von Augmented Reality-Präsentationen mit einem High-Level Authoring System – eine Fallstudie“ [GSR+06], „Development of an augmented reality game by extending a 3D authoring system“ [GSKF07], „Mixed Reality Authoring“ [GS07], „HYUI: a visual framework for prototyping hybrid user interfaces“ [GFLS08] und „Authoring of 3D and AR Applications for Educational Purposes“ [SGDZ08].

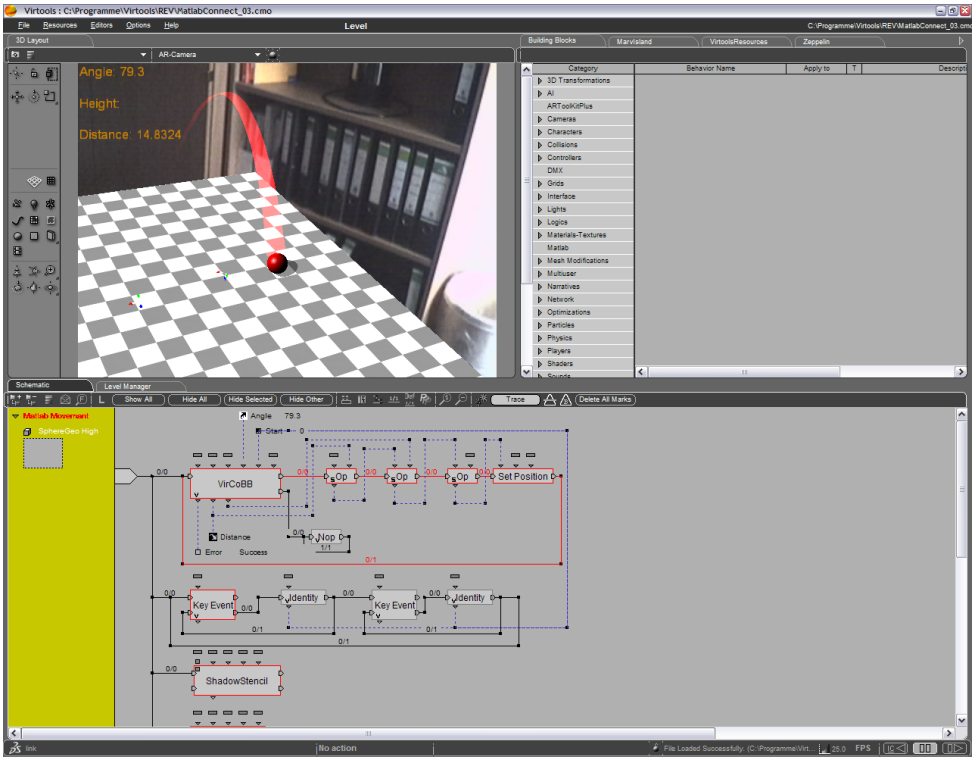


Abbildung 4.19: Die Entwicklungsumgebung 3DVIA Virtools.

3DVIA Virtools - Übersicht

3DVIA Virtools[6] ist eine komplette Entwicklungs- und Verteilungsplattform mit einem innovativen Ansatz zur interaktiven Erstellung von 3D-Inhalten. Eine Innovation dieser Plattform gegenüber anderen Entwicklungsumgebungen in diesem Gebiet besteht darin, dass die imperative Programmierung zum größten Teil durch eine visuelle Programmierung ersetzt wurde, dem so genannten Behavior Graph, zu sehen in Abbildung 4.20$_{138}$. Der Behavior Graph ist ein gerichteter

[6] 3DVIA Virtools wird nachfolgend nur noch *Virtools* genannt.

Graph von miteinander verbundenen Building Blocks (die Grundelemente der visuellen Programmierung in Virtools, siehe Abbildung 4.20_{138}), der das Programm darstellt. Virtools bietet eine große Auswahl an schon vorhandenen Building Block an, die über einfach Aufgaben, wie Transformation eines 3D Objekts, bis hin zu sehr komplexen Aufgaben, wie die Bewegungssteuerung eines Charakters, verfügt. Die Funktion der einzelnen Building Blocks ist sehr gut dokumentiert und meist an Beispielen erklärt, so dass die Entwicklung von Applikationen selbst für programmierunerfahrene Anwender einfach möglich. Durch graphische hierarchische Zusammenfassung schon bestehender Buidling Blocks ist es weiterhin möglich, eigene, neue, wiederverwendbare Building Blocks zu erzeugen, diese zu speichern und in anderen Projekten weiter zu verwenden (In Abbildung 4.20_{138} ist z. B. der Building Block „Get Phantom State“ ein zusammengefasster Building Block). Neben der Wiederverwendbarkeit kann das visuelle Programm durch diese Zusammenfassungen in neue Building Blocks übersichtlicher gestaltet werden.

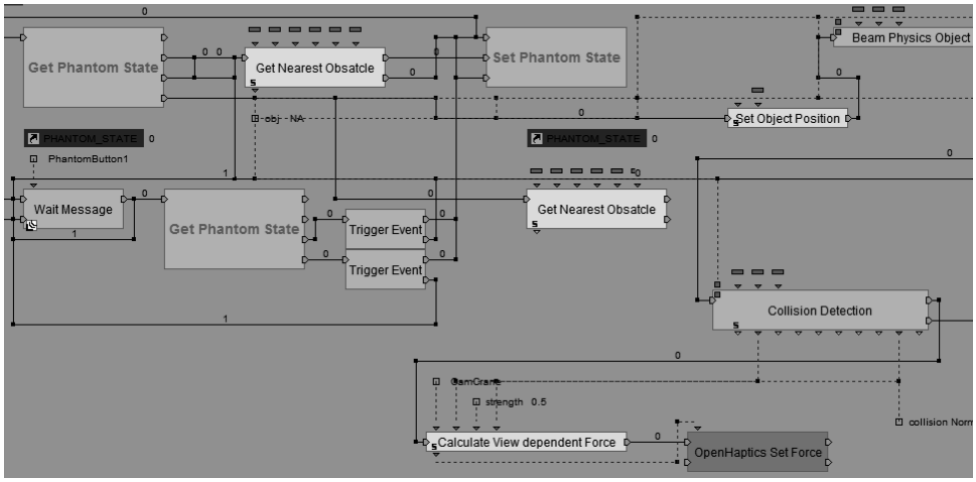


Abbildung 4.20: Behavior Graph mit verbundenen Building Blocks.

Sollte die visuelle Programmierung für manche Probleme nicht ausreichen bzw. die Komplexität der visuellen Programme zu groß werden, gibt es die Möglichkeit, Teile des Programms in einer der Skriptsprachen VSL (**V**irtools **S**cripting **L**anguage) oder LUA [ICdF99] (ab Version 5.0 von Virtools) zu erstellen und in einem Building Block zu kapseln, wie in Abbildung 4.21_{139} zu sehen. Über VSL/LUA kann jede Funktionalität von Virtools verwendet werden, so dass die Programmierung über die Skriptsprache keine Nachteile bietet. Vorteil ist z. B. die kürzere Form der Programme, gerade bei Konstrukten wie Schleifen und Bedingungen.

Sollten die integrierten Building Blocks für spezielle Aufgaben nicht ausreichen bzw. der Programmablauf sowohl über Building Blocks

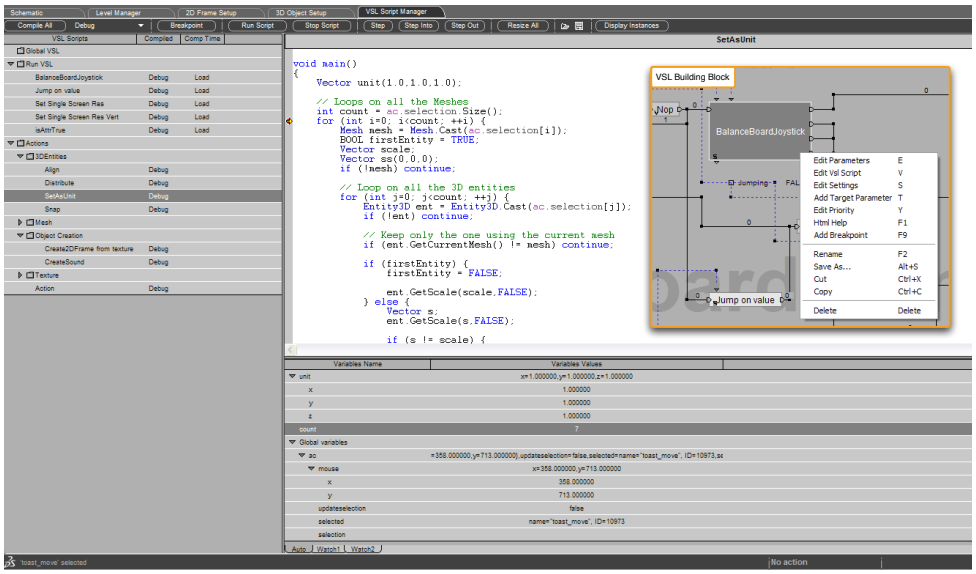


Abbildung 4.21: VSL Skript Manager und VSL Building Block.

als auch über Skripts zu lange dauern bzw. zu komplex werden, besteht die Möglichkeit, eigene Building Blocks über das mitgelieferte SDK in C/C++ zu entwickeln. Das SDK bietet vollen Zugriff auf alle Funktionen in Virtools, es besteht die Möglichkeit sowohl auf die Behavior Engine als auch auf die Render Engine zuzugreifen und dort Manipulationen vorzunehmen. Die eigenen Building Blocks können genau wie die internen in einen Behavior Graphen eingefügt werden und unterscheiden sich grundsätzlich nicht von diesen. Zusätzlich zu Building Blocks können so genannte Manager mit dem SDK entwickelt werden, die globale Funktionen übernehmen können. Manager folgen dem Prinzip eines Singleton (vgl. [GHJV96] Seite 156 ff.), d. h. es existiert immer nur jeweils eine Instanz dieses Managers innerhalb eines Programms. In den einzelnen Building Blocks kann auf diese Manager global zugegriffen werden. Somit lässt sich dort globale Funktionalität kapseln.

Neben den unterschiedlichen Programmierarten bietet Virtools eine komplexe und moderne 3D Rendering Engine, die effizient auch große Szenen darstellen kann. Sie basiert auf Microsofts DirectX 9.0c und integriert fortgeschrittene Techniken wie Schattenwurf, Shaderintegration, Rendertargets, etc., die einfach über die bereitgestellten Bulding Blocks benutzt werden können. Diese Techniken sind auf einer hohen Abstraktionsebene in Virtools implementiert, so dass sich der Entwickler nicht um technische Details kümmern muss. Um Modelle in Virtools benutzten zu können, werden für alle gängigen 3D Design Programme wie Maya oder 3D Studio Max Export-Plug-ins bereitgestellt, die die Modelle in das für Virtools lesbare Format speichern.

Damit ist sichergestellt, dass die Inhalte, egal mit welcher Software entwickelt, mit Virtools weiterverarbeitet werden können.

Bei Fertigstellung der Anwendung kann diese als lauffähiges Programm exportiert bzw. in einem webbasierten Player wiedergegeben werden. Damit kann die Anwendung auf verschiedenen Plattformen und mit unterschiedlichen Benutzergruppen getestet werden. Leider verbietet die strenge Lizenzpolitik von Dassault Systems eine einfache Handhabung, dazu mehr im Abschnitt „3DVIA Virtools - Nachteile“ in Kapitel 4_{148}.

Als Entwicklungsumgebung bietet Virtools zusammenfassend folgende Merkmale:

Behavior Graph: Über den Behavior Graphen kann Programmlogik und Algorithmik visuell programmiert werden.

Bibliothek vordefinierte Programmblöcke: Über die mitgelieferten Building-Blocks von Virtools können visuelle Programme erstellt werden. Sie werden in den Behavior Graphen eingefügt und untereinander verbunden.

Skriptsprachenanbindung: Zu der visuellen Programmierung bietet Virtools zusätzlich die Möglichkeit, Teile der Algorithmik in einer textuellen imperativen Skriptsprache zu realisieren. Dabei stehen die Skriptsprachen VSL und LUA zur Verfügung.

Entwicklung eigener Building Block über SDK: Ist die Implementierung über Building-Blocks nicht ausreichen, zu komplex werden oder die Ausführungszeit bestimmter Programmteile optimiert werden, können über das mitgelieferte SDK diese Abschnitte in C/C++ realisiert und Virtools als eigenständige Building Blocks bzw. Manager zur Verfügung gestellt werden.

Moderne 3D Render Engine: Die Darstellung der 3D-Inhalte wird durch eine moderne auf DirectX 9.0c basierende 3D Render Engine realisiert, die u.a. Schattenberechnung, Shaderintegration und Rendertargets unterstützt.

Export: Die fertige Anwendung kann aus Virtools exportiert werden und sowohl in einem webbasierten Player abgespielt als auch in eine ausführbare Datei abgespeichert werden.

3DVIA Virtools - Konzepte

3DVIA Virtools beinhaltet fünf Schlüsselkomponenten:

- Die graphische Benutzerschnittstelle zur Entwicklung von Anwendungen durch visuelle Programmierung von Objekten und Verhalten.
- Die Behavior Engine zur Ausführung von interaktiven Anwendungen.
- Die Render Engine zur Visualisierung der Anwendung in Echtzeit.
- Die Virtools Skriptsprache für die Low-Level Programmierung bestimmter Funktionen.
- Das SDK für benutzerspezifische Behaviors.

Die graphische Benutzerschnittstelle von Virtools wir in jedem Schritt in der Entwicklung genutzt. Sie beinhaltet u. a. Ein 3D Layout zur Darstellung des Inhalts der Anwendung in Echtzeit. Hier wird die komplette virtuelle 3D Szene dargestellt. Unter Verwendung der zur Verfügung stehenden grafische Werkzeuge können 3D Objekte, Kameras, Lichter, etc. erzeugt, verändert selektiert und manipuliert werden. Über Drag & Drop kann Entitäten einer virtuellen 3D Szene Verhalten hinzugefügt werden. Dieses Verhalten wird über die Behavior Buildung Blocks visuell erzeugt und in einer schematischen Ansicht dargestellt. Ausgeführt wird dieses Verhalten durch die Behavior Engine, die zu Anfang jedes darzustellenden Bildes ausgeführt wird.

Die Behavior Engine führt sowohl selbst entwickelte als auch von Virtools mitgeliefertes Building Blocks aus. Die mitgelieferten Building Blocks umfassen u. a. die Kategorien Kamera, CHaracter, Kollisionserkennung, Optimierung, Pfadfindung, Mesh-Modifikation, Logik, Partikel, Sound, etc. Die Behavior-Bibliothek kann durch selbst entwickelte Building Blocks erweitert werden. Diese werden mit dem mitgelieferten SDK in C++ programmiert und in Virtools eingebunden werden.

Virtools Render Engine kann für viele Plattformen verwendet werden und kann von DirektX 5 über DirectX 9.0c bis OpenGL 2.0 konfiguriert werden, so dass eine große Anzahl an Konfigurationen abgedeckt werden kann. Die Render Engine unterstützt in den neueren Konfigurationen programmierbare Vertex- und Pixel-Shader bis Version 3.0, die in DirextX mit HLSL, CgFX oder Assembler und in OpenGL in GLSL programmiert werden können. Für den Import von Modellen bietet Virtools Plug-ins für alle gängigen 3D Modeling Systeme an, die sowohl Modelle als auch Animationen nach Virtools exportieren können. Dynamische Erzeugung und Löschung von Objekten und

Modellen wird mit der Render Engine komplett unterstützt. Für die Animation von Charakteren bietet die Render Engine ein Skin and Bones System an, mit dem sich Bewegungen natürlich animieren lassen. Die gesamte Funktionalität der Render Engine lässt sich auch für selbst geschriebene Building Blocks über das SDK verwenden.

Die Skriptsprachen VSL und LUA sind in Virtools voll integriert und werden über einen speziellen Editor direkt programmiert. Virtools besitzt für die Skriptprogrammierung eine intelligentes Farbsystem, eine kontextsensitive Vervollständigung und eine Anzeige von Argumenten einer Funktion. Weiterhin bietet Virtools für die Skriptprogrammierung einen kompletten Debug-Mode mit Breakpoints, anzeigen und ändern von Variableninhalten und eine Schritt-Ausführung (Sinlge-Step) an.

Das SDK von Virtools ist eine Sammlung von Entwicklungswerkzeugen, die aus Biblioteken, sowohl statisch als auch dynamisch, und Header-Dateien bestehen. Sie bieten vollen Zugriff auf alle Low-Level Funktionen von Virtools. Mit dem SDK können Entwickler sowohl eigenständige Applikationen, die auf Virtools basieren, als auch Erweiterungen von Virtools selbst entwickeln. Erweiterung können dabei Behaviors, Medien-Importer, Manager, Render Engine Plug-ins oder Rasterizer sein.

3DVIA Virtools - Erweiterungen

Damit Virtools das MRiL-Entwurfsvorgehen zum Teil unterstützten konnte, mussten einige eigene Komponenten mit Hilfe des SDKs realisiert werden. Im Einzelnen waren das folgende Erweiterungen:

Tracking: Diese Building Blocks sind für die Registrierung von realen Objekten in der virtuellen Welt zuständig.

- ReacTIVision Building Blocks und Manager für ein bildbasiertes 2D Tracking [GFLS08]
- ARToolkitPlus Building Blocks und Manager für ein bildbasiertes 3D Tracking [GSKF07]
- OptiTrack Building Blocks und Manager für ein 2D/3D Infrarot-Tracking [GSR+06]
- OpenCV Building Blocks und Manager für ein bildbasiertes 2D/3D Tracking [GFLS08]

Kommunikation: Building Blocks, die für eine Kommunikation zwischen Virtools und anderen Applikationen, z. B. MATLAB/Simulink, zuständig sind.

- *COMMUVIT* Building Block und Manager zur synchronen Kommunikation mit externen Tools [SGDZ08]

Eingabe: Diese Kategorie von Building Blocks beinhaltet die Anbindung verschiedener Eingabe-Hardware an Virtools.

- Wiimote Building Blocks zur Steuerung der virtuellen Inhalte [GSKF07]
- OpenHaptics Building Block und Manager zur Steuerung von haptischen Geräten [GSKF07]

Ausgabe: Building Blocks, die Ergebnisse auf spezieller Hardware ausgeben können.

- Ausgabe über einen externen Midiadapter, um so Midigesteuerte Geräte ansprechen zu können [GFLS08]

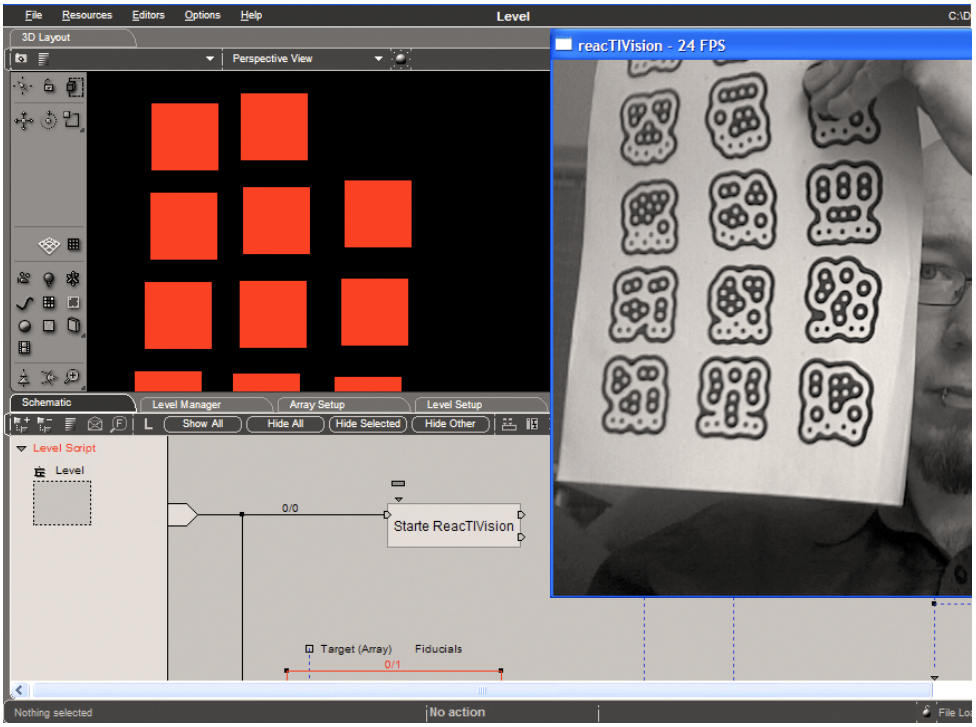


Abbildung 4.22: 2D Tracking mit ReacTIVision in Virtools.

Die wichtigsten Erweiterungen für das MRiL-Entwurfsvorgehen sind die Tracking Building Blocks, mit denen es möglich ist, 2D bzw. 3D

Positionen aus einem Videobild zu berechnen. Das 2D Tracking mit ReacTIVision(siehe Abbildung 4.22) kann für eine einfache Registrierung von realen Objekten genutzt werden, bei denen die Tiefeninformation nicht wichtig ist (z. B. Infotext, der an ein reales Objekt gebunden sein soll). Der Vorteil der 2D Registrierung ist die Performance und die Robustheit dieser Methode. Daher kann das 2D Tracking auch gut für reale Benutzerschnittstellen eingesetzt werden. Die Implementierung von ReacTIVision hat weiterhin den Vorteil, dass die Erkennung durch eine eigenständige Anwendung realisiert wird und die Tracking-Daten mittels TUIO-Protokoll [KBBC05] über ein Netzwerk versendet werden. D. h. Hauptanwendung und Trackinganwendung können auf verschiedenen Rechnern laufen so dass die Performance gesteigert werden kann.

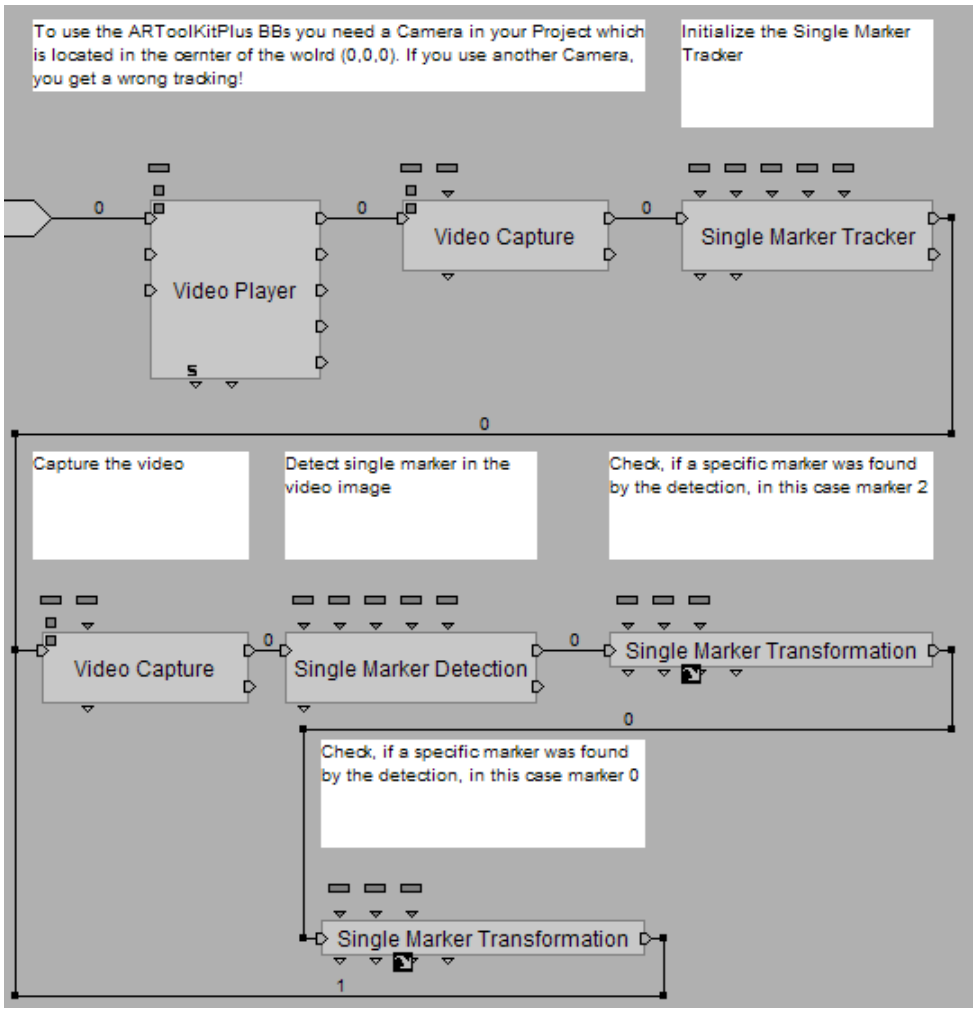


Abbildung 4.23: Tracking mit ARToolKitPlus Building Blocks.

Für das 3D-Tracking sind zwei verschiedene Verfahren verwendet worden, zum einen ein bildbasiertes Tracking über Marker mit Hilfe vom ARToolKitPlus (siehe Abbildung 4.23), zum anderen ein Infrarot-

Tracking mit Hilfe von OptiTrack (siehe Abbildung 4.24$_{145}$). Das Tracking mit ARToolKitPlus geschieht über ein Videobild, in welchem spezielle Marker gesucht werden. Werden diese gefunden, ist ARToolKitPlus in der Lage, die 3D Position dieser Marker im Kameraraum zu bestimmen. Dafür ist aber eine Kalibrierung auf die jeweilige Kamera und auf das jeweilige Objektiv der Kamera notwendig. Für Virtools wurde diese Funktionalität in mehrere logische Building Blocks aufgeteilt. Für die globale Erkennung wurde ein spezieller Building Block entwickelt, der einmal pro Frame aufgerufen wird, das aktuelle Videobild ausliest und dieses analysiert. Damit nur jeweils eine Instanz vom ARToolKitPlus zur Laufzeit aktiv ist, wurde diese in einem Manager gekapselt, der die Funktionalität für die einzelnen Building Blocks im Behavior Graphen zur Verfügung stellt. Einzelne Marker werden durch jeweils einen Building Block im Behavior Graphen abgebildet und reagieren somit nur auf einen spezifischen Marker. Diese Building Blocks fragen den Manager nach ihren Daten. Falls der spezielle Marker im Videobild erkannt wurde liefert der Manager die Position zurück. Je nach Anzahl der im Bild befindlichen Marker und der Größe des Videobildes kann die Erkennung viel Rechenleistung beanspruchen.

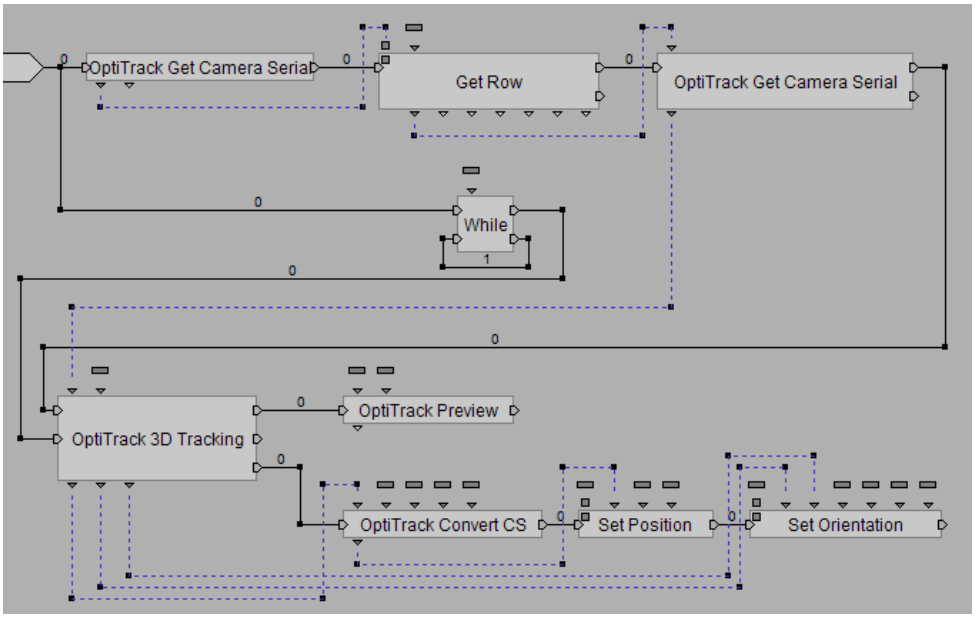


Abbildung 4.24: Infrarot-Tracking mit OptiTrack Building Blocks.

Das Infrarot-Tracking funktioniert im Prinzip wie das oben beschriebene bildbasierte Tracking, mit dem Unterschied, dass die OptiTrack-API zur Erkennung benutzt wird. Beim Infrarot-Tracking kann nicht zwischen verschiedenen Markern unterschieden werden, da diese keine ID-Informationen kodiert haben. Daher ist es im 3D Tracking nur möglich, ein einzelnes Objekt per Infrarot zu tracken und eindeutig

zuzuweisen. Vorteil des Infrarot-Trackings ist die hohe Genauigkeit und die größere Entfernung, gemessen an bildbasierten Trackingverfahren, in der die Erkennung funktioniert. Des Weiteren ist, gegeben durch die schnelle Bildwiederholrate der Infrarot-Kamera (im vorliegenden Fall: 120 Bilder pro Sekunde), ein genaues Tracking möglich, bei dem die Latenzen sehr gering sind. Das kann zwar auch beim bildbasierten Tracking erreicht werden, jedoch werden dann spezielle Kameras benötigt und die Leistung des Rechners muss ausreichend sein um die Einzelbilder zu analysieren.

Eine zweite bildbasierte Tracking-Methode wurde mit Hilfe der offenen Bibliothek OpenCV realisiert. Diese ist aber erst in einem frühen Stadium der Entwicklung und wurde, nachdem auf MiReAS gewechselt wurde, nicht weiter entwickelt. Grund für die Entwicklung eines OpenCV-basierten Trackers war die Unabhängigkeit von anderen Tracking-Verfahren und die dadurch resultierende Freiheit in der Entwicklung.

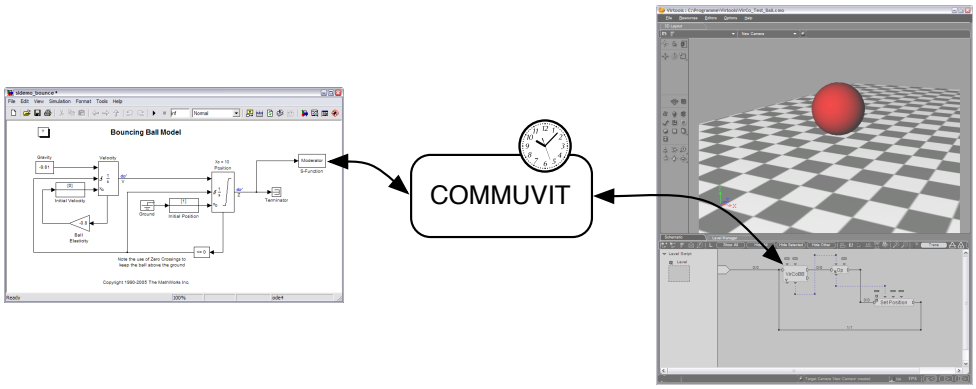


Abbildung 4.25: *COMMUVIT* Building Block und Simulink Modell.

Eine weitere wichtige Erweiterung in Virtools ist die Anbindung an externe Anwendungen. Dies wurde über das von Henning Zabel entwickelte Werkzeug *COMMUVIT* [SGDZ08] [LZE+06] [LZB07] realisiert, das eine Übergabe von jeglichen Daten zwischen Anwendungen zur Verfügung stellt. Ein wichtiger Punkt ist, dass *COMMUVIT* die Anwendungen auch zeitlich koppelt, so dass die Werte immer aktuell sind. In unseren Beispielen haben wir *COMMUVIT* verwendet, um physikalisch korrekte Modelle in MATLAB/Simulink zu berechnen und in Virtools zu visualisieren und damit zu interagieren. Abbildung 4.25 zeigt eine Virtools-Anwendung, die zur Berechnung der Bewegung der roten Kugel ein MATLAB/Simulink Modell hinterlegt hat, das für jeden Frame die physikalisch richtige Position berechnet.

Um neue Interaktionsmöglichkeiten zu schaffen, wurden zwei Building Blocks realisiert, die unterschiedliche Eingabehardware in Vir-

tools zur Verfügung stellen. Zum einen ist das der Wiimote Building Block, der es erlaubt, die Wiimote (die eigentlich zur Benutzung der Spielekonsole Wii von Nintendo gedacht ist) in Virtools-Projekten zu benutzen. Gerade durch die eingebauten Beschleunigungssensoren ist es eine kostengünstige Alternative zu anderen Speziallösungen. Über die Beschleunigungssensoren ist es möglich eine einfache Gestenerkennung zu realisieren, die für die Steuerung benutzt werden kann. Über die eingebaute Kamera wird weiterhin ein 2D Tracking zur Verfügung gestellt, so dass eine Pointer-basierte Interaktion mit Hilfe der Wiimote möglich ist.

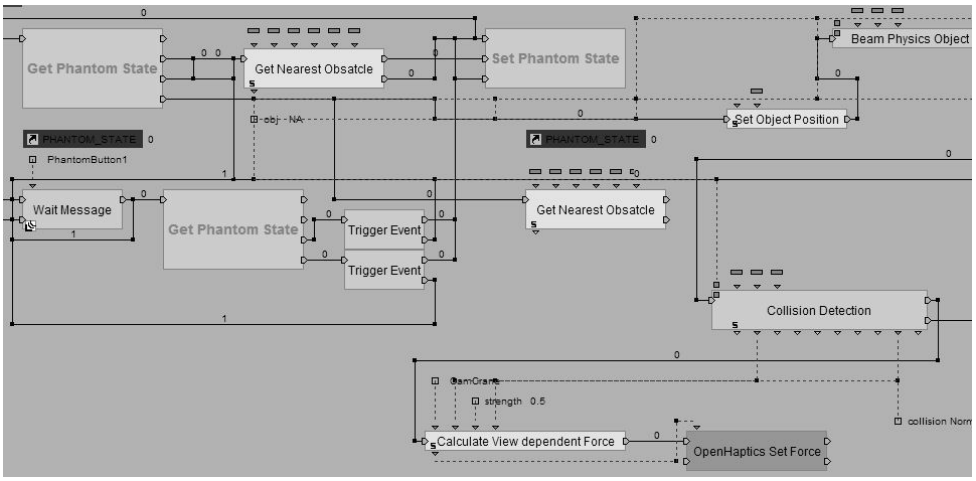


Abbildung 4.26: Eingabe über das OpenHaptics Interface in Virtools.

Der OpenHaptics Building Block (siehe Abbildung 4.26) wurde speziell für das haptische Feedback zum Benutzter realisiert. Gesteuert werden kann damit z. B. ein SensAble Phantom Omni Haptic Device [Sen10], das dem Benutzer als Eingabemedium dient. Über das Phantom kann ein Punkt im 3D Raum incl. Richtung angesteuert werden. Der Programmierer hat die Möglichkeit, eine gewisse Gegenkraft auf das Eingabegerät zu geben, so dass das Gefühl eines Widerstandes entsteht. Über verschiedene Parameter lassen sich unterschiedliche haptische Materialeigenschaften programmieren.

Über die oben beschriebenen Building Blocks, die die Funktionalität von Virtools erweiterten, ließen sich einige Konzepte des MRiL-Entwurfsvorgehens realisieren, wie es in Kapitel 5_{159} beschrieben wird. Allerdings konnten nicht alle Konzepte in Virtools realisiert werden, da die Struktur von Virtools dies nicht zuließ.

3DVIA Virtools - Nachteile

Leider gab es bei der Verwendung von Virtools auch einige Nachteile, die u. a. dafür verantwortlich waren, dass eine eigene Entwicklungsumgebung realisiert wurde. Zu diesen Nachteilen zählen:

MRiL Prozess nicht komplett abbildbar: Infolge der vorgegebenen Struktur von Virtools war es nicht möglich, das gesamte MRiL-Entwurfsvorgehen abzubilden. Damit aber dieser Prozess komplett evaluiert werden konnte, war es essenziell, eine Entwicklungsumgebung zu besitzen, die diese Abbildbarkeit leistete.

Lizenzproblematik: Leider wurde infolge einer Lizenzänderung in neueren Versionen von Virtools untersagt, selbst geschriebene Bulding Blocks weiterhin der Community zur Verfügung zu stellen. Des Weiteren ist der Webplayer in den neueren Versionen nicht mehr in der Lage selbst geschriebene Building Blocks zu laden und auszuführen. Dementsprechend müssen für jede Demo eigene ausführbare Dateien kompiliert werden, falls auf dem Zielsystem kein Virtools installiert ist. Leider müssen diese Demos einen Lizenzschlüssel haben, so dass es schwierig ist, Demos auf Konferenzen oder Messen vorzustellen oder im wissenschaftlichen Rahmen zu veröffentlichen.

Proprietär: Da Virtools eine proprietäre Software ist, gestaltet sich eine Anpassung auf eigene Bedürfnisse äußerst schwierig. Über das SDK kann viel Funktionalität realisiert werden, leider aber nicht alles. Daher war es auch nicht möglich, das komplette MRiL-Entwurfsvorgehen auf Virtools abzubilden, da wichtige Strukturen und Funktionsweisen nicht veränderbar waren.

Aus den oben genannten Gründen war es notwendig, eine eigene, auf das MRiL-Entwurfsvorgehen zugeschnittene Entwicklungsumgebung zu programmieren. Es wurde besonders darauf geachtet, dass eine quelltextoffene Implementation einer 3D Grafikbibliothek als Grundlage diente, um ggf. Konzepte nachträglich ins System integrieren zu können.

4.5.2 MiReAS - Eine Mixed Reality Softwareumgebung

Die Ergebnisse, die mit der Entwicklungsumgebung basierend auf Virtools entstanden sind, waren zum großen Teil akzeptabel. Da jedoch die von uns entwickelten Erweiterungen in Virtools nicht das komplette MRiL-Entwurfsvorgehen abdecken, insbesondere die Rollen von

Akteuren, wurde eine Softwareumgebung entwickelt, die speziell für meinen MRiL Prozess entworfen wurde. Diese Softwareumgebung wurde an der Hochschule Düsseldorf im Rahmen einer Masterarbeit [Pog09] nach den Vorgaben des MRiL-Entwurfsvorgehens entwickelt und trägt den Namen MiReAS: **Mi**xed **Re**ality **A**ctor **S**imulation. MiReAS kann als Werkzeug für schnelles Prototyping sowie für die einfache Entwicklung von Mixed Reality Anwendungen verwendet werden. Durch die konsequente Umsetzung des MRiL Vorgehens ist MiReAS eine ideale Plattform für die Entwicklung und die Tests von Mixed Reality Anwendungen.

Damit MiReAS die Vorraussetzungen für den MRiL-Entwurfsprozess erfüllt, mussten folgende Anforderungen erfüllt werden:

- Bereitstellung einer komponentenbasierten Architektur mit der Möglichkeit, Komponenten einfach zu adaptieren, auszutauschen oder wieder zu verwenden
- Verwendung einer Quelltext-offenen 3D Renderbibliothek basierend auf einem Szenegraphen zur einfachen Erstellung von Szenarios
- Unterstützung essenzieller Mixed Reality Funktionalität, beispielsweise die Verwendung von Videogeräten und Trackingsystemen sowie die einfache Erweiterbarkeit auf neue Trackingsysteme
- Unterstützung einer großen Anzahl von Eingabegeräten bzw. die Möglichkeit, solche einfach in das System einzubinden
- Anbindung an eine Physiksimulation, systemintern über eine schnelle Physikbibliothek aus dem Game-Sektor, darüber hinaus allerdings auch über externe Programme wie z. B. MATLAB/Simulink für mathematisch präzisere Berechnungen
- Einfache Nutzung von Netzwerkschnittstellen zur Verteilung
- Benutzerfreundliche und einfach anwendbare Systemkonfiguration über Konfigurationsdateien im XML Format bzw. grafische Benutzerschnittstellen

Konzepte von MiReAS

Das zentrale Konzept von MiReAS ist das Prinzip des „Actors & Adaptors“. Akteure (engl. Actors) sind aktive Elemente einer Anwendung, die in einem Szenario betrachtet und gesteuert werden sollen. Für eine

typische Mixed Reality Anwendung sind dies Eingabegeräte sowie interaktive und/oder dynamische Szenenelemente. Im Kontext von Mixed Reality sind hier sowohl virtuelle als auch reale steuerbare Systeme als Akteur zu verstehen. Die Grundlagen zum Akteurmodell können in Kapitel 4.3.3$_{120}$ nachgelesen werden.

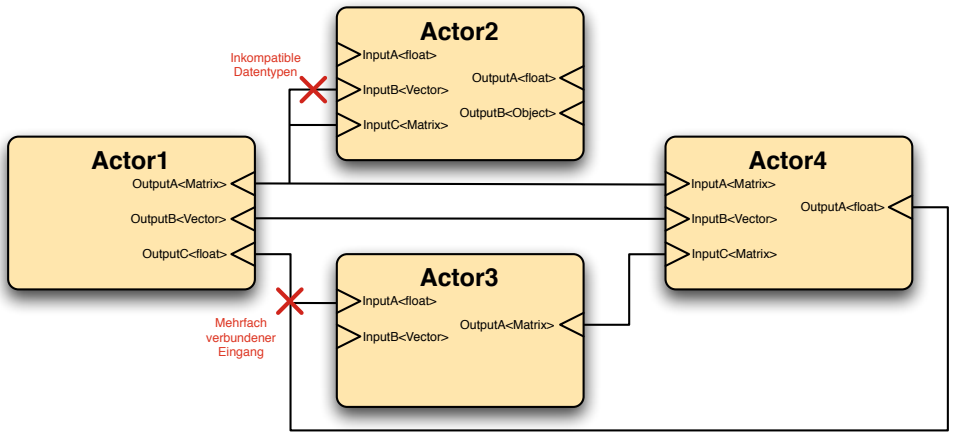


Abbildung 4.27: Datenflussnetzwerk basierend auf Akteure.

In MiReAS sind Akteure in der Lage, Steuer-und Informationsdaten über Ports zu senden beziehungsweise zu empfangen. Ports können dabei einen beliebigen Datentyp annehmen, z. B. vektorielle Werte für zusammenhängende Daten oder skalare Werte für einzelne Datenleitungen. Selbst komplexe Objekte können als Datentyp verschickt bzw. empfangen werden. Die Ports können beliebig miteinander verschaltet werden, allerdings müssen die jeweiligen Datentypen des Ein- und Ausgangs kompatibel sein. Ein Ausgangsport kann mit mehreren Eingangsports verbunden werden, Eingangsports können indes nur mit einem Ausgangsport verbunden sein. In jedem Zeitschritt aktualisiert ein Akteur seinen Zustand indem er die Werte an seinen Eingangsports liest, diese verarbeitet und die neu berechneten Werte an seine Ausgangsports anlegt. Damit ist es möglich ein interaktives Datenflussnetzwerk aufzubauen, welches das Verhalten der Anwendung steuert. In Abbildung 4.27 ist so ein Datenflussnetzwerk zu sehen. Hier wird des Weiteren dargestellt, dass weder zwei Ports miteinander verbunden werden können, die inkompatible Datentypen haben, noch Ausgänge mehrfach belegt werden dürfen.

Akteure, die über die gleichen Ein- und Ausgangsports verfügen, können problemlos untereinander ausgetauscht werden. So können z. B. Prototypen von Akteuren entwickelt werden, die ein bestimmtes Portinterface bieten und direkt in die Software eingebunden werden. Im Laufe der Entwicklung können diese Prototypen mit neueren bzw. anderen Versionen von Akteuren getauscht werden, ohne das das

Datenflussnetzwerk geändert werden muss.

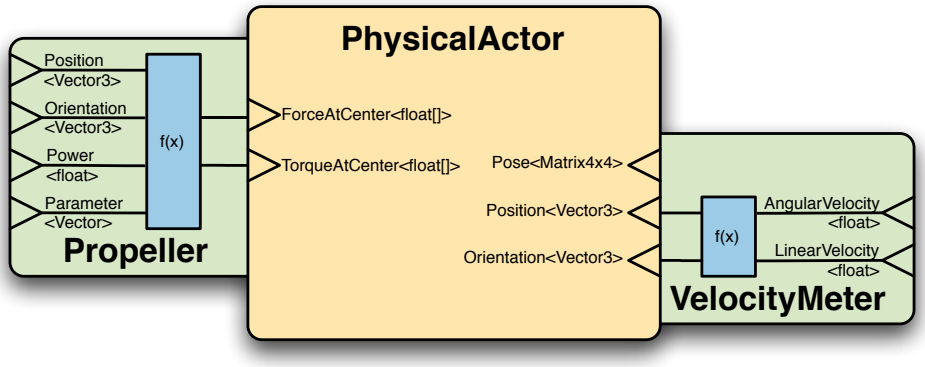


Abbildung 4.28: Kopplung zwischen einem Akteur und Adaptern.

Ein bereits implementierter Akteur kann durch sogenannte Adapter (engl. Adaptors) erweitert werden. Adapter haben ebenso wie Akteure eine Menge an Eingangs- und Ausgangsports. Wird ein Adapter einem Akteur zugewiesen, erhält der Akteur sämtliche Ports des Adapters. Diese Ports können dann von außen so angesprochen werden, als wenn sie Teil des jeweiligen Akteurs wären. Um die Ports des Adapters mit den Ports des Akteurs zu verbinden, muss eine entsprechende Funktionalität in den Adapter implementiert werden. Beispiele für Adapter wären zum einen, inkompatible Ports miteinander zu verbinden, indem der Adapter intern eine Umwandlung durchführt und die gewandelten Signale an die entsprechenden Eingangsports des Akteurs bzw. die Ausgangsports des Adapters weiterleitet. Die grundsätzliche Funktionalität von Adaptern kann in Kapitel 4.3.3$_{120}$ nachgelesen werden.

Der Vorteil eines Adapters gegenüber einer Vorschaltung bzw. Nachschaltung eines neuen Akteurs ist die automatische Verschaltung. Verbindungen müssen nicht neu gesetzt werden, was vor allem bei mehrfachem Verwenden von Adaptern der Übersichtlichkeit des Datenflussnetzwerk dient.

Dieses Konzept gestattet komplexe, aufeinander aufbauende Strukturen, die aus einfachen, wiederverwendbaren Einzelkomponenten zusammengesetzt sind. In einer ersten Iteration des MRiL-Prozesses kann ein Akteur z. B. ein einfaches physikalisches Objekt ohne spezielle Funktionen sein, d. h. ein physikalisches Körper mit Eingängen für Kraft und Drehmoment sowie Ausgängen für Positionsinformationen. Durch das Aufsetzen eines eingangsseitigen Adapters für die Simulation eines Propellers mit Eingängen für Position und Orientierung sowie angelegte Energie wird aus dem einfachen physikalischen Körper ein aktiv steuerbares System. Durch einen ausgangsseitigen

Adapter kann z. B. ein Sensor zur Geschwindigkeitsmessung aufgesetzt werden. Dieses einfache Beispiel für eine Kopplung zwischen einem Akteur und einem Adapter ist in Abbildung 4.28 zu sehen. Andere Arten von Adaptern sind z. B. Typkonvertierungen für inkompatible Ports, Erweiterungen für intelligente Steuerungen (z. B. Reglersteuerungen, etc.) oder alle Arten von Sensoren und Akteuren.

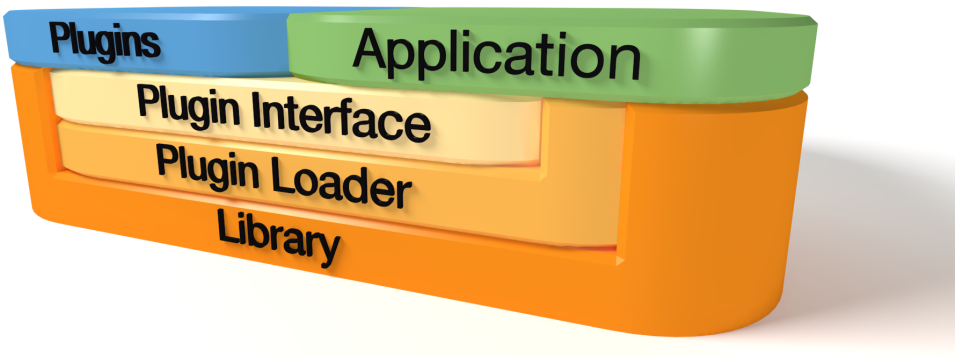


Abbildung 4.29: Das Plug-in-System von MiReAS.

Ein weiteres Konzept von MiReAS ist ein flexibles Plug-in-System, welches erlaubt, alle benötigten Komponenten separat zu implementieren, einzubinden und zu nutzen. Da Plug-ins zur Laufzeit geladen werden können, wird mit ihnen das modulare Konzept sowie die Erweiterbarkeit von MiReAS unterstützt. Für die größtmögliche Flexibilität sind sowohl der 3D-Renderer als auch die Trackingsysteme bzw. Videogeräte als Plug-in realisiert und können während der Laufzeit geladen werden. Auch Akteuren oder Adapter sind über Plug-ins realisiert. Somit ist auch die Entwicklung von Anwendungen in einem großen Team möglich, da sich die einzelnen Programmierer nur auf ihre Plug-ins konzentrieren müssen.

MiReAS unterscheidet zwischen zwei Arten von Plug-ins. System-Plug-ins, die essenziell für die Funktionalität sind und Erweiterungs-Plug-ins, mit denen vor allem Szenenelemente und Funktionserweiterungen für bestimmte Aufgaben erstellt werden können. Unter die System-Plug-ins fallen Komponentien wie der 3D Renderer, Tracker und Videoquellen, wobei Akteure, Adapter und Sensoren unter die Erweiterungs-Plug-ins fallen.

Damit die unterschiedlichen systeminternen sowie extern-angebundenen Komponenten innerhalb eines einzelnen Simulationsschrittes aktualisiert werden können, wurde MiReAS mit einem dreistufigen Simulationszyklus realisiert. Dies geschieht indem die Akteure, die über Ports miteinander verbunden sind, ihre Ausgabeports erst am Ende eines Simulationsschrittes aktualisieren. Diese Technik vermeidet, dass die Reihenfolge der einzelnen Aktualisierungen eine Aus-

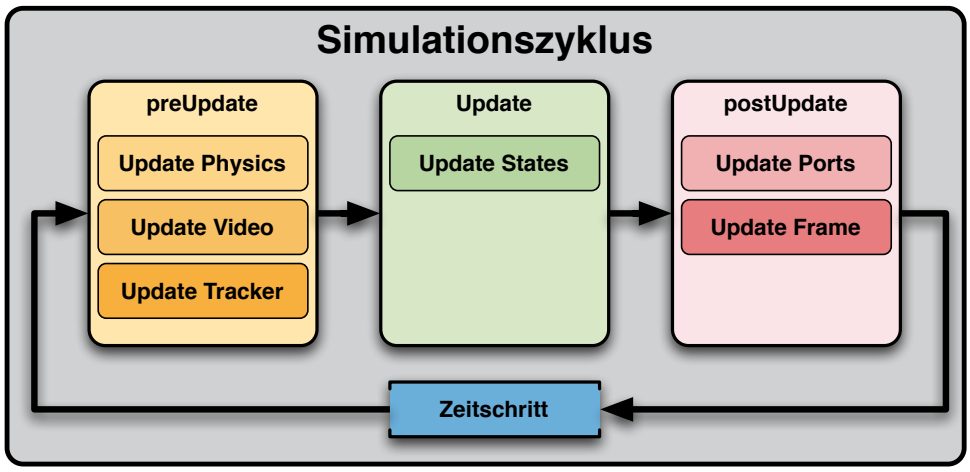


Abbildung 4.30: Der Simulationszyklus von MiReAS

wirkung auf die Simulation hat. Anders ist dies bei Berechnungen, die noch im selben Frame visualisiert werden sollen, wie z. B. Physikberechnungen. Diese Berechnungen müssen vor dem aktuellen Frame behandelt werden. Für eine größtmögliche Flexibilität wurde deshalb ein Simulationszyklus wie in Abbildung 4.30_{153} entworfen. Jede Komponente, die in diesen Simulationszyklus eingebunden wird, kann in jedem dieser drei Abschnitte Funktionalität hinterlegen. Es sollte allerdings weiterhin die Möglichkeit geben, innerhalb dieser Zykluselemente Prioritäten zu vergeben, um einen möglichst verzögerungsfreien Ablauf der Simulation zu gewährleisten. Daher wurde z. B. eine Visualisierung als letzte Phase im Simulationszyklus realisiert.

Um Rapid Prototyping zu unterstützten, wurde eine geeignete Methode zur Konfiguration der Anwendung durch den Entwickler realisiert, die ohne großen Programmieraufwand möglich ist. Es müssen lediglich die benötigten Akteure und Adapter einmal implementiert werden. Die Basis dieser Konfiguration bildet ein hierarchisches Datenmodell, welches im XML-Format abgelegt werden kann. Eine Szene wird durch einen Szenegraph abgebildet, in der alle Akteure, Adapter und Geräte samt Einstellungen integriert sind. Benötigte Dateien werden in einer eigenen Ordnerstruktur abgelegt. Ist eine erste Iteration erfolgreich abgeschlossen, kann die nächste Iteration direkt auf der bisherigen XML-Datei aufbauen und benötigte Änderungen direkt implementieren.

In der derzeitigen Entwicklung befindet sich eine grafische Benutzeroberfläche, die den Zugriff auf alle Systemkomponenten erlaubt, um die Konfiguration des Systems noch benutzerfreundlicher zu gestalten.

```
<Application name="SimplePhysics" dataDirectory="../data/SimplePhysics/" >
    <Logger level="INFO" selection="NONE" />
    <Timestep>0.02</Timestep>
    <Physics><Engine>Bullet</Engine></Physics>
    <Scene name="Scene"></Scene>
    <Renderer plugin="OSGRenderer" configFile="configs/OSGRendererConfig.xml" />

    <Physical name="Plane"  scene="Scene">
        <Mode>STATIC</Mode>
        <Visibility>On</Visibility>
        <Model>models/plane.osg</Model>
        <RenderMode>NORMAL</RenderMode>
        <CastShadow>On</CastShadow>
        <ReceiveShadow>On</ReceiveShadow>
        <Material>Normal</Material>
    </Physical>

    <Physical name="Cube" scene="Scene">
        <Mode>DYNAMIC</Mode>
        <Scale>0.5</Scale>
        <Pose>
            <Position x="0" y="-2.5" z="1"  />
            <Orientation x="0" y="0" z="0"  />
        </Pose>
        <Visibility>On</Visibility>
        <Model>models/simpleCube.flt</Model>
        <RenderMode>NORMAL</RenderMode>
        <CastShadow>On</CastShadow>
        <ReceiveShadow>On</ReceiveShadow>
        <Material>Normal</Material>
        <Mass>0.5</Mass>
    </Physical>

    <Physical name="RedSphere" scene="Scene">
        <Mode>DYNAMIC</Mode>
```

Abbildung 4.31: Konfiguration über XML.

Systemstruktur von MiReAS

Für die Implementierung des MiReAS-Systems wurde C++ als Programmiersprache gewählt. Das Design der Applikation wurde objektorientiert ausgelegt, so dass die nötigen Komponenten modular programmiert werden konnten. Eine Reihe Opensource-Bibliotheken, die Basisfunktionalitäten zur Verfügung stellen, wurden genutzt um den Implementierungsaufwand zu begrenzen. Bei der Auswahl wurde darauf geachtet, dass eine freie Verwendung sowie eine eventuelle Abänderung der Bibliotheken möglich ist. Auch sollten die Bibliotheken plattformübergreifend verwendet werden können. Entwickelt wurde das System jedoch vollständig auf einer Windows-Plattform, so dass eine Portierung auf ein anderes System zwar möglich ist, allerdings nicht entwickelt wurde.

Die verschiedenen Funktionen von MiReAS wurden in eigenen dynamischen Bibliotheken implementiert, um so eine hohe Modularität des Systems zu gewährleisten. Die Bibliotheken können daher unabhängig voneinander entwickelt werden, solange die Schnittstel-

len gleich bleiben. Beispielsweise werden die Grundfunktionen einiger Bibliotheken, z. B. der Physik- und der Input-Bibliothek, über Opensource-Bibliotheken realisiert, die bei Bedarf durch andere Bibliotheken ausgetauscht werden können.

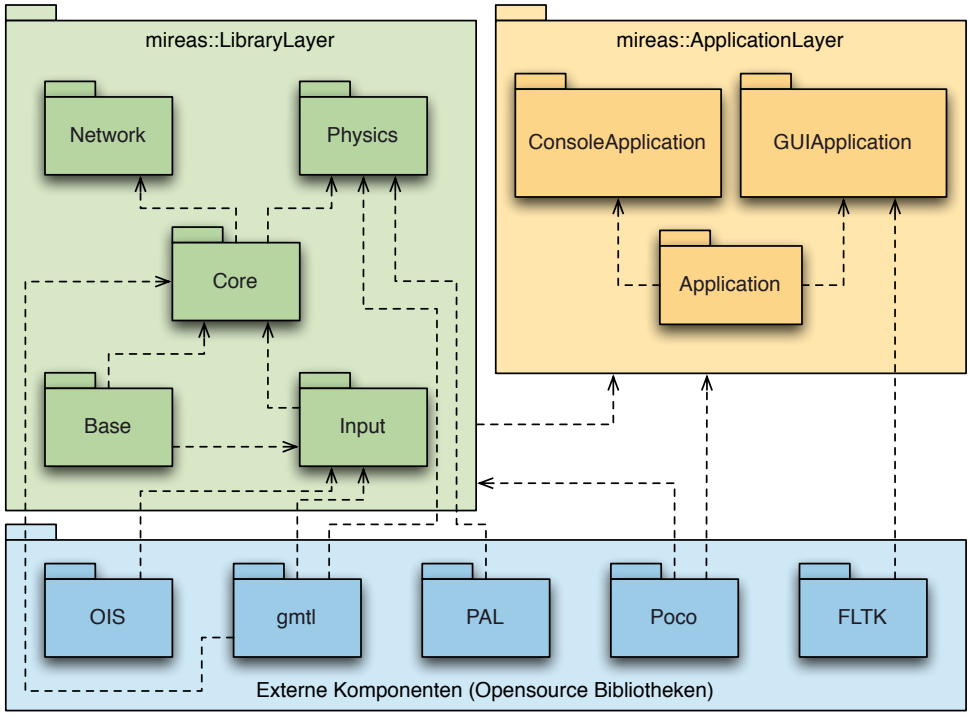


Abbildung 4.32: Die Systemstruktur von MiReAS

In Abbildung 4.32 wird die Softwarestruktur von MiReAS dargestellt. Die externen Komponenten, die über Opensource-Bibliotheken realisiert sind, bieten bereits fertige Strukturen, die von MiReAS genutzt werden, z. B. die Unterstützung von unterschiedlichen Eingabegeräten, eine Physiksimulation, mathematische Funktionen und eine 2D-GUI-Bibliothek. Das eigentliche MiReAS-System kann in zwei Komponenten geteilt werden: den *Library-Layer* und den *Applications-Layer*. Der *Library-Layer* beinhaltet die gesamte Simulationslogik und ist in die Bibliotheken *Base*, *Input*, *Core*, *Physics* und *Network* unterteilt. Der *Applications-Layer* enthält die ausführbaren Anwendungen. Je nach Wunsch des Anwenderprogrammierers kann die fertige Applikation von einer Konsolenanwendung (*ConsoleApplication*), die durch eine XML-Konfigurationsdatei erstellt wird, oder eine Applikation mit grafischer Benutzeroberfläche (*GUIApplication*) abgeleitet werden. Dabei bietet die GUI eine flexiblere Möglichkeit zum Eingriff in das laufende Programm und kann zur Laufzeit nachkonfiguriert werden. In der Bibliothek *Application* werden gemeinsam verwendete Funktionen, wie z. B. das Konfigurationssystem implementiert.

Es wurden folgende externe Opensource-Bibliotheken für die Ent-

wicklung von MiReAS genutzt:

POCO (Portable Components): Eine Bibliothek für die Entwicklung von portablen, netzwerkzentrierten Anwendungen. Implementiert sind z. B. ein Threadsystem, ein System zum Laden von dynamischen Bibliotheken, Sockets und Netzwerkprotokollen, ein Konfigurationssystem und ein XML-Parser [Obi10]. In MiReAS wurde gerade das System zum Einlesen von Konfigurationsinformationen aus XML-Dateien intensiv genutzt.

GMTL (Generic Math Template Library): Eine mathematische, auf Template-Klassen basierende Bibliothek, die Funktionen der linearen Algebra zur Verfügung stellt [BMS10]. Diese werden vor allem in 3D-Anwendungen benötigt. Des Weiteren beinhaltet GMTL allgemeine mathematische Funktionen, die plattformübergreifend dieselbe Funktionalität bieten.

OIS (Object-Oriented Input System): Eine Bibliothek zur Abstraktion von Plattform-Schnittstellen. OIS bietet die Möglichkeit der Abstraktion sämtlicher Standard-Eingabegeräte wie z. B. Tastatur, Maus oder Joystick [Cas10]. Weitere Eingabegeräte, die z. B. für Mixed Reality Anwendung benötigt werden, können über OIS einfach integriert werden. OIS wurde für die Verwendung in MiReAS durch eigene Input-Handler erweitert, z. B. eine bessere Anbindung an die Nintendo Wiimote.

PAL (Physics Abstraction Layer): Bibliothek zur Abstraktion physikalischer Simulationen [Boe09]. In PAL ermöglicht die Nutzung eine Vielzahl von Physik-Bibliotheken innerhalb einer einzelnen Anwendung. Die einzelnen Physik-Bibliotheken müssen jeweils als Plug-in vorhanden sein bzw. als PAL-Plug-in implementiert worden sein. Somit besteht eine große Flexibilität bei der Auswahl der Physik-Bibliotheken. Nachteil der Abstraktion ist jedoch die fehlende Unterstützung spezieller Merkmale bestimmter Physik-Bibliothek, es können nur allgemeine Funktionen genutzt werden. In MiReAS wird das PAL-Plug-in der BulletEngine als Standard Bibliothek genutzt [Cou10].

FLTK (Fast Light Toolkit): Eine Bibliothek, die eine plattformunabhängige grafische Benutzerschnittstelle bereitstellt [Spi10]. Die Bibliothek basiert auf OpenGL und über den integrierten Designer FLUID (**F**ast **L**ight **U**ser-**I**nterface **D**esigner) können Benutzerschnittstellen sehr schnell grafisch erstellt werden.

Diese Bibliotheken sind auch in der Abbildung 4.32_{155} zu sehen und werden von den verschiedenen Layern in der MiReAS Architektur

genutzt. Alle verwendeten Bibliotheken stehen unter einer Open-Source-Lizenz, z. B. GPL, LGPL, Boost License, etc., und sind somit allgemein zugänglich und änderbar. Des Weiteren wurde darauf geachtet, dass die verwendeten Bibliotheken für mehrere Betriebssysteme zur Verfügung stehen, so dass eine Cross-Plattform Tauglichkeit grundsätzlich vorstellbar ist.

Weitere Einzelheiten zur Implementierung können in der Master-Arbeit von Patrick Pogscheba nachgelesen werden [Pog09].

4.6 Zusammenfassung

In diesem Kapitel wurde beschrieben, was das MRiL-Entwurfsvorgehen ist und wie es anwendet werden kann. Zu Beginn wurden die Vorraussetzungen definiert, die benötigt werden, damit das MRiL-Entwurfsvorgehen erfolgreich bei einer Applikationsentwicklung angewendet werden kann. Dabei ist es wichtig, dass die zu entwickelnde Anwendung in die MVCE-Komponetnen aufgeteilt werden kann und aus einem der Bereiche Mixed Reality Applikationen, Mixed Reality Benutzerschittstellen oder mechatronische Systeme stammt. Diese Anwendungen lassen sich sehr gut durch das MRiL-Entwurfsvorgehen entwickeln.

Nach Klärung der Voraussetzungen wurde auf grobe Vorgehensweise des MRiL-Entwurfsvorgehens eingegangen. Hier wurde gezeigt, wie ein Entwickler erfolgreich den Prozess an seiner Applikation anwenden kann. Die einzelnen Schritte bei der Entwicklung wurden beschrieben und erklärt.

Der Vorgehensweise folgte die Beschreibung der für das MRiL-Entwurfsvorgehen entwickelten Methoden. Dazu gehörte das MVCE Architekturmuster, das die Anwendung in vier verschiedene Komponenten einteilt und es ermöglicht, diese Komponenten unabhängig von einander zu entwickeln bzw. verfeinern. Die MRiL-Metrik erlaubt eine Bewertung des Entwicklungsstaus der Applikation anhand der MVCE Komponenten. Über das Akteurmodell wurden dann verschiedene Teile der Applikation nochmals gekapselt. Akteure sind eine Verfeinerung des MVCE Architekturmusters und teilen die jeweiligen Komponenten in kleinere autarke Teile auf. Daraufhin wurde das Entwurfsvorgehen nochmals in allen Details beschrieben und erklärt, wie das MRiL-Entwurfsvorgehen verwendet wird. Diese Akteure lassen sich konzeptionell mit Hilfe von Adaptern erweitern um so eine Verfeinerung der Komponenten mit geringem programmiertechnischen Aufwand zu bewältigen.

Das Entwurfsvorgehen beschreibt den iterativen Entwurfsprozess. Abstrakt gesehen werden die Phasen Konzeption, Implementierung, Tests und Bewertung durchlaufen und ggf. wiederholt. Konkret wird in jeder Initialisierungsphase des MRiL-Entwurfsvorgehens der nachfolgende Verfeinerungsschritt geplant. In der Verfeinerugsphase werden die Änderungen implementiert und so die Komponenten bzw. Akteure verfeinert. Nach der Verfeinerung entsteht ein neuer Prototyp, der über funktionale Tests oder Benutzertests validiert und analysiert wird. Hier entscheidet sich, ob eine weitere Iteration nötig ist oder ob der Prototyp funktional fertig ist. Sollte eine weitere Iteration vorgenommen werden, werden in der Bewertungsphase die Ziele des nächsten Prototypen festgelegt.

Damit der MRiL Prozess softwaretechnisch unterstützt wird, wurden zwei unterschiedliche Softwareumgebungen entwickelt. Als erstes wurde die proprietäre Entwicklungsumgebung Virtools dahingehend erweitert, dass der MRiL Prozess zum größten Teil abgebildet werden konnte. Hierfür wurden mehrere Plug-ins entwickelt, die u. a. das Tracking, Benutzung neuer Eingabegeräte und die Interapplikations-Kommunikation ermöglichten. Damit konnte das MRiL-Entwurfsvorgehen angewendet werden, da die Anwendung in die entsprechenden MVCE-Komponenten aufgeteilt und diese Komponenten dann verfeinert werden konnte. Leider konnte das Prinzip der Akteure nicht in Virtools umgesetzt werden, da es keine Möglichkeit gab, dieses abzubilden. Aus diesem Grund wurde eine komplett neue Softwareumgebung, die speziell auf das MRiL-Entwurfsvorgehen angepasst war, entwickelt. MiReAS wurde auf Opensource-Bibliotheken entwickelt und bietet alle Möglichkeiten, die das MRiL-Entwurfsvorgehen benötigt, vom MVCE Architekturmuster bis hin zur Akteurmodell. Letzteres ist in MiReAS ein zentraler Punkt bei der Entwicklung, da jede Komponente der Applikation über einen Akteur abstrahiert wird. Damit Akteure verfeinert werden können und nicht immer von Grund auf neu implementiert werden müssen, wurde das Prinzip des Adapters, der einen Akteur mit neuer Funktionalität kapseln kann, in MiReAS realisiert.

In diesem Kapitel wurde somit das komplette MRiL-Entwurfsvorgehen mit der entsprechenden Softwareumgebung vorgestellt und erläutert. Im folgenden Kapitel wird nun gezeigt, wie der MRiL Prozess an einem nicht trivialen Beispiel erfolgreich eingesetzt wurde. Es werden die einzelnen Schritte der Entwicklung vorgestellt und es wird auf auftretende Probleme hingewiesen.

KAPITEL 5

Beispiel

Nach der Vorstellung des MRiL-Entwurfsvorgehes im letzten Kapitel konzentriert sich dieses Kapitel auf die Anwendung von MRiL auf ein nicht triviales Beispiel. Als Softwaregrundlage diente hier das Werkzeug MiReAS $4.5.2_{148}$ unter Verwendung u. a. des Akteurmodells und des Prinzip des Adapters $4.3.3_{120}$. In diesem Kapitel wird detailliert die Entwicklung der Prototypen dieser Beispielapplikation beschrieben und jede Iteration der Entwicklung erläutert. Insgesamt wurden während der Entwicklung sieben Prototypen entwickelt, die jeweils eine geforderte Funktionalität implementierten. Angefangen von einer sehr einfachen, abstrakten Applikation, die die Funktionsweise verdeutlichen soll, bis hin zu komplexen Prototypen, die für Tests einer speziellen Ausprägung der Applikation verwendet wurden. Am Ende wurde ein Prototyp entwickelt, der als fertige Applikation bezeichnet werden kann.

5.1 Überblick

An der Fachhochschule Düsseldorf wurde im Fachbereich Elektrotechnik ein ferngesteuerter Indoor-Zeppelin zu Forschungszewecken aufgebaut. In Kooperation mit der VR-Abteilung des Fachbereich Medien der Fachhochschule Düsseldorf sollten im Rahmen des Projektes MoVeIT (Mobilität, Verteilung und Interaktion: Realisierung einer Testumgebung für Multimediaanwendungen) neue, intuitive Steuerstrategien für diesen Zeppelin entwickelt werden. Für die Entwicklung dieser Strategien sollte das in dieser Arbeit vorgestellte MRiL-Entwurfsvorgehen angewendet und mit Hilfe von MiReAS

realisiert werden.

Die Idee für die Entwicklung neuer Steuerstrategien entstand aus der Tatsache, dass die normale Steuerung des Zeppelins mit Hilfe einer Funkfernsteuerung kompliziert und schwer zu erlernen ist. Das liegt zum einen an den wenigen Freiheitsgraden, die durch die Bauweise des Zeppelins gegeben sind, und zum anderen durch die Trägheit des Zeppelins. Um ihn präzise und genau zu steuern ist es für den Benutzer wichtig, die zu erzielende Bewegung in richtigen Bewegungen der einzelnen Freiheitsgrade des Zeppelins aufzuspalten, dabei die Trägheit mit zu berücksichtigen und ggf. gegenzusteuern. Die ersten Versuche, den Zeppelin kontrolliert zu steuern, wurden deshalb in einer großen Halle durchgeführt, um die Kollisionsgefahr mit Hindernissen zu minimieren. Nach mehreren Stunden intensiver Übung war schließlich ein Benutzer in der Lage, den Zeppelin halbwegs sicher zu fliegen.

Um auch einer größeren Gruppe an Benutzern die Bedienung des Zeppelins zu ermöglichen, war die Idee, die klassische Steuerung des Zeppelins durch eine neue, intuitivere Steuerung zu ersetzten. Da die Umsetzung einer neuen intuitiven Steuerung sehr of das „Trail-and-Error"-Prinzip verwendet, gerade wenn es sich um das Finetuning bestimmter Parameter der Steuerung handelt, war der Einsatz des realen Zeppelins für diese Tests schon von Beginn an ausgeschlossen. Durch unsachgemäße Handhabung kann der Zeppelin schnell beschädigt oder zur Gefahr für Personen werden, so dass die Entwicklung einer neuen Steuerung prädestiniert für eine VR Simulation ist. Des Weiteren gab es viele Ideen einer intuitiven Steuerung, so dass mehrere dieser Ideen getestet werden sollten. Auch hier war eine VR Simulation die beste Lösung.

Zu diesem Zeitpunkt entstand die Idee das MRiL-Entwurfsvorgehen zu verwenden und die Prototypen mit Hilfe der MiReAS-Software zu entwickeln. Die Überlegung war, erst eine sehr einfache Applikation zu entwickeln, die grob das Verhalten des Zeppelins simuliert, um so Steuerstrategien mit diesem Prototypen entwickeln und testen zu können. Um in weiteren Entwicklungsschritten auf den vorherigen Prototypen aufzubauen, wurde MiReAS dazu verwendet, langsam die VR-Komponenten in MR bzw. reale Komponenten zu ersetzten. So konnte die Steuerung immer feiner getestet werden, erst an einfachen, später an komplexen Simulationen sowohl virtuell als auch real.

Für die Berechnung der Metriken war es notwendig, den endgültigen Prototypen der Anwendung zu spezifizieren. Das war in allen Bereichen nicht sehr kompliziert, da schon der reale Zeppelin existierte und die Applikation diesen steuern sollte. Über die Metriken war eine

Einschätzung des Entwicklungsstatus mit Hilfe des Kiviatgraphen möglich. Für die Controller-Metrik wurden allerdings keine groß angelegten Benutzertests durchgeführt, da hier die Zeit und die Personen fehlten. Es wurden hier nur die Aussagen der Entwickler berücksichtigt um so eine Einschätzung der verwendeten Controller-Strategie zu erhalten.

Insgesamt wurden für dieses Beispiel zehn aufeinander aufbauende Prototypen entwickelt und getestet, wobei zwei Beispiele nur konzeptionell entwickelt wurden, da hier die Hardware, die eingesetzt werden sollte, nicht rechtzeitig fertiggestellt werden konnte. Alle Prototypen wurden aufeinander aufbauend entwickelt und mit Hilfe von dem Werkzeug MiReAS realisiert.

5.1.1 Der Zeppelin

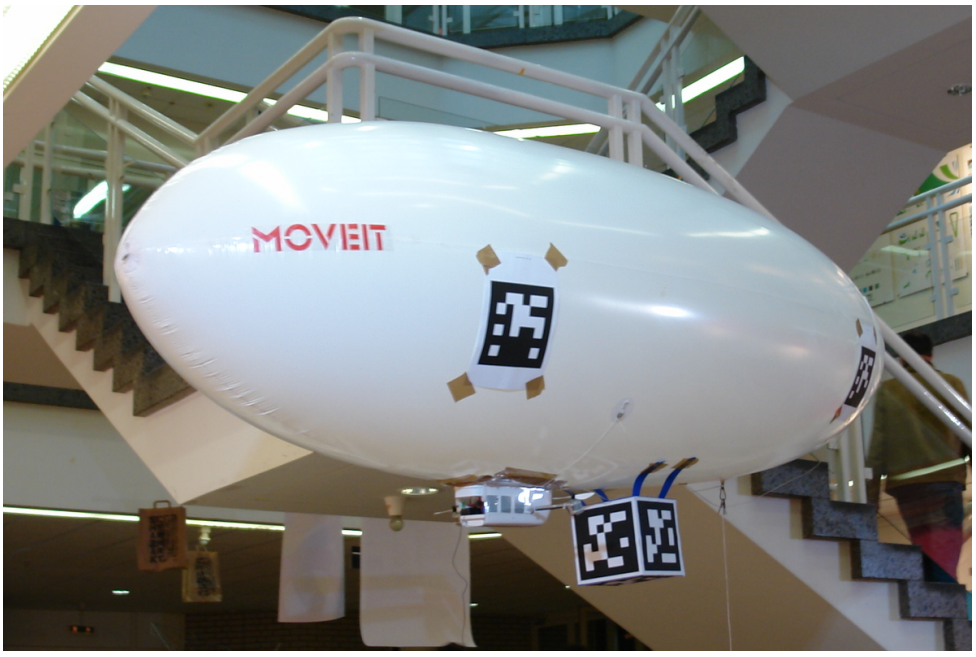


Abbildung 5.1: Modell des Zeppelins.

Das Modell des Zeppelins (Abbildung 5.1, hier beim Testen des AR-Prototypen mit Markern) besteht aus einer speziellen Kunststoffhülle mit einer Länge von drei Metern und einem Durchmesser von einem Meter. Befüllt wird die Hülle mit Helium, was den erforderlichen Auftrieb des Zeppelins liefert. Dabei beträgt die Tragkraft des Zeppelins exklusiv der Elektronik und der Motoren ca. 250 Gramm. Am Heck des Zeppelins befindet sich ein Propeller, der sich über einen DC-Motor rechts bzw. links drehen lässt. Unten an der Hülle ist eine Gondel befestigt, die die Bordelektronik fasst. An beiden Seiten

der Gondel befindet sich jeweils ein weiterer Propeller, der auf einer drehbaren Achse gelagert ist.

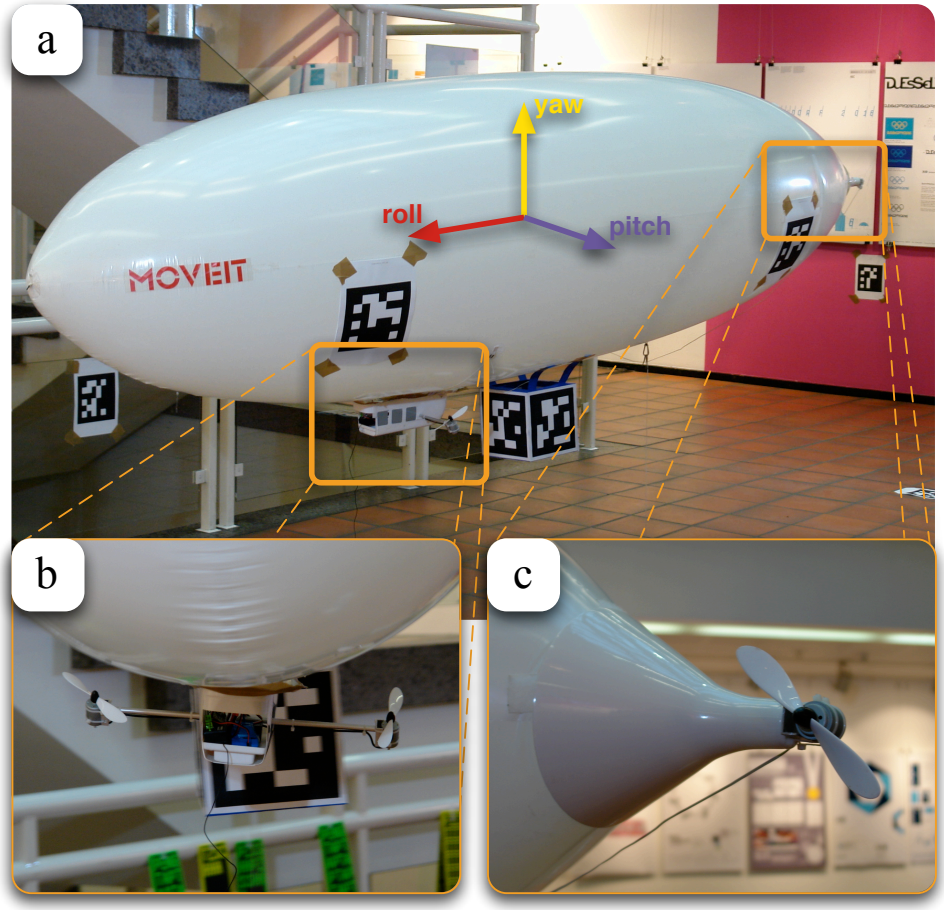


Abbildung 5.2: Der Zeppelin im Detail.

Der Propeller am Heck des Zeppelins (Abbildung 5.2 – c) ist für das Gieren[1] zuständig und kann über den DC-Motor in beide Richtungen betrieben werden. Die beiden Schubmotoren an der Gondel (Abbildung 5.2 – b) ermöglichen die Bewegung in der Ebene, die von der Gierachse und der Rollachse[2] aufgespannt wird, also vorwärts, rückwärts, aufwärts oder abwärts. Beide Schubmotoren sind über eine Drehachse miteinander verbunden, die je nach Stellung die Schubrichtung der Propeller vorgibt. Dabei ist die Drehung um diese Achse auf 360° beschränkt (aus der Normalstellung die horizontal nach vorne zeigt, jeweils 180° in beide Richtungen). Beide Schubmotoren ermöglichen eine maximale Geschwindigkeit von ca. $6,1\frac{m}{s}$.

[1] Die Gierachse, auch Hoch- bzw. Vertikalachse (engl. yaw axis), bezeichnet die vertikale Achse eines Luftfahrzeugs, um die sich das Fahrzeug dreht. Als Gieren bezeichnet man eine Drehbewegung um diese Achse. In Abbildung 5.2 – a sind die Achsen zur Verdeutlichung in den Zeppelin eingezeichnet.

[2] Die Rollachse (engl. roll axis) wird auch als Längsachse bezeichnet.

Abbildung 5.3: Handelsübliche 4-Kanal Funkfernbedienung.

In der Gondel befindet sich eine Platine mit der Bordelektronik, die aus der Motorsteuerung sowie der Kommunikation mit der Fernbedienung besteht. Wie im Flugzeug-Modellbau üblich wird der Zeppelin über eine handelsübliche 4-Kanal Fernbedienung, wie in Abbildung 5.3_{163} zu sehen, gesteuert. Über die einzelnen Hebel der Fernbedienung können der Heckrotor, die Drehachse und die beiden Schubmotoren gesteuert werden. Beide Schubmotoren sind miteinander gekoppelt, so dass sie nur gemeinsam steuerbar sind. Daraus ergibt sich, das drei der vier Kanäle der Fernbedienung mit Funktionen belegt sind. Der linke Hebel ist vertikal für die Steuerung der Schubmotoren belegt, der rechte Hebel ist vertikal für für den Winkel der Schubmotoren zuständig. Des Weiteren ist der linke Hebel auf der horizontalen Achse für den Heckmotor verantwortlich. Die horizontale Achse des rechten Hebels ist nicht belegt.

5.2 Prototypenentwicklung

Wie schon im Überblick erwähnt, bedarf es einer gewissen Erfahrung, den Zeppelin präzise zu steuern. Die korrekte Steuerung der drei beschriebenen Freiheitsgrade (Rotation und Translation in einer Ebene) gelingt aufgrund der Trägheit des Zeppelins nur mit viel Übung. Selbst kleinste Impulse der einzelnen Rotoren können große Auswirkungen auf die Bewegung des Zeppelins haben. Daher ist ein häufiges Gegensteuern für exakte Manöver unumgänglich.

Bei der Prototypentwicklung sollen nun verschiedene Techniken ent-

wickelt werden, die die Steuerung des Zeppelins vereinfachen und auch für ungeübte Benutzer anwendbar sind. Die Kommandos an die einzelnen Rotoren sollen durch eine High-Level Steuerung abstrahiert werden, so dass sich der Zeppelin durch einfach Befehle wie beispielsweise „Vorwärts“, „Rückwärts“, „Aufwärts“ oder „Abwärts“ steuern lässt. Die Fernsteuerung soll weiterhin durch andere Eingabegeräte ersetzt werden, die eine intuitivere Steuerung des Zeppelins versprechen, beispielsweise Gestensteuerung oder Stellvertreterobjekte.

Da die Arbeit am realen Zeppelin einerseits mit hohen laufenden Kosten[3] verbunden ist und andererseits durch die Kooperation der FH Düsseldorf und der Universität Paderborn der Zeppelin nicht an beiden Standorten verfügbar war, sollte die Entwicklung mit Hilfe einer Simulation durchgeführt werden. Die zu entwickelnden Steuerstrategien sollten erst an der Simulation getestet und später dann auf den realen Zeppelin übertragen werden. Diese Schritte sollten mit Hilfe des MRiL-Entwurfsvorgehens realisiert und unter Verwendung des Werkzeuges MiReAS implementiert werden.

Die folgenden Abschnitte beschreiben die iterative Entwicklung der zehn Prototypen mit Hilfe des MRiL-Entwurfsprozesses am Beispiel des Zeppelins. Dabei wurden alle Prototypen mit Hilfe des MiReAS Frameworks entwickelt. Bis auf die letzten drei Prototypen, die nur konzeptionell entwickelt wurden, sind alle Prototypen komplett lauffähig.

5.2.1 Die Initialphase

Zu Beginn der Initialisierungsphase wurde die endgültige Applikation in schriftlicher Form festgehalten, um so eine Basis für die Entwicklung zu erhalten. Die schriftliche Ausarbeitung, die dem ersten Schritt der Vorgehensweise aus Kapitel 4.2_{101} entspricht, wurde so detailliert wie möglich verfasst, um die Entwicklung zu vereinfachen und die Berechnung der Metriken zu ermöglichenoo. Aus der schriftlichen Form wurden im folgenden Schritt die einzelnen Teile der Applikation identifiziert und in die einzelnen Komponenten des MVCE Architekturmusters eingeordnet. Diese erste Einteilung war eine sehr grobe Einteilung der Komponenten, die allerdings sowohl für die Entwicklung als auch für die Berechnung der einzelnen Werte der Metrik ausreichte.

Daraus entstand die unten angegebene Tabelle mit den folgenden Komponenten:

[3] Da durch die Kunststoffhülle des Zeppelin leider immer etwas Helium diffundiert, muss sie häufig nachgefüllt werden, eine Füllung mit Helium kostet derzeit ca. 40,00 €.

Komponente	MVCE-Kategorien
Zeppelin	**Modell**, View
Steuerung	**Controller**, View
Umgebung	**Environment**, Modell, View
Metadaten	**View**, Modell, Controller, Environment

Aus der obigen Tabelle ist ersichtlich, in welche Kategorien die einzelnen Komponenten der Applikation fallen. Dabei entsprechen die hervorgehobenen Komponenten der primären MVCE-Kategorie, der sie zugeordnet werden. Der Zeppelin muss dementsprechend als Modell vorliegen, um das Verhalten abzubilden, sollte aber auch eine visuelle Repräsentation besitzen. Die Steuerung wird im Controller abgebildet, kann jedoch auch eine visuelle Repräsentation haben. Die Umgebung ist sowohl dem Environment zugeordnet, da wir keinen Einfluss in der Software auf Ereignisse der Umgebung haben, kann aber auch eine Entsprechung im Modell und im View haben, wenn in späteren Prototypen die Daten der Umgebung zur Kollisionserkennung benutzt werden sollen. Die Metadaten stehen in diesem Zusammenhang für z. B. Debuginformationen, die in allen Kategorien erzeugt werden können und im View visualisiert werden sollen.

Nach der ersten groben Einteilung der Komponenten kann nun mit der Arbeit am ersten Prototypen begonnen werden. Zunächst werden alle Akteure des ersten Szenarios identifiziert. Das erste Szenario soll einfach gehalten werden und zur Planung und Kommunikation des Entwicklungsteams dienen. Des Weiteren soll die Entwicklung des ersten Prototypen schnell und unkompliziert sein. So sind im ersten Szenario wenig Akteure und auch der View und der Controller sind einfach gehalten. Die Umgebung ist rein virtuell und basiert weder auf simulierten noch auf realen Daten der echten Umgebung. Eine Interaktion mit der Umgebung ist nicht vorhanden, sie besitzt nur eine visuelle Repräsentation. Das Modell des Zeppelins ist repräsentiert durch eine Transformation, die das visuelle Modell des Zeppelins in der virtuellen Welt positioniert. Die Steuerung erfolgt über die Tastatur des Rechners.

Es wurden somit für das erste Szenario die aufgeführten Akteure definiert, die in der nachfolgenden Tabelle angegeben sind:

Akteur	Modell	View	Controller	Environm.
Zeppelin	×	×		
Umgebung		×		
Tastatur			×	

Für die Einordnung des Prototypen und die Aussage über den Entwicklungsstand müssen die Metriken, die in Kapitel 4.3.2$_{109}$ definiert wurden, berechnet werden. Dazu ist es notwendig eine Definition der endgültigen Applikation zu erstellen. Dies wurde schon im ersten Schritt in schriftlicher Form realisiert, so dass jetzt nur noch die jeweiligen MVCE-Komponenten extrahiert und beschrieben werden müssen. Dies wurde für die Modell-Metrik Γ_M, die View-Metrik Θ_V und die Environment-Metrik Ω_E durchgeführt.

In den nun folgenden Tabellen werden die Komponenten, die ausschlaggebend für die finale Version sind, aufgeführt, ob es sich um Ein- bzw. Ausgaben handelt und, wenn dies möglich ist, von welchem Typ sie sind und in welchem Wertebereich sie liegen. Für das endgültige Modell M_{final} ergab sich folgende Einteilung, die als Berechnungsgrundlage der Modell-Metrik Γ_M verwendet wurde:

Modell-Komponenten	Eingabe	Ausgabe
Heckrotor	$[-1, 0, \ldots, 1, 0]$	–
Seitenrotoren	$[-1, 0, \ldots, 1, 0]$	–
Winkel Seitenrotoren	$[-180, 0, \ldots, 180, 0]$	–
Flughöhe	–	$[0, 0, \ldots, 10, 0]$

Das Modell beschreibt hier die Ein- und Ausgaben der Hardware des Zeppelins, die von der Software verwendet werden können. Der Höhensensor wurde dabei speziell entwickelt und soll für die fortgeschrittenen Steuerstrategien zum Einsatz kommen. Dabei wurde der Höhensensor zu Anfang als Softwarekomponente realisiert, die die Höhe des Zeppelins in der virtuellen Umgebung zurückgibt. Später wurde eine spezielle Hardwarekomponente in den Zeppelin verbaut, die die reale Höhe des Zeppelins mit Hilfe von Luftdruck ermittelte. Zählen wir die in der oben angegebenen unterschiedlichen Ein- und

Ausgabequellen für das endgültige Modell M_{final} zusammen erhalten wir einen Wert von Vier ($M_{final} = 4$).

Für das Environment E_{final} wurden folgende Parameter für die Berechnungsgrundlage der Environment-Metrik Ω_E festgestellt:

Environment-Komponenten	Eingabe	Ausgabe
Hindernisse	–	Tracking
Umgebung	–	Tracking
Umwelteinflüsse	–	Gyroskop, Tracking

Beim Environment soll in späteren Prototypen Hindernisse erkannt und die Position der Umgebung relativ zu einer festen Kamera bestimmt werden, um so bestimmte Manöver ausführen zu können. Des Weiteren sollen Umwelteinflüsse, die auf den Zeppelin wirken, wie z.,B. Gegen- oder Seitenwind, mit Hilfe eines Gyroskops[4] oder eines Camera-Tracking erkannt und darauf reagiert werden. Damit ergeben sich laut obiger Tabelle drei Komponenten für das finale Environment ($E_{final} = 3$).

Für den View V_{final} wurden folgende Parameter für Θ_V festgelegt:

View-Komponenten	Eingabe	Ausgabe
Zeppelin	–	Realer Zeppein
Umgebung	–	Reale Umgebung
Zustand	Flughöhe, Position, etc.	Visualisierung in VR/AR
Modell	Modellparameter	Visualisierung des Modells

Für den View soll der endgültige Prototyp der reale Zeppelin in der realen Umgebung sein, vorzugsweise sollen bestimmte Parameter entweder durch AR- oder durch VR-Techniken visualisiert werden. Die Techniken richten sich dabei nach der aktuellen Darstellung. Wir

[4] Da sich der Zeppelin bei Gegenwind um die Querachse (pitch axis in Abbildung 5.2$_{162}$) neigt, kann er über über ein Gyroskop erkannt werden.

erhalten somit vier Komponenten für finalen View ist somit ($V_{final} = 4$).

Der eingesetzte Controller wird (zumindest bei neuen Steuerstrategien) mit Hilfe von Benutzertests gewertet, um einen Wert für die Metrik zu erhalten. Diese Benutzertests sind hier allerdings klein gehalten und werden meist nur von den Entwicklern selbst ausgeführt. Somit ist die Einordnung eher subjektiv.

5.2.2 Der erste Prototyp: Eine einfache VR Version

Wie schon im Kapitel 5.2.1$_{164}$ beschrieben soll der erste Prototyp zur Planung und Kommunikation des Entwicklungsteams dienen und dementsprechend einfach gehalten werden. Über diesen sehr einfachen virtuellen Prototypen können Manöver visualisiert und so besser im Team besprochen werden.

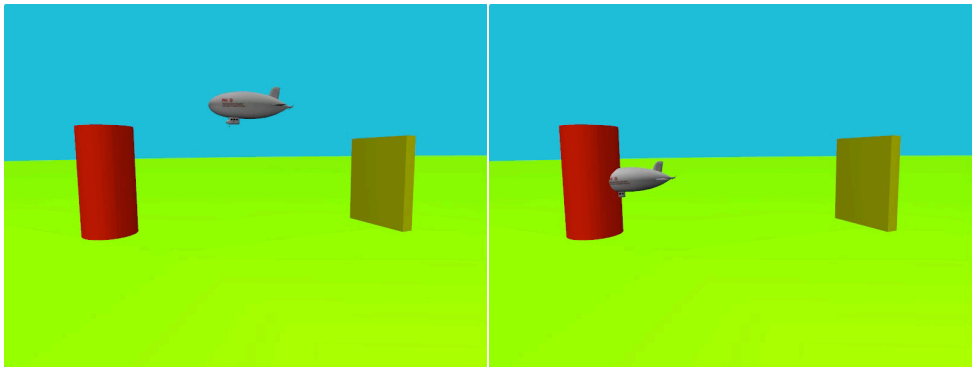


Abbildung 5.4: Bilder aus dem ersten Prototypen.

Die Entwicklungszeit für diesen Prototypen war sehr kurz, dabei wurde die meiste Zeit die Modellierung des 3D-Modells des Zeppelins verwendet. Das 3D Modell wurde in MiReAS als spezieller Akteur (*RenderableActor*) implementiert. Über einen *KeyboardManipulator*, der auf Tastatureingaben reagieren kann, wurde die Transformation des Zeppelins in der virtuellen Welt gesteuert. Die Umgebung wurde durch eine einfache Bodenplatte (*Groundplane*) realisiert. Des Weiteren wurden zwei einfache 3D Objekte in die Szene eingefügt, um die der Zeppelin gesteuert werden kann. Es wurde jedoch keine Kollisionserkennung in diesen Prototypen eingebaut, so dass der Zeppelin auch durch diese Objekte gesteuert werden kann. In Abbildung 5.4 sind einige Screenshots vom ersten Prototypen abgebildet.

Um nun über die Metriken festzustellen, wie weit der Prototyp entwickelt ist, muss dieser mit dem finalen Prototypen verglichen werden. In der folgenden Tabelle werden die in dem Szenario verwendeten

Komponenten entsprechend klassifiziert und der Wert der Metrik berechnet.

Modell-Metrik Γ_M		
σ_i	Typ	ϵ_i
Zeppelin Transformation	Virtuell	0

$$\mathbf{\Gamma_M = \frac{0}{4} = 0}$$

Der Wert für die Modell-Metrik Γ_M ist somit für diesen Prototypen 0, da noch keines der erwarteten Komponenten implementiert ist. Die Bewegung des Zeppelins über eine einfache Manipulation der Transformationsmatrix ist eine rein virtuelle Lösung für diesen Prototypen.

View-Metrik Θ_V		
ω_i	Typ	ϕ_i
3D Modell des Zeppelins	Virtuell	0,5
Virtuelle Umgebung	Temporär	0

$$\mathbf{\Theta_V = \frac{0{,}5}{4} = \frac{1}{8}}$$

Die View-Metrik Θ_V ist für diesen Prototypen $\frac{1}{8}$, da als einzige Komponente der Zeppelin als virtuelles 3D Modell existiert. Es ist zwar eine virtuelle Umgebung in dem Szenario vorhanden, allerdings entspricht es nicht der Realität, ist also nur temporär für diesen Prototypen implementiert.

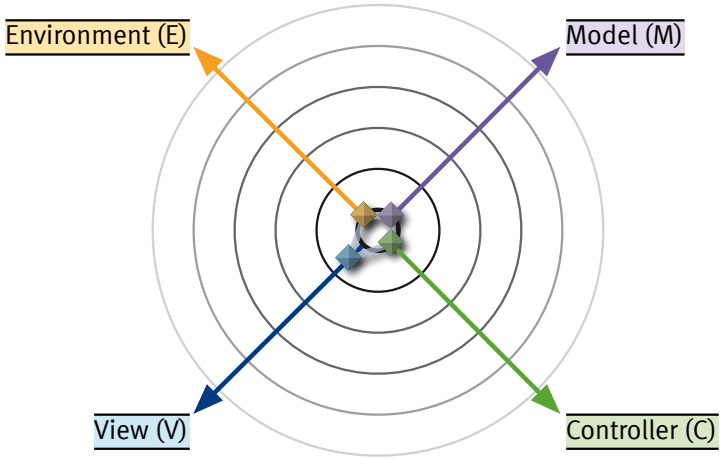


Abbildung 5.5: Kiviatgraph des ersten Prototypen.

Die Environment-Metrik Ω_E ist bei diesem Prototypen 0, da keine Informationen der realen Umgebung, weder simuliert noch per Sensor, ermittelt werden.

Da die Steuerung über die Tastatur des Computers geschieht, wur-

den keine Benutzertests durchgeführt, so dass sich auch hier für die Controller-Metrik Ψ_C 0 ergibt.

Werden nun die einzelnen Metriken auf die entsprechenden Achsen des Kiviatgraphen abgetragen, erhalten wir den Entwicklungsstand für den vorliegenden Prototypen, wie er in Abbildung 5.5_{169} dargestellt ist. An dem Kiviatgraphen kann man gut erkennen, dass die Entwicklung in einem sehr frühen Stadium ist. Der einzige Wert, der nicht Null ist, ist der View, da hier schon ein einigermaßen genaues Modell des Zeppelins als Visualisierung verwendet wird. Alle anderen Komponenten sind nur temporär, sie werden früher oder später ersetzt.

5.2.3 Der zweite Prototyp: Virtueller Prototyp mit Physiksimulation

Auf dem ersten Prototypen aufbauend wurde der zweite Prototyp entwickelt, bei dem das realistische Verhalten des Zeppelins im Mittelpunkt stand. Wurde im ersten Prototypen die Position des Zeppelins über die direkte Manipulation des Transformationsmatrix realisiert, sollte beim zweiten Prototyp ein physikalisches Modell zum Einsatz kommen, welches das Verhalten des Zeppelins realistisch nachbilden sollte.

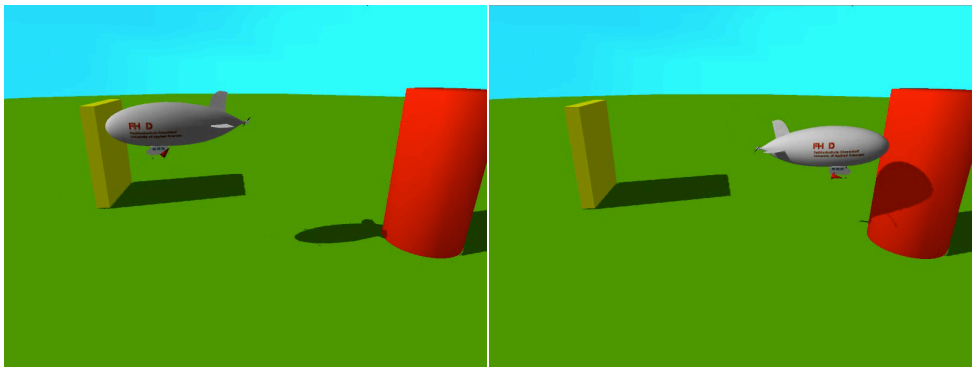


Abbildung 5.6: Bilder des zweiten Prototypen.

Um dieses Ziel zu erreichen, mussten einige Änderungen an dem bisherigen Prototypen vorgenommen werden. Das visuelle Modell des Zeppelins erhielt zu allererst eine Kollisionsgeometrie, so dass es möglich war, Kollisionen mit der Umgebung zu erkennen. Da das 3D Modell des Zeppelins, das zur Darstellung verwendet wurde, eine zu hohe Anzahl an Polygonflächen enthielt, musste eine vereinfachte Version des Zeppelins für die Kollisionserkennung erzeugt werden. In Abbildung 5.7_{171} ist der Unterschied zwischen den beiden Modellen

sichtbar. Zu Erkennen ist, dass die Geometrie zur Kollisionserkennung gegenüber dem visuellen 3D Modell des Zeppelins stark vereinfacht wurde. Die Qualität des vereinfachten Kollisionsmodells reicht jedoch aus, um eine genaue Simulation zu erhalten.

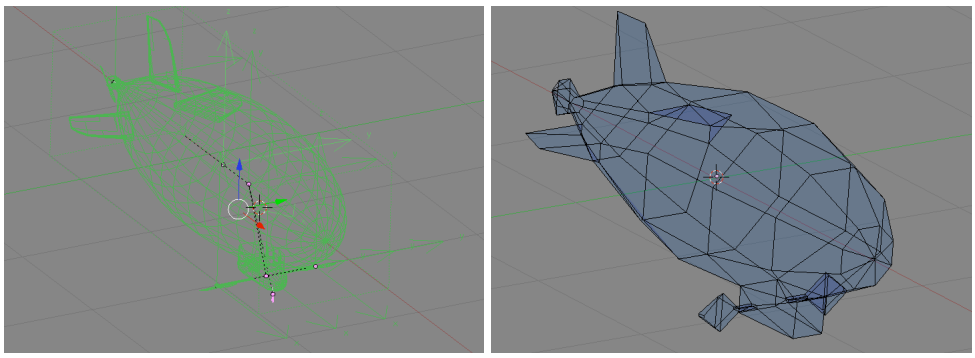


Abbildung 5.7: 3D Modell vs. Kollisionsmodell des Zeppelins.

In MiReAS wurde nun das Modell des Zeppelins um physikalische Eigenschaften erweitert, indem dort der vorhandene *RenderableActor* durch einen *PhysicalActor* ersetzt wurde. Dieser beinhaltet zum einen die 3D Geometrie, die dargestellt werden soll, zum anderen die Kollisionsgeometrie, die unsichtbar nur für die physikalischen Berechnungen genutzt wird. An diesen *PhysicalActor* können nun Aktuatoren angehängt werden, die physikalische Kräfte auf den Zeppelin ausüben. Einerseits existieren die physikalische Effekte wie die Auftriebskraft oder den aerodynamischen Widerstand und andererseits gibt es die einzelnen Propeller des Zeppelins, die dem physikalischen Modell zugefügt werden müssen. Die Achse der Schubrotoren wird über einen modellierten DC-Motor gesteuert. Alle Motoren und Propeller beziehen sich auf den entsprechenden Knoten im 3D-Modell, so dass diese auch die exakte Position besitzen. Damit die Drehrichtung der Schubmotoren in der virtuellen Umgebung besser sichtbar ist, sind auf diese rote Kegel aufgesetzt. Abbildung 5.6_{170} zeigt drei Screenshots aus dem fertigen Prototypen.

Wenn wir nun die Modell-Komponente des Prototypen betrachten, kommen wir auf folgendes Ergebnis für die Berechnung der Modell-Metrik Γ_M:

Modell-Metrik Γ_M		
σ_i	Typ	ϵ_i
Physikalisches Modell	Virtuell	0
Heckrotor	Simuliert	0,5
Seitenrotoren	Simuliert	0,5

Fortsetzung auf der nächster Seite

Fortsetzung von der vorherigen Seite		
σ_i	Typ	ϵ_i
Winkel Seitenrotoren	Simuliert	0,5

$$\mathbf{\Gamma_M = \frac{1,5}{4} = \frac{3}{8}}$$

Das physikalische Modell, was wir in diesem Prototypen eingebaut haben, ist rein virtueller Natur, da es in der finalen Version durch den realen Zeppelin gegeben ist und wir diese Eigenschaft nicht beeinflussen können. Durch das physikalische Modell des Zeppelins sind wir jedoch in der Lage, die Bewegung des Zeppelins durch die korrekten Kräfte an den Rotoren zu simulieren. Damit erhalten wir einen Wert von $\frac{3}{8}$ für die Modell-Metrik Γ_M.

Der View wurde dahingehend aufgewertet, dass Schatten dem Szenario zugefügt wurden. Durch die Visualisierung der Drehrichtung der Rotoren wurde weiterhin eine Darstellung von Modellparametern eingefügt, die die Darstellung erweitert. Da sich allerdings nichts an der visuellen Repräsentation des Umgebung geändert hat, ist der Wert für die View-Metrik nicht viel höher als der des ersten Prototypen:

View-Metrik Θ_V		
ω_i	Typ	ϕ_i
3D Modell des Zeppelins	Virtuell	0,5
Virtuelle Umgebung	Temporär	0
Visualisierung Modellparameter	Virtuell	0,5

$$\mathbf{\Theta_V = \frac{1}{4}}$$

Zu sehen ist, dass sich der Wert für die View-Metrik verbessert hat, da nun Parameter des Modells visualisiert werden und dem Benutzer die Möglichkeit zur Kontrolle bieten. Da die Umgebung noch immer nicht die Realität widerspiegelt, fließt sie nicht in die Bewertung mit ein.

Wie bei dem ersten Prototypen ist der Wert der Environemnt-Metrik Ω_E 0, da keine Daten der Umwelt der Applikation zur Verfügung gestellt werden.

Die Steuerung wurde überarbeitet und es ist nun möglich, den Zeppelin über einen Joystick oder einen Gamecontroller zu steuern. Mit Hilfe einer Fernsteuerung, die über USB an den Computer angeschlossen werden kann (siehe Abbildung 5.8$_{173}$) kann der virtuelle Zeppelin genau so gesteuert werden wie der reale Zeppelin. Ein Benutzer, der geübt in der Steuerung des realen Zeppelins ist, kam auf Anhieb mit der Steuerung des virtuellen Zeppelins zurecht und konnte ihn schnell kontrolliert steuern. Um eine Einschätzung dieser Steuerung

Abbildung 5.8: USB Fernsteuerung.

zu bekommen, wurde ein kleiner Test mit den Entwicklern und dem geübten Benutzer durchgeführt, bei dem der Zeppelin zwischen den beiden Hindernissen in einer Acht gesteuert werden sollte, ohne dabei die Hindernisse zu berühren. Es wurden die Zeit und die Kollisionen protokolliert und mit Hilfe der Contoller-Metrik Ψ_C der Wert 0.2[5] berechnet.

Die Akteure in diesem Prototypen sind somit folgendermaßen definiert:

Akteur	Modell	View	Controller	Environm.
Zeppelin	×	×		
Heckrotor	×	×		
Seitenrotoren	×	×		
Kollisionsmodell	×			
Umgebung	×	×		
Fernsteuerung			×	

Der Zeppelin wurde weiter unterteilt und es wurden der Heckrotor und die beiden Seitenrotoren sowohl im Modell als auch im View zugefügt. Des Weiteren wurde die Kollisionsgeometrie dem Modell zugefügt, damit Kollisionen erkannt werden können. Die Umgebung findet sich auch im Modell wieder, da der Zeppelin mit den beiden Hindernissen kollidieren kann. Die Tastatur wurde durch die USB Fernsteuerung ersetzt und ist nun für die Steuerung zuständig.

In der Abbildung 5.9_{174} sind nun die einzelnen Metriken in den Kiviatgraphen eingetragen worden. Im Gegensatz zum ersten Prototypen ist nun sowohl der Wert für die Modell-Metrik als auch der Wert

[5] Dieser Wert ist durch die kleine Gruppe an Teilnehmern jedoch nicht repräsentativ.

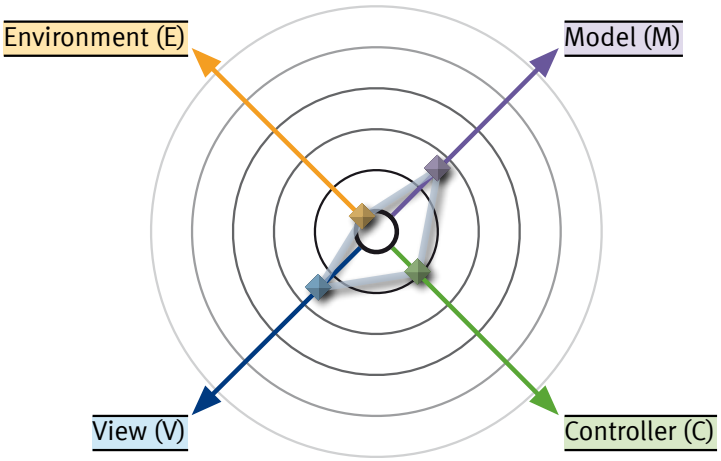


Abbildung 5.9: Kiviatgraph des zweiten Prototypen.

für die Controller-Metrik auf einen Wert > 0 gestiegen. Der Wert für den View konnte sich über die Visualisierung der Modellparameter verdoppeln. Die Schatten werten zwar den visuellen Eindruck auf, spielen allerdings in der Bewertung des Views keine Rolle.

5.2.4 Der dritte Prototyp: Verfeinerung der Steuerung

Mit dem verbesserten Modell, das im letzten Prototypen eingebaut wurde, verhält sich der Zeppelin im allgemeinen Fall realitätsnah. Der Prototyp kann nun als Grundlage für die Entwicklung einer verbesserten Steuerung verwendet werden. Im dritten Prototyp soll nun die Steuerung verbessert werden, um die Erfolgsrate zu erhöhen. Dabei soll das Eingabegerät dasselbe bleiben wie im zweiten Prototyp, allerdings soll der Benutzer nicht mehr direkten Einfluss auf die einzelnen Motoren haben, sondern den Zeppelin intuitiver steuern können.

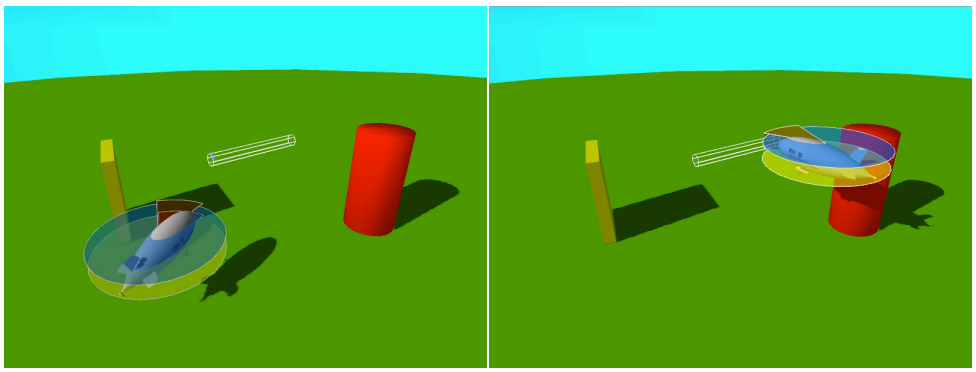


Abbildung 5.10: Bilder aus dem dritten Prototypen.

Bei der zu entwickelnden Steuerung soll der Benutzer die Höhe

und den Kurs des Zeppelins angeben, woraufhin der Zeppelin dann versucht, diese Vorgaben umzusetzen. Hierfür wurden zunächst virtuelle Sensor-Plug-ins entwickelt, die einen Höhensensor sowie einen Kompass(-sensor) abbilden. Zur Regelung der Höhe wurden über einen PID-Regler[6] die Informationen des virtuellen Höhensensors in Steuersignale der Antriebsmotoren so umgesetzt, dass sie den Zeppelin auf der eingestellten Höhe halten.Da die Antriebsrotoren nur zwei Freiheitsgrade besitzen (Vortrieb und Auftrieb), ist die Realisierung eines solchen Reglers aufwändig.

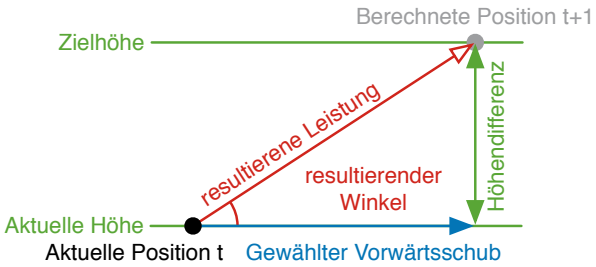


Abbildung 5.11: Berechnung des Winkels und der Leistung.

In Abbildung 5.11 ist die Methode der Berechnung zur Höhenregulierung abgebildet. Dabei wird zunächst die Höhendifferenz zum Zeitpunkt t zwischen der momentanen Höhe des Zeppelins und der gewählten Höhe berechnet. Aus dem gewählten Vorwärtsschub, den der Benutzer eingestellt hat, kann nun der resultierende Winkel für die Antriebsmotoren berechnet werden. Damit der Zeppelin sowohl die Höhe als auch den gewählten Vorwärtsschub erlangt, wird die resultierende Leistung der Motoren berechnet und eingestellt. Damit sollte der Zeppelin rechnerisch zum Zeitpunkt $t + 1$ an die berechnete Position gelangen. Da die Trägheit hier nicht mit berücksichtigt wird, wurde ein PID Regler verwendet, der die Berechnung kontinuierlich neu berechnet und dabei versucht, starke Schwingungen zu mindern. Abbildung 5.12_{176} verdeutlicht noch einmal genau die Zusammenhänge zwischen der Höhenreglung und dem Vorschub, die im Rahmen einer Bachelorarbeit [Lau08] bereits vor der Realisierung des Prototypen konzipiert wurden.

Auch der Kurs wurde über einen entsprechenden Regler, der die Informationen des Kompasssensors verarbeitet, realisiert. Die Steuerwerte für den jeweiligen Kurs werden für den Heckrotor berechnet. Da dieser jedoch nur einen Freiheitsgrad besitzt, ist die Implementierung des Reglers einfacher als bei der Höhensteuerung. Analog zur Kursabweichung wird die Stärke des Heckrotors geregelt.

[6] Der Regelkreises eines PID-Reglers (proportional–integral–derivative controller) besteht aus einem P-Anteil, einem I-Anteil und einem D-Anteil.

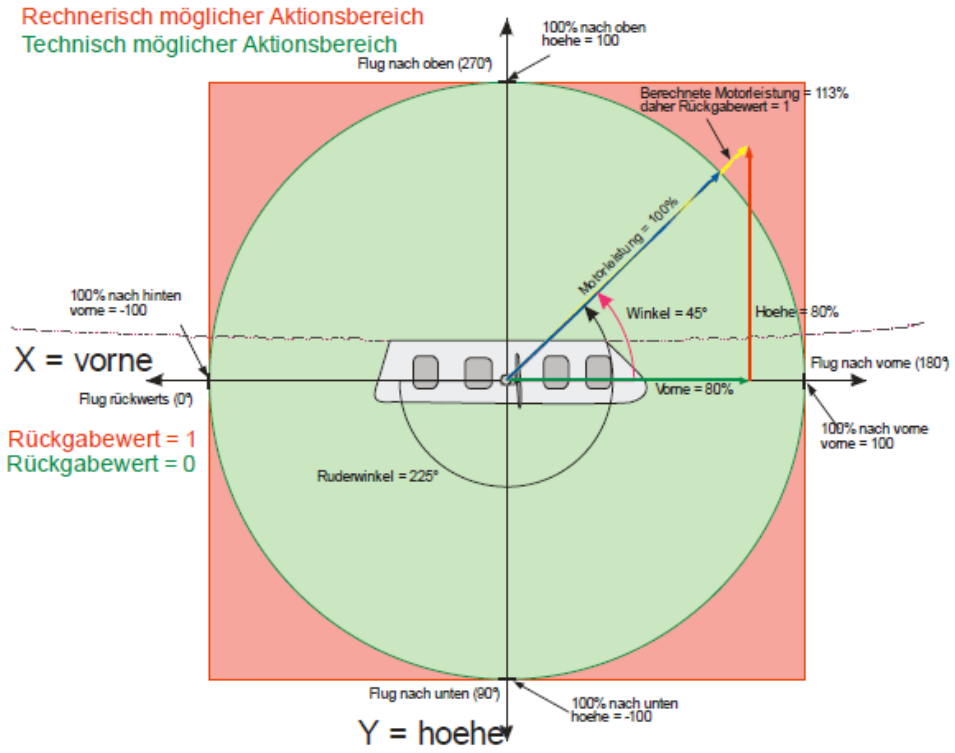


Abbildung 5.12: Berechnungsgrundlage der Höhenregelung [Lau08]

Damit der Benutzer ein Feedback der eingegebenen Höhe und des Kurses hat, wurden diese beiden Parameter visualisiert. In Abbildung 5.10_{174} sind diese Visualisierungen zu sehen. Dabei gibt der blaue Kreis die vom Benutzer eingestellt Höhe an, der gelbe Kreis die aktuelle Höhe des Zeppelins und das rote Kreissegment die Abweichung des eingestellten Kurses zur aktuellen Flugrichtung.

Zum Testen der Kurskontrolle wurde in das Szenario eine virtuelle Windquelle positioniert, In Abbildung 5.10_{174} ist diese Windquelle mit einem Drahtgitter-Zylinder in der Mitte der Szene visualisiert. Befindet sich der Zeppelin innhalb des Wirkbereiches der virtuellen Windquelle, wird das physikalische Modell des Zeppelins von diesem beeinflusst.

Die Zielwerte des Kurses und der Höhensteuerung werden mit Hilfe der Fernbedienung, die schon beim zweiten Prototypen zum Einsatz kam, eingestellt. Solange der Benutzer Steuersignale sendet, wird der jeweilige Zielwert eingestellt. Setzt der Benutzer die Steuerung aus, wird der aktuelle Wert von Höhe bzw. Kurs als Zielwert für die Regelung festgelegt. Dies vereinfacht die Steuerung, da die Zielwerte direkt auf die Eingaben des Benutzers reagieren.

Bei dem dritten Prototypen hat sich weder am View noch am Modell

etwas verändert. Dem View wurde zwar die Visualisierung der Höhe und des Kurses hinzugefügt, allerdings ändert das nicht den Wert der von uns definierten View-Metrik, da schon im zweiten Prototypen Parameter visualisiert wurden. Als Akteure müssen diese beiden Visualisierungen indes mit angeführt werden. Das Modell ist dasselbe wie im zweiten Prototypen, nur dass sich die Methode, wie der Benutzer damit interagiert, verändert hat. Geändert hat sich allerdings der Controller, der nun die Low-Level Motorsteuerung abstrahiert und eine Höhen- und Kurssteuerung anbietet. Des Weiteren wurde eine virtuelle Windquelle implementiert, die Umwelteinflüsse widerspiegelt. Damit ist die erste Komponente im Bereich Environment integriert.

Environment-Metrik Ω_E		
η_i	Typ	δ_i
Umwelteinflüsse	Simuliert	0,5
		$\Theta_V = \frac{1}{6}$

Wie im zweiten Prototypen wurde nur ein Benutzertest im kleinen Rahmen unter den Entwicklern durchgeführt. Der Aufbau war hier auch derselbe wie zuvor. Es sollte wieder zwischen den Hindernissen eine Acht geflogen werden, ohne dass der Zeppelin mit den Hindernissen kollidiert. Erschwerend hinzu kam die Windquelle, die sich in der Mitte zwischen den beiden Hindernissen befand. Hier musste jedoch nicht der Benutzer den Kurs neu setzten, sondern der Controller musste die entstandene Kursänderung korrigieren. Es waren nur die Reaktionen der Benutzer interessant, da die sich auf die automatische Kurskorrektur einrichten mussten. Nach der Bewertung berechnete sich ein Wert für die Controller-Metrik von 0.45[7].

Für den dritten Prototypen ergibt sich folgende Aufteilung:

Akteur	Modell	View	Controller	Environm.
Zeppelin	×	×		
Heckrotor	×	×		
Seitenrotoren	×	×		
Kollisionsmodell	×			
Umgebung	×	×		
Fernsteuerung			×	
Kurs		×		
Höhe		×		
Wind		×		×

Neu als Akteur hinzugekommen sind der Kurs und die Höhe, die

[7]Der Wert wurde zur Übersichtlichkeit im Kiviatgraphen abgerundet.

visualisiert werden. Der virtuelle Wind, der die Umgebung widerspiegelt, wird dem Environment zugeordnet, hat allerdings auch eine visuelle Repräsentation. Als Controller dient wie im zweiten Prototypen die USB Fernsteuerung, die jetzt jedoch anders verwendet wird, da sie nicht mehr direkt die Rotoren steuern.

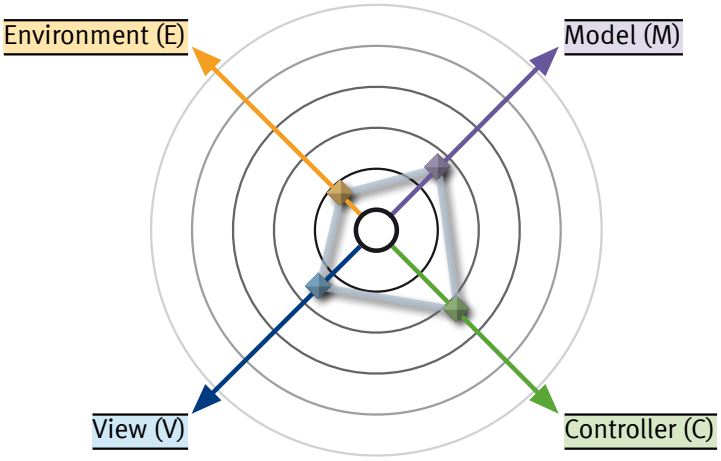


Abbildung 5.13: Kiviatgraph des dritten Prototypen.

Damit ergibt sich für den Kiviatgraphen das Bild, das in Abbildung 5.13 zusehen ist. Der Wert für den Controller hat sich merklich verbessert und durch die Verwendung des virtuellen Windes wurde auch die Environment-Metrik etwas erhöht. Durch die Verwendung der neuen Steuerung ist die Kontrolle des Zeppelins einfacher geworden und die Benutzer lernen schneller den Zeppelin zu beherrschen. Allerdings benötigt diese Steuerung auch eine gewisse Lernphase, weil die Elemente des Controllers erlernt und eingeprägt werden müssen.

Im nächsten Schritt soll nun eine realistischere Umgebung entstehen, um den Zeppelin unter fast realen Bedingungen steuern zu können.

5.2.5 Der vierte Prototyp: Verbesserte real existierende Umgebung

Nach erfolgreichem Test der ersten verbesserten Steuerung wurde für den vierten Prototypen eine verbesserte und real existierende Umgebung angestrebt. Hierzu wurde ein 3D-Modell des Lichthofs der Fachhochschule Düsseldorf modelliert und eingebaut, wie in Abbildung 5.14$_{179}$ zu sehen.

Das visuelle 3D Modell wurde exakt nach den architektonischen Vorgaben modelliert und eine daraus entwickelte, einfachere Version diente zur Kollisionserkennung (Abbildung 5.15$_{179}$). Da der Zeppelin

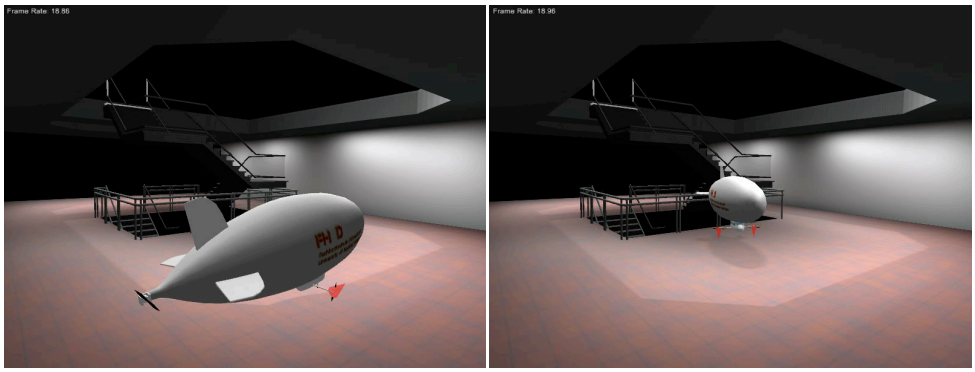


Abbildung 5.14: Bilder aus dem vierten Prototypen.

in dieser Umgebung schon mehrfach geflogen wurde, konnte mit Hilfe des realistischen Umgebungseindrucks die Steuerung und das Verhalten des Zeppelins besser beurteilt werden. Dies sollte noch einmal das physikalische Modell des Zeppelins auf Korrektheit überprüfen.

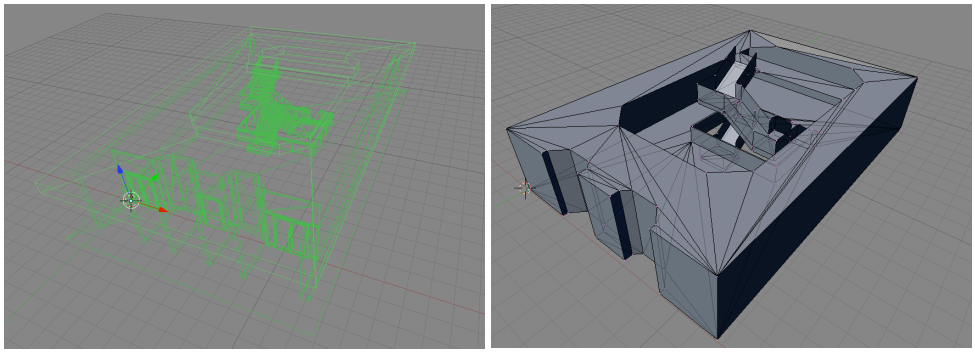


Abbildung 5.15: 3D Modell vs. Kollisionsmodell der Umgebung.

Dieser Prototyp ist ein Zwischenschritt zur Einführung der nächsten Verbesserung der Steuerung. Es war wichtig, dass hier die Umgebung exakt modelliert wurde, damit im nächsten Schritt die neue Steuerung mit dem virtuellen und realen Zeppelin verglichen werden konnte. Deshalb wurden hier auch die Kurs- und Höhenkontrolle wieder durch die einfachere Steuerung aus dem zweiten Prototypen ersetzt.

Es ergeben sich für den vierten Prototypen folgende Akteure:

Akteur	Modell	View	Controller	Environm.
Zeppelin	×	×		
Heckrotor	×	×		
Seitenrotoren	×	×		
Kollisionsmodell	×			
Umgebung		×		×

Fortsetzung auf der nächster Seite

Fortsetzung von der vorherigen Seite				
Akteur	Modell	View	Controller	Environm.
Fernsteuerung			×	

Da die neue Steuerung vom dritten Prototypen wieder entfernt wurde, sind auch die entsprechenden Akteure nicht mehr im Szenario vorhanden. Die Umgebung ist nun auch im Environment vorhanden, da es sich um ein Abbild der realen Umgebung handelt und der Zeppelin mit der Umgebung kollidieren kann. Dabei wird die virtuelle Umgebung als simulierte reale Umgebung gesehen, das bedeutet, die Applikation kann nicht auf die Daten zugreifen, außer es würde in passender Sensor existieren. Die Kollision ist dabei von der visuellen Darstellung gekapselt. Somit erhöht sich der Wert für die Environment-Metrik folgendermaßen:

Environment-Metrik Ω_E		
η_i	Typ	δ_i
Hindernisse	Simuliert	0,5
Umgebung	Simuliert	0,5
		$\Theta_V = \frac{1}{3}$

Die Darstellung wurde durch die real existierende virtuelle Umgebung aufgewertet, so dass sich der Wert der View-Metrik ein wenig erhöht hat. Die Modellparameter aus dem dritten Prototypen (Kurs und Höhe) wurden zwar nicht mehr visualisiert, allerdings wurde weiterhin die Drehrichtung der Rotoren angezeigt, so dass sich der Wert für die Visualisierung von Modellparametern nicht ändert.

View-Metrik Θ_V		
ω_i	Typ	ϕ_i
3D Modell des Zeppelins	Virtuell	0,5
Virtuelle Umgebung	Virtuell	0.5
Visualisierung Modellparameter	Virtuell	0,5
		$\Theta_V = \frac{3}{8}$

Der Wert für die Controller-Metrik ist derselbe wie im zweiten Prototyp. Es wurden keine Benutzertests für die Metrik durchgeführt, allerdings wurde dieser Prototypen von einem erfahrenen Anwender ausführlich getestet, um den virtuellen Prototyp und den realen Zeppelin vergleichen zu können. Das Ergebnis war, dass sich der virtuelle Prototypen für die meisten Fälle hinreichend genau so verhielt, wie der reale Zeppelin. Auf diesem Ergebnis aufbauend konnte die Arbeit an neuen Steuerstrategien aufgenommen werden.

Im Kiviatgraphen in Abbildung 5.16$_{181}$ ist die Veränderung des vier-

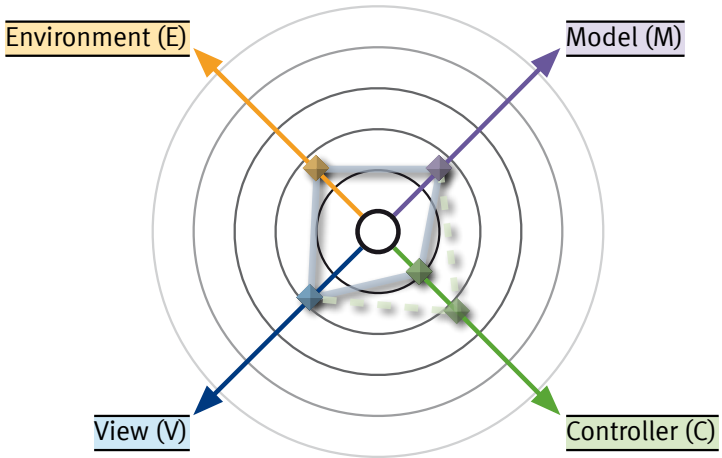


Abbildung 5.16: Kiviatgraph des vierten Prototypen.

ten Prototypen gut zu erkennen. Der Wert der Controller-Metrik ist wieder auf dem Niveau vom zweiten Prototypen, das Modell ist unverändert und die Werte vom View und vom Environment sind gegenüber dem Vorgänger etwas gestiegen. Noch bewegt sich die ganze Entwicklung eher im unteren Drittel des Graphen, Der Grund ist, dass zum vorliegenden Zeitpunkt der ganze Prototyp in VR existiert und noch keine realen Akteuere zur Anwendung gekommen sind.

Am Ende der Entwicklung des vierten Prototypen wurde die verbesserte Steuerung des dritten Prototypen wieder eingebaut, jedoch ohne die Visualisierung des Kurses und der Höhe. Damit ließ sich nun auch diese Steuerung in einer real existierenden Umgebung testen. Die Ergebnisse sind mit denen aus dem dritten Prototypen identisch, so dass der Wert der Controller-Metrik gleich blieb. Die grün gestrichelte Linie in Abbildung 5.16 zeigt diese Ausprägung des vierten Prototypen.

5.2.6 Der fünfte Prototyp: Virtueller Prototyp mit einfacher Gestensteuerung

Nachdem der Zeppelin mit der einfachen und der High-Level Steuerung in einer vertrauten real existierenden Umgebung getestet werden konnte, sollte nun im fünften Prototypen eine weitere Verbesserung der Steuerung implementiert werden.

Die USB Fernsteuerung soll nun durch eine einfache Gestenerkennung ersetzt werden, die Manöver des Zeppelins intuitiver ausführen soll. Der Zeppelin soll über die Kommandos *Hoch*, *Runter*, *Links*, *Rechts*, *Vorwärts* und *Rückwärts* gesteuert werden. Diese Kommandos sollen in

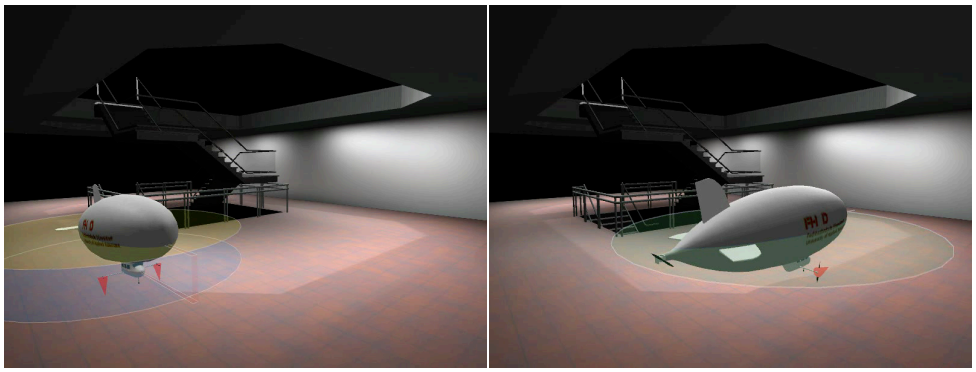


Abbildung 5.17: Bilder aus dem fünften Prototypen.

möglichst intuitive Gesten gewandelt werden. Die Gesten sollen mit Hilfe der Wiimote, einem Eingabegerät der Spielekonsole Nintendo Wii, das über einen 3-Achsen Beschleunigungssensor die Lage des Controllers im Raum erkennen kann, in die entsprechenden Kommandos umgesetzt werden. Die Wiimote kann über Bluetooth mit Hilfe der Bibliothek OIS [Cas10] in MiReAS genutzt werden. Für ein genaues Tracking ist der 3-Achsen Sensor, der in der Wiimote verbaut wurde, zu unpräzise, allerdings reicht er für eine Gestenerkennung aus. Auf Basis der Accelerometer Gesture Recogniser Bibliothek (AGR), die mit Hilfe der Hidden Markov Modellen (HMMs) [SPHB08] eine Gestenerkennung realisiert[8] wurde ein Plug-in für MiReAS entwickelt. AGR erlaubt Gesten für 3D Beschleunigungsdaten zu trainieren und trainierte Gesten zu erkennen.

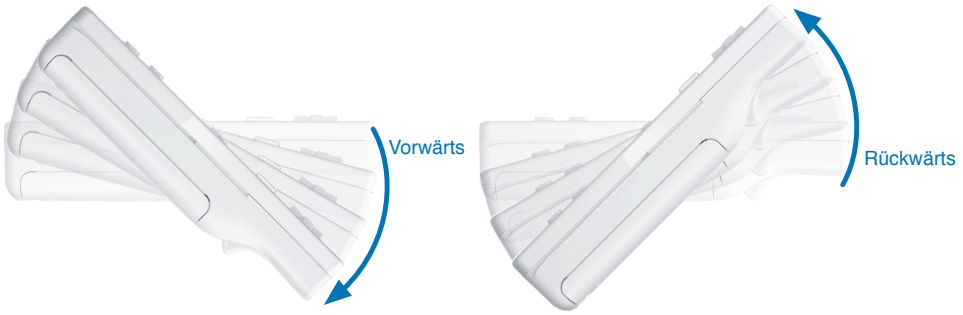


Abbildung 5.18: Gesten zur Schubkraftregulierung des Zeppelins.

Die Kommandos für die Steuerung des Zeppelins wurden über entsprechende Gesten realisiert. In Abbildung 5.19$_{183}$ und Abbildung 5.18 sind die Gesten dargestellt, die den Zeppelin steuern. Sie wurden mit dem von AGR mitgelieferten Programm „GestureCreator“ trainiert und die entsprechenden HMM-Modelle zur Erkennung der

[8]Weitere Details der Realisierung der Gestenerkennung sind in der Masterarbeit von Pogscheba [Pog09], zu finden.

Gesten in MiReAS importiert. Beim Training der Gesten wurde die Bewegung mehrfach aufgenommen, um so Fehler bei der Eingabe zu kompensieren. Bei der Erkennung der Gesten haben sich gekrümmte Bewegungen robuster gegenüber Fehlinterpretationen erwiesen, bei geradlinigen Bewegungen traten zu viele Fehlerkennungen auf.

Die Schubkraft des Zeppelins nach vorne bzw. hinten wird über den Neigungswinkel der Wiimote geregelt, wie in Abbildung 5.18[182] dargestellt. Dieser kann über den 3-Achsen Beschleunigungssensor direkt berechnet werden. Um eine gewisse Toleranz für die Ruhelage der Wiimote zu gewährleisten, wurde der Neigungswinkel erst ab einem bestimmten Schwellwert berücksichtigt.

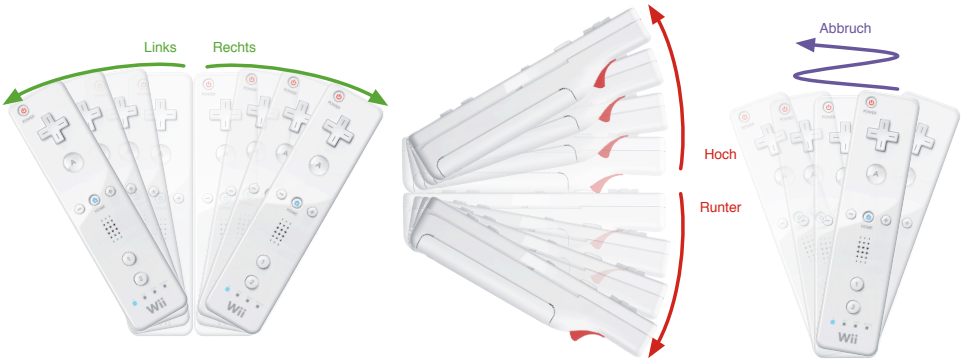


Abbildung 5.19: Gesten zur Navigation des Zeppelins.

Soll der Zeppelin nach links oder rechts fliegen, muss der Benutzer die Wiimote mit einem schnellen Impuls in die entsprechende Richtung bewegen. Soll der Zeppelin hoch oder runter fliegen, muss entsprechend die Wiimote mit einem schnellen Impuls in die gewünschte Richtung bewegt werden. Damit diese Gesten nicht fälschlicherweise, beisielsweise bei der Einstellung der Schubkraft, erkannt werden können, muss der Benutzer die *B*-Taste der Wiimote während der Eingabe der Geste gedrückt halten (angedeutet durch die rot eingefärbten Taste in Abbildung 5.19). Dadurch wird gewährleistet, dass bei ungewollten Bewegungen der Wiimote keine Gesten erkannt werden. Ist eine Geste erkannt, wird die Aktion, die mit der Geste verbunden ist, so lange ausgeführt, bis der Benutzer die *Abbruch*-Geste ausführt. Dadurch werden alle Aktionen, die der Zeppelin gerade ausführt, ohne Verzögerung abgebrochen. Die Geste für den Abbruch aller Aktionen sollte für den Benutzer einfach zu merken und auch einfach in der Ausführung sein, so dass sie schnell in Notsituationen ausgeführt werden kann. Die Entscheidung fiel auf das Schütteln der Wiimote, da diese Geste auch schon erfolgreich z. B. beim iPhone von Apple als entsprechende Geste bekannt war.

Diese Art von Steuerung vereinfacht die Benutzung des Zeppelins,

da sich der Benutzer nur fünf Gesten für die jeweiligen Kommandos merken muss. Die Schwierigkeit in der Steuerung liegt jedoch nun in der strikt sequenziellen Ausführung der Kommandos. Es ist mit den vorgestellten Gesten nicht möglich, eine Kurs- bzw. Höhenänderung während der Vorwärts oder Rückwärtsbewegung des Zeppelins zu initiieren, die Kommandos werden hier strikt getrennt. Das bedeutet bei einer Kurs- bzw. Höhenänderung, dass der Benutzer den Schub auf Null zurücksetzten und die Geste zur Kurs- bzw. Höhenänderung ausführen muss. Sobald der gewünschte Kurz bzw. die gewünschte Höhe erreicht ist, muss der Benutzer die Aktion durch die Abbruchgeste beenden und kann dann wieder den Schub des Zeppelins setzen. Soll sowohl Kurs als auch Höhe verändert werden, muss dies auch sequenziell geschehen, z. B. kann erst die Höhe und danach der Kurs geändert werden. Es wäre auch eine parallele Ausführung der einzelnen Aktionen möglich gewesen, allerdings hätte das zu einer sehr viel komplizierteren Steuerung geführt.

Bei dem fünften Prototypen ergab sich somit folgende Aufteilung der Akteure:

Akteur	Modell	View	Controller	Environm.
Zeppelin	×	×		
Heckrotor	×	×		
Seitenrotoren	×	×		
Kollisionsmodell	×			
Umgebung		×		×
Wiimote			×	
Gestenerkennung			×	
Kurs		×		
Höhe		×		

Die Visualisierung der Höhe und des gewählten Kurses wurde wieder eingebaut, so dass der Benutzer ein Feedback bekam. Anstatt der USB Fernsteuerung wurde nun die Wiimote eingebunden und zusätzlich die Gestenerkennung als Plug-in eingebaut. Die anderen Akteure waren dieselben wie im vierten Prototypen.

Auch bei diesem Prototypen wurde ein kleiner Benutzertest mit den Entwicklern und einem erfahrenen Benutzer, der den Zeppelin mit der normalen Fernbedienung sehr gut steuern konnte, durchgeführt. Die gestellte Aufgabe war, den Zeppelin einmal schnellstmöglich um den Lichthof zu steuern, ohne dabei mit der Umgebung zu kollidieren. Es zeigte sich, dass der erfahrene Benutzer mit der Fernbedienung aus dem vierten Prototypen schneller die Aufgabe löste, jedoch war die Fehlerrate und auch die Zeit, die die unerfahrenen Entwickler benötigten, mit der Gestensteuerung sehr viel besser. Es wurde ein

Wert von 0.65 (gerundet) für die Controller-Metrik berechnet.

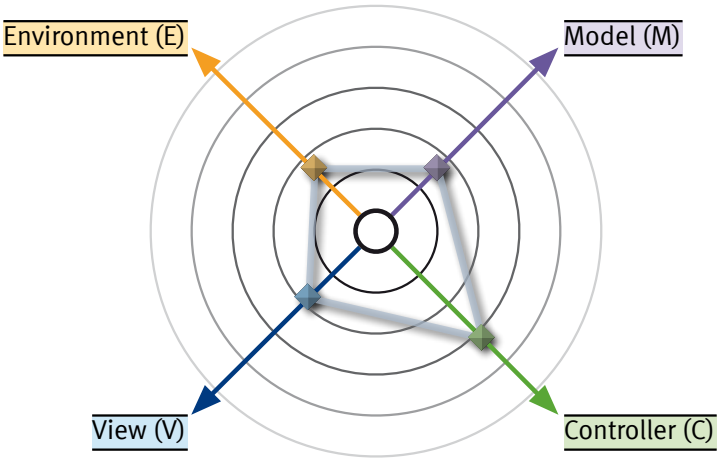


Abbildung 5.20: Kiviatgraph des fünften Prototypen.

Im Kiviatgraphen ist zu sehen, dass sich die Gestensteuerung über die Wiimote positiv auf die Controller-Metrik ausgewirkt hat. Die anderen Werte sind in Bezug auf den vierten Prototypen gleich geblieben, weil sich nur die Steuerung geändert hat.

5.2.7 Der sechste Prototyp: Virtueller Prototyp mit verbesserter Physiksimulation

Nach erfolgreichen Tests der ersten Konzepte für neue Interaktionstechniken sollte nun in diesem sechsten Prototypen eine verbesserte physikalische Simulation eingebaut werden.

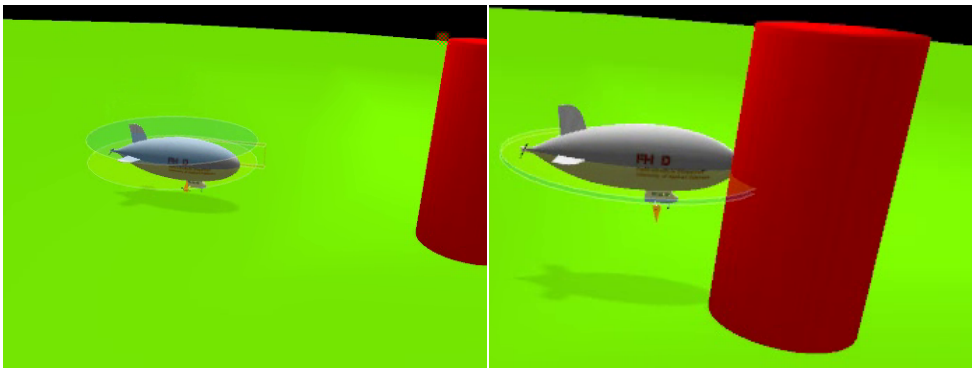


Abbildung 5.21: Bilder des sechsten Prototypen (einfache Umgebung).

Anstatt der aktuell implementierten Game-Physik Bibliothek, die die Simulation des Zeppelins übernimmt, soll ein komplexes und präzises physikalisches Modell des Zeppelin mit Hilfe des Softwarepaketes Software MATLAB/Simulink [Mat11a, Mat11b] erstellt und simuliert

werden. Für MATLAB /Simulink existiert eine Vielzahl an Toolboxen, die Funktionsblöcke zur Verfügung stellen. Diese sind teilweise von Universitäten bzw. ambitionierten MATLAB/Simulink Benutzern erstellt worden und stehen zur freien Verfügung. Daher ist MATLAB/Simulink für präzise Simulation sehr gut geeignet und wird auch in der Industrie und Forschung häufig für die Lösung solcher Probleme eingesetzt.

Das Modell des Zeppelins sollte unabhängig von MiReAS entwickelt und simuliert werden. Um eine Kopplung zwischen der Software MiReAS und MATLAB/Simulink zu realisieren, wurde zunächst *COMMUVIT* benutzt, das auch schon bei der Kopplung von Virtools und MATLAB/Simulink Modellen Verwendung fand (siehe Kapitel 4.5.1$_{136}$). Da es sich bei *COMMUVIT* um eine externe, einzeln ausführbare Anwendung handelte, haben wir uns indes für eine integrierte Lösung entscheiden, die in MiRaAS integriert wurde. So wurde ein spezielles Netzwerk-Plug-in sowohl in MiReAS als auch in MATLAB/Simulink eingebaut, das es erlaubt, zwischen den beiden Programmen zu kommunizieren. Wichtig dabei ist, dass die Übertragung der Daten synchron geschieht, so dass keine Probleme bei der Kopplung entstehen können. Dies ist insbesondere wichtig bei Übertragungen von externen Kräften und Impulsen zum physikalischen Modell.

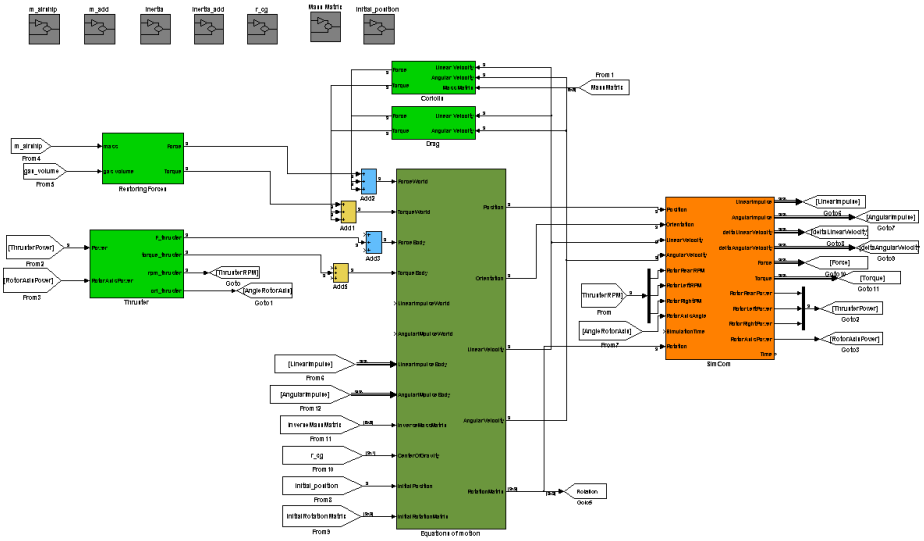


Abbildung 5.22: Das MATLAB/Simulink Modell des Zeppelins.

In Simulink werden die Bewegungsgleichungen berechnet, die Pogscheba in seiner Masterarbeit [Pog09] erarbeitet und implementiert hat. Hierzu wird ein spezieller *Equations of Motion*-Block in MATLAB/Simulink erstellt, wie in Abbildung 5.22 zu sehen ist. Dieser Block führt eine doppelte Integration von Kraft und Drehmoment aus,

um die korrekte Position und Orientierung zu errechnen. In dem Modell existieren des Weiteren Blöcke zur Simulation der Corioliskraft, des aerodynamisches Widerstandes, der Auftriebskraft und der Antriebsmotoren. Die so berechneten Kräfte werden summiert und dem Block, der für die Bewegungsgleichungen zuständig ist, bereitgestellt. Dieser errechnet mit den Werten die Position, Orientierung und Geschwindigkeit des Zeppelins und sendet diese Informationen über den Block „SimCom“, der eine Netzwerkschnittstelle kapselt, an MiReAS. Die von MiReAS empfangenen Daten können nun benutzt werden, um das visuelle Modell des Zeppelins korrekt auszurichten und zu animieren. Die Informationen über die Motorleistungen, die über den Controller eingestellt werden, und weitere externe Kräfte, liefert MiReAS im Gegenzug an das MATLAB/Simulink Modell zurück. Auf dieser Basis berechnen sich dann die neuen Werte für den nächsten Simulationsschritt.

Das in MATLAB/Simulink entworfene Modell des Zeppelins ist alleine schon durch die Verwendung der exakten Berechnungsmethoden genauer als die allgemeine Festkörpersimulation der Game-Physik Engine. Um eine noch genauere Simulation zu erhalten, könnten z. B. Verfahren, die in [Koro6] vorgestellt wurden, implementiert werden. Das implementierte Modell des Zeppelins reicht allerdings komplett für unsere Zwecke aus, so dass auf eine weitere Verfeinerung verzichtet wurde.

Eine Kollisionserkennung wurde der Einfachheit halber direkt in MiReAS mit Hilfe von PAL (Physics Abstraction Layer, siehe Kapitel 4.5.2$_{148}$) implementiert. Kontaktpunkte zwischen dem Zeppelin und anderen Objekten werden mit Hilfe von PAL generiert, nach MATLAB/Simulink geschickt und dort in Impulse und Kräfte umgerechnet, die den Zeppelin vom Kollisionsobjekt abstoßen, um so eine Durchdringung der beiden Objekte zu verhindern. Leider funktionierte die Kollisionsverarbeitung mit Hilfe von PAL nicht so gut wie die Kollisionserkennung der Game-Physik Engine. Hier wäre Bedarf der Verbesserung, auf die jedoch aus zeitlichen Gründen verzichtet wurde.

Betrachten wir nun die Akteure, die sich in diesem Prototypen wiederfinden, so erhalten wir folgende Aufteilung:

Akteur	Modell	View	Controller	Environm.
Zeppelin		×		
Heckrotor	×	×		
Seitenrotoren	×	×		

Fortsetzung auf der nächster Seite

Fortsetzung von der vorherigen Seite

Akteur	Modell	View	Controller	Environm.
Kollisionsmodell	×			
Simulink Modell	×			
Kommunikation	×			
Umgebung		×		×
Wiimote			×	
Gestenerkennung			×	
Kurs		×		
Höhe		×		

Durch die Auslagerung des Zeppelin-Modells nach MATLAB/Simulink ist in MiReAS nur noch das visuelle Zeppelinmodell vorhanden. Hinzu kam das Simulink-Modell, das extern die physikalischen Berechnungen ausführt und sie MiReAS zur Verfügung stellt. Dazu wird ein Kommunikations-Akteur benötigt, der Daten von MATLAB/Simulink empfängt bzw. diese dorthin sendet.

Der Wert für die Modell-Metrik hat sich nun geändert, da nun auch noch die Berechnung der Flughöhe hinzukam:

Modell-Metrik Γ_M		
σ_i	Typ	ϵ_i
Physikalisches Modell	Virtuell	0
Heckrotor	Simuliert	0,5
Seitenrotoren	Simuliert	0,5
Winkel Seitenrotoren	Simuliert	0,5
Flughöhe	Simuliert	0,5
		$\mathbf{\Gamma_M = \frac{2}{4} = \frac{1}{2}}$

Zu Beginn der Entwicklung dieses Prototypen wurde auf eine einfache Umgebung und eine einfache Steuerung zurückgegriffen, um sich auf die Entwicklung des Modells in MATLAB/Simulink bzw. die Kommunikation zwischen MiReAS und MATLAB/Simulink zu konzentrieren. Bilder aus dieser Phase sind in Abbildung 5.21$_{185}$ zu sehen. Später wurde allerdings wieder in dieselbe Umgebung und auf dieselbe Steuerung wie in Prototyp 5 umgestellt, so dass sich für diese beiden Varianten des sechsten Prototypen folgender Kiviatgraph ergab:

Die Punkte, die mit der transparent-gestrichelten Linie verbunden sind, repräsentieren den Prototypen mit der einfachen Steuerung und Umgebung, die anderen den endgültigen sechsten Prototypen. Das Modell ist etwas verbessert, da nun auch die Flughöhe mit simuliert wird.

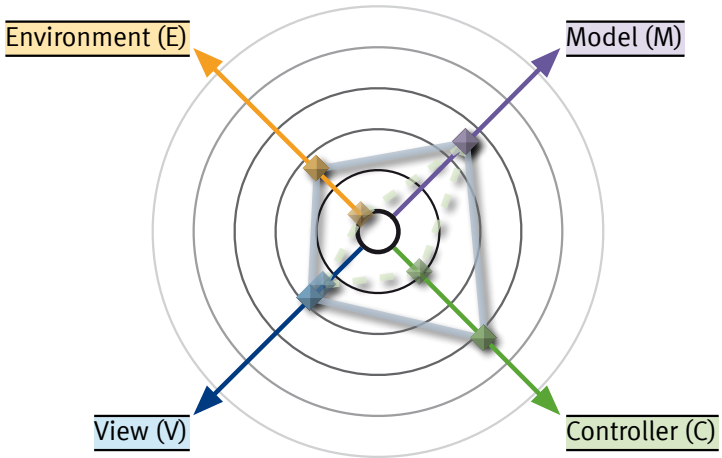


Abbildung 5.23: Kiviatgraph des sechsten Prototypen.

5.2.8 Der siebte Prototyp: Virtueller Prototyp in realer Umgebung

Die Entwicklung der virtuellen Prototypen ist nun bis auf wenige Kleinigkeiten abgeschlossen. Als nächster Schritt folgt nun der Übergang von der virtuellen Umgebung in den realen Raum.

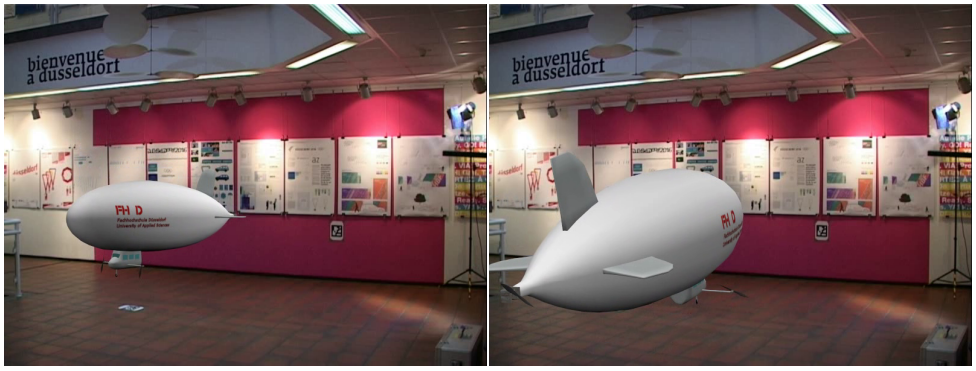


Abbildung 5.24: Bilder des siebten Prototypen.

Hierzu wird die virtuelle Umgebung aus aus dem Prototypen entfernt. Das Kollisionsmodell wird jedoch beibehalten, um die Interaktion zwischen realem Raum und virtuellem Zeppelin zu ermöglichen. Die entfernte virtuelle Umgebung wird durch ein Live-Video desselben Raumes ausgetauscht. Um die reale Umgebung sowohl mit dem virtuellen Zeppelin als auch mit dem Kollisionsmodell zu registrieren, werden in der realen Umgebung Marker an definierten Punkten platziert (Abbildung 5.24). Diese Marker können von MiReAS erkannt werden und stellen eine Beziehung zwischen der virtuellen und realen Welt her. Erst über diese Beziehung ist es möglich, den virtuellen Zeppelin im realen Raum fliegen zu lassen und auch auf die reale

Umgebung zu reagieren.[9]

Bei diesem Prototypen würde sich zur Visualisierung der Szene ein Head-Mounted Display (HMD) anbieten, wir haben uns allerdings für eine normale Videokamera entschieden, da wir so auch in der Lage waren, nachträglich mit Hilfe des aufgenommenen Videomaterials Offline-Versuche durchzuführen. Bei den Aufnahmen stellten sich schnell einige Probleme beim Tracking der Marker heraus. Sind die mit 20cm Größe relativ großen Marker zu weit entfernt, konnte ein durchgängiges Verfolgen der Marker nicht immer garantiert werden. Auch spielte die Ausleuchtung und der Winkel zwischen Marker und Kamera eine große Rolle bei der Erkennung. Diese Probleme sind jedoch bei visuellem Markertracking bekannt und können durch geschickte Platzierung und gute Ausleuchtung behoben werden.

Ein weiteres Problem sind Verdeckungen von virtuellen Objekten mit realen Objekten. Da nur das Videobild vorhanden ist, fehlen die Tiefeninformationen, um das virtuelle Modell verdecken zu können. Hier kann jedoch das Kollisionsmodell verwendet werden, das bei der Berechnung der sichtbaren Pixel erkennen lässt, ob das Videobild oder das virtuelle Objekt zu sehen ist. So kann entweder das Pixel vom Videobild oder das Pixel vom Zeppelin je nach Entscheidung gezeichnet werden. Diese Methode kann direkt auf modernen Grafikkarten entschieden werden und benötigt kaum Rechenzeit. Diese Technik wurde u. A. in der Arbeit „Entwicklung virtueller Kreaturen in 3D- und AR-Umgebungen“ [GSS04c] vorgestellt.

Bei den Akteuren hat sich im siebten Prototypen Folgendes geändert:

Akteur	Modell	View	Controller	Environm.
Zeppelin		×		
Heckrotor	×	×		
Seitenrotoren	×	×		
Kollisionsmodell	×			
Simulink Modell	×			
Kommunikation	×			
Umgebung				×
Fernsteuerung			×	

Da in diesem Prototypen der Schwerpunkt auf die Integration des virtuellen Zeppelins in die reale Umgebung gelegt wurde, haben wir die Gestensteuerung wieder durch die einfache USB Fernbedienung ersetzt. So konnte uns der geübte Benutzer ein Feedback über die Qualität der Visualisierung und des Modells geben, da er diesel-

[9] Vorraussetzung zur realistischen Interaktion zwischen virtuellem Zeppelin und realer Umgebung ist die exakte Modellierung des Kollisionsmodells.

ben Manöver einmal mit dem virtuellen Prototypen und einmal mit dem realen Zeppelin nachfliegen und beurteilen konnte. Hier zeigte sich ein weiteres Mal, dass das physikalische Modell, das in MATLAB/Simulink berechnet wurde, sehr exakt war. Leider war, wie auch schon zuvor erwähnt, die Kollisionserkennung zwischen Zeppelin und Kollisionsmodell teilweise unpräzise und träge. Da keine visuelle Repräsentation der Umgehung mehr in dem Prototypen vorhanden war, entfiel auch der entsprechende Akteur. Somit war die Umgebung nur noch durch einen Environment Akteur repräsentiert, hinter dem ein Tracking System stand. Das Tracking System erkannte die visuellen Marker in der Videoaufnahme und berechnete daraus die die Position und Orientierung des Kollisionsmodells. Über die Lage des Kollisionsmodells konnten nun die Kollisionskräfte zwischen dem Zeppelin und der Umgebung in MATLAB/Simulink berechnet werden.

So ergab sich für die Environment-Metrik folgernder Wert:

Environment-Metrik Ω_E		
η_i	Typ	δ_i
Hindernisse	Simuliert	0,5
Umgebung	Real	1
		$\boldsymbol{\Theta_V = \frac{1}{2}}$

Die Hindernisse sind zwar der realen Umgebung angepasst, allerdings wird die Kollision noch simuliert. Die Umgebung ist komplett real und muss mit Hilfe eines Trackingverfahrens erkannt werden. Da eine bekannte Umgebung gewählt wurde, reicht ein einfaches visuelles Marker-Tracking aus, um die Position und Orientierung der Kamera im Raum zu berechnen und damit sowohl den Zeppelin als auch das Kollisionsmodell auszurichten.

An der View-Metrik hat sich auch etwas geändert, da nun nicht eher die virtuelle, sondern die reale Umgebung visualisiert wird.

View-Metrik Θ_V		
ω_i	Typ	ϕ_i
3D Modell des Zeppelins	Virtuell	0,5
Umgebung	Real	1,0
Visualisierung Modellparameter	Virtuell	0,5
		$\boldsymbol{\Theta_V = \frac{1}{2}}$

Die Visualisierung der realen Umgebung ließ den Wert der View-Metrik nun auf $\frac{1}{2}$ ansteigen. Durch diese Veränderungen ergibt sich folgender Kiviatgraph:

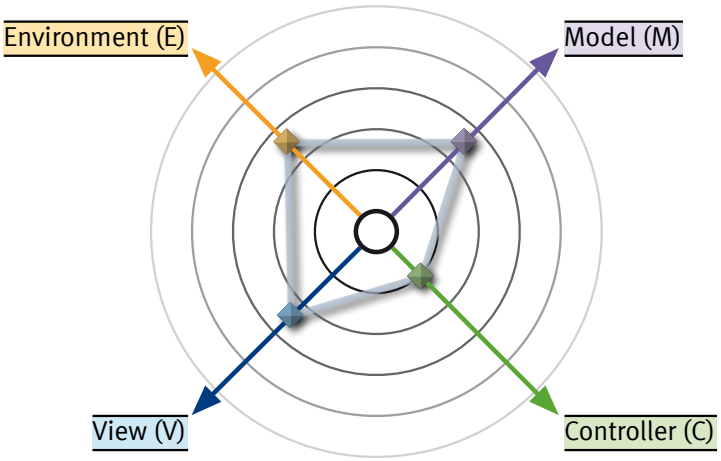


Abbildung 5.25: Kiviatgraph des siebten Prototypen.

Die Werte der Environment-Metrik und der View-Metrik sind gestiegen, da jedoch die einfachere Steuerung wieder benutzt wurde, ist der Wert der Controller-Metrik gefallen. Für die Modell-Metrik ist der Wert gleich geblieben bzgl. des sechsten Prototypen, da auch hier das MATLAB/Simulink Modell aus dem sechsten Prototypen verwendet wurde.

5.2.9 Der achte Prototyp: Realer Zeppelin mit AR-Unterstützung und verbesserter Steuerung

Die folgende Iteration konnte leider auf Grund fehlender Hardwareunterstützung nur theoretisch betrachtet werden. Es wurde allerdings das Konzept entwickelt und könnte in Zukunft umgesetzt werden, wenn die Hardware entwickelt wurde. Im achten Prototypen geht es um die Einbindung des realen Zeppelins mit der im dritten Prototypen aus Kapitel 5.2.4$_{174}$ vorgestellten Steuerung. Hierzu soll die Fernbedienung des Zeppelins (Abbildung 5.26$_{193}$, rechts) über USB mit den entsprechenden Werten für die Motorensteuerung gespeist werden. Die Möglichkeit, die Fernsteuerung über USB anzusteuern ist allerdings nicht fertig entwickelt.

In den Zeppelin wird ein Embedded System Modul eingebaut, das aus einer Controller-Einheit, einem Höhensensor und einem Kompass besteht. Links in Abbildung 5.26$_{193}$ ist die Platine des Moduls zu sehen, wie es in der Gondel des Zeppelins platziert ist. Das Modul soll später komplett für die Kurs- und Höhenregelung alleine zuständig sein, zunächst jedoch werden die Daten der beiden Sensoren über die Fernbedienung an MiReAS weitergeleitet. Mit diesen realen Werten kann die schon entwickelte Steuerung arbeiten und es ist möglich,

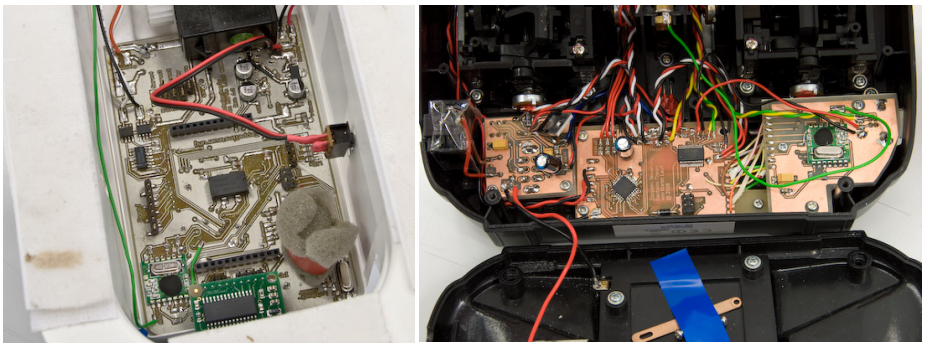


Abbildung 5.26: Zeppelinplatine und Fernbedienung mit USB Modul

den Algorithmus zur Kurs- bzw. Höhenregulierung auch mit dem realen Zeppelin zu testen. Später kann dieser Algorithmus dann auf das Embedded System Modul übertragen werden und so autonom die Regelung übernehmen. In diesem Prototypen soll der Zeppelin mit Hilfe eines Trackings verfolgt werden, um so die gelieferten Werte auch validieren zu können. Dabei können die Position und die Höhendaten durch AR-Techniken am getrackten Zeppelin visualisiert werden. Alternativ zum Embedded System Modul könnte auch das Tracking die Informationen zum Kurs und zur Höhe liefern.

Im achten Prototypen haben wir demzufolge diese Akteure:

Akteur	Modell	View	Controller	Environm.
Zeppelin	×			×
Heckrotor	×			
Seitenrotoren	×			
Kollisionsmodell				×
Umgebung				×
Umwelteinflüsse				×
Kommunikation	×			
Fernsteuerung			×	
Flughöhe	×	×		
Kurs	×	×		

In diesem Prototypen befindet sich der realen Zeppelin in der realen Umgebung, allerdings wird immer noch ein Modell des Zeppelins zur Berechnung des Kurses und der Höhenregulierung benötigt. In diesem Modell sind Heck- und Seitenmotoren vorhanden, da die Umdrehung und der Stellwinkel für diese am realen Zeppelin berechnen werden müssen. Die Umgebung und die Umwelteinflüsse sowie das Kollisionsmodell gehören hierbei zum Environment. Die Kommunikation zwischen der Fernbedienung und dem MiReAS System muss als Akteur vorhanden sein, damit die benötigten Daten empfangen werden können.

Bei der Betrachtung der Environment-Metrik kann folgender Wert ermittelt werden:

Environment-Metrik Ω_E		
η_i	Typ	δ_i
Hindernisse	Real	1,0
Virtuelle Umgebung	Real	1,0
Umwelteinflüsse	Real	1,0
		$\mathbf{\Theta_V = 1}$

Hier haben wir nun einen Wert von 1, so dass wir alle im Vorfeld definierten Punkte realisiert haben. Der Wert für die View-Metrik sieht dementsprechend so aus:

View-Metrik Θ_V		
ω_i	Typ	ϕ_i
3D Modell des Zeppelins	Real	1,0
Umgebung	Real	1,0
Visualisierung Modellparameter	Real	1,0
Visualisierung Zustandsparameter	Real	1,0
		$\mathbf{\Theta_V = 1}$

Auch bei dieser Metrik ermitteln wir einen Wert von 1, da alle Vorgaben erreicht worden sind. Dasselbe gilt auch für den Wert der Modell-Metrik:

Modell-Metrik Γ_M		
σ_i	Typ	ϵ_i
Physikalisches Modell	Virtuell	0
Heckrotor	Real	1,0
Seitenrotoren	Real	1,0
Winkel Seitenrotoren	Real	1,0
Flughöhe	Real	1,0
		$\mathbf{\Gamma_M = 1}$

Wir benötigen immer noch ein virtuelles physikalisches Modell, um den Kurs und die Höhenregulierung zu realisieren. Alle anderen Bestandteile des Modells spiegeln die realen Bauteile des Zeppelins wieder. Auch die Flughöhe ist ein realer Wert, der über den Höhensensor im Zeppelin gemessen wird.

Zusammenfassend ergibt sich nun folgender Kiviatgraph:

Wir machen hier die Annahme, dass der Wert der Controller-Metrik

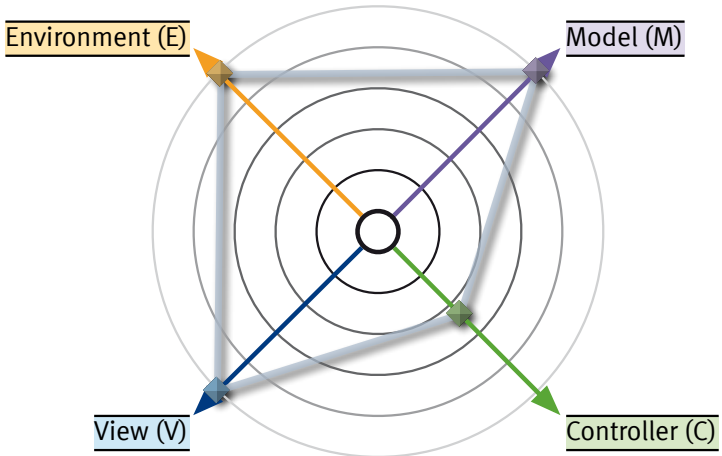


Abbildung 5.27: Kiviatgraph des achten Prototypen.

demselben Wert aus dem dritten Prototypen aus Kapitel 5.2.4$_{174}$ entspricht, da es sich um dieselbe Steuerung handelt. Da wir diesen Prototypen nur konzeptionell entwickelt haben, konnten wie keine Benutzertests durchführen. Wir hätten nun den Prototypen in den drei übrigen Metriken soweit entwickelt, wie wir es zuvor festgelegt hatten. Bei der Entwicklung kann nun der Schwerpunkt auf eine verbesserte Steuerung gelegt werden.

5.2.10 Der neunte Prototyp: Realer Zeppelin mit AR-Unterstützung und verbesserter Hardware-Steuerung

Der neunte Prototyp ist eine konsequente Weiterentwicklung des achten Prototypen und konnte daher nur konzeptionell entwickelt werden. Die Steuerung ist in diesem Prototypen nun komplett auf dem Embedded System umgesetzt worden, nachdem die Algorithmen zunächst im achten Prototypen mit MiReAS getestet worden sind.

Abbildung 5.28: Bilder des neunten Prototypen.

Wie im achten Prototypen werden Informationen direkt an dem mit Markern verfolgten Zeppelin visualisiert. Hierzu gehören die Anzeige des zurückgelegten Pfades sowie die aktuelle Richtung, in der sich der Zeppelin bewegt. Des Weiteren werden die Positionsdaten über dem Zeppelin angezeigt. Durch ein entsprechendes Plug-in werden diese Zusatzinformationen im Kreis um den Zeppelin angeordnet. Leider konnte die Höhenregelung auf Grund der fehlenden Anbindung der Fernbedienung an MiReAS, die die Sensor- und Reglerdaten senden sollte, nicht visualisiert werden. Aus demselben Grund konnte auch die Gestensteuerung nicht in diesen Prototypen implementiert werden.

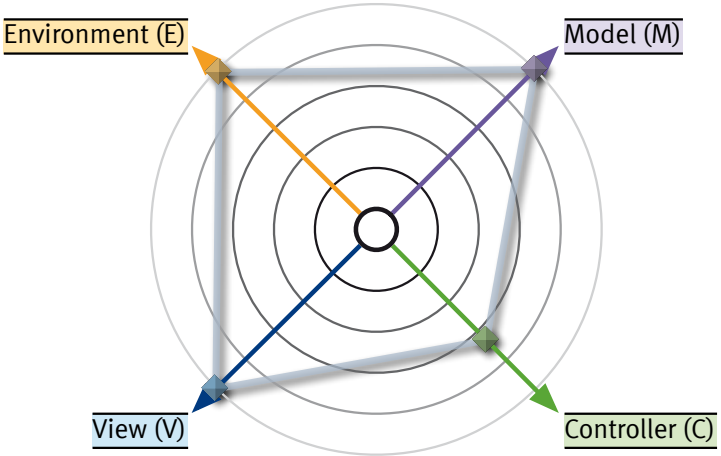


Abbildung 5.29: Kiviatgraph des neuneten Prototypen.

Gerne hätten wir die Gestensteuerung des fünften Prototypen aus Kapitel 5.2.6[181] eingebaut, jedoch war das wegen der fehlenden Anbindung der Fernbedienung an MiReAS nicht möglich. Da sich ansonsten an diesem Prototypen gegenüber dem achten Prototyp nichts weiter geändert hat als die konzeptionelle Idee der Gestensteuerung, ändert sich am Kiviatgraph auch nur der Wert der Controller-Metrik. Der hier eingetragene Wert ist einfach vom fünften Prototypen übertragen worden, würde man den Prototypen realisieren, müsste man den Wert allerdings noch validieren.

5.2.11 Der zehnte Prototyp: Realer Zeppelin in virtueller Umgebung

Auch dieser Prototyp wurde nur konzeptionell entwickelt, es wurde jedoch schon teilweise mit der Entwicklung begonnen. Leider war der Prototyp zum Zeitpunkt des Verfassens dieses Textes noch nicht fertig, so dass hier nur auf Teilergebnisse zurückgegriffen werden kann.

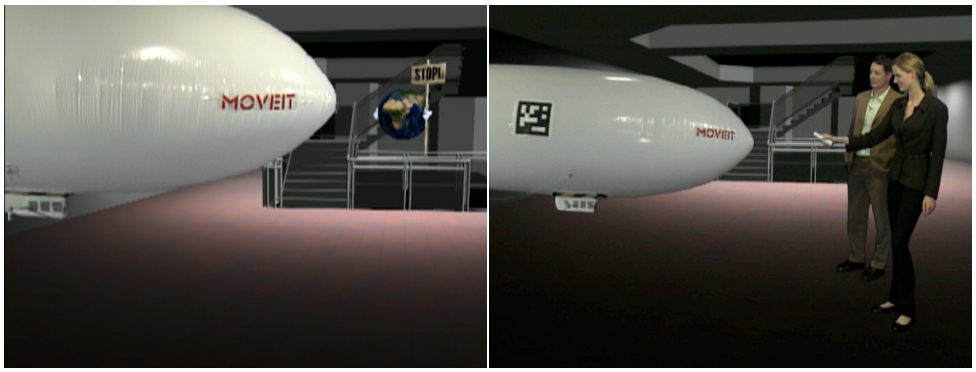


Abbildung 5.30: Konzeptbilder des zehnten Prototypen.

In diesem Prototypen wird gezeigt, dass bei der Weiterentwicklung der Steuerung wieder zurück in die Virtualität gewechselt werden kann. Allerdings soll hier nicht der virtuelle Zeppelin zum Einsatz kommen, sondern der reale Zeppelin, der sich in einer virtuellen Umgebung bewegt. So ist es möglich, neue Konzepte der Steuerung direkt am realen Zeppelin zu testen, jedoch die Gefahr zu minimieren, den Zeppelin zu beschädigen. Denkbar wäre ein intelligentes Steuersystem, das Fehler des Benutzers erkennt und zu korrigieren versucht. Beispiel hierfür wäre eine Kollisionsvermeidung auf Basis von Ultraschallsensoren, die den Abstand des Zeppelins zu möglichen Hindernissen misst und ggf. in die Steuerung eingreift, sollte sich der Zeppelin einem Hindernis nähern.

Damit im Zuge der Testreihen dieser Ausweichstrategien der Zeppelin nicht beschädigt wird, bietet sich hier eine virtuelle Umgebung an, in der sich der Zeppelin bewegt. Eine virtuelle Umgebung bietet ein hohes Maß an Kontrolle, da die Gestaltung der Umgebung frei gewählt werden kann, um darin dann die Steuerung zu testen. Weiterhin könnten virtuelle Kräfte, wie schon im dritten Prototypen aus Kapitel 5.2.4$_{174}$ vorgestellt, eingebaut werden, die die Steuerung des Zeppelins erschweren und den intelligenten Steueralgorithmus testen sollen.

Um virtuelle Kräfte und Kollisionen mit virtuellen Objekten zu realisieren, muss auf die Rotorensteuerung des Zeppelins zugegriffen werden. Kollidiert der Zeppelin mit einem virtuellen Objekt, muss die resultierende Kraft und die Richtung berechnet werden und der Zeppelin in diese Richtung gesteuert werden. Dabei wird die Kontrolle der Benutzers bzw. des intelligenten Algorithmus für kurze Zeit außer Kraft gesetzt. Leider funktioniert diese Methode nur bedingt, da bei hohen Impulsen, die normalerweise bei Kollisionen auftreten, der Zeppelin durch seine Trägheit nicht sofort seine Richtung ändert. Für den Test einer Steuerung ist dieser Nachteil allerdings zu vernachlässigen.

Um eine virtuelle Umgebung zu realisieren, würde man den Zeppelin in einer ausreichend großen Halle fliegen lassen. Über ein festinstalliertes Tracking-System, das die Position einer Kamera (oder eines HMDs) in dieser Halle ermittelt, kann nun die virtuelle Umgebung eingeblendet werden. Der Benutzer muss nun entweder über einen Monitor oder über ein HMD den Zeppelin in dieser virtuellen Umgebung fliegen. Um virtuelle Objekte auch vor dem realen Zeppelin darstellen zu können, müsste auch der Zeppelin mit Hilfe des Tracking-Systems verfolgt und die Position im Raum bestimmt werden. So kann bei der Bilddarstellung entschieden werden, welches Objekt näher zum Betrachter liegt und somit dargestellt wird.

Betrachten wir die Environment-Metrik bei diesem Prototypen, erhalten wir folgenden Wert:

Environment-Metrik Ω_E		
η_i	Typ	δ_i
Hindernisse	Virtuell	0,5
Virtuelle Umgebung	Real	0,5
Umwelteinflüsse	Virtuell	0,5
		$\mathbf{\Theta_V = \frac{1}{2}}$

Da die komplette Umgebung wieder virtuell ist, erhalten wir einen Wert von $\frac{1}{2}$. Bei der View-Metrik haben wir folgende Aufteilung:

View-Metrik Θ_V		
ω_i	Typ	ϕ_i
3D Modell des Zeppelins	Real	1,0
Umgebung	Virtuell	0,5
Visualisierung Modellparameter	Real	1,0
Visualisierung Zustandsparameter	Virtuell	0,5
		$\mathbf{\Theta_V = \frac{3}{4}}$

Die Umgebung und die Zustandsparameter sind virtuell[10], Modellparameter und der Zeppelin sind real, so dass sich ein Wert für die View-Metrik von $\frac{3}{4}$ ergibt. Am Wert der Model-Metrik verändert sich nichts, da es sich um den realen Zeppelin handelt und somit die realen Baugruppen angesprochen werden. Ein virtuelles physikalisches Modell muss indes immer noch vorhanden sein, um Kollisionskräfte berechnen zu können.

[10]Die Zustandsparameter beziehen sich auf die virtuelle Umgebung und sind aus diesem Grund als virtuell anzusehen.

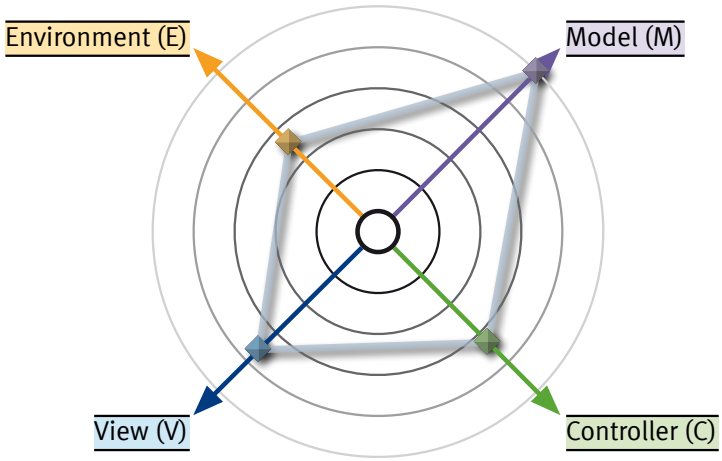


Abbildung 5.31: Kiviatgraph des zehnten Prototypen.

In Abbildung 5.31$_{199}$ ist der Kiviatgraph für den zehnten Prototypen zu sehen. Es wurde angenommen, dass die Steuerung aus dem neunten Prototypen verwendet wurde. Für die intelligente Steuerung kann leider keine Angabe über den Wert der Controller-Metrik gemacht werden, da diese Steuerung nur konzeptionell existiert. Man erkennt, dass der Wert für das Environment und für den View geringer ist als im neunten Prototypen, da die virtuelle Umgebung verwendet wird. Das Modell bleibt jedoch bezogen auf den neunten Prototypen gleich.

5.3 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie sich eine nicht triviale Applikation mit dem MRiL-Entwurfsvorgehens entwickeln lässt. Mit Hilfe der Software MiReAS, die das MRiL-Entwurfsvorgehen unterstützt, wurden insgesamt sieben Prototypen komplett und drei Prototypen konzeptionell entwickelt. Die Aufteilung der Applikation, einerseits in die MVCE-Komponenten und andererseits in Akteure, hat die Entwicklung der Prototypen beschleunigt. Da am Anfang der Entwicklung definiert wurde, wie der endgültige Prototyp aussehen sollte, konnte mit Hilfe der vorgestellten Metriken der Entwicklungsstatus über einen Kiviatgraphen grafisch angegeben werden. So war die Klassifizierung der Prototypen nach Status der Entwicklung schnell möglich.

Es zeigt sich, dass die Initialisierungsphase kritisch in der Entwicklung der Applikation ist. Hier haben Fehler im grundlegenden Design schwere Folgen für die spätere Entwicklung. Deshalb sollte für die Initialisierungsphase ausreichend Zeit veranschlagt werden, so dass Probleme schon zu Beginn erkannt und gelöst werden können. Gerade

die Einteilung in die MVCE-Kategorien muss sorgfältig geschehen, allerdings ist es auch später möglich, eine Komponente in zwei MVCE Kategorien aufzuteilen. Es kann aber sein, dass dann einige Akteure, die zu dieser Komponente gehören, getrennt werden müssen, was zusätzlich Arbeit bedeutet.

Ist die Initialisierungsphase beendet, kann nun begonnen werden, den ersten Prototypen zu entwickeln. Dies muss nicht, wie es hier der Fall war, eine sehr einfache Version der Applikation sein, es kann auch sofort ein komplexer Prototyp entwickelt werden, was jedoch zeitlich aufwändiger wird. Des Weiteren ist es sinnvoll schnell eine einfache Anwendung zu haben, in der auch später schnell einige Ideen realisiert werden können, beispielsweise in unserem Fall die verbesserte physikalische Simulation aus Kapitel 5.2.7[185], bei der wir zurück zur einfachen Umgebung gewechselt haben, um besser die Simulation debuggen zu können.

Abschließend ist zu sagen, dass die Entwicklung der Prototypen mit Hilfe von MiReAS über das MRiL-Entwurfsvorgehen schnell zu realisieren war. Ein großer Vorteil war der akteurbasierte Aufbau, so dass bei der Entwicklung schnell auf schon vorhandene Komponenten zurückgegriffen und so z. B. die Steuerung einfach gewechselt werden konnte. So war es möglich, Fehler, die sich bei der Programmierung eingeschlichen hatten, schnell zu isolieren und zu entfernen, auch wenn an mehreren Akteuren gleichzeitig entwickelt wurde. Indem zu einem stabilen Stand gewechselt wurde, konnten die einzelnen Akteure bzw. Komponenten getrennt validiert und so Fehler schneller gefunden werden.

KAPITEL 6

Synopsis

In diesem Kapitel möchte ich noch einmal kurz mein „Mixed Reality in the Loop“-Entwurfsvorgehen mit den herausragenden Arbeiten in diesem Bereich vergleichen und die Gemeinsamkeiten und Unterschiede der Verfahren aufzeigen.

6.1 Mixed Reality in the Loop im Vergleich

Das vorgestellte „Mixed Reality in the Loop“-Entwurfsvorgehen bietet eine Lösung zur Entwicklung von Mixed Reality Anwendungen unter Zuhilfenahme des Mixed Reality Kontinuums. Das ist einzigartig in der Literatur, es lehnt sich allerdings an anerkannte Verfahren einiger Wissenschaftler an. Die Arbeiten, auf die sich „Mixed Reality in the Loop“ bezieht, habe ich bereits in Kapitel 3_{59} beschrieben. Ich möchte hier, nachdem ich mein Verfahren vorgestellt habe, eine Zusammenfassung der Gemeinsamkeiten und Unterschiede an drei in Kapitel 3_{59} vorgestellten Arbeiten geben.

A model-based design process for interactive virtual environments

Cuppens verwendet in seinem Vorgehen (siehe Kapitel 3_{66}) ein modellbasierten Entwurfsprozess, um mit Hilfe eines visuell dargestellten *Task Model* und einem textbasierten *Interaction Description Model* eine lauffähige virtuelle Anwendung zu generieren. Die visuelle „Programmierung“ des *Task Model* ist ein wesentlicher Aspekt in seiner Arbeit um die Entwicklung auch für Anwender, die wenig oder keinen Hin-

tergrund in der imperativen Programmierung haben, anwendbar zu machen. Diesen Schritt bin ich in meiner ersten Werkzeugumgebung, die mit Hilfe des proprietären Autorenwerkzeuges Virtools entstanden ist, auch gegangen. Hier lag der Fokus auf der Benutzbarkeit der Werkzeugumgebung durch einen Personenkreis, der eher im Bereich Entwickler für 3D-Inhalte anstatt in der reinen Programmierung von Anwendungen angesiedelt war. Mit Hilfe der visuellen Programmierung und den von mir angebotenen Plug-ins war es dem Personenkreis möglich, Mixed Reality Anwendungen ohne das technische Wissen um die Basistechnologie zu entwerfen. Mit Hilfe der Abstraktion wurde einem größeren Kreis von Personen die Entwicklung von Mixed Reality Anwendungen ermöglicht. Dieses Prinzip findet man nicht nur bei Cuppens, auch und gerade MacIntyre zeigt mit seiner Werkzeugumgebung *DART* (siehe Kapitel 3_{90}), dass sich mit Hilfe von Personen, die im Gebiet von Kunst und Design bewandert sind, sehr interessante MR Anwendungen entwickeln lassen.

Bei Cuppens existiert zwar kein iteratives Entwurfsvorgehen, jedoch ist die automatische Generierung der abstrakten Darstellung in eine ausführbare Anwendung vergleichbar mit einem iterativen Prozess. So können auch hier die Ergebnisse schnell evaluiert werden und ggf. über das Model und einer entsprechend neuen Generierung der Anwendung in sehr kurzer Zeit ein neuer Prototyp entwickelt werden. Diese Methode entspricht in Allgemeinen dem iterativen, prototypenbasierten Verfahren, das ich vorgestellt habe. Das Model von Cuppens bietet allerdings nicht explizit die Möglichkeit der Verfeinerung der Komponenten an.

Mein Vorgehen unterscheidet sich von der Methode bei Cuppens, da es bei der Entwicklung einer Anwendung entlang des Mixed Reality Kontinuums führt, Cuppens indes rein virtuelle Anwendungen erzeugt und keine Möglichkeit bietet, reale Komponenten in seinem Prozess zu verarbeiten. Auch beschränkt er sich auf die Implementierung von Benutzerschnittstellen in virtuellen Umgebungen, sein Ansatz kann allerdings auch für andere rein virtuelle Anwendungen Verwendung finden.

Mixed Reality: A model of Mixed Interaction

In der Arbeit von Coutrix et al. (siehe Kapitel 3_{68}) wird auf die Verknüpfung der digitalen mit der physikalischen Welt fokussiert. Mit Hilfe des *Mixed Interaction Model* wird dem Entwickler hier ein Framework angeboten, das ihn bei der die Realisierung seiner Anwendung unterstützen soll. Dabei ist das Hauptkonzept das *mixed object*, eine

Komponente, die gewisse Eigenschaften des digitale Modells mit den entsprechenden Eigenschaften des physikalischen Objekts verknüpft. Diese *mixed objects* beinhalten somit die komplette Basistechnologie, um die physikalischen Objekte zu erkennen und die entsprechenden Eigenschaften zu extrahieren. Auch ermöglichen sie die Ausgabe an physikalische Objekte, die jedoch im gegebenen Beispiel auf die Ausgabe auf ein HMD beschränkt sind.

Die Verwendung von *mixed objects* ist ein sinnvoller Schritt um physikalische Objekte in einer digitalen Anwendung zu erfassen und zu kapseln. Die Basistechnologie kann hier perfekt vor dem Entwicker versteckt werden, so dass dieser sich nur die Eigenschaften der *mixed objects* definieren muss. Denselben Weg gehe ich mit den Akteuren, die auch die Funktionalität in sich kapseln. Allerdings können die Akteure nicht nur eine Verbindung zwischen digitalen und physikalischen Daten sein, sondern können auch rein virtuell bzw. rein physikalisch sein. Je nach Entwicklungsstand kann zwischen den einzelnen Versionen der Akteure gewechselt werden, vorausgesetzt es existieren die entsprechenden Implementierungen. Jedoch würden die bei Coutrix et al. vorgestellten *mixed objects* in meinem Ansatz zwei getrennte Akteure ergeben, da die Hin- und Rückrichtung getrennt betrachtet würde. Auch sieht mein Ansatz vor, dass physikalische Objekte, die nicht unter der Kontrolle der Anwendung stehen, der Komponente *Environemnt* angehören. Diese kann von der Anwendung nur abgefragt, allerdings nie verändert werden. Beispiel wäre ein getracktes physikalisches Objekt, das zwar vom Benutzer der Anwendung verändert werden kann (durch Manipulation des realen Objekts), aus der Anwendung heraus jedoch keine Möglichkeit der Manipulation besteht.

Daher ist der Ansatz von Coutrix et al. etwas unterschiedlich zu meiner Sichtweise, was sich auch in den Softwarekomponenten widerspiegelt. Des Weiteren sieht Coutrix et al. keine iterative Vorgehensweise für seine Methode vor und behandelt ausschließlich die Entwicklung von Interaktionstechniken in Mixed Reality.

A Design-Oriented Information-Flow Refinement of the ASUR Interaction Model

ASUR von Dubois (siehe Kapitel 3_{75}) bietet ein Modell und eine grafische Notation zur Entwicklung von Mixed Reality Anwendungen mit dem Schwerpunkt auf Benutzerinteraktion. Das auf grafische Darstellung basierende Modell beschreibt dabei die Interaktion zwischen dem Benutzer und dem Mixed Reality System. Es soll helfen, die digitale

und physikalische Welt miteinander zu verbinden und eine benutzerfreundliche Anwendung zu erhalten. *ASUR* unterscheidet in seinem Modell zwischen unterschiedlichen Komponenten, erwähnenswert sind hier die *Adapter* und die realen Entitäten.

Wie in meinem Ansatz verwendet *ASUR* in seinem Modell eine Art *Environment*, das hier aus den zwei Komponenten *Adapter* und *Real Entity* zusammengesetzt sind. Die *Adapter* können sowohl für die Eingabe von Daten (z. B. von einer Kamera) als auch für die Ausgabe (z. B. auf einem Monitor) verwendet werden. Ein *Adapter* stellt somit die Schnittstelle zur physikalischen Welt dar und kann Daten aus ihr extrahieren. Allerdings definiert *ASUR* die grafische Ausgabe auf einem Monitor genau mit solch einem Ausgabeadapter; dies wird im Gegensatz dazu in meinem Vorgehen über die View-Komponenten realisiert, die als nicht physikalisch angesehen werden.

In *ASUR* existieren zudem *Adapter*, die eine Schnittstelle zur physikalischen Welt darstellen, die *Real Entities*, also reale Objekte in der physikalischen Welt, deren Eigenschaften über die *Adapter* den Digitalen Komponenten zur Verfügung gestellt werden. Die Kombination aus beidem, den *Adaptern* und den *Real Entities* ermöglicht erst die Entwicklung von Mixed Reality Anwendungen. In meinem Ansatz entspricht genau diese Kombination den E-Komponenten aus dem MVCE-Architekturmuster, mit dem oben beschriebenen Unterschied, dass bei mir die grafische Ausgabe nicht als Ausgabeadapter verstanden wird. E-Komponenten in meinem Vorgehen sind reale Objekte, die in der realen, „physikalischen“ Umgebung existieren, auf die die Anwendung keinen Einfluss nehmen, sie jedoch auslesen kann. Dass die Umgebung physikalisch existiert, ist nur teilweise richtig, da die Entwicklung nach meinem Ansatz entlang des Mixed Reality Kontinuums geschieht, und so auch die Umgebung in frühen Phasen virtuell ist. Der Unterschied hier ist allerdings, dass die Umgebung nie unter der Kontrolle der Anwendung steht, so wie es auch bei den *Real Entities* in *ASUR* der Fall ist.

Den *Real Entities* stehen in *ASUR* noch die digitalen Entitäten zur Seite, die mit Hilfe des Adapters verschiedene Eigenschaften der realen Objekte auslesen können. All das ist in einer E-Komponente in meinem Vorgehen gekapselt. Sollen mehrere Eigenschaften eines realen Objektes erfasst werden, so kann der Entwickler entweder eine weitere E-Komponente integrieren oder die vorhandene E-Komponente dahingehend erweitern, dass sie auch die gewünschten Informationen liefert.

Da die Entwicklung bei *ASUR* nicht entlang des Mixed Reality Kontinuums geschieht und des weiteren kein iterativer, prototypenbasierter

Prozess ist, wird die Transition von virtuellen zu realen Objekten nicht unterstützt. Die Möglichkeit der Transition ist allerdings prinzipiell gegeben, auch wenn einige Teile des entwickelten Modells verändert werden müssen. So kann der Entwickler die virtuellen Komponenten in *Real Entities* (durch Ersetzung) umwandeln, muss jedoch ggf. Adapter und neue digitale Entitäten hinzufügen, die die Eigenschaften des realen Objektes abbilden. Da diese Transition nicht durch Weiterentwicklung sondern durch Ersetzung geschieht, ist eine Wiederverwendung der schon vorhandenen Komponenten nicht möglich. Anders als in meinem Vorgehen, bei dem die Komponenten (bei mir Akteure) durch Adapter erweitert werden und so die restlichen Komponenten der Anwendung nicht angepasst werden müssen. Da *ASUR* keine automatische Generierung von lauffähigen Programmcode hat, ist die Änderung des Modells aufwändiger als beispielsweise bei Cuppens. Mein Ansatz bietet zwar auch keine automatische Generierung, da allerdings die vorhandenen Komponenten weiterentwickelt werden und so nur kleine Änderungen anfallen, hält sich der Aufwand in annehmbaren Grenzen.

6.2 Zusammenfassung

In diesem Kapitel habe ich kurz die Gemeinsamkeiten und die Unterschiede meines „Mixed Reality in the Loop“-Entwurfsvorgehens mit anderen, relevanten Arbeiten aus der Literatur, die ich im Kapitel 3_{59} ausführlich beschrieben habe, vorgestellt. Im Allgemeinen ist festzustellen, das es sinnvoll ist, Mixed Reality Anwendungen einerseits getrennt von der Basistechnologie, andererseits entlang des Mixed Reality Kontinuums zu entwickeln. Da sich die Basistechnologien schnell ändern können, hat man mit der Abstraktion eine Möglichkeit, die unterliegende Technik zu ändern ohne die Anwendung an sich anzupassen. Dieses Vorgehen wird in der Softwareentwicklung sehr häufig eingesetzt, es ist also eine konsequente Weiterentwicklung bei Mixed Reality Anwendungen. Die Entwicklung entlang des Mixed Reality Kontinuums ist genau deshalb auch sinnvoll. So kann bei fehlender Basistechnologie bzw. vor Fertigstellung dieser die Entwicklung der Anwendung schon begonnen werden und in späteren Prototypen die Komponenten mit der Basistechnologie erweitert werden. Ein anderer wichtiger Punkt ist die Entwicklung der Anwendung aus einer vom Programmierer kontrollierbaren virtuellen Umgebung hin in zur realen Umgebung. So können sicherheitsrelevante Algorithmen zuerst in der virtuellen Umgebung auf ihre Funktionsweise hin überprüft werden, bevor sie in der realen Welt verwendet werden.

Aus dem Stand der Forschung geht hervor, dass mein Verfahren auf re-

nommierten Ansätzen der Entwicklung von im Bereich der Mixed Reality basiert. Das zeigen die hier vorgestellten Arbeiten. Weiterführend stellt mein Entwurfsvorgehen eine konsequente Weiterentwicklung dieser bekannten Verfahren darstellt, in dem es versucht, die Vorteile der einzelnen Verfahren zu kombinieren und so die Entwicklung vom MR Anwendungen weiter vereinfacht.

KAPITEL 7

Zusammenfassung und Ausblick

Dieses Kapitel schließt meine Arbeit mit einer Zusammenfassung der Ergebnisse und einem Ausblick ab. Die Zusammenfassung bietet einen Überblick über die entstandenen Ergebnisse, die ich in dieser Arbeit vorgestellt habe. Ich versuche die Ergebnisse in Beziehung zu den Zielen zu stellen, die ich in Kapitel 1.2_4 vorgestellt habe. Darüber hinaus biete ich am Ende dieses Kapitel einen Ausblick, wie sich MRiL in Zukunft weiter entwickeln ließe und welche Erweiterungen vorstellbar wären.

7.1 Zusammenfassung

In der hier vorliegenden Arbeit habe ich ein werkzeuggestütztes , prototypenbasiertes, iteratives Entwurfsvorgehen für Mixed Reality Anwendungen vorgestellt. MRiL wird entlang des Mixed Reality Kontinuums angewendet, so dass die Entwicklung einer Anwendungen aus der Virtualität in die Realität stattfindet. Mit Hilfe des iterativen prototypenbasierten Ansatzes ist eine ständig testbare Designrepräsentation der Anwendung vorhanden, die zu Evaluationszwecken verwendet werden kann. Folgende Punkte wurden im Einzelnen vorgestellt:

Entwurfsvorgehen: Das Entwurfsvorgehen stellt die zentrale Vorgehensweise bei der Entwicklung einer Mixed Reality Anwendung mit MRiL dar. Basierend auf einem iterativen Prototyping Prozess verfeinert es in jedem Schritt die jeweiligen Teile einer

Anwendung und stellt nach jeder Iteration einen testbaren Prototypen zur Verfügung.

Architekturmuster: Um den Entwurf einer Mixed Reality Anwendung zu vereinfachen, steht das MVCE Architekturmuster zur Verfügung, das Teile der Anwendung bestimmten Komponenten zuordnet. Ist eine Anwendung nach dem MVCE Architekturmuster klassifiziert, so lassen sich die jeweiligen Teile unabhängig voneinander weiterentwickeln. Dabei legt MVCE die Beziehungen fest, wie welche Komponenten miteinander interagieren können. Beispielsweise ist es einer Mixed Reality Anwendung nicht möglich, Daten der *Environment*-Komponente zu ändern, da sich die Manipulation physikalischer Eigenschaften realer Objekte nicht durch die Anwendung steuern lässt. Die Umgebung kann nur erfasst, allerdings nicht geändert werden. Mit diesem Prinzip lassen sich auch virtuelle Objekte bzw. Umgebungen realisieren, die jedoch von der Anwendung als reale, nicht kontrollierbare Komponenten angesehen werden.

Akteurmodell: Eine Verfeinerung der Aufteilung in MVCE Komponenten bietet das Akteurmodell. Hinsichtlich der vorhandenen Werkzeugunterstützung ist die Aufteilung in Akteure sinnvoll, da sie einerseits eine fein granularere Weiterentwicklung der Mixed Reality Anwendung bietet und andererseits direkt softwareseitig durch ein Entwicklungswerkzeug unterstützt wird. Dem Akteur steht im Entwicklungswerkzeug der Adapter zur Seite, der es erlaubt, den Akteur mit mehr Funktionalität auszustatten und die Schnittstellen des Akteurs zu erweitern.

Metrik: Zur genaueren Analyse des Entwicklungsstandes der Mixed Reality Anwendung wurden für jede der MVCE-Komponenten eigene Metriken entwickelt. Diese Metriken sollen den Stand der Entwicklung entlang einer bestimmten Komponente ermitteln und können mit Hilfe eines Kiviatgraphen dargestellt werden. Alle Komponenten basieren auf der Annahme der Funktionalität der fertig entwickelten Anwendung, einzig die Metrik für den Controller bildet eine Ausnahme. Sie stellt die Benutzbarkeit der vorliegenden Anwendung dar, vorausgesetztes es werden spezielle Nutzertests durchgeführt. Dies ist gerade in der Entwicklung von neuen Mixed Reality Interaktionstechniken nützlich, so die Metrik dem Entwickler ein relativ gutes Feedback gibt.

Werkzeugunterstützung: Mit dem Ziel, dass das MRiL-Entwurfsvorgehen bei Mixed Reality Anwendungen ebenfalls auf Softwareebene unterstützt wird, wurden insgesamt zwei Softwareumgebungen in dieser Arbeit vorgestellt. Die erste, die mehr an

Designer gerichtet war anstatt an Programmierer, basierte auf dem proprietären Werkzeug 3DVIA Virtools und wurde mit Hilfe von Plug-ins realisiert, die dem Anwendungsentwickler zur Verfügung standen. Diese Umgebung entstand zu Beginn dieser Arbeit und deckt nicht alle Aspekte von MRiL ab. Damit MRiL komplett von einer Werkzeugumgebung unterstützt wird, wurde MiReAS entwickelt. MiReAS wurde unter Berücksichtigung von MRiL entworfen und unterstützt den Entwurfsprozess daher nativ. Hier wird mit Hilfe von Akteuren und Adaptern die Mixed Reality Anwendung entworfen und implementiert. Da MiReAS auf der Programmiersprache C++ und der OpenSceneGraph-Grafikbibliothek basiert, ist die Entwicklung von Mixed Reality Anwendungen jedoch eher für Entwickler mit Programmierhintergrund geeignet als für Designer.

Testbare Designrepräsentation: Angesichts der Tatsache, dass das hier vorgestellte iterative MRiL-Entwurfsvorgehen nach jeder Iteration einen lauffähigen Prototypen als Ergebnis liefert, sind Benutzertests auch in frühen Entwicklungsphasen möglich. So können Fehler in der frühen Entwicklungsphasen schnell erkannt und korrigiert werden. Weiterhin erlaubt die Aufteilung in die MVCE Komponenten und die Wiederverwendung von Akteure und deren Erweiterung durch Adapter gezielt entwickelte Prototypen, die speziell eine Komponente für Benutzertests bereitstellen.

MRiL wurde an mehreren exemplarischen Beispielen angewendet, wobei sowohl nur die Vorgehensweise als auch beide Werkzeuge verwendet wurden. Das erste Beispiel, das im Kapitel 4.4_{128} beschrieben wurde, basiert auf dem reinen Vorgehen und wurde ohne Zuhilfenahme eines Werkzeuges entwickelt. Hier sollte das Entwurfsvorgehen angewendet und evaluiert werden, um zu erfahren, ob ein sinnvoller Einsatz möglich ist. Die Werkzeugumgebung, die auf Virtools basiert, wurde schon im Vorfeld bei mehreren Projekten und Veröffentlichungen verwendet, die in Kapitel $4.5.1_{136}$ erwähnt wurden. Für die schnelle und einfache Entwicklung von Demonstratoren von, in Programmiersprachen nicht versierten, Entwickler ist diese Umgebung sehr gut geeignet, da die Programmierung der Anwendungen visuell erfolgte. Leider deckte die auf 3DVIA Virtools basierende Werkzeugumgebung nicht das komplette Entwurfsvorgehen abdeckt, was es nötig machte, eine eigene Lösung zu entwickeln. Mit MiReAS, vorgestellt in Kapitel $4.5.2_{148}$, erschien eine Werkzeugumgebung, die alle Aspekte des MRiL-Entwurfsvorgehens beinhaltete. Mit MiReAS wurde ein sehr umfangreicher Demonstrator erfolgreich realisiert, der in Kapitel 5_{159} beschrieben wurde. Mit Hilfe der Entwicklung dieses

Demonstrators war dann möglich, MRiL komplett von Entwicklern zu evaluieren.

Das „Mixed Reality in the Loop“-Entwurfsvorgehen bietet ein sinnvolles Verfahren zur Entwicklung von Mixed Reality Anwendungen. Durch die Werkzeugunterstützung mit MiReAS ist es nicht nur konzeptionell möglich, eine MR Anwendung zu entwerfen, sondern auch schnell Prototypen in einer sehr frühen Phase der Entwicklung zu realisieren. Mit Hilfe des iterativen Prozesses können die einzelnen Komponenten sehr feingranular verfeinert werden und durch kurze Iterationszyklen erhält der Entwickler einen ständig testbaren Prototypen, der für die Evaluation verwendet werden kann. Die vorgestellte Metrik bietet ein Maß sowohl für den Entwicklungstand der einzelnen Komponenten als auch für die Benutzerfreundlichkeit der Anwendung, wenn Benutzertests für die Controller-Komponente durchgeführt wurden.

7.2 Ausblick

Für die Zukunft könnte man sich einige Erweiterungen für MRiL vorstellen, die die Entwicklung von Mixed Reality Anwendungen weiter vereinfachen und verkürzen könnten. Vorstellbar wäre eine grafische Notation für die modellbasierte Entwicklung mit MRiL. Um die grafische Notation in der Entwicklung auch sinnvoll einsetzten zu können, wäre eine Werkzeugunterstützung vorstellbar, in der der Entwickler seine MR Anwendung entwirft. Hieraus würde sich dann die dritte Erweiterung ergeben, die automatische Generierung von ausführbaren Prototypen aus dem grafischen Modell. Beide Erweiterungen würden die Entwicklung von Mixed Reality Anwendungen mit dem MRiL-Entwurfsvorgehen mehr in den modellbasierten Entwurf heben, so dass sich der Entwickler nicht mehr um die technische Implementierung der Anwendung Gedanken machen müsste. Es wäre ein konsequenter Schritt in Richtung des schnellen Prototypings.

7.2.1 Visuelle Notation

MRiL hat zur Zeit keine grafische Repräsentation für das Modell der Anwendung. Das liegt an der Entwicklungsgeschichte von MRiL. Da die erste softwareseitige Realisierung auf 3DVIA Virtools basierte und dieses Werkzeug eine visuelle Programmiersprache anbot, wurde auf eine grafische Notation verzichtet. Im späteren Verlauf bei der Entwicklung von MiReAS wurde kein Fokus auf eine grafische Notation gelegt, da die Entwicklung der einzelnen Akteure und Ad-

apter und deren Schnittstellen auch textuell sehr gut funktionierten. Das lag größtenteils an den Entwicklern, die ein tiefgehendes Hintergrundwissen in imperativen Programmiersprachen hatten und mit der textuellen Darstellung einer Anwendung gut arbeiten konnten. Den Entwicklern, die zuvor mit Hilfe von 3DVIA Virtools die Anwendungen entwickelt hatten, fiel die Verwendung von der MiReAS Werkzeugumgebung schwerer, da sie nicht das Hintergrundwissen hatten. Um auch diese Entwickler zu unterstützten, wäre eine visuelle Notation des Modells der Anwendung sinnvoll.

Vorstellen könnte man sich eine Notation in der Art von ASUR (siehe Kapitel 3_{75}), um das Modell der Mixed Reality Anwendung zu definieren. Hier konnten die Beziehungen zwischen den einzelnen Akteuren und damit die Schnittstellen zwischen ihnen definiert werden. Um das Prinzip der Adapter zu realisieren, könnte man sich eine hierarchische Notation vorstellen, ähnlich dem Komponentendiagramm in UML 2 [JRH+07]. Da ein Akteur auch nach der Erweiterung mit einem Adapter ein Akteur bleibt, ist es in der obersten Ebene eines Modells unwichtig, ob und mit wie vielen Adaptern ein Akteur erweitert wurde. Einzig die Art und die Anzahl der Schnittstellen können sich per Erweiterung des Akteurs ändern.

Mit Hilfe der visuellen Notation könnte die Mixed Reality Anwendung einfach beschrieben und konzeptionell aufgebaut werden. Dieser Prozess könnte bei der Vorgehensweise aus Kapitel 4.2_{101} den zweiten bis vierten Punkt visuell unterstützen und würde eine kompakte visuelle Ansicht auf die zu entwickelnde Mixed Reality Anwendung geben. Gerade bei einem Team von mehreren Entwicklern dürfte sich hier die Definition der einzelnen Schnittstellen als einfacher erweisen.

7.2.2 Entwicklungswerkzeug für die visuelle Notation

Aufgesetzt auf MiReAS wäre ein Entwicklungswerkzeug denkbar, das es erlaubt, mit Hilfe der grafischen Notation die Mixed Reality Anwendung zu entwerfen. Die Entwicklung von MR Anwendungen würde sich von der Programmierung zur Modellierung verschieben. Iteriert würde nur noch im visuellen Modell der Anwendung. Das Entwicklungswerkzeug müsste eine Möglichkeit bieten, aus dem Modell entweder Konfigurationsdateien für MiReAS, Quelltext oder, wie in nächsten Abschnitt vorgeschlagen, einen ausführbaren Prototypen zu generieren. Die Austauschbarkeit der Komponenten mit älteren Versionen im visuellen Modell, die dem der Softwarekomponenten in MiReAS entsprechen, könnte über eine Versionierung realisiert werden. Bei einer Verfeinerung eines Akteurs (beispielsweise durch das Anfügen eines Adapters), könnte der Akteur eine neue Versi-

onsnummer erhalten. Im späteren Verlauf der Entwicklung hätte der Anwender nun die Möglichkeit im grafischen Modell anzugeben, welche Version er für die einzelnen Akteure verwenden möchte. Dabei ist zu beachten, dass sich die Schnittstellen bei den aktuellen Versionen zu den älteren Versionen des Akteurs geändert haben können. Dadurch entstandene Probleme müssten entweder durch den Entwickler manuell entfernt werden, wobei das Entwicklungswerkzeug Lösungsvorschläge bieten könnte. Auch eine automatische Auflösung von inkompatiblen Schnittstellen wäre denkbar, da alle Informationen sowohl über die neuen als auch über die alten Schnittstellen vorhanden sind. Für das Entwicklungswerkzeug könnten noch weitergehende Techniken aus dem modellbasierten Entwurf integriert werden, wie beispielsweise automatische Optimierungsverfahren oder Fehlererkennungen.

7.2.3 Automatische Generierung von ausführbaren Prototypen

Eine konsequente Erweiterung zum in Abschnitt 7.2.2$_{211}$ vorgestellten Entwicklungswerkzeug für die visuelle Notation ist die automatische Generierung von ausführbaren Prototypen, wie sie beispielsweise bei der Arbeit von Cuppens et al. (Kapitel 3$_{66}$) und bei NiMMiT von De Boeck et al. (Kapitel 3$_{71}$) der Fall ist. Anstatt Konfigurationdateien für MiReAS oder Quelltext zu generieren, könnte sofort ein lauffähiger Prototyp aus dem Modell erzeugt werden, der für die Evaluation verwendet werden kann. Wäre ein Entwicklungswerkzeug für die visuelle Notation vorhanden, würde die automatische Generierung grundsätzlich unkompliziert zu realisieren sein. Im ersten Schritt würden die Akteure und Adapter in MiReAS realisiert und daraus Quelltext erzeugt. Nachfolgend könnte man die MR Anwendung mit Hilfe der normalen C/C++ Kompilier übersetzten und würde ein ausführbares Programm erhalten.

Akteure, die neue Hardware kapseln, müssten sowohl im Entwicklungswerkzeug also auch in MiReAS implementiert werden. Ähnlich wie bei AMIRE (siehe Kapitel 3$_{88}$) würde mehrere Arten von Entwicklern existieren, zum einen der Anwenderentwickler, der für die Realisierung der MR Anwendung zuständig ist, zum anderen der Systementwickler, der die erforderlichen Komponenten in das Entwicklungssystem integriert. Dabei würde der Anwendungsentwickler mit Hilfe der visuellen Notation entwickeln, der Systementwickler allerdings mit einer imperativen Programmiersprache (bei MiReAS wäre das C++).

7.3 Zusammenfassung

In diesem Kapitel habe ich die Ergebnisse meiner Arbeit zusammengefasst und noch einmal die wichtigsten Punkte des „Mixed Reality in the Loop“-Entwurfsvorgehens aufgezeigt. Im zweiten Teil habe ich einen Ausblick gegeben, welche forschungsrelevanten Weiterentwicklungen bei MRiL möglich sind. Dabei handelte es sich einerseits um Arbeiten im theoretischen Umfeld, wie die Entwicklung einer visuellen Notation für MRiL, und andererseits im praktischen Umfeld, wie die Entwicklung eines Werkzeuges für die Verwendung der visuellen Notation.

Abschließend ist zu erwähnen, dass das MRiL-Entwurfsvorgehen die Entwicklung von Prototypen und die daraus resultierende Möglichkeit, schon in frühen Phasen der Entwicklung eine testbare Designrepräsentation zu erhalten und durch Benutzertest zu überprüfen, durchaus erleichtert und zu besser benutzbaren MR Anwendungen führen kann.

Abbildungsverzeichnis

Abkürzungsverzeichnis

η_i ..... Datum in der Environment-Metrik
Γ_M ..... Modell-Metrik
ω_i ..... Datum in der View-Metrik
Ω_E ..... Environment-Metrik
Ψ_C ..... Controller-Metrik
σ_i ..... Datum in der Modell-Metrik
Θ_V ..... View-Metrik
ACO ..... Ant Colony Optimization
AGR ..... Accelerometer Gesture Recogniser
AMIRE ..... Authoring Mixed Reality
API ..... Application Programming Interface
AR ..... Augmented Reality
AV ..... Augmented Virtuality
CARE ..... Complementarity, Assignment, Redundancy and Equivalence
CAVE ..... Cave Automatic Virtual Environment
CPU ..... Central Processing Unit
DART ..... Designer's AR-Toolkit
DC ..... Direct Current
DIN ..... Deutsches Institut für Normung
DSL ..... Domain-Specific Language
EDV ..... Elektronische Datenverarbeitung
FDD ..... Feature Driven Development
FLTK ..... Fast Light Toolkit
FLUID ..... Fast Light User-Interface Designer
FPS ..... Frames Per Second
FPU ..... Floating Point Unit
GLSL ..... Graphics Library Shading Language
GMTL ..... Generic Math Template Library
GPL ..... GNU General Public License
GPU ..... Graphics Processing Unit
GUI ..... Graphical User Interface
HCI ..... Human Computer Interaction
HIL ..... Hardware in the Loop
HLSL ..... High Level Shading Language
HMD ..... Head-Mounted Display

HMM ..... Hidden Markov Model
HUI ..... Haptical User Interface
HYUI ..... Hybrid User Interface
IEC ..... International Electrotechnical Commission
IO ..... Input-Output
ISO ..... International Organization for Standardization
IST ..... Informationsgesellschaftstechnologien
JNI ..... Java Ntive Iintrface
LGPL ..... GNU Lesser General Public License
LUA ..... Virtools Scripting Languale
MDSD ..... Model Driven Softare Development
MIL ..... Model in the Loop
MiReAS ..... Mixed Reality Actor Simulator
MoVeIT ..... Mobilität, Verteilung und Interaktion
MR ..... Mixed Reality
MRiL ..... Mixed Reality in the Loop
MVC ..... Model-View-Cotroller
MVCE ..... Model-View-Cotroller-Environmnet
NiMMiT ..... Notation for Modeling Multimodal Interaction Techniques
OIS ..... Object-oriented Input System
OMG ..... Object Management Group
OOTC ..... Object-Oriented Technology Center
OpenCL ..... Open Computing Language
OpenCV ..... Open Source Computer Vision
OpenGL ..... Open Graphics Library
OS ..... Operating System
PAC ..... Pesentation-Abstraction Layer
PAL ..... Physics Abrstaction Layer
PID-Regler .. Proportional–Integral–Derivative Regler
POCO ..... Portable Components
PSO ..... Particle Swarm Optimization
RUP ..... Rational Unified Process
RV ..... Reality-Virtuality Kontinuum
SDK ..... Software Development Kit
SEARIS ..... Software Engineering and Architectures for Realtime Interactive Systems
SIL ..... Software in the Loop
TUI ..... Tangible User Interface
TUIO ..... Tangible User Interface Object
UML ..... Unified Modeling Language
URP ..... Urban Planning Workbench
VR ..... Virtual Reality
VSL ..... Virtools Scripting Languale
WIMP ..... Windows Icons Menus Pointer
XMI ..... XML Metadata Interchange
XML ..... Extensible Markup Language

XNA ..... XNA's Not Acronymed
XP ..... Extreme Programming

Literaturverzeichnis

[AH02] Ambler, Scott W. und Theresa Hudson: *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, März 2002. ISBN 0-471-20282-7.

[Azu97] Azuma, Ronald T.: *A Survey of Augmented Reality*. In: *Presence: Teleoperators and Virtual Environments*, Nummer 6 in 4, Seiten 355–385, August 1997.

[BA04] Boxall, Marcus A. S. und Saeed Araban: *Interface metrics for reusability analysis of components*. In: *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, ASWEC '04, Seiten 40–51, Washington, DC, USA, September 2004. IEEE Computer Society.

[Bal08] Balzert, Helmut: *Lehrbuch der Softwaretechnik: Softwaremanagement*. Spektrum Akademischer Verlag, zweite Auflage, Februar 2008. ISBN 3-8274-1161-0.

[BBvB+01] Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ken Schwaber und Jeff Sutherland: *Manifesto for Agile Software Development*. 2001, URL (zuletzt besucht am 10.11.2009): `http://agilemanifesto.org/`.

[BDS+99] Beedle, Mike, Martine Devos, Yonat Sharon, Ken Schwaber und Jeff Sutherland: *SCRUM: An extension pattern language for hyperproductive software development*. Pattern Languages of Program Design, Januar 1999.

[Bec00] Beck, Kent: *Extreme Programming. Das Manifest*. Addison-Wesley, zweite Auflage, September 2000. ISBN 3-8273-1709-6.

[Ben56] Benington, Herbert D: *Production of large computer programs*. Proceedings of the 9th international conference on advanced programming methods for digital computers, Januar 1956.

[BHB08] Broll, Wolfgang, Jan Herling und Lisa Blum: *Interactive Bits: Prototyping of Mixed Reality Applications and Interaction Techniques through Visual Programming*. 3DUI IEEE Symposium on 3D User Interfaces, 0:109–115, März 2008. ISBN 1-4244-2047-6.

[BJH+01] Bierbaum, Allen, Christopher Just, Patrick Hartling, Kevin Meinert, Albert Baker und Carolina Cruz-Neira: *VR Juggler: a virtual platform for virtual reality application development*. In: *Virtual Reality, 2001. Proceedings. IEEE*, Seiten 89 – 96, März 2001.

[BKK92] Budde, Reinhard, Karin Kuhlenkamp und Karlheinz Kautz: *Prototyping: an approach to evolutionary system development*. Springer Verlag, New York, März 1992. ISBN 0-387-54352-X.

[BL04] Beaudouin-Lafon, Michel: *Designing interaction, not interfaces*. In: *Proceedings of the working conference on Advanced visual interfaces*, AVI '04, Seiten 15–22, New York, NY, USA, 2004. ACM. ISBN 1-58113-867-9.

[BMRS98] Buschmann, Frank, Regine Meunier, Hans Rohnert und Peter Sommerlad: *Pattern-orientierte Software-Architektur . Ein Pattern-System*. Addison-Wesley, zweite Auflage, Januar 1998. ISBN 3-8273-1282-5.

[BMS10] Bierbaum, Allen, Kevin Meinert und Ben Scott: *Graphics Math Template Library (GMTL)*. 2010, URL (zuletzt besucht am 20.09.2010): `http://ggt.sourceforge.net/`.

[Boe79] Boehm, Barry W.: *Guidelines for Verifying and Validating Software Requirements and Design Specifications*. In: Samet, P. A. (Herausgeber): *Euro IFIP 79*, Seiten 711–719. North Holland, 1979.

[Boe81] Boehm, Barry W.: *Software engineering economics*. classes.cec.wustl.edu, 1981.

[Boe88] Boehm, Barry W.: *A spiral model of software development and enhancement*. Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research, Mai 1988.

[Boe09] Boeing, Adrian: *Physics Abstraction Layer (PAL)*. Februar 2009, URL (zuletzt besucht am 03.02.2011): `http://pal.sourceforge.net/`.

[Bol10] Bolte, Mario: *Entwurf und Vergleich wissensbasierter Verfahren zur Wegeplanung autonomer Einheiten*. Bachelorarbeit, Universität Paderborn, März 2010.

[Cas10] Castaneda, Phillip: *Object Oriented Input System (OIS)*. 2010, URL (zuletzt besucht am 01.02.2011): `http://sourceforge.net/projects/wgois/`.

[CCN97] Calvary, Gaëlle, Joëlle Coutaz und Laurence Nigay: *From single-user architectural design to PAC*: a generic software architecture model for CSCW*. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '97, Seiten 242–249, New York, NY, USA, 1997. ACM. ISBN 0-89791-802-9.

[CMPH08] Commerell, Walter, Heinz-Theo Mammen, Klaus Panreck und Joachim Haase: *Simulation technischer Systeme Anforderungen und Perspektiven*. Advances in Simulation for Production and Logistics Applications, Oktober 2008.

[CN06] Coutrix, Céline und Laurence Nigay: *Mixed reality: a model of mixed interaction*. In: *AVI '06: Proceedings of the working conference on Advanced visual interfaces*, Seiten 43–50, New York, NY, USA, 2006. ACM. ISBN 1-59593-353-0.

[Cou87] Coutaz, Joëlle: *PAC, an Object Oriented Model for Dialog Design*. In: Bullinger, Hans-Jörg und Brian Shackel (Herausgeber): *Human-Computer Interaction - INTERACT '87*, Seiten 431–436. Elsevier Science Publishers, 1987.

[Cou10] Coumans, Erwin: *Game Physics Simulation*. 2010, URL (zuletzt besucht am 21.09.2010): `http://bulletphysics.org`.

[CRC06] Cuppens, Erwin, Chris Raymaekers und Karin Coninx: *A model-based design process for interactive virtual environments*. Lecture Notes in Computer Science, Seiten 225–236, Mai 2006. ISBN 3-540-34145-1.

[CS89] Connell, John L und Linda Shafer: *Structured rapid prototyping: an evolutionary approach to software development*. Computing Series. Yourdon Press, März 1989. ISBN 0-13-853573-6.

[Das09] Dassault Systems: *3DVIA Virtools - A complete development and deployment platform with an innovative approach to interactive 3D content creation*. Mai 2009, URL (zuletzt besucht am 18.05.2010): `http://www.3ds.com/products/3dvia/3dvia-virtools/`.

[DBS06] Dorigo, Marco, Mauro Birattari und Thomas Stützle: *Ant colony optimization*. Computational Intelligence Magazine, IEEE, 1(4):28–39, 2006.

[DBVRC07] De Boeck, Joan, Davy Vanacken, Chris Raymaekers und Karin Coninx: *High-level modeling of multimodal interaction techniques using NiMMiT*. Journal of Virtual Reality and Broadcasting, 4(2), September 2007.

[DCD05] Dupuy-Chessa, Sophie und Emmanuel Dubois: *Requirements and Impacts of Model driven engineering on Mixed Systems Design*. In: *1ères Journées sur l'Ingénierie Dirigée par les Modèles (IDM'05)*, Seiten 43 – 54, 2005.

[DeM03] DeMarco, Tom: *Bärentango: Mit Risikomanagement Projekte zum Erfolg führen*. Hanser Fachbuch, März 2003. ISBN 3-446-22333-9.

[DFL06] Dachselt, Raimund, Pablo Figueroa und Irma Lindt: *Specification of Mixed Reality User Interfaces: Approaches, Languages, Standardization*. Workshop on IEEE VR 2006, März 2006.

[DG08] Dubois, Emmanuel und Philip Gray: *A Design-Oriented Information-Flow Refinement of the ASUR Interaction Model*. In: Gulliksen, Jan, Morton Harning, Philippe Palanque, Gerrit van der Veer und Janet Wesson (Herausgeber): *Engineering Interactive Systems*, Band 4940 der Reihe *Lecture Notes in Computer Science*, Seiten 465–482. Springer Berlin / Heidelberg, 2008.

[DGHP02] Dörner, Ralf, Christian Geiger, Michael Haller und Volker Paelke: *Authoring mixed reality - a component and framework based aproach*. First International Workshop on Entertainment Computing (IWEC 2002), Mai 2002.

[DGN02] Dubois, Emmanuel, Philip D. Gray und Laurence Nigay: *ASUR++: A Design Notation for Mobile Mixed Systems*. In: *Proceedings of the 4th International Symposium on Mobile Human-Computer Interaction*, Mobile HCI '02, Seiten 123–139, London, UK, 2002. Springer-Verlag. ISBN 3-540-44189-1.

[DGN10] Dubois, Emmanuel, Philip Gray und Laurence Nigay (Herausgeber): *The Engineering of Mixed Reality Systems*. Springer, London, 2010. ISBN 978-1-84882-732-5.

[DL98] De Luca, Jeff: *The Original FDD Processes*. Februar 1998, URL (zuletzt besucht am 26.05.2011): `http://www.nebulon.com/articles/fdd/originalprocesses.html`.

[DL04] De Luca, Jeff: *Version 1.3 of the Feature Driven Development processes*. November 2004, URL (zuletzt besucht am 26.05.2011): `http://www.nebulon.com/articles/fdd/download/fddprocessesA4.pdf`.

[DMJ05] Darken, R., P. McDowell und E. Johnson: *Projects in VR: the Delta3D open source game engine*. Computer Graphics and Applications, IEEE, 25(3):10 – 12, Mai 2005.

[DS90] DeGrace, Peter und Leslie Hulet Stahl: *Wicked Problems, Righteous Solutions: A Catolog of Modern Engineering Paradigms*. Prentice Hall, Mai 1990. ISBN 0-13-590126-X.

[EB08] Encarnação, J.L. und G. Brunetti: *Cybertechnologien als Werkzeug im Bauwesen*. Schriftenreihe der Stiftung Bauwesen, 13:13 – 31, 2008.

[Fia05] Fiala, Mark: *ARTag, a Fiducial Marker System Using Digital Techniques*. Computer Vision and Pattern Recognition, IEEE Computer Society Conference on, 2:590 – 596, 2005.

[FIB95] Fitzmaurice, George W., Hiroshi Ishii und William A. S. Buxton: *Bricks: laying the foundations for graspable user interfaces*. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '95,

Seiten 442–449, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co. ISBN 0-201-84705-1.

[Fis02] Fisher, Scott S.: *An Authoring Toolkit for Mixed Reality Experiences*. In: *In Proceedings of the International Workshop on Entertainment Computing (IWEC2002): Special Session on Mixed Reality Entertainment*, Seiten 487 – 494, 2002.

[Gam86] Game, The New New Product Development: *Hirotaka Takeuchi and Ikujiro Nonaka*. Harvard Business Review, Januar-Februar 1986.

[GHJV96] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 1996. ISBN 3-8273-1862-9.

[GHP+02] Grimm, Paul, Michael Haller, Volker Paelke, Silvan Reinhold, Christian Reimann und Jürgen Zauner: *AMIRE - Authoring Mixed Reality*. In: *The First IEEE International Augmented Reality Toolkit Workshop*, Darmstadt, Germany, September 2002.

[Har87] Harel, David: *Statecharts: A visual formalism for complex systems*. Sci. Comput. Program., 8:231–274, Jui 1987.

[HBR+94] Hill, Ralph D., Tom Brinck, Steven L. Rohall, John F. Patterson und Wayne T. Wilner: *The Rendezvous language and architecture for constructing multi-user applications*. ACM TOCHI, (1):81 – 125, 1994.

[ICdF99] Ierusalimschy, Roberto, Waldemar Celes und Luiz Henrique de Figueiredo: *Lua - An Extensible Extension Language*, Band 26 der Reihe *Software: Practice and Experience*, Seiten 635–652. John Wiley & Sons, Ltd., Januar 1999.

[IEE98] *ISMAR – The IEEE International Symposium on Mixed and Augmented Reality*, 1998.

[IOOTC97] IBM Object-Oriented Technology Center, CORPORATE: *Developing object-oriented software: an experience-based approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997. ISBN 0-13-737248-5.

[IRP+04] Ishii, Hiroshi, Carlo Ratti, Ben Piper, Y. Wang, Assaf Biderman und E. Ben-Joseph: *Bringing Clay and Sand into Digital Design – Continuous Tangible user Interfaces*. BT Technology Journal, 22:287–299, Oktober 2004.

[Ish08] Ishii, Hiroshi: *The tangible user interface and its evolution*. Communications of the ACM, 51(6):32–36, Juni 2008.

[ISO95] ISO/IEC: *12207, Information technology – Software life cycle processes*, 1995.

[ISO99] ISO/IEC: *13407, Benutzer-orientierte Gestaltung interaktiver Systeme*, 1999.

[ISO11] ISO/IEC: *9241-210, Ergonomie der Mensch-System-Interaktion – Teil 210: Prozess zur Gestaltung gebrauchstauglicher interaktiver Systeme*, 2011.

[IU97] Ishii, Hiroshi und Brygg Ullmer: *Tangible bits: towards seamless interfaces between people, bits and atoms*. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '97, Seiten 234–241, New York, NY, USA, 1997. ACM. ISBN 0-89791-802-9.

[JBR99] Jacobson, Ivar, Grady Booch und James Rumbaugh: *The Unified Software Development Process*. Addison-Wesley, Februar 1999. ISBN 0-201-57169-2.

[JRH+07] Jeckle, Mario, Chris Rupp, Jürgen Hahn, Barbara Zengler und Stefan Queins: *UML 2 glasklar*. Hanser Fachbuchverlag, 2007. ISBN 3-446-22575-7.

[KB99] Kato, Hirokazu und Mark Billinghurst: *Marker Tracking and HMD Calibration for a Video-based Augmented Reality Conferencing System*. In: *2nd International Workshop on Augmented Reality (IWAR 99)*, Seiten 85–94, Los Alamitos, CA, USA, Oktober 1999. IEEE Computer Society.

[KBBC05] Kaltenbrunner, Martin, Till Bovermann, Ross Bencina und Enrico Costanza: *TUIO - A Protocol for Table Based Tangible User Interfaces*. In: *6th International Workshop on Gesture in Human-Computer Interaction and Simulation (GW 2005)*, 2005.

[KE95] Kennedy, James und Russell Eberhart: *Particle swarm optimization*. Neural Networks, 1995. Proceedings., IEEE International Conference on, 4:1942–1948, 1995.

[Kor06] Kornienko, Andrei: *System Identification Approach for Determining Flight Dynamical Characteristics of an Airship from Flight Data*. Doktorarbeit, Universität Stuttgart, August 2006.

[KOSS11] Kropatschek, Martin, Alexander Obert, Roland Sattler und Stefan Schmied: *Softwareentwicklungsmodelle – Prototyping*. April 2011, URL (zuletzt besucht am 18.04.2011): `http://cartoon.iguw.tuwien.ac.at/fit/fit01/prototyping/welcome.html`.

[KR96] Kruchten, Philippe und WInston W. Royce: *A rational development process*. CrossTalk, Januar 1996.

[Kra08] Krassi, Boris: *Dynamic Virtual Prototyping for Control Engineering*. VDM Verlag Dr. Müller Aktiengesellschaft & Co. KG, Saarbrücken, November 2008. ISBN 3-639-08926-X.

[Kru99] Kruchten, Philippe: *Der Rational Unified Process. Eine Einführung*. Addison-Wesley, zweite Auflage, August 1999. ISBN 3-8273-1543-3.

[Lau08] Lauff, Martin: *Entwicklung und Test einer Zeppelinsteuerung auf Basis eines Atmel MEGA 2560 8Bit Mikrokontrollers*. Masterarbeit, Fachhochschule Düsseldorf, 2008.

[LCCV03] Luyten, Kris, Tim Clerckx, Karin Coninx und Jean Vanderdonckt: *Derivation of a Dialog Model from a Task Model by Activity Chain Extraction*. In: *Interactive Systems. Design, Specification, and Verification*, Band 2844 der Reihe *Lecture Notes in Computer Science*, Seiten 83–83. Springer Berlin / Heidelberg, 2003. ISBN 3-540-39929-1.

[LN02] Laurillau, Yann und Laurence Nigay: *Clover architecture for groupware*. In: *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, CSCW '02, Seiten 236–245, New York, NY, USA, 2002. ACM. ISBN 1-58113-560-2.

[LQH06] Lei, Kaiyou, Yuhui Qiu und Yi He: *A novel path planning for mobile robots using modified particle swarm optimizer*. Systems and Control in Aerospace and Astronautics, 2006. ISSCAA 2006. 1st International Symposium on, Seiten 981–984, 2006.

[LZB07] Lietsch, Stefan, Henning Zabel und Jan Berssenbruegge: *Computational Steering of Interactive and Distributed Virtual Reality Applications*. ASME Conference Proceedings, 2007(48035):1023–1032, 2007.

[LZE+06] Lietsch, Stefan, Henning Zabel, Martin Eikermann, Veit Wittenberg und Jan Berssenbrügge: *Light Simulation in a Distributed Driving Simulator*. In: Bebis, George, Richard Boyle, Bahram Parvin, Darko Koracin, Paolo Remagnino, Ara Nefian, Gopi Meenakshisundaram, Valerio Pascucci, Jiri Zara, Jose Molineros, Holger Theisel und Tom Malzbender (Herausgeber): *Advances in Visual Computing*, Band 4291 der Reihe *Lecture Notes in Computer Science*, Seiten 343 – 352. Springer Berlin / Heidelberg, 2006.

[Mat11a] MathWorks: *MATLAB – Die Sprache für technische Berechnungen*. 2011, URL (zuletzt besucht am 02.02.2011): `http://www.mathworks.de/products/matlab/`.

[Mat11b] MathWorks: *Simulink – Simulation und Model-Based Design*. 2011, URL (zuletzt besucht am 02.02.2011): `http://www.mathworks.de/products/simulink/`.

[MGDB04] MacIntyre, Blair, Maribeth Gandy, Steven Dow und Jay David Bolter: *DART: a toolkit for rapid design exploration of augmented reality experiences*. UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology, 6:197 – 206, Oktober 2004. ISBN 1-58113-957-8.

[MGRB06] McEwan, Gregor, Saul Greenberg, Michael Rounding und Michael Boyle: *Groupware Plug-ins: A Case Study of Extending Collaboration Functionality through Media Items*. In: *CollabTech 2006*, Seiten 42 – 47. University of Calgary, 2006.

[Mic11] Microsoft: *Microsoft Surface 2.0*. April 2011, URL (zuletzt besucht am 12.04.2011): `http://www.microsoft.com/surface`.

[MRZ11] Müller, Markus, Markus Rerych und Mario Zittera: *Softwareentwicklungsmodelle – Wasserfallmodell*. April 2011, URL (zuletzt besucht am 18.04.2011): `http://cartoon.iguw.tuwien.ac.at/fit/fit01/wasserfall/welcome.html`.

[MTUK94] Milgram, Paul, Haruo Takemura, Akira Utsumi und Fumio Kishino: *Augmented reality: A class of displays on the reality-virtuality continuum*. Telemanipulator and Telepresence Technologies, SPIE Vol. 2351:282–292, Januar 1994.

[NC97] Nigay, Laurence und Joëlle Coutaz: *Multifeature Systems: The CARE Properties and Their Impact on Software Design*. In: *Multimedia Interfaces: Research and Applications, chapter 9*. AAAI Press, 1997.

[Nie93] Nielsen, Jakob: *Iterative user-interface design*. In: *Computer*, Band 26, Seiten 32–41. IEEE Computer Society, November 1993.

[Nie94] Nielsen, Jakob: *Usability inspection methods*. In: *Conference companion on Human factors in computing systems*, CHI '94, Seiten 413–414, New York, NY, USA, 1994. ACM. ISBN 0-89791-651-4.

[NT95] Nonaka, Ikujiro und Hirotaka Takeuchi: *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*. Oxford University Press, September 1995. ISBN 0-19-509269-4.

[Obi10] Obiltschnig, Günter: *POCO C++ Libraries*. 2010, URL (zuletzt besucht am 20.09.2010): `http://pocoproject.org/`.

[OLWF07] Oda, Ohan, Levi J. Lister, Sean White und Steven Feiner: *Developing an augmented reality racing game*. In: *Proceedings of the 2nd international conference on INtelligent TEchnologies for interactive enterTAINment*, INTETAIN '08, Seiten 2:1 – 2:8, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-963-9799-13-4.

[Ope10] OpenSceneGraph Community: *OpenSceneGraph website*. 2010, URL (zuletzt besucht am 01.09.2010): `http://www.openscenegraph.org`.

[Pat99] Paterno, Fabio: *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, London, UK, 1st Auflage, 1999. ISBN 1-85233-155-0.

[PF02] Palmer, Stephen R. und John M. Felsing: *A Practical Guide to the Feature-Driven Development Prentice Hall International*. Prentice Hall International, 2002. ISBN 0-13-067615-2.

[Phi99] Phillips, G.: *Architectures for Synchronous Groupware*. Technischer Bericht 1999-425, School of Computing — Queen's University, Kingston, Ontario, Canada, Mai 1999.

[Pog09] Pogscheba, Patrick: *Mixed Reality in the Loop*. Masterarbeit, Fachhochschuhle Düsseldorf, November 2009.

[PRI02] Piper, Ben, Carlo Ratti und Hiroshi Ishii: *Illuminating clay: a 3-D tangible interface for landscape analysis*. In: *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, CHI '02, Seiten 355–362, New York, NY, USA, 2002. ACM. ISBN 1-58113-453-3.

[PW94] Pomberger, Gustav und Rainer Weinreich: *The Role of Prototyping in Software Development*. Proceedings of the 13th Intl. Conference on the Technology of Object-Oriented Languages and Systems (TOOLS Europe), März 1994.

[Rab00a] Rabin, Steve: *A* Aesthetic Optimizations*, Band 1 der Reihe *Game Programming Gems*, Kapitel 3.4, Seiten 264–271. Charles River Media, 2000.

[Rab00b] Rabin, Steve: *A* Speed Optimizations*, Band 1 der Reihe *Game Programming Gems*, Kapitel 3.5, Seiten 272–287. Charles River Media, 2000.

[Ree03] Reenskaug, Trygve: *The Model-View-Controller (MVC)*. Its Past and Present", Januar 2003.

[RG96] Roseman, Mark und Saul Greenberg: *Building real-time groupware with GroupKit, a groupware toolkit*. ACM Trans. Comput.-Hum. Interact., 3:66–106, März 1996.

[RHT95] Rix, Joachim, Stefan Haas und José Teixeira: *Virtual Prototyping - Virtual environments and the product design process*. Chapman & Hall, London, 1995. ISBN 0-412-72160-0.

[RN95] Rekimoto, Jun und Katashi Nagao: *The world through the computer: computer augmented interaction with real world environments*. In: *Proceedings of the 8th annual ACM symposium on User interface and software technology*, UIST '95, Seiten 29–36, New York, NY, USA, 1995. ACM. ISBN 0-89791-709-X.

[Roy87] Royce, W. W.: *Managing the development of large software systems: concepts and techniques*. In: *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, Seiten 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. ISBN 0-89791-216-0.

[RS01] Reitmayr, Gerhard und Dieter Schmalstieg: *OpenTracker-An Open Software Architecture for Reconfigurable Tracking based on XML*. In: *VR '01: Proceedings of the Virtual Reality 2001 Conference (VR'01)*, Seite 285, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-0948-7.

[Sch04] Scherf, Helmut E.: *Modellbildung und Simulation dynamischer Systeme. Eine Sammlung von Simulink-Beispielen*. Oldenbourg Wissenschaftlicher Verlag, zweite Auflage, September 2004. ISBN 3-486-20018-6.

[Sch08] Schlager, Martin: *Hardware-in-the-Loop Simulation: A Scalable, Component-based, Time-triggered Hardware-in-the-loop Simulation Framework*. VDM Verlag Dr. Müller Aktiengesellschaft & Co. KG, Saarbrücken, April 2008. ISBN 3-8364-6216-8.

[SEA08] SEARIS Working Group: *Software Engineering and Architectures for Realtime Interactive Systems*. Juli 2008, URL (zuletzt besucht am 28.7.2009): `http://www.searis.net/index.php5/Main_Page#Past_Workshops`.

[Sen10] SensAble Technologies Inc.: *PHANTOM Omni Haptic Device*. August 2010, URL (zuletzt besucht am 15.08.2010): `http://www.sensable.com/haptic-phantom-omni.htm`.

[SFH+02] Schmalstieg, Dieter, Anton Fuhrmann, Gerd Hesina, Zsolt Szalavari, L. Miguelm Encarnacao, Michael Gervautz und Werner Purgathofer: *The Studierstube Augmented Reality Project*. Presence, Massachusetts Institute of Technology, 11(1):33 – 54, Februar 2002.

[SGR08] Schimmel, Brian, Christian Geiger und Holger Reckter: *Projektgruppe „3D Desktop Tower Defence"*. Technischer Bericht, Hochschule Harz, Medieninformatik, 2008.

[SH02] Schmalstieg, Dieter und Gerd Hesina: *Distributed Applications for Collaborative Augmented Reality*. Virtual Reality Conference, IEEE, 0:59, 2002.

[SOBF05] Sandor, Christian, Alex Olwal, Blaine Bell und Steven Feiner: *Immersive mixed-reality configuration of hybrid user interfaces*. In: *Fourth IEEE and ACM International Symposium on Mixed and Augmented Reality, 2005. Proceedings.*, Seiten 110 – 113, Oktober 2005.

[SPHB08] Schlömer, Thomas, Benjamin Poppinga, Niels Henze und Susanne Boll: *Gesture recognition with a Wii controller*. In: *Proceedings of the 2nd international conference on Tangible and embedded interaction*, TEI '08, Seiten 11–14, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-004-3.

[Spi10] Spitzak, Bill: *FLTK: Fast Light Toolkit*. 2010, URL (zuletzt besucht am 21.09.2010): `http://www.fltk.org/`.

[Sto00] Stout, Brian: *The Basics of A* for Path Planning*, Band 1 der Reihe *Game Programming Gems*, Kapitel 3.3, Seiten 254–263. Charles River Media, 2000.

[SVEH07] Stahl, Thomas, Markus Völter, Sven Efftinge und Arno Haase: *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. d.punkt Verlag, zweite Auflage, Mai 2007. ISBN 3-89864-448-0.

[SW07] Schmalstieg, Dieter und Daniel Wagner: *Experiences with Handheld Augmented Reality*. Mixed and Augmented Reality, IEEE / ACM International Symposium on, 0:1 – 13, 2007. ISBN 978-1-4244-1749-0.

[Sys06] System, Chrysler Comprehensive Compensation: *Chrysler Goes To "Extremes"*. Chrysler Group, LLC, Juni 2006.

[Tec11] Technologies, Kongsberg Oil & Gas: *Coin3D – 3D Graphics Development Kit*. 2011, URL (zuletzt besucht am 15.04.2011): `http://www.coin3d.org/`.

[UG99] Urnes, Tore und T.C. Nicholas Graham: *Flexibly Mapping Synchronous Groupware Architectures to Distributed Implementations*. In: *In Proceedings of Design, Specification and Verification of Interactive Systems*, Seiten 133 – 148. Springer-Verlag, 1999.

[UI99] Underkoffler, John und Hiroshi Ishii: *Urp: a luminous-tangible workbench for urban planning and design*. In: *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, CHI '99, Seiten 386–393, New York, NY, USA, 1999. ACM. ISBN 0-201-48559-1.

[VCBT04] Vanderdonckt, Jean, Chow Kwok Chieu, Laurent Bouillon und Daniela Trevisan: *Model-based design, generation, and evaluation of virtual user interfaces*. In: *Web3D '04: Proceedings of the ninth international conference on 3D Web technology*, Seiten 51–60, New York, NY, USA, 2004. ACM. ISBN 1-58113-845-8.

[VN00] Vernier, Frederic und Laurence Nigay: *A Framework for the Combination and Characterization of Output Modalities*. In: *in Proceedings of DSV-IS2000, LNCS, Springer-Verlag*, Seiten 32–48. Springer- Verlag, 2000.

[Wei91] Weiser, Mark: *The computer for the 21st century*. Scientific American, 3(265):94–104, September 1991.

[Wik11] Wikimedia Foundation Inc.: *Wikipedia, die freie Enzyklopädie*. April 2011, URL (zuletzt besucht am 18.04.2011): `http://de.wikipedia.org/`.

[WIN+06] Wahli, Ueli, Majid Irani, Ana Negrello, Celio Palma, Jason Smith und Matt Magee: *Rational Business Driven Development for Compliance*. IBM Redbooks publication, November 2006. ISBN 0-7384-9657-X.

[WRL05] Wolf, Henning, Stefan Roock und Martin Lippert: *eXtreme Programming*. dpunkt, zweite Auflage, 2005. ISBN 3-89864-339-5.

[WYF03] Washizaki, Hironori, Hirokazu Yamamoto und Yoshiaki Fukazawa: *A Metrics Suite for Measuring Reusability of Software Components*. In: *Proceedings of the 9th International Symposium on Software Metrics*, Seiten 211–223, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1987-3.

Publikationen

[GFLS08] Geiger, Christian, Robin Fritze, Anke Lehmann und Jörg Stöcklein: *HYUI: A Visual Framework for Prototyping Hybrid User Interfaces*. TEI '08: Proceedings of the 2nd International Conference on Tangible and Embedded Interaction, Seiten 63 – 70, Februar 2008. ISBN 1-60558-004-3.

[GPR+02] Geiger, Christian, Volker Paelke, Christian Reimann, Waldemar Rosenbach und Jörg Stöcklein: *Testable Design Representations for Mobile Augmented Reality Authoring*. ISMAR '02: Proceedings of the 1st International Symposium on Mixed and Augmented Reality, Seite 281, September 2002. ISBN 0-7695-1781-1.

[GPS+09] Geiger, Christian, Patrick Pogscheba, Jörg Stöcklein, Hartmut Haehnel und Malte C. Berntssen: *Modellbasierter Entwurf von Mixed Reality-Interaktionstechniken für ein Indoor-Zeppelin*. Augmented & Virtual Reality in der Produktentstehung, Seiten 205 – 223, Juni 2009. ISBN 978-3-939350-71-2.

[GRSP02] Geiger, Christian, Christian Reimann, Jörg Stöcklein und Volker Paelke: *JARToolKit – A Java Binding for ARToolKit*. Augmented Reality Toolkit, The First IEEE International Workshop, September 2002. ISBN 0-7803-7680-3.

[GS07] Geiger, Christian und Jörg Stöcklein: *Mixed Reality Authoring*. Journal of Three Dimensional Images, 21(1):41 – 46, März 2007.

[GSBP09] Geiger, Christian, Jörg Stöcklein, Jan Berssenbrügge und Volker Paelke: *Mixed Reality Design of Control Strategies*. ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, August 2009.

[GSKF07] Geiger, Christian, Jörg Stöcklein, Florian Klompmaker und Robin Fritze: *Development of an Augmented Reality Game by Extending a 3D Authoring System*. ACE '07: Proceedings of the International Conference on Advances in Computer Entertainment Technology, Seiten 230 – 231, Juni 2007. ISBN 1-59593-640-0.

[GSR+06] Geiger, Christian, Jörg Stöcklein, Holger Reckter, Stephan Streuber und Robin Fritze: *Entwicklung von Augmented Reality-Präsentationen mit*

einem High-Level Authoring System - eine Fallstudie. Augmented & Virtual Reality in der Produktentstehung, 188:145 – 149, Juni 2006. ISBN 3-939350-07-9.

[GSS04a] Geiger, Christian, Tim Schmidt und Jörg Stöcklein: *Rapid Development of Expressive AR Applications*. ISMAR '04: Proceedings of the 3rd IEEE/ACM International Symposium on Mixed and Augmented Reality, November 2004.

[GSS04b] Geiger, Christian, Jörg Stöcklein und Tim Schmidt: *Entwicklung physikbasierter AR-Anwendungen mit Java*. Augmented & Virtual Reality in der Produktentstehung, 149:51 – 61, Juli 2004. ISBN 3-935433-58-1.

[GSS04c] Geiger, Christian, Jörg Stöcklein und Tim Schmidt: *Entwicklung virtueller Kreaturen in 3D- und AR-Umgebungen*. Virtuelle und Erweiterte Realität, 1. Workshop der GI-Fachgruppe VR/AR, 1:253 – 264, September 2004. ISBN 3-8322-3225-7.

[PSG+02] Paelke, Volker, Jörg Stöcklein, Lennart Groetzbach, Christian Geiger und Waldemar Rosenbach: *The AR-ENIGMA: A PDA based Interactive Illustration*. SIGGRAPH 2002: International Conference on Computer Graphics and Interactive Techniques, Seite 260, Juli 2002. ISBN 1-58113-525-4.

[PSHG10] Pogscheba, Patrick, Jörg Stöcklein, Jens Herder und Christian Geiger: *Iteratives Mixed-Reality-Prototyping und virtuelle Studiopräsentation einer Steuerung für ein Indoor-Lufschiff*. In: *7. GI-Workshop „Virtuelle und Erweiterte Realität"*, Band 7. Fachgruppe VR/AR der Gesellschaft für Informatik e.V., September 2010.

[SBK+10] Stöcklein, Jörg, Mario Bolte, Florian Klompmaker, Christian Geiger und Karsten Nebe: *Interaktive Illustration heuristischer Optimierungsverfahren zur Wegeplanung*. Augmented & Virtual Reality in der Produktentstehung, 274:191 – 207, Juni 2010. ISBN 978-3-939350-93-4.

[SGDZ08] Stöcklein, Jörg, Christian Geiger, Gundula Dörries und Henning Zabel: *Authoring of 3D and AR Applications for Educational Purposes*. International Conference REV, Seite CRROM, Juni 2008.

[SGP+09] Stöcklein, Jörg, Christian Geiger, Volker Paelke, Patrick Pogscheba und Anke Lehmann: *MVCE – A Design Pattern to Guide the Development of Next Generation User Interfaces*. IEEE Symposium on 3D User Interfaces 2009, März 2009.

[SGP10] Stöcklein, Jörg, Christian Geiger und Volker Paelke: *Mixed Reality in the Loop – Design Process for Interactive Mechatronical Systems*. Virtual Reality Conference (VR), 2010 IEEE, Seiten 303 – 304, März 2010. ISBN 978-1-4244-6236-0.

[SGPP09] Stöcklein, Jörg, Christian Geiger, Volker Paelke und Patrick Pogscheba: *A Design Method for Next Generation User Interfaces inspired by the Mixed Reality Continuum*. HCI International 2009: 12th International Conference on Human-Computer Interaction, Juli 2009.

[SPGP10] Stöcklein, Jörg, Patrick Pogscheba, Christian Geiger und Volker Paelke: *MiReAS: A Mixed Reality Software Framework for Iterative Prototyping of Control Strategies for an Indoor Airship*. AFRIGRAPH '10: Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa, Seiten 27 – 36, Juni 2010. ISBN 978-1-4503-0118-3.