Universität Trier
Fachrichtung IV – Informatik
Prof. Dr. Stephan Diehl
Lehrstuhl für Softwaretechnik

Diplomarbeit

Java-API für Mehrbenutzerinteraktion mit Wii-Remotes

Betreuer	Prof. Dr. Stephan Diehl
Betreuender Assistent:	Dipl.-Inform. Mathias Pohl
Bearbeiter	Sabine Müller geb. Löchl Auf der Redoute 20 54296 Trier Matrikelnummer 754262 12. Studiensemester Studiengang: Diplominformatik
Eingereicht am:	18.12.2008

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Diplomarbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher auch nicht veröffentlicht.

Ort, Datum

Unterschrift

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Erarbeitung meiner Diplomarbeit und während meines Studiums tatkräftig unterstützt haben.

Da es mir wichtig war, eine praxis- und hardwarenahe Diplomarbeit anfertigen zu dürfen, gilt mein besonderer Dank Herrn Prof. Dr. Stephan Diehl für die Vergabe dieses interessanten und aktuellen Diplomarbeitsthemas und seine umfangreiche Unterstützung und Betreuung während meiner Diplomarbeit. Durch die zahlreichen und ergiebigen Besprechungstermine fühlte ich mich zudem im Lehrstuhl sehr gut aufgehoben und unterstützt.

Mein weiterer Dank gilt Herrn Dipl.-Inform. Mathias Pohl, der mich durch seine engagierte Betreuung bei der Erstellung dieser Arbeit sehr unterstützte. Er war jederzeit bereit mit Fachwissen und Hilfestellungen sofort zu reagieren, was mir bei auftretenden Problemen oftmals sehr weiterhalf.

Weiter bedanke ich mich bei meiner Familie, insbesondere meinen Eltern, die mir durch ihren Rückhalt und auch finanzielle Unterstützung dieses Studium erst ermöglichten. Großen Dank an meinen Onkel, der mir, als mein Vorbild, den Weg zur Informatik wies und letztlich auch als Korrekturleser eine große Hilfe war.

Letztendlich bedanke ich mich bei meinem Mann für seine große Geduld und seine seelische und moralische Unterstützung, insbesondere während der Phase der Prüfungen und der Diplomarbeit. Ohne ihn hätte ich mein Studium wohl nicht zu Ende führen können. Danke für die vielen Stunden des Korrekturlesens und der Hilfe bei der Suche verständlicher Formulierungen.

Inhaltsverzeichnis

Abkürzungsverzeichnis

CCD-Chip	Charge Coupled Device-Chip
CSCW	Computer-Supported Cooperative Workwim
HID-Standard	Human Interface Device-Standard
IR	Infrarot
JNI	Java Native Interface
LED	Light Emitting Diode
Wiimote	Wii-Remote
WIMP	Window Icon Menu Pointer

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel I

1.1 Einleitung

Mit Erscheinen der Spielkonsole **Wii** läutete Nintendo im Jahre 2006 ein neues Zeitalter für Computerspiele ein. Anstatt wie bisher mit einem mehr oder weniger kompliziert zu bedienenden Controller starr vor einem Bildschirm zu sitzen, war von nun an Körpereinsatz gefragt, um sich durch die Spielewelten der Konsole zu bewegen.

Möglich machte diesen Fortschritt die Entwicklung des innovativen Controllers **Wii-Remote**, der kabellos via Bluetooth mit der Konsole verbunden wird. Beschleunigungssensoren und eine Infrarotkamera im Controller ermöglichen, in Verbindung mit einem Infrarotsensor, die Erkennung der Bewegung des Spielers im dreidimensionalen Raum.

Die Anwendung der von Nintendo verwendeten Technologie, auf die ich in Kapitel 22.1 näher eingehen werde, ist so innovativ, dass ihr Gebrauch durchaus nicht auf die Verwendung mit Spielkonsolen beschränkt bleibt.

Es bieten sich zahlreiche Ansätze der Verwendung, über die es sich lohnt nachzudenken. Einen davon zeige ich mit dieser Diplomarbeit. Es haben sich bereits mehrere Projekte und Abhandlungen mit neuen Ideen und deren Realisierung beschäftigt, aber die Möglichkeiten sind noch lange nicht ausgereizt.

1.2 Verwandte Arbeiten

Mittlerweile wohl weltweit bekannt wurde mit seinen Wii-Remote Projekten der Amerikaner Lee von der HCII – Carnegie Mellon University [LeeJ]. Lee nutzt die Infrarotkamera der Wii-Remote, um Handbewegungen zu erkennen. Er baute sich dazu mit Infrarot-LEDs einen Infrarot-Strahler, postierte diesen in Nähe der befestigten Wii-Remote, versah einen Handschuh mit Reflektoren und ist nun mittels selbst geschriebener Software in der Lage, ein Computerprogramm mittels Handbewegungen im freien Raum zu steuern (nach seiner eigenen Aussage so gesehen in dem Film „Minority Report“).

Ein weiteres seiner Projekte ermöglicht eine dreidimensionale Darstellung auf einem Computermonitor. Hierzu versah er eine Brille seitlich mit Infrarot-LEDs. Eine auf dem Monitor angebrachte Wii-Remote ermittelt mit ihrer Infrarotkamera den jeweiligen Abstand der LEDs zur Konsole und ist somit in der Lage, die Kopfbewegung des Anwenders an den Computer zu übertragen. Ein Computerprogramm berechnet dann für die jeweilige Lage des Kopfes die zugehörige Ansicht des darzustellenden Objektes. Das Ergebnis ist

eine realistische, dreidimensionale Ansicht eines Objektes auf dem Computermonitor, in Abhängigkeit des Betrachtungswinkels.

Poppinga von der Universität Oldenburg [Poppinga 07] beschäftigte sich im Rahmen des Projektes „Beschleunigungsbasierte 3D-Gestenerkennung mit dem Wii-Controller“ mit der Frage, inwieweit die Wii-Remote für Zwecke der Gestenerkennung genutzt werden kann. Mit der Wii-Remote ist man nicht mehr auf zwei Dimensionen der Bewegungserkennung, wie sie z. B. eine Computermaus vorschreibt, eingeschränkt. Die enormen Vorteile liegen auf der Hand. Poppinga beschreibt die unterschiedlichen Algorithmen, die für die Mustererkennung herangeführt werden können und zeigt die möglichen Anwendungsgebiete der entstandenen Gestenerkennungssoftware. Schlömer et al. führen dieses Projekt weiter und zeigen, dass es Computern möglich ist, Gesten mit einer minimalen Anzahl von Trainingsdurchläufen lernen zu lassen [Schlömer 08].

Einen weiteren interessanten Ansatz zeigt das Projekt von Lee et al., das in dem Artikel „WiiArts: creating collaborative art experience with Wii Remote interaction“ [LeeH 08] veröffentlicht wurde. Lee et al. erforschen hierbei die Möglichkeiten der Verwendung einer Spielkonsole, um gemeinsam, interaktiv, Kunst zu erschaffen. WiiArts ist ein experimentelles Video-, Audio- und Bildverarbeitungs-Kunstprojekt, das die Nintendo Wii Remote und die Sensorbar zur weiteren Umsetzung verwendet. Hierzu haben Lee et al. mehrere WiiArt-Prototypen entwickelt:

- Illumination (draWiing)
- Beneath (Waldo)
- Time Ripples
- WiiBand

Diese Projekte ermöglichen es bis zu drei Personen, gemeinsam Kunst zu erschaffen. Im Beispiel der Illumination können sie gemeinsam Zeichnungen auf einer Leinwand erstellen: flüssige Kerzenlichtspuren. Dies geschieht in einem abgedunkelten Raum und soll ein Erlebnis für die Sinne sein.

Das Projekt „A Wii remote, a game engine, five sensor bars and a virtual reality theatre.“ [Schou 07] von Schou et Gardner hingegen beschäftigt sich mit der Integration der Wiimote in ein **Virtual Reality Theatre**. Hierbei stellte sich unter anderem heraus, dass der Einsatz einer einzigen Sensorbar Probleme bereitet, da der Radius für den Infrarotempfang bzw. für das Aussenden des Infrarot-Signals sehr eng ist und man diesen Kegel erweitern müsste. Sie kamen auf die Idee, mehrere Sensorbars zu benutzen und haben dies auch in einem anschließenden Artikel „A surround interface using the Wii controller with multiple sensor bars“ [Schou 08] veröffentlicht.

Mit der Erstellung einer Java-API haben sich auch schon mehrere Projekte beschäftigt. Unter anderem haben die Programmierer der WiiuseJ [WiiuseJ] eine Java-API basierend

auf der **wiiuse**-API entwickelt und eine Beispielanwendung bereitgestellt, mit der man die Wiimote einstellen und Daten auslesen kann.

Zu meiner jetzigen Arbeit bestehen jedoch wesentliche Unterschiede. So lag zum einen mein Schwerpunkt darin, die Möglichkeiten der Wiimote mit bekannten Entwurfsmustern zur Verfügung zu stellen. Zum anderen lehnt sich der Aufbau meiner API der Maus-API an, was eine Integration der Wiimote in Anwendungen drastisch erleichtert. Zusätzlich stelle ich noch weitere Funktionalitäten der Wiimote bereit. Zudem basiert meine API auf einer Client-Server-Anwendung, die weitere Möglichkeiten bietet, auf die ich in Kapitel III – Implementierung näher eingehe.

1.3 Motivation

Teamwork: Dies bedeutet *gleichzeitig* an *einem* Projekt zu arbeiten. Dieses Thema war schon immer von großer Bedeutung und erhielt wohl mit Einzug des Computers einen noch höheren Stellenwert. Die große Bandbreite an dessen Möglichkeiten musste man *gemeinsam* effektiver nutzen können.

Leider war diese Grundidee bislang aber weitgehend nur ein Wunschgedanke. Ein Anwender, ein Computer, eine Tastatur und eine Computermaus (ein so genanntes WIMP-Interface: **W**indow **I**con **M**enu **P**ointer) war bei der Computerarbeit, seit der Erfindung der Maus als Zeigegerät, vor rund zwei Jahrzehnten durch Douglas C. Engelbart, bisher immer die Regel. Ein gemeinsames, paralleles Arbeiten mehrerer Personen war oftmals nur mit klassischen Mitteln, wie Papier und Bleistift um einen runden Tisch herum, möglich. Die erarbeiteten Ergebnisse wurden danach bestenfalls von einer Person in den PC eingegeben, um Daten zu sichern.

Auch die gängigen Betriebsysteme von Computern waren bislang nicht in der Lage, mehrere Eingabe- und Zeigegeräte gleichzeitig zu verwalten. Microsoft erkannte dieses Manko und veröffentlichte am 23. Mai 2007 MultiPoint SDK [Microsoft] und bot damit die Möglichkeit, für neue Systeme wie Windows XP SP2 (32-Bit-Version) und Windows Vista (32-Bit-Version), Software für den Multi-Userbetrieb zu schreiben. So entwickelte z. B. die Firma –„Wunderworks“ die Software „Teamplayer“ [Wunderworks].

Die Möglichkeit des Computer-supported collaborative work (kurz: CSCW) [CSCW] wurde somit für Microsoft-User geschaffen; nicht jedoch für andere Betriebssysteme. Der Begriff CSCW findet sich auf der Internetseite der Gesellschaft für Informatik e.V.: „Unter "Computer-Supported Cooperative Work" oder CSCW versteht man das Forschungsgebiet hinter dem Einsatz von Software zur Unterstützung von Zusammenarbeit (Collaboration), das Einflüsse aus den Forschungsgebieten Organisations- und Führungslehre, Psychologie, Informatik, Soziologie, u.a. zusammenfasst.“ [CSCW]. Der Begriff findet sich daher auch im Titel der mittlerweile zum sechsten Mal veranstalteten Fachkonferenz wieder (zuletzt im Jahr 2008) [Begole].

Mit meiner Diplomarbeit verfolgte ich im Wesentlichen zwei Ziele:

Teil 1:

Die Erstellung einer Java-API (**A**plication **P**rogramming **I**nterface), die es Programmierern nach vertrauten Entwurfsmustern ermöglicht, die Wii-Remote am PC nutzbar zu machen.

Diese API soll analog zur bis dahin bekannten Maus-API in einem beliebigen Programm benutzbar sein und zusätzlich den Programmierern die umfangreichen Möglichkeiten der Wii-Remote zur Verfügung stellen.

Die Datenübermittlung zwischen Wii-Controller und Computer soll, unter Verwendung der freien Bibliothek **wiiuse**, mittels Bluetooth erfolgen.

Für die Anbindung an diese Bibliothek soll eine netzwerkbasierte Client-Server-Architektur verwendet werden. Diese ermöglicht die räumliche Trennung des Rechners auf dem die Serveranwendung läuft vom Rechner der Client-Anwendung.

Teil 2:

Die Programmierung einer Beispielanwendung, die erstmals den Einsatz der beschriebenen API und die Leistungsfähigkeit meiner Implementierung nachweist. Diese soll zudem die Nutzung des Wii-Controllers im Mehrbenutzerbetrieb demonstrieren.

Die Anwendung soll die Erstellung einer kollaborativen Netzwerkerhebung ermöglichen. Die Konstruktion dieser Netzwerke fände zum Beispiel in der Software-Entwicklung Verwendung.

Kapitel II – Theorie und Grundlagen

Dieses Kapitel gibt einen Einblick in die Hardware und damit in die innovative Technologie, die den Spielkonsolenmarkt revolutionierte. Dabei gehe ich nicht näher auf die Spielkonsole Wii selbst, sondern vielmehr auf den verwendeten Controller Wii Remote ein.

Des Weiteren stelle ich die zur Hardware-Anbindung verwendete Bibliothek **wiiuse** vor und erläutere, warum ich diese und nicht eine der anderen frei verfügbaren Bibliotheken verwendet habe. Hierbei konzentriere ich mich auf die wichtigsten Funktionen der Bibliothek, die ich in meiner SIGMAii-API verwende.

Anschließend gehe ich auf die Grundüberlegungen zur Erstellung der in Java geschriebenen SIGMAii-API näher ein. Hier zeige ich auch erstmals deren Struktur.

2.1 Grundlagen: Wii-Remote-Technologie

Die von Nintendo im Dezember 2006 im europäischen Raum auf den Markt gebrachte Wii-Konsole setzt nicht auf die bis dahin von anderen namhaften Herstellern immer wieder verbesserte Grafik der Computerspiele, sondern auf ein neues Konzept der Spielebedienung, nämlich dem mit viel Körpereinsatz. Realisiert wurde das mit dem neu entwickelten Wii-Controller. Dieser wird wegen seiner äußerlichen Erscheinung, die mehr an eine Fernbedienung, als an einen Spiel-Controller erinnert, auch „Wii-Remote“ oder kurz (und im Folgenden) „Wiimote“ genannt.

Abbildung 2.1 zeigt die Wiimote und deren Tasten mit den zugehörigen Bezeichnungen.

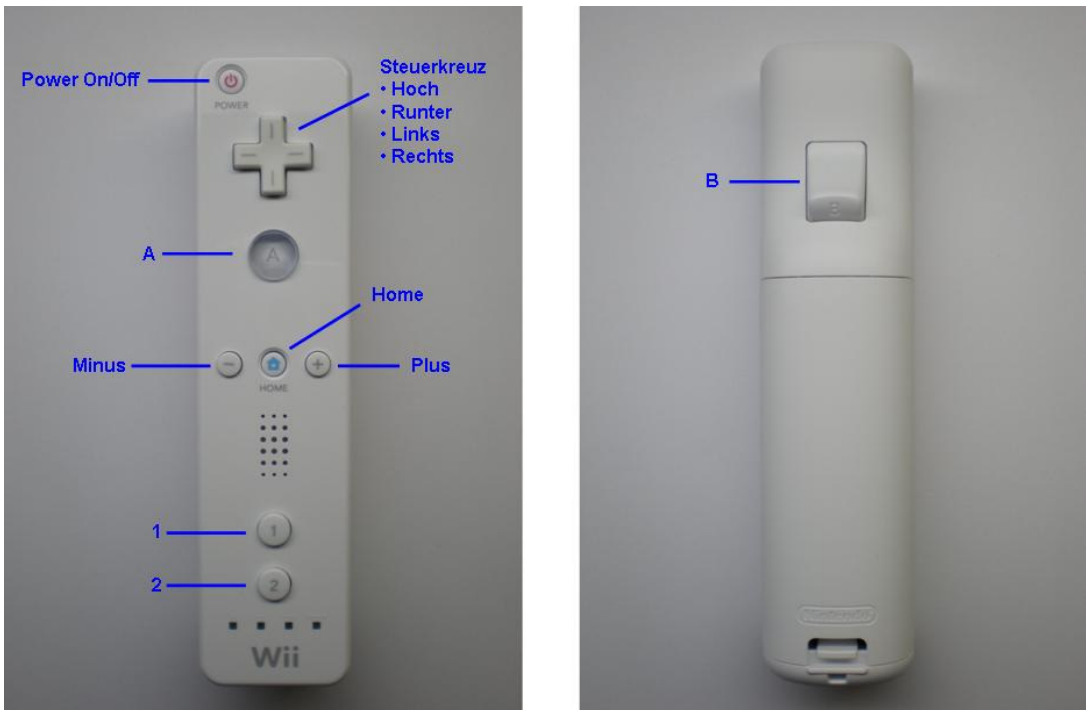

Abbildung 2.1: Nintendo's Wii-Remote: Ober- und Unterseite

Tabelle 2.1 zeigt die Tastenbezeichnung und den zugehörigen Tastencode, der vom Controller bei Tastenbetätigung übertragen wird.

Button	Hexadezimalwert
2	0x0001
1	0x0002
B	0x0004
A	0x0008
MINUS	0x0010
HOME	0x0080
LINKS	0x0100
RECHTS	0x0200
RUNTER	0x0400
HOCH	0x0800
PLUS	0x1000

Tabelle 2.1: Tastenbezeichnungen und Tastencode

Am unteren Ende des Controllers befindet sich eine Erweiterungsbuchse zum Anschluss zusätzlicher Eingabegeräte (z.B. der Nintendo Nunchuck oder der Wii MotionPlus), um die Steuerungsmöglichkeiten der Spielewelten weiter zu vergrößern.

Abbildung 2.2: Nintendo's Wii-Remote: Fuß- und Kopfansicht

Im Kopfende befindet sich, verdeckt durch einen Infrarotfilter der optische Sensor in Form einer Infrarotkamera.

Des Weiteren verfügt der Controller über vier LEDs, einen Lautsprecher für die Ausgabe akustischer Signale und ein so genanntes „Rumble-Feature", ähnlich dem Vibrationsalarm bei Handys für taktile Benachrichtigungen an den Anwender.

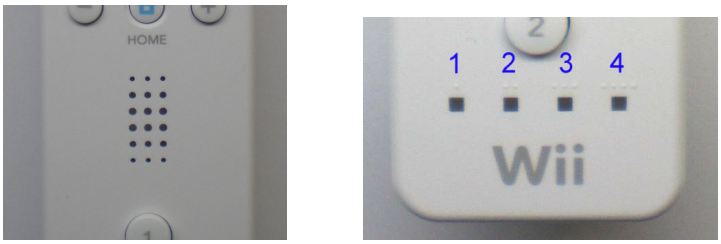

Abbildung 2.3: Lautsprecher und LEDs

2.1.1 Infrarotkamera

Mit Hilfe der Infrarotkamera der Firma „PixArt" wird die Position des Controllers relativ zu der so genannten „Sensorbar" bestimmt, die ober- oder unterhalb des Monitors oder z. B. einer Leinwand angebracht wird. Diese „Sensorbar" ist lediglich eine Leiste, in deren Enden jeweils mehrere, mit Infrarotfilter abgedeckte, Infrarot-LEDs untergebracht sind.

PixArt's Multi-Object Tracking™ engine (kurz: MOT) [PixArt] ist in der Lage, bis zu vier Infrarotpunkte gleichzeitig zu erkennen und mit sehr hoher Geschwindigkeit die genaue Position und damit die reale, auch sehr schnelle Bewegung des Controllers zu berechnen.

Für jeden Infrarotpunkt ermittelt der Controller ein xy-Werte-Paar mit einer Auflösung von 1024x768. Diese Auflösung ergibt sich aus einer sehr empfindlichen, monochromen optischen Kamera und deren CCD-Chip mit entsprechender Auflösung.

Der über der Kamera angebrachte Infrarotfilter verhindert, dass der Controller auch auf sichtbares Licht reagiert.

Im Wesentlichen ist es diese Kamera, die es ermöglicht, die Wii-Konsole mittels dieses Controllers, ähnlich dem PC mittels einer Computermaus, zu bedienen.

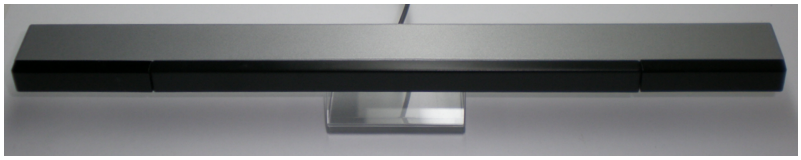

Abbildung 2.4: Sensorbar der Wii

2.1.2 Beschleunigungssensor ADXL330

Der im Controller untergebrachte Beschleunigungssensor ADXL330 setzt grob gesagt mechanische Bewegungen in elektrische Spannungen um. Er kann Beschleunigungen und Drehungen mit einer Genauigkeit von +/-10% der realen Bewegung erfassen und reagiert mit einer Empfindlichkeit von +/-300mV/g sehr genau. Der Sensor erkennt dabei Bewegungen in x-, y- und z-Richtung.

Der ADXL330 wurde für neue Anforderungen bei elektronischen Konsumgütern, wie Mobiltelefone (Ausrichtung des Bildschirms je nach Lage des Gerätes), Spielgeräte (Steuerung der Spielwelten durch Gestenerkennung), Laptops (Erkennung von Stürzen und damit der Schutz von Festplatten, indem der Schreib-/Lesekopf „geparkt“ wird), Digitalkameras (Verwackelungsschutz) sowie Sport- und Gesundheitsvorrichtungen entwickelt [ADXL330].

Erst durch die Entwicklung und kommerzielle Vermarktung dieses Chips wurde Nintendo in die Lage versetzt, ein so gravierend anderes Spiele-Steuerkonzept wie für die Wii-Konsole zu verwirklichen.

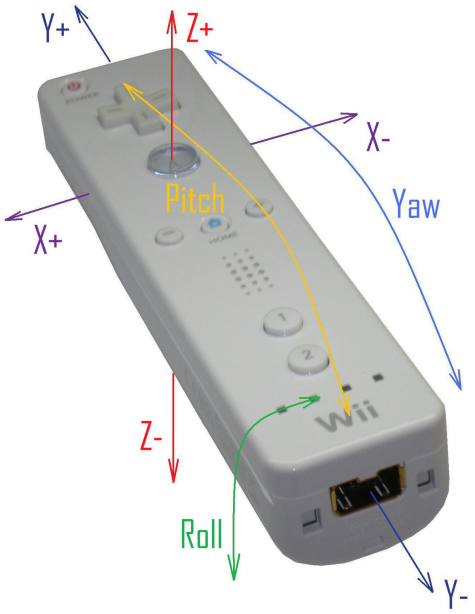

Abbildung 2.5: Wii-Remote-Achsen

Abbildung 2.5 zeigt die möglichen Bewegungs- bzw. Beschleunigungsrichtungen:

- +/- x-Richtung
- +/- y-Richtung
- +/- z-Richtung
- Pitch, Drehung um die x-Achse
- Roll; Drehung um die y-Achse
- Yaw; Drehung um die z-Achse

2.1.3 Bluetooth

Die Datenübertragung des Controllers erfolgt drahtlos über Bluetooth. Der maximale Abstand zum Empfänger beträgt im Normalfall ca. 10m. Bei Nutzung der Wiimote als Zeigegerät wird die maximale Entfernung des Controllers zur Sensorbar jedoch infrarot-bedingt auf 5m reduziert.

Der verwendete Bluetooth-Chip Broadcom BCM2042, einem Single-Chip Bluetooth® Device mit einem voll integrierten Human Interface Device (HID), ermöglicht es, die Wiimote mit dem heimischen PC über Bluetooth 2.0 zu verbinden, sofern dieser über einen entsprechenden Empfänger verfügt. Der HID-Standard ist heutzutage in fast jedem Bluetooth-Empfänger integriert und kommt in jedem moderneren Handy und Notebook zum Einsatz [BCM2042].

Dadurch ist es problemlos möglich, eine drahtlose (wireless) Datenübertragung zwischen PC und Wiimote mittels des Betriebsystems aufzubauen.

Da hier eine Client-Server-Anwendung besteht, erfolgt die Kommunikation zur Wiimote Backend über die freie Bibliothek **wiiuse** [wiiuse] via Bluetoothstack (unter Linux standardmäßig „BlueZ“). Die Anbindung der Wiimote an den PC erfolgt für meine Zwecke folgendermaßen:

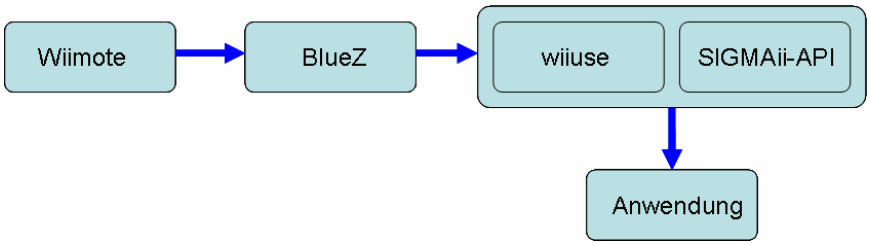

Abbildung 2.6: Anbindung der Wiimote über BlueZ an eine Anwendung

2.2 Die Bibliothek „wiiuse“

Da es die Aufgabenstellung dieser Arbeit nicht forderte, sich mit dem Auslesen der Daten der Wiimote auf Hardwareebene zu beschäftigen, erfolgt, wie in Kapitel 2.1.3 bereits erwähnt, die Anbindung der Wiimote Backend über die freie Bibliothek **wiiuse**.

Zu Beginn des Projekts stand die Überlegung, welche Bibliothek zu verwenden wäre. Ein Hauptkriterium war, dass sie sowohl auf Windows-, wie auch auf Linuxsystemen gleichermaßen implementierbar sein sollte, was die Auswahl bereits erheblich einschränkte. Treiberprojekte wie z.B. GlovePie [GlovePie] sind meist auf ein bestimmtes Betriebssystem zugeschnitten und bieten nicht die erwünschte Möglichkeit, erdenklich viele Wiimotes gleichzeitig anschließen zu können.

Dass die **wiiuse** diese Kriterien erfüllt und zudem auch sehr intuitiv zu verwenden ist, machte die Auswahl schließlich leicht. Sie arbeitet sehr zuverlässig, wird ständig weiterentwickelt und so auch durch zusätzliche Features erweitert. Zudem gibt es eine sehr gute Dokumentation.

Die **wiiuse**-Bibliothek ist in C geschrieben und lizenziert unter GNU GPLv3 und GNU LGPLv3 (nicht gewerblich). Sie unterstützt die Betriebssysteme Linux, Windows 2000, Windows XP und Windows Vista und arbeitet mit den Bluetoothstacks Microsoft-Stack, BlueSoleil-Stack, Widcomm Stack und dem bisher noch ungetesteten Toshiba-Stack problemlos zusammen.

Sie unterstützt die Bewegungs- und Infrarotabfrage. Zusätzlich können neben der Wiimote noch Controllererweiterungen wie z.B. „Nunchuk“, „Guitar Hero“ und der „Classic Controller“ abgefragt werden. Eine Unterstützung für das Wii Balance Board, das am 25.04.2008 in Deutschland auf den Markt kam, gibt es leider noch nicht.

An Ausgabemöglichkeiten werden die visuelle und die haptische Ausgabe unterstützt, jedoch gibt es in der aktuellen Version 12, die auch in meiner Diplomarbeit verwendet wird, noch keine Soundunterstützung.

Die **wiiuse**-API bietet eine Vielzahl an Funktionen, die aber letztendlich in der SIGMAii-API nicht alle Verwendung finden.

Hier die für meine Arbeit wichtigsten Funktionen im Überblick [wiiuse]:

- wiiuse_init()
- wiiuse_find()
- wiiuse_connect()
- wiiuse_poll()
- wiiuse_rumble()
- wiiuse_motion_sensing()
- wiiuse_set_leds()
- wiiuse_set_ir()
- wiiuse_set_ir_vres()

2.2.1 wiiuse_init()

Mit dem Funktionsaufruf

wiimotes = wiiuse_init(MAX_WIIMOTES)

wird auf die Funktion

WIIUSE_EXPORT struct wiimote_t wiiuse_init(int wiimotes)**

zugegriffen und ein Array von Wiimote Strukturen initialisiert.

Als Parameter übergibt man die gewünschte maximale Anzahl von Wiimotes. In unserem Fall ist **MAX_WIIMOTES** gleich 4. Die Anzahl kann fast beliebig erhöht werden und ist nur durch die Bluetooth-Bandbreite begrenzt. Als Rückgabewert erhält man ein Array von initialisierten **wiimote_t** Strukturen, die man für weitere Funktionen benötigt.

2.2.2 wiiuse_find()

Um wartende Wiimotes zu finden, wird der Funktionsaufruf

wiiuse_find(wiimotes, MAX_WIIMOTES, 5)

verwendet. Hierbei wird auf die Funktion

WIIUSE_EXPORT int wiiuse_find(struct wiimote_t ** wm, int max_wiimotes, int timeout)

zugegriffen.

Als Parameter werden das mit **wiiuse_init()** initialisierte Array von initialisierten **wiimote_t**-Strukturen, die Anzahl der Wiimotes im **wiimote_t**-Array (hier die vier **MAX_WIIMOTES**, die mit **wiiuse_init()** initialisiert wurden) und eine Timeoutzeit übergeben.

Die Timeoutzeit beträgt in unserem Fall fünf Sekunden. Mit diesem Funktionsaufruf wird also nach einer maximalen Anzahl von vier Wiimotes maximal fünf Sekunden gesucht. Wurden bis Ablauf der Timeoutzeit keine Wiimotes gefunden, bricht die Suche ab.

Die Funktion gibt uns die Anzahl der gefundenen Wiimotes zurück.

2.2.3 wiiuse_connect()

In einem Windows-System wird der Aufruf **wiiuse_connect()** nicht benötigt, da mit dem Finden der wartenden Wiimotes (**wiiuse_find()**) schon automatisch versucht wird, den vom System verwendeten Bluetoothstack zu verbinden. Für LINUX-Systeme ist der Aufruf jedoch notwendig.
Mit dem Zugriff auf die Funktion

WIIUSE_EXPORT int wiiuse_connect(struct wiimote_t ** wm, int wiimotes)

werden die zuvor gefundenen Wiimotes an die Anwendung gebunden und wir erhalten die Anzahl der erfolgreich angebundenen Wiimotes zurück.

Anschließend kann jede einzelne Wiimote direkt angesprochen werden, da jede eine eigene ID und Adresse hat, die in dem Array von **wiimote_t**-Strukturen hinterlegt ist.

2.2.4 wiiuse_poll()

Wiiuse arbeitet als ein Nichtblockierendes Pollingsystem [wiiuse]. Um auftretende Events abzufragen, muss kontinuierlich eine Schleife durchlaufen werden.
Mit dem Funktionsaufruf

wiiuse_poll(wiimotes, MAX_WIIMOTES)

wird auf die Funktion

WIIUSE_EXPORT int wiiuse_poll(struct wiimote_t ** wm, int wiimotes),

zugegriffen. Als Parameter werden ein Array von Pointern zu den **wiimote_t**-Strukturen sowie die Anzahl der **wiimote_t**-Strukturen in dem Array **wm** übergeben.

Als Rückgabe erhält man die Wiimotes, an denen ein Event aufgetreten ist. Daraufhin wird überprüft, welches Event vorliegt. Für uns sind nur die **WIIUSE_EVENT**s von Interesse. Darüber hinaus ist es auch möglich, folgende Events abzurufen:

- **WIIUSE_NONE**: kein Event aufgetreten
- **WIIUSE_STATUS**: Statusbericht wurde von der Wiimote erhalten
- **WIIUSE_DISCONNECT**: die Wiimote wurde getrennt
- **WIIUSE_READ_DATA**: angeforderte Daten vom Wiimote ROM/Register wurden zurückgegeben
- **WIIUSE_NUNCHUK_INSERTED**: Nunchuk wurde angeschlossen
- **WIIUSE_NUNCHUK_REMOVED**: Nunchuk wurde entfernt
- **WIIUSE_CLASSIC_CTRL_INSERTED**: Classic Controller wurde angeschlossen
- **WIIUSE_CLASSIC_CTRL_REMOVED**: Classic Controller wurde entfernt
- **WIIUSE_GUITAR_HERO_3_CTRL_INSERTED**: Guitar Hero wurde angeschlossen
- **WIIUSE_GUITAR_HERO_3_CTRL_REMOVED**: Guitar Hero wurde entfernt

Bei der Abfrage der **WIIUSE_EVENT**s kann überprüft werden, ob ein Button betätigt wurde, ob eine Zeigerbewegung stattfand (nur von Interesse, wenn der IR-Empfang durch **wiiuse_set_ir()** für die entsprechende Wiimote aktiviert wurde) oder ob eine Beschleunigung erfolgte (vorausgesetzt dieses Event wurde durch **wiiuse_motion_sensing()** für die jeweilige Wiimote aktiviert).

Welches Event ausgelöst wurde, muss für *jede* Wiimote überprüft werden. Wie das geschieht, wird im Kapitel III – Implementierung erläutert.

2.2.5 wiiuse_rumble()

Mit dem Funktionsaufruf

wiiuse_rumble(wiimotes[i], 1)

wird für jede Wiimote **i** die Funktion

void wiiuse_rumble(struct wiimote_t ** wm, int status)

aufgerufen und die Wiimote **i** sowie der Status (1 für Rumble Ein, 0 für Aus) übergeben.

Diese Funktion ermöglicht es, ein Haptik-Feedback auszulösen.

2.2.6 wiiuse_motion_sensing()

Die Abfrage von Beschleunigungsdaten ist standardmäßig nicht aktiviert. Eine Aktivierung erreicht man mit dem Funktionsaufruf

wiiuse_motion_sensing(wiimotes[i], 1)

wodurch man die Funktion

void wiiuse_motion_sensing(struct wiimote_t ** wm, int status)

aufruft und die Wiimote **i** sowie den Status (1 für Ein, 0 für Aus) übergibt.

Da in unserem Fall die Übertragung der Beschleunigungsdaten von erheblichem Interesse ist, wird sie standardmäßig für *jede* Wiimote aktiviert. Die Nutzung wirkt sich allerdings negativ auf die Batterieleistung der Wiimote aus.

2.2.7 wiiuse_set_leds()

Die LEDs der Wiimote lassen sich mit

wiiuse_set_leds()

und dem Funktionsaufruf

WIIUSE_EXPORT void wiiuse_set_leds(struct wiimote_t ** wm, int leds)

ansprechen.
Als Übergabeparameter haben wir wieder die Wiimote **i**, deren LEDs geschaltet werden sollen und die zu schaltende(n) LED(s).

Hier gilt:

WIIMOTE_LED_NONE für alle LEDs aus
WIIMOTE_LED_1 für die LED 1,
WIIMOTE_LED_2 für die LED 2,
WIIMOTE_LED_3 für die LED 3 und
WIIMOTE_LED_4 für die LED 4.

LED	Hexadezimalwert
WIIMOTE_LED_NONE	0x00
WIIMOTE_LED_1	0x10
WIIMOTE_LED_2	0x20
WIIMOTE_LED_3	0x40
WIIMOTE_LED_4	0x80

Tabelle 2.2: Konstanten zur LED-Ansteuerung

Folglich können über die Anzeige der LEDs 2^4-1 Wiimotes differenziert werden.

2.2.8 wiiuse_set_ir()

Die Abfrage von Infrarot-Daten ist ebenfalls standardmäßig nicht aktiviert. Da in unserem Fall diese Abfrage von größtem Interesse ist, wird der IR-Empfang automatisch für jede Wiimote aktiviert. Dies erfolgt mit Aufruf der Funktion

void wiiuse_set_ir(struct wiimote_t* wm, int status)

und Übergabe des Status 1 für jede Wiimote **i** (1 für Infrarotfunktion Ein, 0 für Aus). Auch hier hat die Aktivierung eine Reduzierung der Batterieleistung zur Folge.

Wie in Kapitel 2.1.1 bereits erwähnt, wird für die IR-Abfrage die Wii Sensorbar benötigt, die in unserem Fall oberhalb des Bildschirms angebracht werden sollte. Da die Infrarotkamera in der Lage ist, bis zu vier Infrarotpunkte gleichzeitig abzufragen und deren Positionen zu berechnen, bietet auch **wiiuse** diese Möglichkeit:

Wiiuse fragt alle möglichen vier IR-Punkte ab. Ist ein Punkt sichtbar, berechnet **wiiuse** durch Interpolation die zugehörigen xy-Werte. Die xy-Position wird dabei relativ zu einem virtuellen Bildschirm gesehen, der standardmäßig für 16:9 bei (660x370) Pixel und für 4:3 bei (560x420) Pixel liegt.

Die Abmaße des tatsächlichen Bildschirms können jedoch vom Benutzer mit dem Funktionsaufruf **wiiuse_set_ir_vres()** verändert werden, der im nächsten Absatz behandelt wird.

Wiiuse ermöglicht die Ausgabe der aktuellen Werte bezogen auf die tatsächliche Bildschirmauflösung. Die Koordinate (0,0) ist dabei die linke obere Ecke des Bildschirms.

2.2.9 wiiuse_set_ir_vres()

Wie im vorherigen Abschnitt schon angesprochen, ist es notwendig, den standardmäßigen virtuellen Bildschirm zu ändern, um die tatsächlichen xy-Koordinaten des jeweiligen Cursors, bezogen auf die tatsächlich benutzte Bildschirmauflösung, zu erhalten.

Dies geschieht mit dem Funktionsaufruf

wiiuse_set_ir_vres(wiimotes[i], width, height)

Hierbei wird für jede Wiimote **i** der virtuelle Bildschirm auf die Breite **width** und die Höhe **height** des verwendeten Bildschirms gesetzt. Die Einheit ist Pixel.

2.3 Die SIGMAii-API

Eine der beiden Hauptaufgaben dieser Arbeit bestand darin, die SIGMAii-API-Struktur unabhängig für beliebige Anwendungen zu gestalten und für den Programmierer nach vertrauten Entwurfsmustern zugänglich zu machen. Für die Entwicklung musste ich hauptsächlich folgende Aspekte in Betracht ziehen:

- Welche Funktionen stellt die Hardware zur Verfügung
- Wie soll die Struktur der API aussehen
- Wie erfolgt die Anmeldung der API in der Anwendung

Es stellte sich also die Frage, welche Funktionen die Wiimote zur Verfügung stellt. Sie verfügt über folgende Input- und Outputmöglichkeiten:

Input:

- Tasten:
  11 Tasten (Steuerkreuz, A, B, 1, 2, +, -), die jeweils den Status betätigt/nicht betätigt haben können. Die Powertaste ist nicht verwendet.
- Beschleunigungssensoren:
  Erkennung der Beschleunigungen über drei Achsen: +/- x, +/- y und +/- z.
- Rotation:
  Erkennung der Rotationen um drei Achsen. Die x-Achse „PITCH“, die y-Achse „ROLL“ und die z-Achse „YAW“.
- Infrarotkamera:
  Abfrage von bis zu vier Infrarot-Punkten und Ausgabe der jeweiligen xy-Koordinate.

Zusätzlich kann relativ zum verwendeten Bildschirm die xyz-Koordinate ausgegeben werden.

Output:

- LEDs:
  Vier LEDs, die jeweils den Status ein-/ausgeschaltet haben können. Der Benutzer kann anhand der eingeschalteten LED(s) und der Nummerierung der Cursor, *seinen* Cursor erkennen, ohne Bewegungen mit der Wiimote ausführen zu müssen.
- Haptisches Feedback Rumble:
  Ich werde die haptische Ausgabe benutzen, um den Benutzer darauf aufmerksam zu machen, dass er versucht, eine für ihn gerade gesperrte Funktion, aufzurufen. Der Vorteil gegenüber z.B. einer Warnmeldung besteht darin, dass keine zusätzliche Aktivität des Benutzers notwendig ist, wie z.B. das „Wegklicken“ eines Meldefensters. Sobald der jeweilige Cursor den gesperrten Bereich verlässt, wird auch die Rumble-Funktion ausgeschaltet. Ein weiterer Vorteil ist, dass die Information nur für den auslösenden Benutzer erkennbar ist.
- Lautsprecher:
  Grundsätzlich ermöglicht die Wiimote die Ausgabe akustischer Informationen (auch Sprache) über den Lautsprecher. Da die Wiiuse diese Funktion in der aktuellen Version aber noch nicht unterstützt, konnte ich sie in meiner API leider nicht nutzen. Eine spätere Implementierung ist natürlich möglich.

Nachdem ich mir den Überblick über die Funktionen der Wiimote verschafft hatte, musste ich mir Gedanken über die Struktur der API machen.

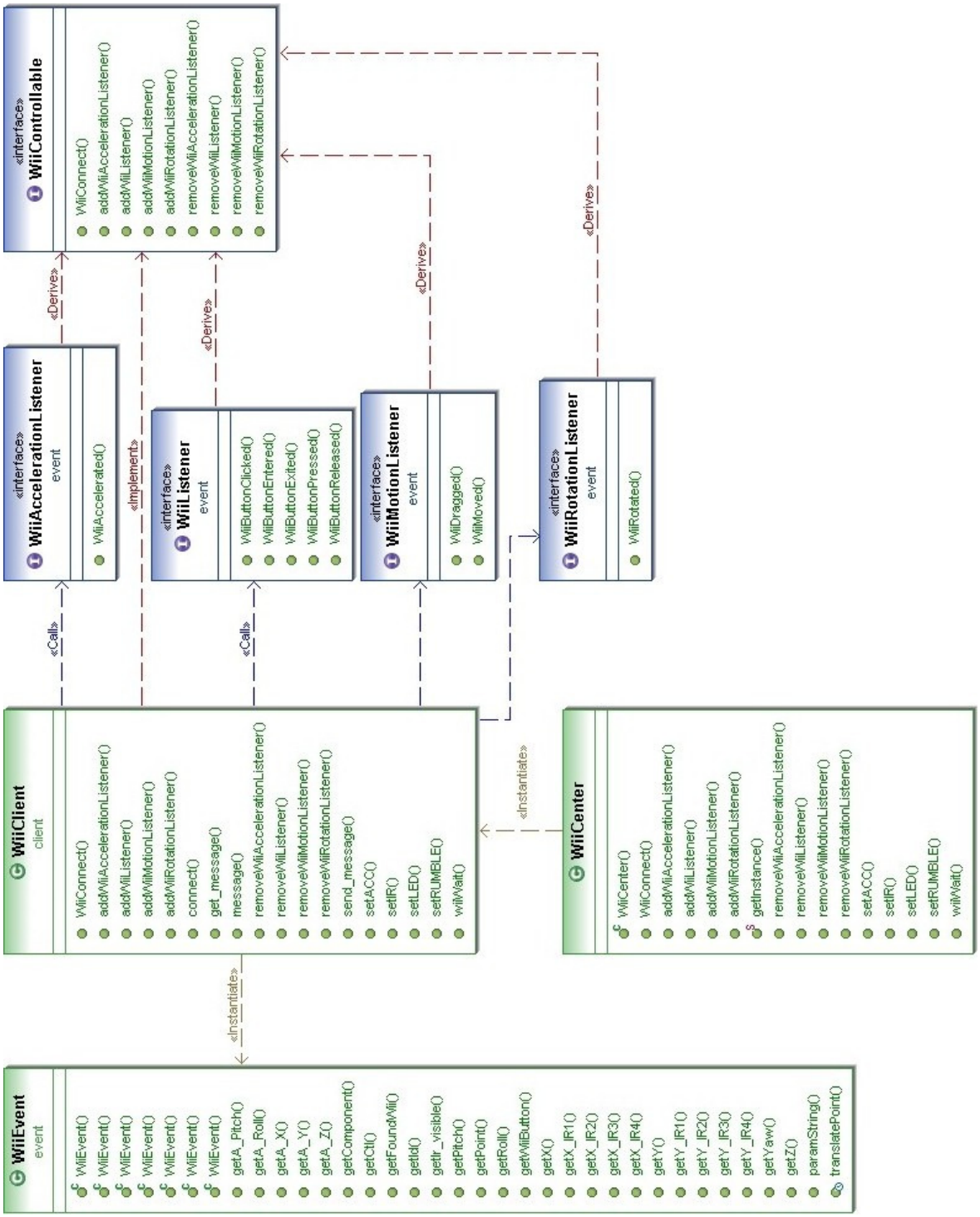

Abbildung 2.7: Die SIGMAii-API im Überblick

Für den Eingang der Daten der Wiimotes benötigt man Listener für die Callbackfunktion. Wenn sich der Zustand des Wii-Controllers ändert, müssen sich als Folge davon die Zustände aller Objekte ändern, die sich auf das geänderte Subjekt beziehen. Diese Änderung soll automatisch erfolgen, ohne dass die angebundenen Objekte ein Refresh aufrufen müssen. Man benötigt zudem die Möglichkeit, dass mehrere Objekte das Subjekt beobachten. Die Verwendung des Entwurfsmodells Observer-Pattern bietet dazu einen Lösungsansatz.

Der Observer muss bei einem zu beobachtenden Subjekt registriert und auch wieder abgemeldet werden können. Das bedeutet, dass das Subjekt diese Methoden bereitstellen muss. Dies funktioniert über die Methoden **addObserver()** und **removeObserver()**. In unserem Fall ist das zu beobachtende Objekt die Wiimote (bzw. der Client, der vom Server die Daten der Wiimote empfängt).

Die Wiimote als Ereignisquelle erzeugt dabei gewisse Events. Für jeden Ereignistyp muss es daher Schnittstellen mit Methoden, so genannte Interfaces, geben, die im Falle des Auftretens eines Events aufgerufen werden.

Die Wiimote erzeugt vier verschiedene Eventtypen:

- Button-Event
- Beschleunigungs-Event
- Bewegungs-Event
- Infrarot-Event

Demzufolge müsste es für jedes dieser Events einen eigenen Listener geben.

Anfangs stellte ich mir die Frage, ob es sinnvoll wäre, ein zentrales Interface zu haben, das alle auftretenden Events behandelt. Letztendlich bietet aber eine Unterteilung eine viel bessere Übersicht über die einzelnen Events. Man kann klar filtern, welches Event behandelt werden soll, bzw. welchen Listener man für ein Objekt registrieren möchte. Zudem sollte das Eventmodell an das MouseEvent-Modell angelehnt sein, das folgende Interfaces mit Methoden zur Verfügung stellt:

- MouseListener:
  - void mouseClicked(MouseEvent e);
  - void mouseEntered(MouseEvent e);
  - void mouseExited(MouseEvent e);
  - void mousePressed(MouseEvent e);
  - void mouseReleased(MouseEvent e);
- MouseMotionListener:
  - void mouseDragged(MouseEvent e);
  - void mouseMoved(MouseEvent e);

Somit würde man zumindest drei verschiedene Listenerinterfaces benötigen. Äquivalent zum **MouseListener** und zum **MouseMotionListener** gibt es in unserem Modell den:

- WiiListener und den
- WiiMotionListener

Der **WiiListener** ist für die Button-Events zuständig und soll somit die Methoden

- WiiButtonClicked()
- WiiButtonEntered()

- WiiButtonExited()
- WiiButtonPressed()
- WiiButtonReleased()

zur Verfügung stellen.

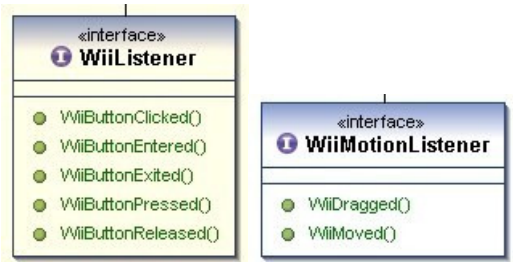

Abbildung 2.8: WiiListener und WiiMotionListener

Das Interface **WiiMotionListener** soll für die Events zuständig sein, die durch die Bewegung der Wiimote, also in Abhängigkeit der IR-Punkte, ausgelöst werden. Es stellt die Methoden

- WiiDragged()
- WiiMoved()

zur Verfügung. Damit sind die Button- und Infrarot-Events behandelt. Es fehlen noch die Bewegungs- und Beschleunigungs-Events.
Das Interface für die Beschleunigung heißt **WiiAccelerationListener** und stellt die Methode

- WiiAccelerated()

zur Verfügung. Als letztes Interface fehlt noch das Rotations-Interface, der **WiiRotationListener**. Es offeriert die Methode

- WiiRotated()

Abbildung 2.9: WiiRotationListener und WiiAccelerationListener

Die auftretenden Events werden in der Klasse **WiiEvent** gespeichert:

- **Button-Events**
  - Welcher Button wurde gedrückt
  - Welches Event ist aufgetreten:
    - WiiButtonPressed(),
    - WiiButtonReleased() oder

        - WiiButtonClicked()
    - Welcher Controller löste dieses Event aus
- **IR-Events**
    - Die Position der Wiimote: xyz-Koordinate
    - Welches Event ist aufgetreten:
        - WiiMoved() oder
        - WiiDragged()
    - Welcher Controller löste dieses Event aus
    - # der sichtbaren IR-Punkte
    - xy-Koordinaten der einzelnen IR-Punkte
- **Bewegungs-Events**
    - Welche Bewegung ist erfolgt:
        - ROLL,
        - PITCH oder
        - YAW
    - Welches Event ist aufgetreten:
        - WiiRotated()
    - Welcher Controller löste dieses Event aus
- **Beschleunigungs-Events**
    - In welche Richtung erfolgte die Beschleunigung:
        - +/- x,
        - +/- y, und/oder
        - +/- z
    - Welches Event ist aufgetreten:
        - WiiAccelerated()
    - Welcher Controller löste dieses Event aus

Das **WiiEvent** ist also die Klasse für alle Ereignisobjekte. Tritt ein Event auf, erzeugt der Client das Ereignisobjekt **WiiEvent**. Jeder eingetragene Listener wird über den Aufruf der Methode aus dem Listener informiert [Ullenboom 2007].

Abbildung 2.10: WiiEvent

Es wird ein zentrales Interface zur Verwaltung der einzelnen Listener benötigt, das **WiiControllable**. Dieses Interface stellt die Methoden zum Hinzufügen (addXXX) und Entfernen (removeXXX) der einzelnen Listener bereit und bietet die Möglichkeit, den Client mit dem Server zu verbinden bzw. Daten über die dll von der **wiiuse**-Bibliothek zu erhalten.

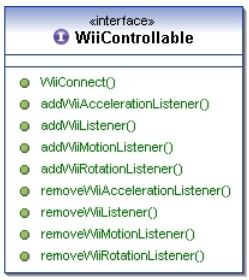

Abbildung 2.11: WiiControllable

Die Anmeldung erfolgt über eine zentrale Klasse. In unserem Fall ist es das **WiiCenter**, die Wii-Zentrale also. Die zentrale Instanz muss als Singleton deklariert sein. Das bedeutet, dass nur ein einziges Objekt erzeugt werden kann. In unserem Fall also nur ein einziger Client, der die angehängten Listener über die Events informiert. Diese Deklaration verhindert, dass man unendlich viele Subjekte erzeugen kann. Da es beim Observer-Pattern ja darum geht, dass *viele* Observer *ein* Subjekt beobachten können, ist diese Deklaration notwendig. **WiiCenter** soll zudem die Möglichkeit bieten, Listener an Komponenten anzuhängen bzw. diese wieder entfernen zu können.

Abbildung 2.12: WiiCenter

Das **WiiCenter** bietet die Möglichkeit mit

- addWiiAccelerationListener(),
- addWiiListener(),
- addWiiMotionListener() und
- addWiiRotationListener()

diese Listener an eine Komponente anzuhängen und mit

- removeWiiAccelerationListener(),
- removeWiiListener(),
- removeWiiMotionListener() und
- removeWiiRotationListener()

diese auch wieder von der Komponente zu entfernen.

Kapitel III – Implementierung

Dieses Kapitel befasst sich mit der Implementierung der SIGMAii-API.

Ich stelle das hier verwendete Entwurfsmuster vor. Anhand der Implementierung der SIGMAii-API erläutere ich deren Klassen bzw. Interfaces und zeige aufgetretene Probleme und die entsprechenden Lösungen. Das Kapitel beinhaltet zudem detaillierte Erklärungen zur Client-Server-Architektur, sowie die genaue Beschreibung der Nachrichtenformate.

3.1 Entwurfsmuster

Wie in Kapitel 2.3 erwähnt, bietet das Observer-Pattern für die vorgestellte API einen Lösungsansatz.

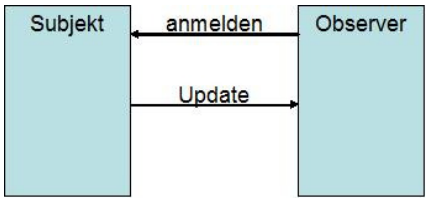

Abbildung 3.1: Observer-Pattern

Beim Observer-Pattern geht es darum, dass ein zu beobachtendes Subjekt von möglichst vielen Beobachtern überwacht werden kann und diese über Zustandsänderungen des Subjekts informiert werden, um darauf reagieren zu können. In unserem Fall ist das Subjekt der **WiiClient**. Dieser erhält durch den **WiiServer** Informationen von der Wiimote, die er durch einen Push an die Listener weitergibt.

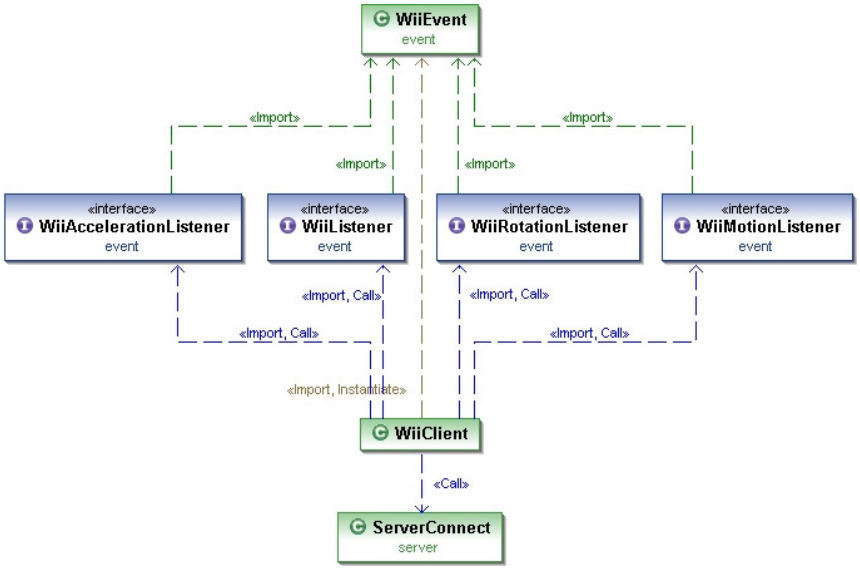

Abbildung 3.2: WiiClient als Observer

3.2 DLL

Da die Anwendung auch ohne die Client-Server-Architektur auf einem Rechner lauffähig sein soll, war es nötig, das Java Native Interface (kurz: JNI) zu nutzen. Man benötigt dafür eine C-Datei (Server) und eine Javaklasse, aus der eine Headerdatei erstellt werden muss.

Die existierende C-Datei aus der Client-Server-Architektur wäre mit wenigen Ergänzungen bereits nutzbar gewesen. Ich habe jedoch einen eigenen Server aus dem ursprünglichen Programm erstellt und einige Funktionen weggelassen (wie zum Beispiel die Socketverbindung), die hier nicht weiter benötigt werden.

Folgende Klassen müssen von Java aufrufbar sein:

- connectWiimotes()
- getWiimotes()
- getDataFromWii()
- getBUTTON()
- getCURSOR()
- getACC()
- setIR()
- setACC()
- setLED()
- setRUMBLE()
- disconnectWiimotes()

Die Erstellung der Javaklasse erfolgt, indem man die Methodenrümpfe deklariert, die später von Java aufgerufen werden. Durch das Schlüsselwort „native“ wird festgelegt, dass Java einen Aufruf über JNI ausführen soll.

In unserem Fall sieht das wie folgt aus:

```
public class ServerConnect {
    public static native void connectWiimotes();
    public static native int getWiimotes();
    public static native void getDataFromWii();
    public static native String getBUTTON();
    public static native String getCURSOR();
    public static native String getACC();
    public static native void setIR(int i, int status);
    public static native void setACC(int i, int status);
    public static native void setLED(int i, int led);
    public static native void setRUMBLE(int i, int status);
    public static native void disconnectWiimotes();
}
```

Hier fällt auf, dass die Darstellung einem Interface sehr ähnelt. Es handelt sich jedoch um eine Klasse. Mit einem Interface wäre ein Aufruf über JNI nicht möglich.

Nach den Deklarationen muss die Javaklasse kompiliert werden, um aus der kompilierten Datei eine Headerdatei zu erstellen. Die Kompilierung kann auf zwei Arten geschehen.

Über die Kommandozeile

javac ServerConnect.java

oder durch Verwendung einer Entwicklungsumgebung wie zum Beispiel „Eclipse“, die beim Speichern der Datei automatisch kompiliert. Das Ergebnis ist die kompilierte Datei ***ServerConnect.class***.

Im nächsten Schritt erfolgt die JNI-Headergenerierung. Diese Headerdatei wird mit dem in C geschriebenen Server im dll-Projekt verwendet, aus dem die dll erstellt wird.

Die Headergenerierung erfolgt mit dem Aufruf

javah –o serverconnect.h ServerConnect

Verwendet man die ServerConnect wie in unserem Fall in einem Paket, muss der vollständige Paketpfad angegeben werden, da sonst die dll nicht funktioniert. Also:

javah –o serverconnect.h Server/ServerConnect

Der generierte Header sieht nun folgendermaßen aus:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class server_ServerConnect */
#ifndef _Included_server_ServerConnect
#define _Included_server_ServerConnect
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     server_ServerConnect
 * Method:    connectWiimotes
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_server_ServerConnect_connectWiimotes
    (JNIEnv *, jclass);
/*
 * Class:     server_ServerConnect
 * Method:    setWindow
 * Signature: (II)V
 */
JNIEXPORT void JNICALL Java_server_ServerConnect_setWindow
    (JNIEnv *, jclass, jint, jint);
/*
 * Class:     server_ServerConnect
 * Method:    getWiimotes
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_server_ServerConnect_getWiimotes
    (JNIEnv *, jclass);
/*
 * Class:     server_ServerConnect
 * Method:    getDataFromWii
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_server_ServerConnect_getDataFromWii
    (JNIEnv *, jclass);
/*
 * Class:     server_ServerConnect
 * Method:    getBUTTON
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_server_ServerConnect_getBUTTON
    (JNIEnv *, jclass);
/*
 * Class:     server_ServerConnect
 * Method:    getCURSOR
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_server_ServerConnect_getCURSOR
    (JNIEnv *, jclass);
```

```
/*
 * Class:     server_ServerConnect
 * Method:    getACC
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_server_ServerConnect_getACC
    (JNIEnv *, jclass);
/*
 * Class:     server_ServerConnect
 * Method:    setIR
 * Signature: (II)V
 */
JNIEXPORT void JNICALL Java_server_ServerConnect_setIR
    (JNIEnv *, jclass, jint, jint);
/*
 * Class:     server_ServerConnect
 * Method:    setACC
 * Signature: (II)V
 */
JNIEXPORT void JNICALL Java_server_ServerConnect_setACC
    (JNIEnv *, jclass, jint, jint);
/*
 * Class:     server_ServerConnect
 * Method:    wait
 * Signature: (II)V
 */
JNIEXPORT void JNICALL Java_server_ServerConnect_wiiWait
    (JNIEnv *, jclass, jint, jint);
/*
 * Class:     server_ServerConnect
 * Method:    setLED
 * Signature: (II)V
 */
JNIEXPORT void JNICALL Java_server_ServerConnect_setLED
    (JNIEnv *, jclass, jint, jint);
/*
 * Class:     server_ServerConnect
 * Method:    setRUMBLE
 * Signature: (II)V
 */
JNIEXPORT void JNICALL Java_server_ServerConnect_setRUMBLE
    (JNIEnv *, jclass, jint, jint);
/*
 * Class:     server_ServerConnect
 * Method:    disconnectWiimotes
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_server_ServerConnect_disconnectWiimotes
    (JNIEnv *, jclass);
#ifdef __cplusplus
}
#endif
#endif
```

Im Server müssen nun noch einige Änderungen vorgenommen werden. Die ursprünglichen Methoden in der Form

void setLED(int i, int leds)

müssen durch die Form

JNICALL Java_server_ServerConnect_setLED
(JNIEnv * pEnv, jclass clazz, jint i, jint leds)

ersetzt werden.

Möchte man einen char eines C-Programms an einen String in Java übergeben, ändert man Folgendes, da C keinen eigentlichen String kennt:

return (*pEnv)->NewStringUTF(pEnv, buttons);

anstelle von

return buttons;

“buttons” ist ein char-Array.

Nach diesen Anpassungen kann man aus dem Projekt die ServerConnect.dll erstellen, um diese in das vorhandene Java-Projekt einzubinden. Das Einbinden erfolgt über den Befehl

System.loadLibrary.

Da dieser Aufruf nur einmal erfolgen darf, muss man die dll statisch einbinden:

```
static{
    System.loadLibrary("ServerConnect");
}
```

Zusätzlich fügt man eine Abfrage hinzu, ob die dll bereits vorhanden ist. Ist dies nicht der Fall, soll eine Exception ausgelöst werden.

Die vollständige Javaklasse sieht nun folgendermaßen aus:

```
package server;
public class ServerConnect {
    public static native void connectWiimotes();
    public static native void setWindow(int width, int height);
    public static native int getWiimotes();
    public static native void getDataFromWii();
    public static native String getBUTTON();
    public static native String getCURSOR();
    public static native String getACC();
    public static native void setIR(int id_wiimote, int status);
    public static native void setACC(int id_wiimote, int status);
    public static native void setLED(int id_wiimote, int led);
    public static native void setRUMBLE(int id_wiimote, int status);
    public static native void disconnectWiimotes();
```

```
    public static boolean init(){
        try{
            System.loadLibrary ("ServerConnect");
            return true;
        }catch (UnsatisfiedLinkError e){
            return false;
        }
    }
}
```

3.3 SIGMAii-API: Implementierung

Abbildung 2.7 zeigt das Klassendiagramm der SIGMAii-API.

Für eine bessere Übersicht, werden die einzelnen Klassen und Interfaces in verschiedene Pakete unterteilt.

- **Control:**
  - WiiCenter()
  - WiiControllable()
- **Client:**
  - WiiClient()
- **Server:**
  - ServerConnect()
- **Event:**
  - WiiAccelerationListener()
  - WiiListener()
  - WiiMotionListener()
  - WiiRotationListener()
  - WiiEvent()

Im weiteren Verlauf gehe ich näher auf die einzelnen Pakete und die beinhalteten Klassen bzw. Interfaces ein.

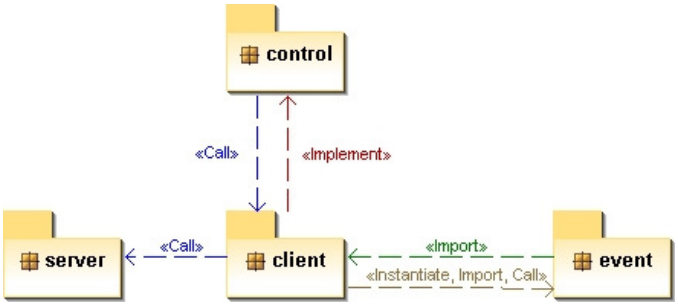

Abbildung 3.3: Abhängigkeiten der einzelnen packages

3.3.1 Control

Das Paket Control beinhaltet die zentrale Klasse **WiiCenter()** und das zentrale Interface **WiiControllable()**.

3.3.1.1 WiiCenter()

Die zentrale Klasse ist als Singleton deklariert, um zu verhindern, dass mehrere Instanzen dieser Klasse eine Verbindung vom **WiiClient()** zum **WiiServer()** bzw. zur ServerConnect.dll herstellen können.

Beim Aufruf der Funktion **WiiConnect()** wird zunächst die Größe des Bildschirms abgefragt, um diese an die **wiiuse**, per Nachricht an den Server, weiterzugeben. Dies ist notwendig für die korrekte Umrechnung der Koordinaten. Ich musste jedoch feststellen, dass, entgegen den Angaben der Programmierer der **wiiuse**-Bibliothek, nicht alle Bildschirmauflösungen unterstützt werden. Deshalb ist es erforderlich, die genutzte Bildschirmgröße auf eine Standardgröße umzurechnen (für 4:3 = 560*420 und für 16:9 = 660*370).

Hier ein Beispiel für eine Bildschirmauflösung von 1440x900:

$$\text{ratio} = \frac{\text{Breite}}{\text{Höhe}} = \frac{1400}{900} \approx 1{,}56$$

```
if ((1.4 < ratio) && (ratio < 1.9)){
    dim_width = 660;
    dim_height = 370;
}else if ((1.1 < ratio) && (ratio < 1.4)){
    dim_width = 560;
    dim_height = 420;
}
```

Ohne oben genannte Umrechnung ließe sich zum Beispiel der Cursor nur auf einem Teil des tatsächlichen Bildschirms bewegen. Der ermittelte Umrechnungsfaktor (**ratio**) wird

später zur Umrechnung der Cursordaten relativ zur tatsächlichen Bildschirmgröße benötigt.

Im nächsten Schritt wird überprüft, ob die **ServerConnect.dll** vorhanden ist oder ob ein Aufbau der Verbindung zum Server über Netzwerk erfolgen soll. Wenn die **ServerConnect.dll** vorhanden ist, wird per Aufruf die Verbindung zu den Wiimotes hergestellt, die Bildschirmgröße übertragen und der Thread für den Empfang der Daten der Wiimote auf dem **WiiClient()** gestartet.

Ist sie nicht vorhanden, wird eine Nachricht an den Server zusammengestellt und per Aufruf an den **WiiClient()** übergeben, der seinerseits die Nachricht an den Server schickt. Zudem wird der Thread zum Empfang der Nachrichten durch den Server gestartet. Damit ist der Verbindungsaufbau zu den Wiimotes abgeschlossen und der Client ist für den Empfang der Daten zuständig.

Das **WiiCenter()** stellt zudem die Funktionen bereit, um die jeweiligen Listener an- bzw. abzumelden. Außerdem sind Einstellungen der Wiimote durchführbar. Entweder direkt durch den Funktionsaufruf über die dll oder aber durch Nachrichten über den Client an den Server. Folgende Funktionsaufrufe sind möglich:

- Abfrage von Beschleunigungs- und Rotationsdaten durch **setACC()**)
- Aktivierung bzw. Deaktivierung des IR-Empfang durch **setIR()**)
- Senden von visuellen bzw. haptischen Informationen an den Anwender, indem man den Status der LEDs setzt (**setLED()**) bzw. den RUMBLE de-/aktiviert (**setRUMBLE()**).
- Einstellung der Pausenzeit zwischen Ein- und Ausschalten des RUMBLEs mittels **wiiWait()**.

3.3.1.2 WiiControllable()

Das zentrale Interface **WiiControllable()** beinhaltet die Methodenaufrufe zum Hinzufügen bzw. Entfernen der einzelnen Listener und den Methodenaufruf **WiiConnect()**, um eine Verbindung zur Wiimote herzustellen.

Eine zentrale Verwaltung dieser Methodenaufrufe hat den Vorteil, dass man dieses Interface durch Vererbung an die einzelnen Interfaces weitergeben kann, um nicht in jedem Interface diese Methodenaufrufe implementieren zu müssen.

Im jeweiligen Programm, das diese Listener verwenden möchte, erfolgt nur einmal der Aufruf

```
WiiCenter wiicenter = new WiiCenter();
```

Die Methodenaufrufe müssen folgendermaßen ergänzt werden:

```
public void WiiConnect() {
    wiicenter.WiiConnect();
}
public void addWiiAccelerationListener(WiiAccelerationListener listener) {
    wiicenter.addWiiAccelerationListener(this);
}
public void removeWiiAccelerationListener(WiiAccelerationListener listener) {
    wiicenter.removeWiiAccelerationListener(this);
}
public void addWiiListener(WiiListener listener) {
    wiicenter.addWiiListener(this);
}
public void removeWiiListener(WiiListener listener) {
    wiicenter.removeWiiListener(this);
}
public void addWiiMotionListener(WiiMotionListener listener) {
    wiicenter.addWiiMotionListener(this);
}
public void removeWiiMotionListener(WiiMotionListener listener) {
    wiicenter.removeWiiMotionListener(this);
}
public void addWiiRotationListener(WiiRotationListener listener) {
    wiicenter.addWiiRotationListener(this);
}
public void removeWiiRotationListener(WiiRotationListener listener) {
    wiicenter.removeWiiRotationListener(this);
}
```

3.3.2 Client/Server

Für die Anbindung an die freie Bibliothek **wiiuse** wurde eine netzwerkbasierte Client-Server-Architektur verwendet. Im Prinzip kann sich dadurch der Rechner, auf dem sich die Serveranwendung befindet, in einem anderen Raum befinden als der Rechner, auf dem die Clientanwendung läuft. Es wäre zum Beispiel nicht notwendig, dass der Rechner, an dem ein Beamer angeschlossen ist, über einen kompatiblen Bluetoothstack verfügt.

Die **ServerConnect.dll** bzw. die Client-Server-Verbindung über **localhost** bieten jedoch auch die Möglichkeit, die Anwendung auf *einem* Rechner laufen zu lassen.

Des Weiteren steht es offen, durch diese Client-Server-Architektur zusätzlich andere Clients mit dem Server durch eine Multicastarchitektur (die bislang nicht implementiert ist) zu verbinden.

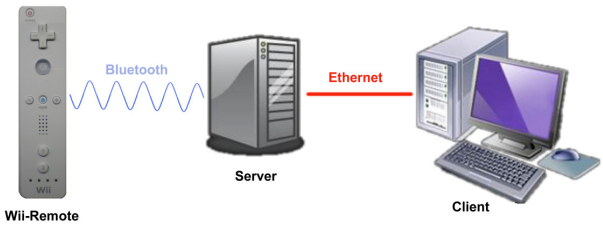

Abbildung 3.4: Client-Server-Architektur mit Wiimote

Die Verbindung zwischen Client und Server kann auf zwei verschiedene Arten erfolgen: Per **UDP** oder per **TCP**

TCP ist verbindungsorientiert, das heißt, es muss zuerst eine Verbindung aufgebaut werden, bevor Daten gesendet werden können. Diese Verbindung ist sehr sicher und TCP kann zum Beispiel verloren gegangene Datenpakete wiederbeschaffen. Möglich ist das, weil der Empfänger den Empfang der Daten innerhalb eines gewissen Zeitraums dem Sender bestätigen muss. Geschieht dies nicht, werden die Daten erneut gesendet. In punkto Sicherheit ist TCP gegenüber UDP klar im Vorteil. Der Nachteil ist die relativ geringe Geschwindigkeit, vor allem bei großen Datenmengen.

UDP dagegen ist nicht verbindungsorientiert. Daten können zwischen Server und Client gesendet werden, ohne dass eine feste Verbindung zwischen den beiden besteht. Mit der Funktion **sendto()** wird genau definiert, zu welchem Empfänger das Paket gesendet wird. UDP kontrolliert allerdings nicht den Empfang. Diese Verbindung wird deshalb vorwiegend dort verwendet, wo vereinzelt Daten verloren gehen können (zum Beispiel bei der Videoübertragung). Der Vorteil dieser Verbindungsart liegt in ihrer hohen Geschwindigkeit und wird daher besonders für Echtzeit-Anwendungen genutzt.

Ich habe mich für die UDP-Verbindung entschieden, da der Eventstrom einen großen Umfang hat und die Cursorbewegungen zur Echtzeit erfolgen sollen. Zu vernachlässigen ist hierbei auch der eventuelle Verlust von einzelnen Datenpaketen. Eine gewünschte Aktion, die aufgrund von Datenverlust nicht ausgeführt wurde, würde der Anwender ein zweites Mal aufrufen. Einen solchen Verlust von Daten konnte ich allerdings bislang noch nicht beobachten.

3.3.2.1 Server

Den Server habe ich in C geschrieben, da er direkt mit der wiiuse-Bibliothek, die ebenfalls in C geschrieben ist, kommunizieren und die erforderlichen Daten abfragen soll.

Der grundsätzliche Vorgang beim Anmelden der Wiimotes am Server funktioniert folgendermaßen:

- Die wartenden Wiimotes (in unserem Fall bis zu vier Stück) werden per

wiiuse_find(wiimotes, MAX_WIIMOTES, 5)

gesucht und per

wiiuse_connect(wiimotes, MAX_WIIMOTES)

verbunden. Wichtig ist, dass die Wiimotes bereits vorher am Betriebssystem angemeldet sind.

- Jede Wiimote erhält ihre visuelle ID, indem die entsprechende LED an der Wiimote eingeschaltet wird. Dies erleichtert später dem Benutzer die Zuordnung seines Controllers zum nummerierten Cursor.
- Jede Wiimote wird kurz über den RUMBLE angesprochen, um dem Benutzer auch haptisch den Anschluss seiner Wiimote mitzuteilen.

Nach der Anmeldung der Wiimotes muss eine Socketverbindung aufgebaut werden.

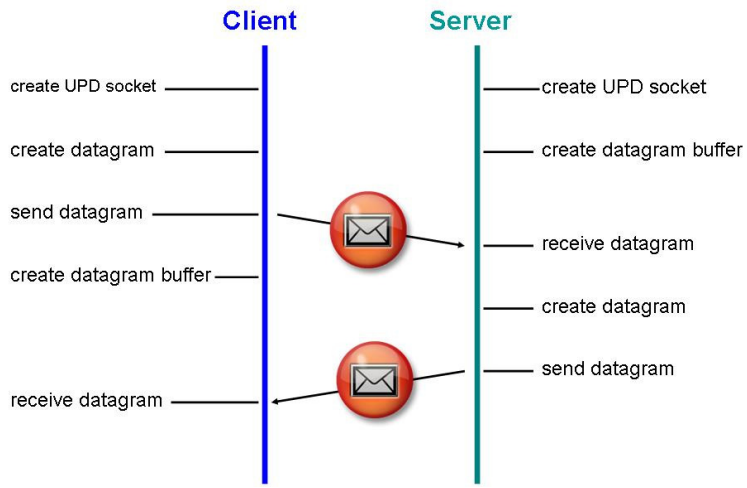

Abbildung 3.5: UDP-basierte Kommunikation

Nach dem erfolgreichen Aufbau einer Socketverbindung, wartet der Server auf die Meldung des Clients, dass er bereit ist, Nachrichten des Servers zu empfangen. Diese erste Nachricht des Clients an den Server enthält unter anderem die Bildschirmauflösung des vom Clientrechner benutzten Monitors und wird der **wiiuse**-Bibliothek per

wiiuse_set_ir_vres(wiimotes[i], width, height)

mitgeteilt. Danach schaltet der Server für jede Wiimote den Empfang für Infrarot-, Bewegungs- und Beschleunigungsdaten ein.

Anschließend sendet der Server seine erste Nachricht an den Client, in der er ihm mitteilt, wie viele Wiimotes verbunden sind. Der Client befindet sich daraufhin im Empfangs- und der Server im Sendemodus.

Der Server ruft nun durch Polling ständig die gewünschten Daten ab, zum Beispiel ob eine Taste betätigt wurde. Ist zusätzlich der IR-Empfang und/oder der Empfang von Beschleunigungsdaten eingeschaltet (was hier der Fall ist), ist auch die Polling-Abfrage nach einem IR-Event oder einem Acceleration-Event aktiviert. Die Abfrage ist in einer While-Schleife verpackt.

Die empfangenen Daten werden, in Form eines Strings, an den Client geschickt und dort zerlegt. Der Client erhält also drei mögliche Datenstrings: IR-Daten, Button-Daten und Acceleration-Daten.

Der Server prüft zudem nach jeder Abfrage, ob der Client eine Nachricht schickt. Ist das nicht der Fall, bricht der Server nach einem Timeout das Warten ab und ruft das Polling erneut auf.

Hat der Client eine Nachricht an den Server geschickt, wird die Nachricht verarbeitet. Der Client kann durch eine Nachricht folgende Änderungen vornehmen:

- Ein-/Ausschalten des IR-Empfangs
- Ein-/Ausschalten des Empfangs von Beschleunigungs- bzw. Bewegungsdaten
- Setzen einzelner LEDs
- RUMBLE de-/aktivieren

Die einzelnen LEDs werden für das Setzen durch folgende Funktion binär codiert:

```
if (led == 0){
    led = WIIMOTE_LED_NONE;
}else if (led == 1){
    led = WIIMOTE_LED_1;
}else if (led == 2){
    led = WIIMOTE_LED_2;
}else if (led == 3){
    led = WIIMOTE_LED_1 + WIIMOTE_LED_2;
}else if (led == 4){
    led = WIIMOTE_LED_3;
}else if (led == 5){
    led = WIIMOTE_LED_3 + WIIMOTE_LED_1;
}else if (led == 6){
    led = WIIMOTE_LED_3 + WIIMOTE_LED_2;
}else if (led == 7){
    led = WIIMOTE_LED_3 + WIIMOTE_LED_2 + WIIMOTE_LED_1;
}else if (led == 8){
    led = WIIMOTE_LED_4;
}else if (led == 9){
    led = WIIMOTE_LED_4 + WIIMOTE_LED_1;
}else if (led == 10){
    led = WIIMOTE_LED_4 + WIIMOTE_LED_2;
}else if (led == 11){
    led = WIIMOTE_LED_4 + WIIMOTE_LED_2 + WIIMOTE_LED_1;
}else if (led == 12){
    led = WIIMOTE_LED_4 + WIIMOTE_LED_3;
}else if (led == 13){
    led = WIIMOTE_LED_4 + WIIMOTE_LED_3 + WIIMOTE_LED_1;
}else if (led == 14){
    led = WIIMOTE_LED_4 + WIIMOTE_LED_3 + WIIMOTE_LED_2;
}else if (led == 15){
    led = WIIMOTE_LED_4 + WIIMOTE_LED_3 + WIIMOTE_LED_2 + WIIMOTE_LED_1;
}
```

Diese Nachrichten haben folgenden Aufbau:

FIRSTMESSAGE:

Parametername	Wertebereich	Zweck
firstmessage	String	„HALLO“
found_wii	int	#gefundener Wiimotes
Antwort an den Client , dass seine Anmeldung akzeptiert wurde und die Anzahl der gefundenen Wiimotes		

Tabelle 3.1: Server-Nachricht: FIRSTMESSAGE

BUTTON:

Parametername	Wertebereich	Zweck
wiimote_id	int	ID der Wiimote, die das Event ausgelöst hat
button	String	Welcher Button wurde gedrückt
length	int	Länge der folgenden Zeichen
status	String	= PRESSED: Button wurde gedrückt = RELEASED: Button wurde losgelassen
Übergabe der Buttons, die gedrückt bzw. losgelassen wurden		

Tabelle 3.2: Server-Nachricht: BUTTON

CURSOR:

Parametername	Wertebereich	Zweck
wiimote_id	int	ID der Wiimote, die das Event ausgelöst hat
ir_visible	int	Sichtbare Infrarotpunkte
length	int	Länge des nachfolgenden Wertes
x	int	x-Koordinate
length	int	Länge des nachfolgenden Wertes
y	int	y-Koordinate
length	int	Länge des nachfolgenden Wertes
z	int	z-Koordinate
IRDOTS	String	Übergabe der Werte der sichtbaren Infrarotpunkte
Übergibt die Infrarotkoordinaten der Wiimote		

Tabelle 3.3: Server-Nachricht: CURSOR

IRDOTS:

Parametername	Wertebereich	Zweck
ir_dots1	int	„1“: Infrarotpunkt 1 wurde erkannt
length	int	Länge des nachfolgenden Wertes
x_ir1	int	x-Koordinate
length	int	Länge des nachfolgenden Wertes
y_ir1	int	y-Koordinate
ir_dots2	int	„2“: Infrarotpunkt 2 wurde erkannt
length	int	Länge des nachfolgenden Wertes
x_ir2	int	x-Koordinate
length	int	Länge des nachfolgenden Wertes
y_ir2	int	y-Koordinate
ir_dots3	int	„3“: Infrarotpunkt 3 wurde erkannt
length	int	Länge des nachfolgenden Wertes
x_ir3	int	x-Koordinate
length	int	Länge des nachfolgenden Wertes
y_ir3	int	y-Koordinate
ir_dots4	int	„4“: Infrarotpunkt 4 wurde erkannt
length	int	Länge des nachfolgenden Wertes
x_ir4	int	x-Koordinate
length	int	Länge des nachfolgenden Wertes
y_ir4	int	y-Koordinate
Übergabe der Werte der sichtbaren Infrarotpunkte. Ist ein IR-Punkt nicht sichtbar, wird kein Wert übergeben.		

Tabelle 3.4: Server-Nachricht: IRDOTS

ACC:

Parametername	Wertebereich	Zweck
wiimote_id	int	ID der Wiimote, die das Event ausgelöst hat
length	int	Länge des nachfolgenden Wertes
roll	double	Drehung um die y-Achse
length	int	Länge des nachfolgenden Wertes
a_roll	double	absolute Drehung um die y-Achse (ungeglättet)
length	int	Länge des nachfolgenden Wertes
pitch	double	Drehung um die x-Achse
length	int	Länge des nachfolgenden Wertes
a_pitch	double	absolute Drehung um die x-Achse (ungeglättet)
length	int	Länge des nachfolgenden Wertes
yaw	double	Drehung um die z-Achse
length	int	Länge des nachfolgenden Wertes
a_x	double	Beschleunigung in +/- x-Richtung
length	int	Länge des nachfolgenden Wertes
a_y	double	Beschleunigung in +/- y-Richtung
length	int	Länge des nachfolgenden Wertes
a_z	double	Beschleunigung in +/- z-Richtung
Die Daten des Bewegungssensors werden übergeben.		

Tabelle 3.5: Server-Nachricht: ACC

Die Verbindung bleibt so lange bestehen, bis sie vom Server oder dem Client beendet wird. Ein Hinzufügen eines weiteren Clients ist bislang in dieser Anwendung nicht vorgesehen, aber möglich.

3.3.2.2 Client

Während der Server bereits gestartet wurde und auf bereite Clients wartet, wird nun der Client gestartet, der seinerseits eine Socketverbindung herstellt.

Anschließend schickt er seine erste Nachricht mit der Bildschirmauflösung an den Server und wechselt vom Sendemodus in den Empfangsmodus um auf die erste Nachricht des Servers zu warten. Erst nach dem Empfang dieser Nachricht, werden weitere Nachrichten vom Server behandelt.

Die Abfrage von Nachrichten erfolgt im Client in einem eigenständigen Thread, der ständig läuft und nur für das Senden von Nachrichten an den Server kurz unterbrochen wird. Der Empfang der Nachrichten wird zudem durch einen Timeout begrenzt. Wenn für einen bestimmten Zeitraum keine Daten empfangen werden, bricht der Empfang ab und es wird eine Testnachricht an den Server gesendet, um Server und Client wieder zu synchronisieren. Dadurch wird ein Deadlock vermieden.

Die empfangenen Stringpakete werden im Client nach ihren Eigenschaften zerlegt und die entsprechenden Listener sowie die Klasse **WiiEvent()** über die Zustandsänderungen informiert:

Im Beispiel der Button-Daten geschieht dies folgendermaßen:

```
if ((get_msg.startsWith("BUTTON")) && (get_msg.length() >= 8)){
    position = (get_msg.indexOf("BUTTON")) + ("BUTTON").length();
    ids = get_msg.substring(position,position+1);
    id = Integer.valueOf(ids).intValue();
    position = position + (ids).length();

    button_pressed = get_msg.substring(position,position+6);
    buttons = Integer.parseInt(get_msg.substring(position+2, position+6), 16);

    position = position + (button_pressed).length();
    coordinates_length = Integer.valueOf(get_msg.substring(position,
                         position+1)).intValue();
    position = position +(get_msg.substring(position,position+1)).length();

    button_status = get_msg.substring(position, position+coordinates_length);

    position = position + (button_status).length();

    /* WiiEvent wird gefeuert */
    if (button_status.equals("PRESSED")){
        WiiEvent w = new WiiEvent(this, BUTTON_PRESSED, id, buttons, found_wii,
                        ir_data[id][1], ir_data[id][2], ir_data[id][3]);
        for (int i = 0; i<wiilistener.size();i++){
            (wiilistener.get(i)).WiiButtonPressed(w);
        }
    }else if (button_status.equals("RELEASED")){
        WiiEvent w = new WiiEvent(this, BUTTON_RELEASED, id, buttons,   found_wii,
                        ir_data[id][1], ir_data[id][2], ir_data[id][3]);
        for (int i = 0; i<wiilistener.size();i++){
            (wiilistener.get(i)).WiiButtonReleased(w);
```

```
            }

            if (but_data[id][2].equals("PRESSED")){
                WiiEvent w2 = new WiiEvent(this, BUTTON_CLICKED, id, buttons, found_wii,
                                ir_data[id][1], ir_data[id][2],
                                ir_data[id][3]);
                for (int i = 0; i<wiilistener.size();i++){
        (wiilistener.get(i)).WiiButtonClicked(w2);
                }
            }//if
        }//else if

    but_data[id][0] = ids;
    but_data[id][1] = button_pressed;
    but_data[id][2] = button_status;

    get_msg = "";
    }//if Button wurde gedrueckt
```

Der WiiClient stellt zudem die Möglichkeit bereit, erforderliche Zustandsänderungen der Wiimote, durch Nachrichten an den WiiServer, auszulösen. Der Programmierer kann folgende Zustände setzen:

- setLED()
- setRumble()
- setIR()
- setACC()

Die Nachrichtenpakete an den Server haben folgende Form:

FIRSTMESSAGE:

Parametername	Wertebereich	Zweck
firstmessage	String	„HALLO“
width	int	bereinigte Breite des Bildschirms (4:3) = 560; (16:9) = 660
height	int	bereinigte Höhe des Bildschirms (4:3) = 420; (16:9) = 370
Verbindungswunsch des Clients als Nachricht an den Server		

Tabelle 3.6: Client-Nachricht: FIRSTMESSAGE

LEDS:

Parametername	Wertebereich	Zweck
ctl	int	ID der Wiimote, an der die Änderungen vorgenommen werden sollen
leds	int	welche LED soll gesetzt werden
Setzen der LED		

Tabelle 3.7: Client-Nachricht: LEDS

RUMBLE:

Parametername	Wertebereich	Zweck
ctl	int	ID der Wiimote, an der die Änderungen vorgenommen werden sollen
status	int	1: Einschalten des RUMBLE 0: Ausschalten des RUMBLE
Aktivierung bzw. Deaktivierung des RUMBLE		

Tabelle 3.8: Client-Nachricht: RUMBLE

IRDOT:

Parametername	Wertebereich	Zweck
ctl	int	ID der Wiimote, an der die Änderungen vorgenommen werden sollen
status	int	1: Einschalten des IR-Empfangs 0: Ausschalten des IR-Empfangs
Aktivierung bzw. Deaktivierung des IR-Empfangs		

Tabelle 3.9: Client-Nachricht: IRDOT

ACCS:

Parametername	Wertebereich	Zweck
ctl	int	ID der Wiimote, an der die Änderungen vorgenommen werden sollen
status	int	1: Einschalten des ACC-Empfangs 0: Ausschalten des ACC-Empfangs
Aktivierung bzw. Deaktivierung des Beschleunigungssensors		

Tabelle 3.10: Client-Nachricht: ACCS

WAIT:

Parametername	Wertebereich	Zweck
lin	int	Wartezeit für Linux-Systeme
win	int	Wartezeit für Windows-Systeme
Wartezeit (z.B. zwischen Ein- und Ausschalten des RUMBLE)		

Tabelle 3.11: Client-Nachricht: WAIT

3.3.3 Interfaces und Event

In diesem Abschnitt beschreibe ich die einzelnen Interfaces und das **WiiEvent**. Wie schon im Kapitel 2.3 erwähnt, gibt es keine separate Klasse Event für die auslösbaren Events. Die Klasse **WiiEvent** ist für die Verwaltung der einzelnen Events zuständig.

3.3.3.1 Interface: WiiAccelerationListener()

```
package event;
import control.WiiControllable;

public interface WiiAccelerationListener extends WiiControllable{
    public void WiiAccelerated (WiiEvent w);
}
```

Klassen, die den **WiiAccelerationListener()** implementiert haben, können die Beschleunigungsdaten der Wiimote abfragen. Hierzu gehören die entsprechen Achsen x, y, z und die ID der Wiimote, die dieses Event ausgelöst hat.

- getCtl() ID der auslösenden Wiimote
- getA_x() Beschleunigung in +/- x-Richtung
- getA_y() Beschleunigung in +/- y-Richtung
- getA_z() Beschleunigung in +/- z-Richtung

3.3.3.2 Interface: WiiListener()

```
package event;
import control.WiiControllable;

public interface WiiListener extends WiiControllable{
    public void WiiButtonPressed(WiiEvent w);
    public void WiiButtonReleased(WiiEvent w);
    public void WiiButtonClicked(WiiEvent w);
    public void WiiButtonEntered(WiiEvent w);
    public void WiiButtonExited(WiiEvent w);
}
```

Durch die Implementierung des Interfaces **WiiListener()** werden die angehängten Listener über die Zustände der Buttons informiert. Eine Information erfolgt, sobald eine Taste betätigt wurde. Die Listener erhalten neben den Button-Zuständen (**PRESSED**, **RELEASED**, **CLICKED**) auch die Information, welche Wiimote dieses Event ausgelöst hat. Zudem erhält man die aktuelle Position (xyz-Koordinaten) der Wiimote, an der das Event ausgelöst wurde. Vorraussetzung ist, dass der Infrarotempfang der Wiimote aktiviert ist und sich die Wiimote in Sichtweite der Sensorbar befindet.

Die Verwendung des **WiiButton()** ermöglicht die Abfrage, ob der Cursor den Button betritt (**WiiButtonEntered()**) oder den Button verlässt (**WiiButtonExited()**). Diese Methoden existieren nur für den **WiiButton**, können aber für jede selbst definierte Komponente ergänzt werden. Dies zeige ich am Beispiel des **WiiButton**:

```
public class WiiButton extends JButton
                        implements WiiMotionListener{

    private static WiiCenter wiic;
    private WiiEvent wiievent;
    private boolean entered = false;
    private Component component;

    public WiiButton(WiiCenter wiic){
        WiiButton.wiic= wiic;
        this.addWiiMotionListener(this);
    }

    public void WiiDragged(WiiEvent w){}

    public void WiiMoved(WiiEvent w){
        component = this.getComponentAt(w.getX(w.getCtl()), w.getY(w.getCtl()));

        if ((component != null) && (!entered)){
            entered = true;
            /* WiiEvent wird gefeuert */
            wiievent = new WiiEvent(wiic.getWiiClient(), WiiClient.BUTTON_ENTERED,
                            w.getCtl(), component);
            for (int i=0;i<wiic.getWiiClient().wiilistener.size();i++){
```

```
                (wiic.getWiiClient().wiilistener.get(i)).WiiButtonEntered(wiievent);
            }
        }else if((component == null) && (entered)){
            entered = false;
            /* WiiEvent wird gefeuert */
            wiievent = new WiiEvent(wiic.getWiiClient(), WiiClient.BUTTON_EXITED,
                            w.getCtl(),component);
            for (int i=0;i<wiic.getWiiClient().wiilistener.size();i++){
                (wiic.getWiiClient().wiilistener.get(i)).WiiButtonExited(wiievent);
            }
        }
    }

    public void WiiConnect(){
        wiic.WiiConnect();
    }

    /**
     * Listener werden hinzugefuegt
     */
    public void addWiiMotionListener(WiiMotionListener listener){
        wiic.addWiiMotionListener(this);
    }

    public void removeWiiMotionListener(WiiMotionListener listener){
        wiic.removeWiiMotionListener(this);
    }

    public void addWiiAccelerationListener(WiiAccelerationListener listener){}
    public void addWiiListener(WiiListener listener){}
    public void addWiiRotationListener(WiiRotationListener listener){}
    public void removeWiiAccelerationListener(WiiAccelerationListener listener){}
    public void removeWiiListener(WiiListener listener){}
    public void removeWiiRotationListener(WiiRotationListener listener){}
}
```

Somit haben die Interfaces **WiiListener()** und **MouseListener()** weitestgehend den gleichen Funktionsumfang:

- getCtl() ID der auslösenden Wiimote
- getWiiButton() Button, der gedrückt wurde
- getX() x-Koordinate, an der das Ereignis aufgetreten ist
- getY() y-Koordinate, an der das Ereignis aufgetreten ist
- getZ() z-Koordinate, an der das Ereignis aufgetreten ist
- getComponent() Komponente, die betreten oder verlassen wurde (nur aktiv, wenn, wie im Beispiel, separat ergänzt)

3.3.3.3 Interface: WiiMotionListener()

```
package event;
import control.WiiControllable;

public interface WiiMotionListener extends WiiControllable{
    public void WiiMoved (WiiEvent w);
    public void WiiDragged (WiiEvent w);
}
```

Klassen, die den **WiiMotionListener()** implementiert haben, werden über die Bewegung der Wiimote, in Abhängigkeit der Sichtbarkeit der Infrarotpunkte, informiert. Dieses Interface ist ähnlich dem **MouseMotionListener()**-Interface. **WiiMoved()** entspricht dabei **MouseMoved()** und **WiiDragged()** entspricht **MouseDragged()**.
Abfragen lassen sich die xyz–Koordinaten der Infrarotdaten und die ID der Wiimote, die dieses Ereignis ausgelöst hat.

- getCtl() ID der auslösenden Wiimote
- getX() x-Koordinate, an der das Ereignis aufgetreten ist
- getY() y-Koordinate, an der das Ereignis aufgetreten ist
- getZ() z-Koordinate, an der das Ereignis aufgetreten ist

3.3.3.4 Interface: WiiRotationListener()

```
package event;
import control.WiiControllable;

public interface WiiRotationListener extends WiiControllable{
    public void WiiRotated (WiiEvent w);
}
```

Die Implementierung des **WiiRotationListener()** ermöglicht es, den Klassen Drehungen der Wiimote mitzuteilen. Möglich sind Drehungen um die x-, y- und z-Achse. Hierbei liegen die Werte für **ROLL** und **PITCH** zwischen -180° und +180° und **YAW** zwischen -26° und +26°. **YAW** ist nur verfügbar, wenn der IR-Empfang aktiviert ist und IR-Punkte sichtbar sind. Ansonsten ist **YAW** = 0. Des Weiteren ist eine Abfrage der absoluten Drehung um die x- bzw. y-Achse, für **ROLL** und **PITCH** möglich: **A_ROLL** und **A_PITCH**. Dies sind die absoluten, nicht durch einen Algorithmus geglätteten, Werte. Außerdem wird, wie bei jedem auftretenden Event, die ID der auslösenden Wiimote übermittelt:

- getCtl() ID der auslösenden Wiimote
- getRoll() Drehung um die y-Achse
- getA_Roll() absolute Drehung um die y-Achse
- getPitch() Drehung um die x-Achse
- getA_Pitch() absolute Drehung um die x-Achse

- getYaw() Drehung um die z-Achse

3.3.3.5 WiiEvent()

Das **WiiEvent()** wird ausgelöst und an die entsprechenden Listener weitergegeben, sobald ein Ereignis an der Wiimote auftritt. Hierzu hat die Klasse **WiiEvent()** verschiedene Konstruktoren, die bei dem jeweiligen Ereignis aufgerufen werden. Die Konstruktoren „unterteilen“ das **WiiEvent()** praktisch in die einzelnen Events:

- ButtonEvent()
- IREvent()
- ACCEvent()
- ComponentEvent()

ButtonEvent():

Dem Event werden die Parameter

- id Status des Events
- ctl ID der auslösenden Wiimote
- WiiButton Kennung des betätigten Buttons
- found_wii #der gefundenen Wiimotes
- x x-Koordinate, an der das Ereignis ausgelöst wurde
- y y-Koordinate, an der das Ereignis ausgelöst wurde
- z z-Koordinate, an der das Ereignis ausgelöst wurde

übergeben. Die xyz-Werte, werden nur übergeben, wenn der IR-Empfang der Wiimote aktiviert ist. Ansonsten sind die Werte standardmäßig 0. Die übergebenen Parameter werden, zugehörig zur ID der Wiimote, in einem Array gespeichert. Somit ist jederzeit für jede Wiimote das entsprechende Ereignis abrufbar.

1. IREvent():

Dem Event werden die Parameter

- id Status des Events
- ctl ID der auslösenden Wiimote
- found_wii #der gefundenen Wiimotes
- x x-Koordinate, an der das Ereignis ausgelöst wurde
- y y-Koordinate, an der das Ereignis ausgelöst wurde
- z z-Koordinate, an der das Ereignis ausgelöst wurde

übergeben und zugehörig zur ID der Wiimote in einem Array gespeichert. Bei diesen xyz-Koordinaten handelt es sich um, relativ zur Position des Bildschirms, *berechnete* Werte.

Im folgenden **2. IREvent()** werden die Parameter der einzelnen IR-Punkte übergeben:

- id Status des Events
- ctl ID der auslösenden Wiimote
- found_wii #der gefundenen Wiimotes
- ir_visible #der sichtbaren IR-Punkte
- x_ir1 IR1: x-Koordinate, an der das Ereignis ausgelöst wurde
- y_ir1 IR1: y-Koordinate, an der das Ereignis ausgelöst wurde
- x_ir2 IR2: x-Koordinate, an der das Ereignis ausgelöst wurde
- y_ir2 IR2: y-Koordinate, an der das Ereignis ausgelöst wurde
- x_ir3 IR3: x-Koordinate, an der das Ereignis ausgelöst wurde
- y_ir3 IR3: y-Koordinate, an der das Ereignis ausgelöst wurde
- x_ir4 IR4: x-Koordinate, an der das Ereignis ausgelöst wurde
- y_ir4 IR4: y-Koordinate, an der das Ereignis ausgelöst wurde

Diese werden zugehörig zur ID der Wiimote in einem Array gespeichert.

In einem **3.IREvent()** können die Parameter der ersten beiden zusammen übergeben werden.

ACCEvent():

Dieses Event tritt auf, wenn eine Bewegung der Wiimote erfolgt und die Beschleunigungssensorabfrage aktiviert ist. Als Parameter wird folgendes übergeben:

- ctl ID der auslösenden Wiimote
- found_wii #der gefundenen Wiimotes
- roll Drehung um y-Achse
- a_roll absolute Drehung um y-Achse
- pitch Drehung um x-Achse
- a_pitch absolute Drehung um x-Achse
- yaw Drehung um z-Achse
- a_x Beschleunigung in +/- x-Richtung
- a_y Beschleunigung in +/- y-Richtung
- a_z Beschleunigung in +/- z-Richtung

und in das entsprechende Array geschrieben.

ComponentEvent():

Das **ComponentEvent()** tritt auf, wenn eine Komponente betreten bzw. verlassen wird. Dieses Event kann nicht direkt im Client übergeben werden, da dieses dort nicht auftritt. Dazu muss eine Komponente wie im Beispiel von Kapitel 3.3.3.2 neu definiert werden. Wird bei einer IR-Bewegung **WiiMoved** die Komponente betreten bzw. verlassen, wird dieses Event an die entsprechenden Listener übergeben („gefeuert“).
Als Parameter werden folgende Werte übergeben:

- id Status des Events
- ctl ID der auslösenden Wiimote
- component Komponente, an der das Ereignis auftrat

Die Werte werden ebenfalls in ein entsprechendes Array geschrieben.

Kapitel IV – Anwendung

Dieses Kapitel behandelt die implementierte Anwendung **SIGMAii-SocialNet**.

Es handelt sich hierbei um eine Beispielanwendung, die den Einsatz der beschriebenen API und die Leistungsfähigkeit meiner Implementierung erstmals nachweist. Mit **SIGMAii-SocialNet** ist es möglich, kollaborativ Netzwerke zu konstruieren. Eine beliebige Anzahl von Objekten können in einem Fenster frei platziert, mit Bildern versehen und Kanten zwischen diesen Objekten hinzugefügt werden.

Ein Einsatzgebiet dieser Anwendung wäre zum Beispiel die Verwendung in der Software-Entwicklung. Hier könnten CRC-Karten eingescannt und kollaborativ in einem Netzwerk dargestellt werden.

CRC steht für „Class, Responsibilities and Collaborators“. CRC-Karten wurden von Cunningham und Beck Ende der 80er Jahre entwickelt und in ihrem Artikel „A Laboratory for Teaching Object-Oriented Thinking“ das erste Mal erwähnt. Sie dienen der Simulierung objektorientierter Softwareentwicklung [Cunningham 89] und werden als Hilfsmittel zur objektorientierten Analyse eingesetzt [Diehl 08].

Ich erläutere in diesem Kapitel die während der Erstellung der Anwendung aufgetretenen Probleme und meine Lösungen.

Abbildung 4.1 zeigt die Anwendung (**mywindow**, **gui** und **net.javajeff.jtwain)** und deren Anbindung an die SIGMAii-API (**control**, **client**, **server**, **event**).

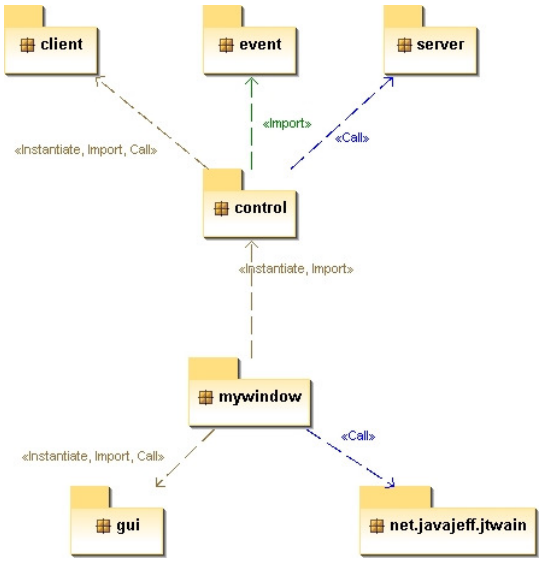

Abbildung 4.1: Paketübersicht der Anwendung

Anwendung

Diese Anwendung ist ein in Java geschriebenes Programm, das die Möglichkeiten der SIGMAii-API verdeutlicht. Die Funktionalität des Programms besteht darin, eine Art Stammbaum, allgemeiner *Soziale Netze*, zu verwalten.

SIGMAii-SocialNet ermöglicht es unter anderem, einem Hauptfenster neue Komponenten hinzuzufügen und in diese Bilder von der Festplatte oder von einem Scanner zu laden. Dass es unter Java keine Scanneranbindung gibt, stellte ein Problem dar. Auf die Lösung dieses Problems gehe ich im weiteren Verlauf genauer ein.

Die Komponenten sind in Position, Größe und Ausrichtung veränderbar. Zudem lassen sich Relationen zwischen einzelnen Komponenten durch Kanten hinzufügen und wieder löschen.

Da soziale Verbindungen meist nicht in einer Baumstruktur darstellbar sind, war diese Darstellungsart auch nicht gefordert. Eine Abfrage, ob ein Knoten zum Beispiel bereits eine eingehende Kante hat, war somit nicht nötig.

Jede Komponente ist mit einem Text versehbar. Zudem passt sich dieses Textfeld, Änderungen an Lage und Position der zugehörigen Komponente automatisch an. Die Eingabe des Textes kann mit der Wiimote erfolgen und ist nicht an eine Tastatur gebunden. Ebenso ist eine *undo*-Funktion für den jeweils letzten durchgeführten Befehl eines Cursors enthalten. Drucken, Speichern und Laden des Arbeitsfensters ist ebenfalls implementiert.

Die Anwendungsfunktionen im Überblick:

- Komponenten hinzufügen mit Bild aus Datei oder vom Scanner und wieder entfernen
- Position, Größe und Ausrichtung der Komponenten ändern
- Text der Komponente hinzufügen und ändern
- Verbindungslinien (Kanten) zwischen zwei Komponenten ziehen
- Letzte Änderung (cursorabhängig) rückgängig machen
- Speichern, Laden, Drucken des Arbeitsfensters

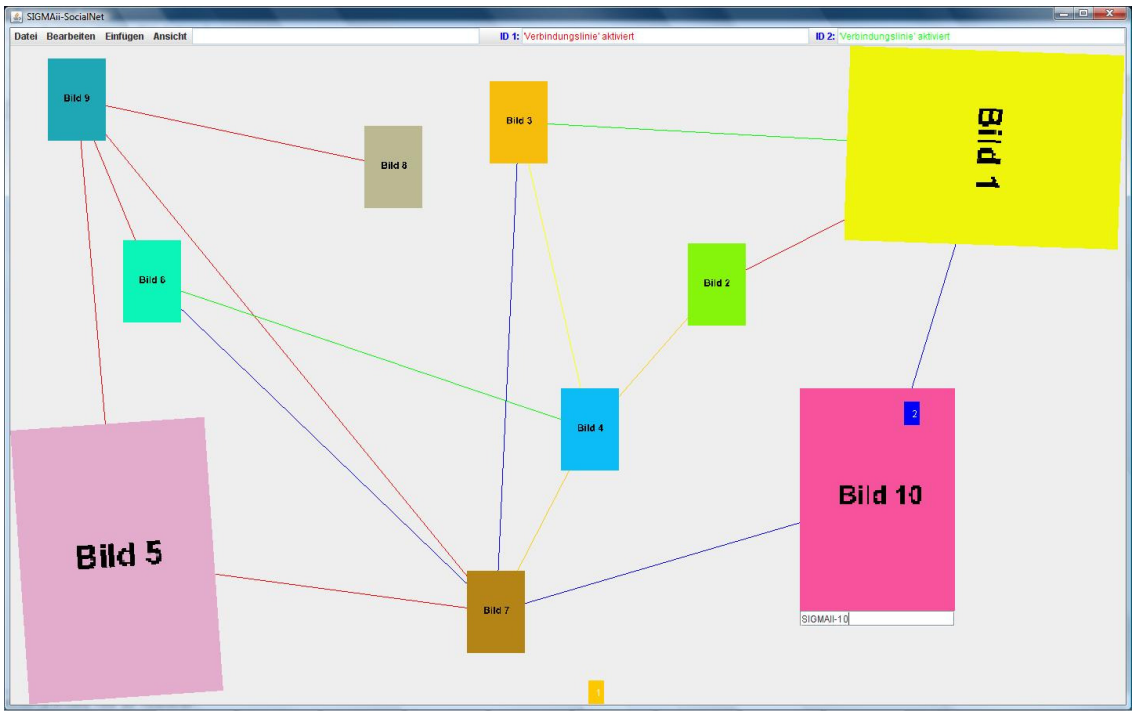

Abbildung 4.2: „SocialNet“ mit Komponenten

Nachdem der Funktionsumfang der Anwendung feststand, stellte sich mir die Frage, welche Funktionen ich welchen Tasten der Wiimote zuordnen sollte. Um alle Funktionen zuzuordnen, hätte ich teilweise erweiterte und für den Anwender schwierig zu merkende Tastenkombinationen einbauen müssen. Um dies zu verhindern, verzichtete ich auf Kombinationen für die Funktionen „Beenden“, „Speichern“, „Laden“ und „Drucken“. Die nachfolgende Tabelle zeigt die umgesetzten Tastenkombinationen und deren Funktionen:

Buttonkombination	Funktion
A	Komponente markieren
A + B	Komponente bewegen
B + PLUS	Komponente hinzufügen mit Bild von Scanner
B + HOCH	Komponente in 1° Schritten nach rechts drehen
B + RUNTER	Komponente in 1° Schritten nach links drehen
B + RECHTS	Komponente um 90° nach rechts drehen
B + LINKS	Komponente um 90° nach links drehen
B + HOME	Letzte Änderung rückgängig machen
1	Text zu markierter Komponente hinzufügen
1 dann HOCH	Buchstabenauswahl aufwärts blättern
1 dann RUNTER	Buchstabenauswahl abwärts blättern
1 dann RECHTS	Buchstabenauswahl wählen
1 dann LINKS	Buchstabenauswahl löschen
2	Linieneingabe aktivieren
HOCH	Komponente schrittweise vergrößern
RUNTER	Komponente schrittweise verkleinern
RECHTS	Komponente auf kompletten Bildschirm vergrößern
LINKS	Komponente auf festgelegte Ursprungsgröße ändern
HOME	Maus simulieren
PLUS (+)	Komponente hinzufügen mit Bild aus Datei
MINUS (–)	markierte Komponente entfernen

Tabelle 4.1: Wiimote-Buttonkombinationen und deren zugeordnete Funktionen

Abbildung 4.1 zeigt die einzelnen Pakete der Anwendung. Im weiteren Verlauf des Kapitels gehe ich auf die Pakete

- gui und
- net.javajeff.jtwain

genauer ein.

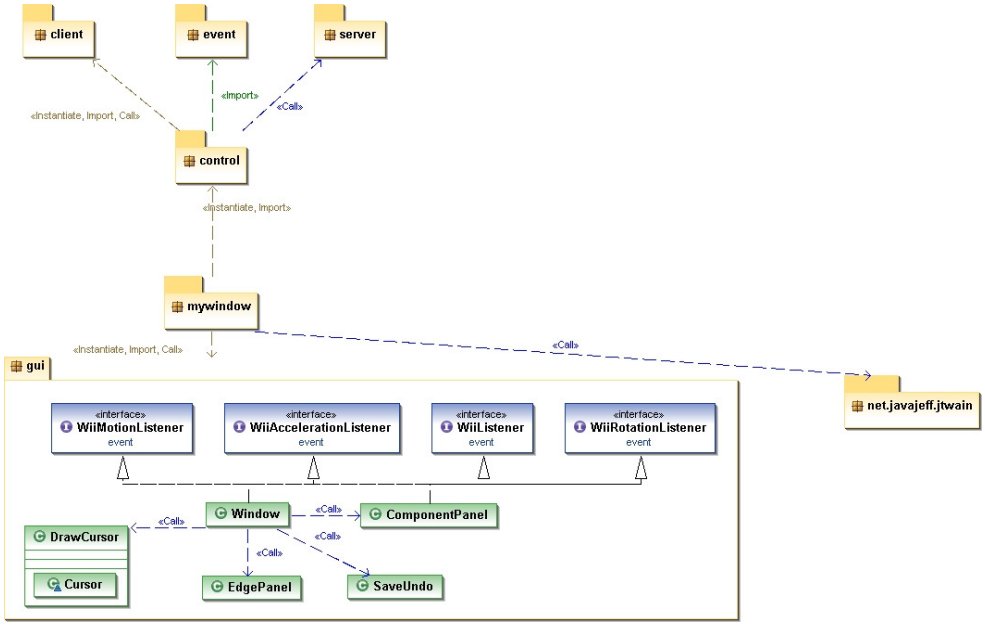

Abbildung 4.3: Das Paket gui

4.1 gui: Window()

Die einzelnen Funktionen sind auch über ein Menü des Hauptfensters aufrufbar. Hier sind alle Funktionen erreichbar; entweder mit der Maus oder mit einer, in den Mausmodus versetzen, Wiimote. Dieser Mausmodus wird benötigt, da man die Wiimote nicht ohne weiteres an die vorgefertigten Menüs bzw. die FileChooser von Java und das Scanfenster anhängen kann. Eine in den Mausmodus versetzte Wiimote bewegt den Mauscursor durch die Klasse **Robot()** [SUN: java.awt.Robot].

Mit der Klasse **Robot()** lassen sich Tastatur- und Mausereignisse (zum Beispiel die Cursorbewegung) erzeugen. Die Maus lässt sich so simulieren. Ich verwende folgende Methoden:

- mouseMove()
- mousePress()
- mouseRelease()

Eine Bewegung der Wiimote im Mausmodus ruft die Methode

mouseMove()

auf. Wird zum Beispiel die Taste A gedrückt, die in unserem Fall der linken Taste der Maus entspricht, wird die Methode

mousePress()

aufgerufen. Äquivalent dazu im Fall des Loslassens der Taste die Methode

mouseRelease().

Das Hauptfenster ist auf den kompletten verfügbaren Bildschirm angepasst. Abbildung 4.3 zeigt die Einordnung der Klasse Window.

4.2 gui: DrawCursor()

Da man im Betriebssystem normalerweise nur über *einen* Mauscursor verfügt, muss man sich für die Wiimote eigene Cursor definieren. Diese sind dann betriebssystemunabhängig.

Dies geschieht durch die Klasse **DrawCursor()**. Hierzu wird für jeden benötigten Cursor ein Fenster erzeugt. Dieses Fenster erhält eine ID und wird so der jeweiligen Wiimote angehängt. Zum Bewegen des Cursors muss nun folgende Ergänzung vorgenommen werden:

```
public void WiiDragged(WiiEvent w) {
    cursors.getCursor(w.getCtl()).setLocation(w.getX(w.getCtl()), w.getY(w.getCtl()));
}
public void WiiMoved(WiiEvent w) {
    cursors.getCursor(w.getCtl()).setLocation(w.getX(w.getCtl()), w.getY(w.getCtl()));
}
```

Der Cursor wird dadurch mit jeder Bewegung der Wiimote (vorausgesetzt die IR-Punkte sind sichtbar und die Abfrage ist aktiviert) an die entsprechende Position auf dem Bildschirm bewegt; entsprechend dem Mauscursor durch die Bewegung der Maus.

4.3 gui: ComponentPanel(): Komponenten hinzufügen

Eine neue Komponente kann entweder über die Wiimote oder direkt über das Menü des Hauptfensters hinzugefügt werden. Zwei Möglichkeiten stehen zur Auswahl. Zum einen kann der Komponente ein Bild aus einer Datei über den Funktionsaufruf

insert_graphic_fromfile(int id_cursor)

hinzugefügt werden. Als Parameter wird die ID des Cursors übergeben. Im Falle der Nutzung der Maus, ist die Cursor-ID = 0.
Zum anderen kann ein Bild über den Funktionsaufruf

insert_graphic_fromscanner(int id_cursor)

eingescannt werden. Auch hier wird als Parameter die ID des Cursors übergeben.

Abbildung 4.4: Menüpunkt: Grafik

4.3.1 Bild aus Datei

Die erste Variante war einfacher zu implementieren, da Java diese Funktion bereitstellt. Das Hinzufügen einer Bild-Datei in eine Komponente, kann entweder über das Menü (Abbildung 4.4) erfolgen oder über die Taste „**+**“ der Wiimote. Zum Öffnen einer Datei wird der **JFileChooser** eingebunden. Diese Komponente dient zur Auswahl von Dateien. Durch die Benutzung des Filters **setFileFilter()** kann die Auswahl der Dateiformate eingeschränkt werden. In unserem Fall ist nur die Verwendung von JPG- und GIF-Dateien zulässig. Dies wird durch Folgendes erreicht:

```
filechooser.setFileFilter(new FileFilter(){
    @Override
    public boolean accept(final File f){
        return f.isDirectory()
            || f.getName().toLowerCase().endsWith( ".jpg" )
            || f.getName().toLowerCase().endsWith( ".gif" );
    }

    @Override
    public String getDescription(){
        return "*.jpg;*.gif";
    }
});
```

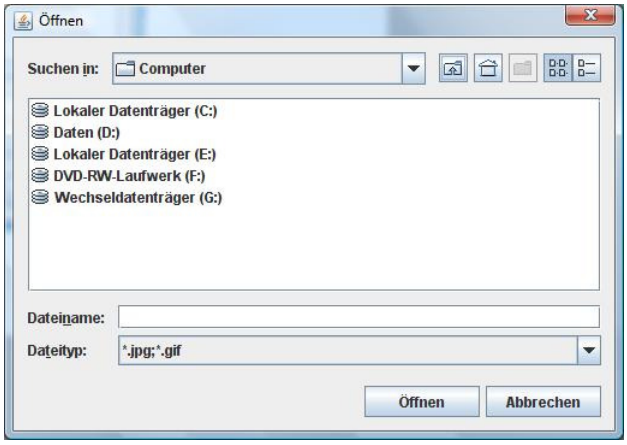

Abbildung 4.5: JFileChooser

Auf die zweite Variante, nämlich das Einfügen eines Bildes über den Scanner, gehe ich in Kapitel 4.3.2 genauer ein.

Ist das Bild geladen, wird durch den Funktionsaufruf

```
components = new ComponentPanel(id_component, image, wiic, dim_win_x, dim_win_y,
                                dim_win_width, dim_win_height);
```

eine neue Komponente mit dem ausgewählten Bild erzeugt und in die obere linke Ecke des Bildschirms (Koordinate: 0,0) positioniert. Die Parameter, die dabei übertragen werden, sind:

- id_component: ID der Komponente
- image: geladenes Bild
- wiic: initialisiertes WiiCenter() zum Hinzufügen bzw. Entfernen der Listener
- dim_win_x: x-Koordinate des Zeichenbereichs
- dim_win_y: y-Koordinate des Zeichenbereichs
- dim_win_width: Breite des Zeichenbereichs
- dim_win_height: Höhe des Zeichenbereichs

Durch Hinzufügen der Komponente zum Fenster wird diese auch sichtbar. Anschließend müssen noch die entsprechenden Listener hinzugefügt werden, damit die Komponente auf die Wiimote reagiert.

4.3.2 Scannen

Offiziell gibt es unter Java keine Scannerunterstützung. Möchte man dennoch einen Scanner unter Java verwenden, ohne auf eines der vielen kommerziellen Programme zurückgreifen zu müssen, bietet der etwas umständlichere Weg über die JNI (Java Native Interface) eine Lösung. Hierbei musste ich beachten, dass es unterschiedliche Treiber für

die verschiedenen Betriebssysteme gibt. Unter Windows gibt es den TWAIN-Treiber und unter Mac, Linux, BSD und Unix den SANE-Treiber. Das macht eine plattformübergreifende Lösung schwieriger. Man sollte sich vorab über das verwendete Betriebssystem im Klaren sein, da der zu programmierende Scannertreiber betriebssystemabhängig ist.

Bei meiner Umsetzung über JNI bot sich die Möglichkeit, zumindest einen Teil des benötigten Quellcodes frei zu erwerben, durch die Verwendung des Programms von Jeff Friesen. Auf seiner Seite `http://javajeff.mb.ca` bietet er ein Paket inklusive eines in C++ geschriebenen Programms zur Scanneranbindung zum download an [Daum 07, S.311-313]. Zusätzlich beinhaltet es ein in Java geschriebenes Beispielprogramm.

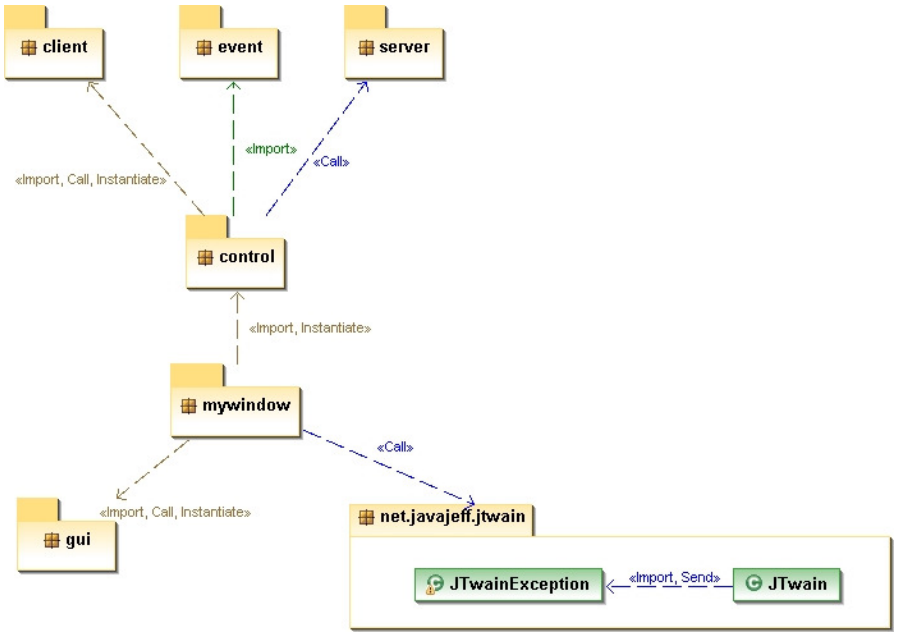

Abbildung 4.6: Das Paket net.javajeff.jtwain

Die Javaklasse für die JNI-Headergenerierung ist bereits vorhanden und aus dieser muss man die Headerdatei **jtwain.h** generieren. Zu beachten ist dabei die genaue Angabe des Pfades des Paketes, in der die Javaklasse hinterlegt ist. Anschließend erstellt man aus den vorhandenen Dateien **jtwain.cpp**, **jtwain.h** und **twain.h** eine dll-Datei. Diese dll-Datei wird dann im Projekt eingebunden oder in den entsprechenden Ordner (in meinem Fall **c:\windows**) kopiert. Dort befindet sich auch die **TWAIN32.dll**.

Wie schon erwähnt enthält das Paket von Jeff Friesen eine Beispielanwendung, mit der sich die Scan-Funktion testen lässt.

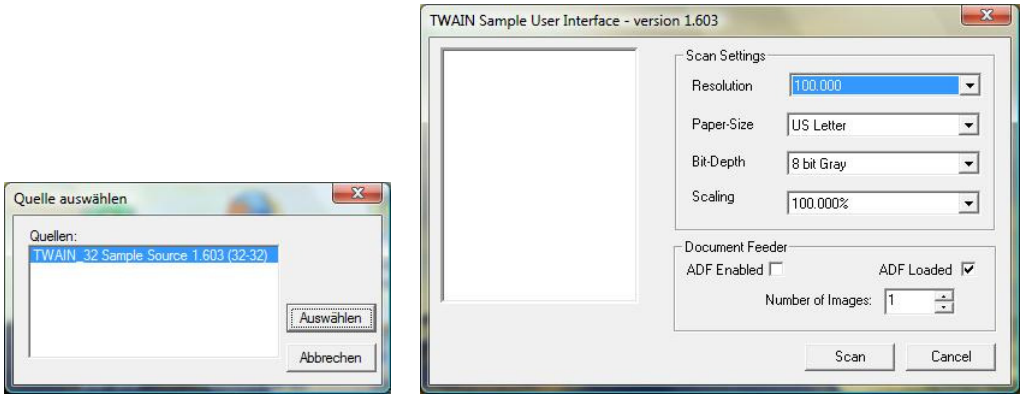

Abbildung 4.7: Scannen

4.4 Komponente verschieben

Für das Verschieben einer Komponente muss die Komponente der Funktion bekannt sein. Es gibt zwar die Möglichkeit, die Komponente an einer bestimmten Position mittels

getComponentAt(int x, int y),

zu ermitteln, was aber in meiner Anwendung nicht die gewünschten Resultate brachte. Zum Beispiel lieferte mir diese Funktion lediglich die Information für die letzte hinzugefügte Komponente. Ein Zugriff auf vorherige Komponenten war durch diesen Funktionsaufruf nicht möglich. Ich entschloss mich, eine eigene Funktion für die Ermittlung der ID der Komponente zu programmieren.

In dieser werden die Informationen der Komponente wie Start- und Endpunkt auf der x-Achse und Start- und Endpunkt auf der y-Achse zu Anfang in ein zweidimensionales Array geschrieben, das für die Verwaltung der Komponenteninformationen verwendet wird:

```
/* x_min: linke x-Koordinate des Image */
component[id_component][8] = dim_win_left + dim_win_x;
/* x_max: rechte x-Koordinate des Image */
component[id_component][9] = width_out + dim_win_left + dim_win_x;
/* y_min: obere y-Koordinate des Image */
component[id_component][10] = dim_win_top + dim_win_menue + dim_win_y;
/* y_max: untere y-Koordinate des Image */
component[id_component][11] = height_out + dim_win_top + dim_win_menue + dim_win_y;
```

Dabei ist **id_component** die ID der Komponente, die an die erste Position der ersten Spalte des Arrays geschrieben wird:

***component*[id_component][0] = id_component;**

Der Funktion

getComponentIDAt()

werden durch den Funktionsaufruf

***id_components*[id_cursor][0] =**
components.getComponentIdAt(f.getXOnScreen(), f.getYOnScreen());

durch die Maus und

***id_components*[id_cursor][0] =**
components.getComponentIdAt(w.getX(id_cursor), w.getY(id_cursor));

durch die Wiimote die Parameter x und y der aktuellen Position des Cursors auf dem Bildschirm übergeben. Durch folgende Abfrage wird dann überprüft, welche Komponente sich an dieser Position befindet und die ID der Komponente wird zurückgegeben:

```
for(int i = 0; i <= 1023; i++){
    if ((component[i][8] <= x)  && (x <= component[i][9])
                                && (component[i][10] <= y)
                                && (y <= component[i][11])){
        id_component = component[i][0];
        i = 1023;
    }else{
        id_component = 0;
    }
}
return id_component;
```

Befindet sich keine Komponente an der überprüften Position, wird 0 als ID der Komponente zurückgeliefert. Der Cursor befindet sich dann außerhalb aller Komponenten.

Wurde eine Komponente gefunden, ist deren ID jetzt bekannt. Zum Verschieben der Komponente löst man durch Drücken der entsprechenden Tastenkombination folgenden Aufruf aus:

componentpanel.get(*id_components*[id_cursor][0]).setLocation(point);

Die Komponente wird mit dem Cursor über den Bildschirm zur gewünschten Stelle verschoben. Die Variable **point** ist der neu berechnete Startpunkt der Komponente. Dieser wird durch den Funktionsaufruf

point = components.coordinates(x, y, *id_components*[id_cursor][0]);

im **ComponentPanel()** folgendermaßen berechnet:

```
width_x = dim_win_width - dim_win_left - dim_win_right;
height_y = dim_win_height - dim_win_bottom - dim_win_top - dim_win_menue;

width = (int) component[id_component][3];
height = (int) component[id_component][4];

if((cursor_x >= dim_win_width)  || (cursor_y >= dim_win_height)
                                || (cursor_x < 0)
                                || (cursor_y < 0)){
    x_new = (int) component[id_component][8];
    y_new = (int) component[id_component][10];
}else{
```

```
        x_new = (int) (cursor_x - distance.getX());
        y_new = (int) (cursor_y - distance.getY());
    }

    if (x_new <= 0){
        x_new = 0;
    }
    if ((x_new + width) > (width_x)){
        x_new = width_x - width;
    }
    if (y_new <= 0){
        y_new = 0;
    }
    if ((y_new + height) > (height_y)){
        y_new = height_y - height;
    }

    x_new = x_new + dim_win_left + dim_win_x;
    y_new = y_new + dim_win_top + dim_win_menue + dim_win_y;

    component[id_component][8] = x_new;
    component[id_component][9] = x_new + width;
    component[id_component][10] = y_new;
    component[id_component][11] = y_new + height;

    return new Point(x_new, y_new);
```

Hierbei wird überprüft, ob die neue Position der Komponente innerhalb des Anwendungsfensters liegt. Würde sich die Komponente aus dem Fenster hinausbewegen, werden die neuen Werte auf eine Position innerhalb des Fensters begrenzt. Die neuen Werte für Start- und Endpunkt auf der x-Achse und Start- und Endpunkt auf der y-Achse werden wieder im Array gespeichert und die alten Positionswerte dieser ID dabei überschrieben.

Auch jede andere Änderung einer Komponente, wie zum Beispiel das Drehen, hat eine Aktualisierung der Array-Werte zur Folge.

4.5 Drehung der Komponenten

Bei der Drehung einer Komponente musste ich die daraus resultierende Änderung der Komponentengröße, wie auch die korrekte Darstellung des gedrehten Bildes der Komponente, berücksichtigen.

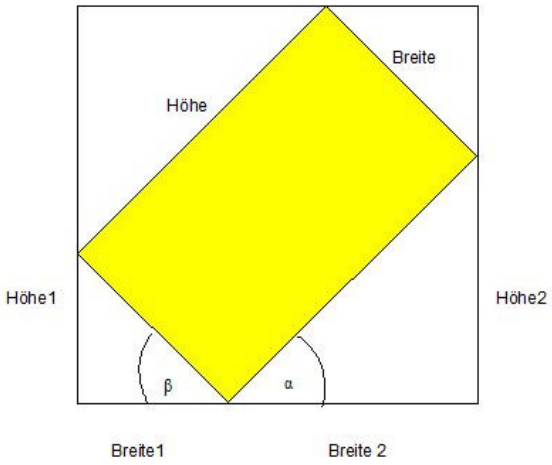

Abbildung 4.8: Berechnung der neuen Komponentengröße in Abhängigkeit des Drehwinkels

Bei einer Drehung um 0°, 180° oder 360° bleibt die Ursprungsgröße der Komponente gleich. Hierfür muss also keine neue Berechnung erfolgen, da

- Breite_neu = Breite_alt und Höhe_neu = Höhe_alt.

Bei 90° und 270° gilt die Berechnung:

- Breite_neu = Höhe_alt und Höhe_neu = Breite_alt.

Bei jedem anderen Winkel muss die Berechnung jedoch explizit erfolgen. Hierfür muss die neue Komponentengröße über die Sinus- bzw. Kosinusfunktion neu berechnet werden.

Der Sinus eines Winkels ist definiert als: $\frac{\text{Gegenkathete des Winkels}}{\text{Hypothenuse}}$

Der Kosinus eines Winkels ist definiert als: $\frac{\text{Ankathete des Winkels}}{\text{Hypothenuse}}$

Für die Komponenten bedeutet dies:

- $\cos(\beta) = \frac{\text{Breite1}}{\text{Breite}}$ => Breite1 = $\cos(\beta) \cdot$ Breite
- $\sin(\beta) = \frac{\text{Höhe1}}{\text{Breite}}$ => Höhe1 = $\sin(\beta) \cdot$ Breite
- $\cos(\alpha) = \frac{\text{Breite2}}{\text{Höhe}}$ => Breite2 = $\cos(\alpha) \cdot$ Höhe
- $\sin(\alpha) = \frac{\text{Höhe2}}{\text{Höhe}}$ => Höhe2 = $\sin(\alpha) \cdot$ Höhe

Damit ergibt sich als Größe für die neue Komponente:

- Breite_neu = |Breite1 + Breite2|
- Höhe_neu = |Höhe1 + Höhe2|

Der Winkel β ist der Winkel, um den die Komponente gedreht wird. Den Winkel α kann man nach dem Winkelsatz berechnen, der aussagt, dass die Summe der Winkel eines Halbkreises 180° ergeben:

180° - 90° - β = α

Für die Drehung des Bildes muss dessen Mittelpunkt ermittelt werden:

- $\text{Mittelpunkt\_x} = \frac{\text{Breite\_alt}}{2}$
- $\text{Mittelpunkt\_y} = \frac{\text{Höhe\_alt}}{2}$

Anschließend erfolgt vor dem Neuzeichnen des Bildes, die Berechnung der Bildposition innerhalb der Komponente mit folgender Formel:

- $x = \frac{\text{Breite\_alt-Breite\_neu}}{2}$
- $y = \frac{\text{Höhe\_alt-Höhe\_neu}}{2}$

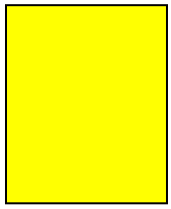

Abbildung 4.9: Position und Ausrichtung des Bildes bei 0°, 180° und 360°

Die Abbildung 4.9 zeigt die Position und Ausrichtung des Bildes bei 0°, 180° und 360°. Ein darunter liegendes, blaues Rechteck wird hier vollkommen verdeckt. Es hat sich also gegenüber der ursprünglichen Position keine Änderung ergeben, da die Drehung um den Mittelpunkt des Bildes erfolgt ist. Daher ist die Position:

- $x = \frac{\text{Breite\_alt-Breite\_neu}}{2} = 0$
- $y = \frac{\text{Höhe\_alt-Höhe\_neu}}{2} = 0$

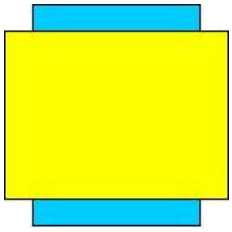

Abbildung 4.10: Position und Ausrichtung des Bildes bei 90° und 270°

Abbildung 4.10 dagegen zeigt die Drehung des Bildes der Komponente um den Mittelpunkt bei 90° und 270°. Das blaue Rechteck zeigt dabei die Position und Ausrichtung des alten Bildes sowie die Größe der alten Komponente. Das gelbe Rechteck zeigt hingegen die Position und Ausrichtung des neuen Bildes und die neue Größe der Komponente.

Die Position des neuen Bildes errechnet sich wie folgt:

- $x = \frac{\text{Breite\_alt-Breite\_neu}}{2} = -\frac{|\text{differenz\_breite}|}{2}$
- $y = \frac{\text{Höhe\_alt-Höhe\_neu}}{2} = \frac{|\text{differenz\_höhe}|}{2}$

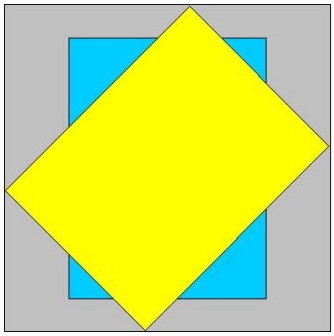

Abbildung 4.11: Position und Ausrichtung des Bildes bei einem frei gewählten Winkel

In Abbildung 4.11 sieht man die Drehung eines Bildes in einem frei gewählten Winkel. Das blaue Rechteck ist die Ausrichtung des ursprünglichen Bildes und gleichzeitig auch die Größe der ursprünglichen Komponente. Das gelbe Rechteck zeigt die neue Ausrichtung des Bildes. Das graue Rechteck macht die dadurch entstandene neue Größe der Komponente deutlich. Da die Drehung des Bildes um den Mittelpunkt erfolgt, kann man leicht die neue Position des Bildes berechnen.
Für frei gewählte Winkel gilt:

- $x = \frac{\text{Breite\_alt-Breite\_neu}}{2}$
- $y = \frac{\text{Höhe\_alt-Höhe\_neu}}{2}$

Drei Beispiele:
Breite der Komponente und des Bildes: 2,64 cm
Höhe der Komponente und des Bildes: 4,37 cm

$$\text{Mittelpunkt\_x} = \frac{\text{Breite\_alt}}{2} = \frac{2{,}64\text{ cm}}{2} = 1{,}32\text{ cm}$$

$$\text{Mittelpunkt\_y} = \frac{\text{Höhe\_alt}}{2} = \frac{4{,}37\text{ cm}}{2} = 2{,}185\text{ cm}$$

1. Beispiel: $\beta = 30°$

- $\alpha = 180° - 90° - 30° = 60°$
- $\text{Breite1} = \cos(\beta) \cdot \text{Breite} = \cos(30°) \cdot 2{,}64\text{ cm} \approx 2{,}29\text{ cm}$
- $\text{Breite2} = \cos(\alpha) \cdot \text{Höhe} = \cos(60°) \cdot 4{,}37\text{ cm} \approx 2{,}19\text{ cm}$
- $\text{Höhe1} = \sin(\beta) \cdot \text{Breite} = \sin(30°) \cdot 2{,}64\text{ cm} = 1{,}32\text{ cm}$
- $\text{Höhe2} = \sin(\alpha) \cdot \text{Höhe} = \sin(60°) \cdot 4{,}37\text{ cm} \approx 3{,}78\text{ cm}$

- Breite_neu= |Breite1 + Breite2| = |2,29 cm + 2,19 cm| = 4,48 cm
- Höhe_neu = |Höhe1 + Höhe2| = |1,32 cm + 3,78 cm| = 5,1 cm
- $x = \frac{\text{Breite\_alt-Breite\_neu}}{2} = \frac{2{,}64\text{ cm} - 4{,}48\text{ cm}}{2} = -0{,}92\text{ cm}$
- $y = \frac{\text{Höhe\_alt-Höhe\_neu}}{2} = \frac{4{,}37\text{ cm} - 5{,}1\text{ cm}}{2} \approx -0{,}58\text{ cm}$

2. Beispiel: $\beta = 90°$

- $\alpha = 180° - 90° - 90° = 0°$
- Breite1= $\cos(\beta) \cdot$ Breite = cos(90°) · 2,64 cm = 0 cm
- Breite2= $\cos(\alpha) \cdot$ Höhe = cos(0°) · 4,37 cm = 4,37 cm
- Höhe1 = $\sin(\beta) \cdot$ Breite = sin(90°) · 2,64 cm = 2,64 cm
- Höhe2 = $\sin(\alpha) \cdot$ Höhe = sin(0°) · 4,37 cm = 0 cm
- Breite_neu= |Breite1 + Breite2| = |0 cm + 4,37 cm| = 4,37 cm
- Höhe_neu = |Höhe1 + Höhe2| = |2,64 cm + 0 cm| = 2,64 cm

$\Rightarrow$ Breite_neu= Höhe_alt
Höhe_neu = Breite_alt

- $x = \frac{\text{Breite\_alt-Breite\_neu}}{2} = \frac{2{,}64\text{ cm} - 4{,}37\text{ cm}}{2} = -0{,}865\text{ cm}$
- $y = \frac{\text{Höhe\_alt-Höhe\_neu}}{2} = \frac{4{,}37\text{ cm} - 2{,}64\text{ cm}}{2} \approx 0{,}865\text{ cm}$

3. Beispiel: $\beta = 180°$

- $\alpha = 180° - 180° - 90° = -90°$
- Breite1= $\cos(\beta) \cdot$ Breite = cos(180°) · 2,64 cm = -2,64 cm
- Breite2= $\cos(\alpha) \cdot$ Höhe = cos(-90°) · 4,37 cm = 0 cm
- Höhe1 = $\sin(\beta) \cdot$ Breite = sin(180°) · 2,64 cm = 0 cm
- Höhe2 = $\sin(\alpha) \cdot$ Höhe = sin(-90°) · 4,37 cm = -4,37 cm
- Breite_neu= |Breite1 + Breite2| = |-2,64 cm + 0 cm| = 2,64 cm
- Höhe_neu = |Höhe1 + Höhe2| = |0 cm + (-4,37 cm)| = 4,37 cm

$\Rightarrow$ Breite_neu= Breite_alt
Höhe_neu = Höhe_alt

- $x = \frac{\text{Breite\_alt-Breite\_neu}}{2} = \frac{2{,}64\text{ cm} - 2{,}64\text{ cm}}{2} = 0\text{ cm}$
- $y = \frac{\text{Höhe\_alt-Höhe\_neu}}{2} = \frac{4{,}37\text{ cm} - 4{,}37\text{ cm}}{2} \approx 0\text{ cm}$

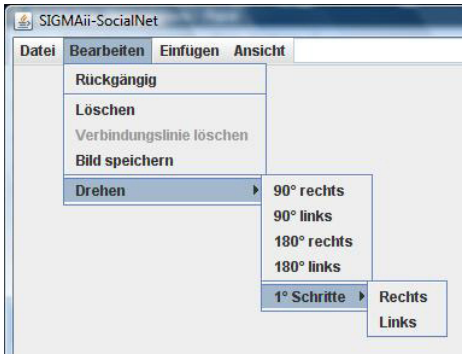

Abbildung 4.12: Menüpunkt: Drehen

Das Menü der Anwendung (Abbildung 4.12) bietet die Möglichkeit, die Komponente entweder um

- 90° rechts degree 90; direction: right
- 90° links degree: 90; direction: left
- 180° rechts degree: 180; direction: right
- 180° links degree: 180; direction: left
- 1° Schritt rechts degree: 1; direction: right
- 1° Schritt links degree: 1; direction: left

zu drehen.
Die Wiimote bietet folgende Möglichkeiten, die Komponente zu drehen:

- 90° rechts degree 90; direction: right
- 90° links degree: 90; direction: left
- 1° Schritt rechts degree: 1; direction: right
- 1° Schritt links degree: 1; direction: left

Die Drehung der Komponente wird durch folgenden Funktionsaufruf erzeugt:

edit_rotate(String direction, int degree, int id_cursor)

Hierbei werden die Parameter:

- direction: Die Richtung, in die die Komponente gedreht werden soll
- degree: Gradzahl der Komponentendrehung
- id_cursor: ID des aufrufenden Cursors

übergeben. Die Drehung selbst erfolgt im **ComponentPanel()** wie oben beschrieben und wird in der Funktion **edit_rotate()** durch den Funktionsaufruf

componentpanel.get(i).rotate(i, direction, degree);

ausgelöst. Der Parameter **i** ist dabei die zu drehende Komponente. Hierbei wird eine Drehung für alle vorher markierten Komponenten durch eine For-Schleife ausgelöst. Dadurch ist es möglich, mehrere Komponenten gleichzeitig zu drehen.

4.6 Änderung der Komponentengröße

Bei der Größenänderung einer Komponente und damit des angezeigten Bildes, ist es wichtig zu berücksichtigen, dass die Errechnung der neuen Größe immer aus der ursprünglichen Originalgröße erfolgt. Ansonsten stimmen die prozentualen Größenänderungen nicht.

Zudem muss nach einer Größenberechnung, das Bild immer neu geladen werden, um es neu zu zeichnen. Dafür wurde das Bild zu Anfang in einem Array gespeichert. Diese Methode verhindert, dass das dargestellte Bild mit zunehmender Größe immer unschärfer wird.

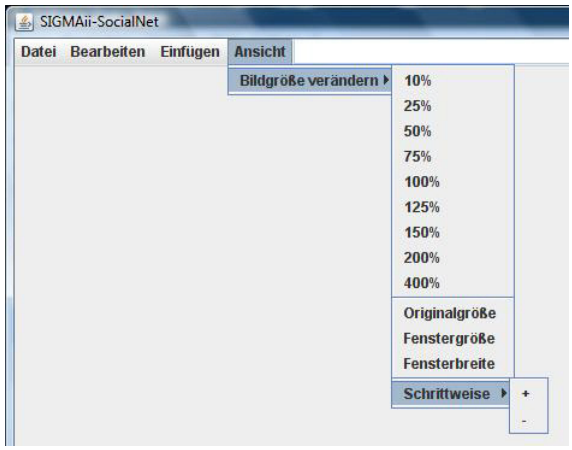

Abbildung 4.13: Menüpunkt: Bildgröße verändern

Nachfolgende Aufzählungen zeigen Menüpunktbezeichnungen und die verwendeten Variablen **size** und **percent** im ausgelösten Aufruf.

Im Menü der Anwendung (Abbildung 4.13) gibt es die Möglichkeit, die Größe der Komponente auf die festgelegten Werte

- 10% size: fix; percent: 10
- 25% size: fix; percent: 25
- 50% size: fix; percent: 50
- 75% size: fix; percent: 75
- 100% size: fix; percent: 100
- 125% size: fix; percent: 125
- 150% size: fix; percent: 150
- 200% size: fix; percent: 200
- 400% size: fix; percent: 400

der Originalgröße zu ändern oder sie schrittweise jeweils um

- 1% size: enlarge; percent: 1

zu vergrößern oder um

- 1% size: reduce; percent: 1

zu verkleinern.

Zusätzlich gibt es noch die Möglichkeit, die Komponente auf folgende Größen anzupassen:

- Originalgröße: size: originalsize; percent: 0
- Fenstergröße: size: windowsize; percent: 0
- Fensterbreite: size: windowwidth; percent: 0

Über die Wiimote kann die Größe der Komponente auf folgende Maße verändert werden:

- Fenstergröße: size: windowsize; percent: 0
- feste Größe: size: back; percent: 0

Oder aber schrittweise jeweils um

- 1% size: enlarge; percent: 1

vergrößert oder um

- 1% size: reduce; percent: 1

verkleinert werden.
Die Größenänderung der Komponente wird durch folgenden Funktionsaufruf erreicht:

view_size(String size, int percent, int id_cursor)

Hierbei werden die Parameter:

- size: Größenänderung
- percent: Größenänderung in %
- id_cursor: ID des aufrufenden Cursors

übergeben. Die eigentliche Größenänderung aber erfolgt wieder im ComponentPanel() durch den Aufruf:

componentpanel.get(i).resize(i, size, percent, dim_win_width, dim_win_height);

Hier erfolgt, wie bei der Drehung für jede markierte Komponente **i**, die entsprechende Größenänderung. Sind mehrere Komponenten markiert, sind die Menü-Punkte:

- Originalgröße
- Fenstergröße
- Fensterbreite
- 400%

deaktiviert; ebenso bei Benutzung der Wiimote. Zusätzlich erhält hier der Benutzer ein Rumble-Feedback und die Markierung der Komponenten wird aufgelöst.

Ist dagegen nur eine Komponente markiert, wird diese Komponente auf die ausgewählte Größe geändert.

4.7 Einer Komponente Text hinzufügen

Wird über die Wiimote das Editieren des Textes das erste Mal für eine Komponente aktiviert, wird dieser ein **JTextPanel()** mit dem Index der Komponente an die untere Kante angedockt. Über das Steuerkreuz der Wiimote lässt sich anschließend der Text editieren.

Abbildung 4.14: Komponente mit Textfeld

Das Editieren des Textes kann auch über die Maus aktiviert werden. Dann erfolgt die Eingabe über die Pfeiltasten der Tastatur. Die Belegung entspricht hierbei der Belegung des Wiimote-Steuerkreuzes.

Abbildung 4.15: Menüpunkt: Text

4.8 Komponente löschen

Eine hinzugefügte und markierte Komponente kann auch wieder gelöscht werden. Dies geschieht entweder über den Menüpunkt „Löschen“ oder über die Wiimote-Taste „**–**“ (Minus). Dabei wird die Funktion:

edit_delete(int id_cursor)

aufgerufen und als Parameter die ID des auslösenden Cursors an die Funktion übergeben. Zum Löschen der markierten Komponenten **i** ist es wichtig, dass ebenfalls alle zuvor gespeicherten Informationen über diese Komponenten im **ComponentPanel()** gelöscht werden. Dies geschieht durch den Aufruf

components.removeData(i);

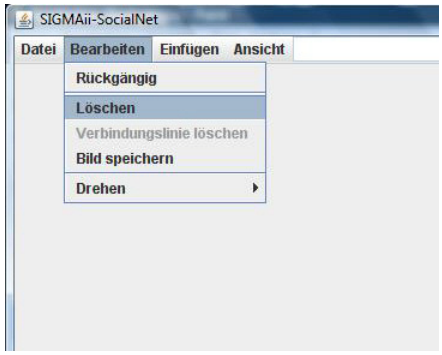

Abbildung 4.16: Menüpunkt: Löschen

4.9 gui: SaveUndo(): Letzte Änderung rückgängig machen

Der Anwender sollte die Möglichkeit haben, die letzte Änderung des jeweiligen Cursors an einer Komponente wieder rückgängig machen zu können. Wurde noch keine Komponente geändert, ist diese Funktion gesperrt. Ein Aufrufen dieses gesperrten Punktes mit der Wiimote löst ein Rumble-Feedback aus.
Folgende Änderungen können durch den Funktionsaufruf

edit_undo(int id_cursor)

rückgängig gemacht werden:

- Hinzufügen einer Komponente
- Löschen einer Komponente
- Drehen einer Komponente
- Größenänderung einer Komponente
- Texteingabe
- Hinzufügen einer Verbindungslinie zwischen zwei Komponenten

Abbildung 4.17: Menüpunkt: Rückgängig

Hierfür werden die Informationen einer Komponente *vor* der Änderung, in der Klasse **SaveUndo()** gespeichert. Die Speicherung ist cursorabhängig, das heißt, die letzte Aktion des jeweiligen Cursors lässt sich rückgängig machen.

4.10 gui: EdgePanel() : Verbindungslinie zwischen Komponenten

Da die Anwendung ein soziales Netz darstellen soll, muss es auch möglich sein, Abhängigkeiten zwischen den einzelnen Komponenten darstellen zu können. Abhängigkeiten zwischen zwei Komponenten können durch Verbindungslinien zwischen diesen Komponenten illustriert werden. Da es verschiedene Beziehungen zwischen den Komponenten geben kann, sind fünf verschiedene Farben zur Darstellung auswählbar. Es besteht keine vordefinierte Festlegung der Bedeutung der Farben. So kann der Anwender diese frei zuordnen und ist somit nicht auf ein bestimmtes soziales Netz festgelegt.

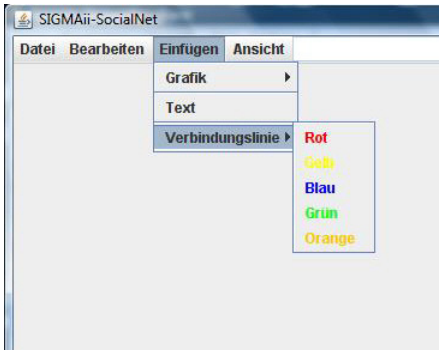

Abbildung 4.18: Menüpunkt: Verbindungslinie

Das Hinzufügen der Verbindungslinie in der gewünschten Farbe muss der Anwender entweder über das Menü oder über die Wiimote aufrufen. Dieser Aufruf ist gesperrt, wenn weniger als zwei Komponenten existieren. Im Menü stehen dem Anwender die Verbindungslinien in den verschiedenen Farben zur Auswahl. Über die Wiimote erfolgt die Farbauswahl über mehrfaches Drücken der Auswahltaste „**2**“. Im Informationsfenster der entsprechenden Wiimote wird die aktuelle Farbe angezeigt.

Die Aktivierung selbst erfolgt durch folgenden Funktionsaufruf:

insert_line(int id_cursor)

Nach der Farbauswahl muss man zwei Komponenten markieren, zu denen man eine Verbindungslinie hinzufügen möchte. Nach dieser Markierung wird durch den Funktionsaufruf

edges(int id_cursor)

und damit in der Funktion

lines = new EdgePanel(*id_line*, nodes[id_cursor][0], nodes[id_cursor][1], x_begin, y_begin, x_end, y_end, count_line[id_cursor][0]);

eine neue Verbindungslinie erzeugt. Die Linie verläuft zu den Mittelpunkten der ausgewählten Komponenten.
Mit erfolgreichem Zeichnen einer Linie ist diese Funktion deaktiviert. Ein weiteres Hinzufügen einer Verbindungslinie zwischen zwei Komponenten muss erst erneut aktiviert werden.

Abbildung 4.19: Menüpunkt: Verbindungslinie löschen

Das Entfernen der Verbindungslinie ist nur über das Menü möglich. Hierzu wird die Funktion

delete_edges(final int id_cursor)

aufgerufen. Nach der Aktivierung dieses Punktes, müssen die zwei Komponenten markiert werden, zwischen denen die Verbindungslinie entfernt werden soll. Auch das Entfernen der Verbindungslinie ist nur für *eine* Verbindungslinie aktiviert. Ein weiteres Entfernen muss erst wieder aktiviert werden.

4.11 Speichern

Es gibt zwei Menüpunkte und damit zwei Möglichkeiten zu speichern:

- Bild speichern (Menü „Bearbeiten“ -> „Bild speichern“)
- Soziales Netz speichern (Menü „Datei“ -> „Speichern)

Abbildung 4.20: Menüpunkt: Bild speichern
Abbildung 4.21: Menüpunkt: Speichern

Nach dem Einscannen eines Bildes kann es gespeichert werden, um es später gegebenenfalls erneut hinzufügen zu können. Zum Speichern des Bildes einer Komponente, muss diese zunächst markiert werden. Eine Markierung mehrerer Komponenten ist möglich. Anschließend erfolgt durch Auswahl des Menüpunktes „Bild speichern“ der Funktionsaufruf:

edit_store(int id_cursor)

Das Dialogfeld zur Eingabe des Speicherortes und des Dateinamens wird über

savefile = new FileDialog(this,"Datei speichern",FileDialog.*SAVE*);

aufgerufen. Die eingegeben Daten werden dann durch den Funktionsaufruf

componentpanel.get(i).storeImage(i, directory, filename);

dem **ComponentPanel** als Parameter, neben der ID **i** der Komponente, übergeben. Das Bild bzw. die Bilder werden dann im **ComponentPanel()** mit der entsprechenden Funktion im JPG-Format gespeichert. Zur Zuordnung werden dem angegebenen Dateinamen die IDs der ausgewählten Komponenten hinzugefügt:

```
bufferedimage = (BufferedImage) images[id_component][0];
try{
    name = directory + filename+ Integer.toString(id_component) + ".jpg";
    ImageIO.write(bufferedimage , "jpeg", new File(name));
} catch (final IOException e){
    e.printStackTrace();
}
```

Als zweite Möglichkeit steht dem Anwender das Abspeichern des kompletten sozialen Netzes zur Verfügung. Die Aktivierung des Menüpunktes „Speichern“ löst den Funktionsaufruf

file_store()

aus. Hier werden die kompletten Informationen des sozialen Netzes:

- Komponente mit:
  - ID
  - Bild
  - Größe

    - Ausrichtung
    - Position
- Verbindungslinien mit:
    - Knoten, von denen sie ausgehen
    - ID
    - Position
- Textfelder mit:
    - Text
    - ID der dazugehörigen Komponente

durch Serialisierung abgespeichert. Serialisierung ist eine Methode, um Objekte persistent zu speichern. Bei der Serialisierung werden die Felder eines Objekts in eine Serie von Bytes (einen Datenstrom) umgewandelt [Daum 07, S.331-332]. Dies geschieht über den ObjectOutputStream:

```
fileoutputstream_text = new FileOutputStream("object.sigmaii");
bufferedoutputstream_text = new bufferedOutputStream(fileoutputstream_text);
objectoutputstream_text = new ObjectOutputStream(bufferedoutputstream_text);
objectoutputstream_text.writeObject(text_components);
objectoutputstream_text.close();
```

Die Komponenten, Textfelder und Verbindungslinien werden separat serialisiert. Das Dialogfeld zur Eingabe des Speicherortes und des Speichernamens der Datei wird über

savefile = new FileDialog(this,"Datei speichern",FileDialog.*SAVE*);

aufgerufen. Nach der Benutzereingabe wird durch folgende Funktion ein neuer Ordner mit dem Speichernamen der Datei erzeugt:

```
createdirectory= new File(directory   + File.separatorChar
                                      + filename
                                      + File.separatorChar
                                      + filename);
createdirectory.mkdirs();
```

In diesen Ordner werden die Informationen der Komponenten gespeichert, die später wieder geladen werden können. Die Bilder und Informationen über Verbindungslinien und Textfelder werden in einen Unterordner desselben Namens gespeichert. Da sich Bilder nicht serialisieren lassen, werden sie separat, mit Anhängen der ID der entsprechenden Komponente, im JPG-Format gespeichert. Die gespeicherten Dateien für Komponenten, Textfelder und Verbindungslinien sind vom Dateityp

sigmaii.

Am Beispiel der Textfelder demonstriere ich kurz den Ablauf der Serialisierung:

1. Auswahl des Speicherortes der zu speichernden Datei:

```
savefile = new FileDialog(this, "Datei speichern", FileDialog.SAVE);
savefile.setVisible(true);
directory = savefile.getDirectory();
filename = savefile.getFile();
```

2. Wir wollen die Datei im Verzeichnis **d:\SIGMAii** unter dem Namen **socialnet1** abspeichern. Also haben die Variablen **directory** und **filename** folgende Werte:
   - directory: d:\SIGMAii\
   - filename: socialnet1
3. Mit der Initialisierung von **createdirectory** und dem Befehl **createdirectory.mkdirs()** wird ein neuer Ordner im Verzeichnis **d:\SIGMAii** mit dem Namen **socialnet1** und ein Unterordner **socialnet1** erzeugt:

```
createdirectory = new File(directory  + File.separatorChar
                                      + filename
                                      + File.separatorChar
                                      + filename);
createdirectory.mkdirs();
```

4. Da wir hier Informationen über die Textfelder abspeichern wollen, muss das Verzeichnis entsprechend geändert werden. Unsere Datei wird im Unterverzeichnis

   d:\sigmaii\socialnet1\socialnet1\ abgespeichert:

```
directory = directory
               + File.separatorChar
               + filename
               + File.separatorChar
               + filename
               + File.separatorChar;
```

5. Nun werden die Informationen der einzelnen Textfelder in ein Array geschrieben:

```
for (int i = 1; i < textfields.size(); i++){
    text_components[i][0] = textfields.get(i).getText();
}
```

6. Das im vorangegangenen Schritt aktualisierte Array wird nun serialisiert:

```
fileoutputstream_text = new FileOutputStream(directoy
                                              + filename
                                              + "_text"
                                              + ".sigmaii");
bufferedoutputstream_text = new bufferedOutputStream(fileoutputstream_text);
objectoutputstream_text = new ObjectOutputStream(bufferedoutputstream_text);
objectoutputstream_text.writeObject(text_components);
objectoutputstream_text.close();
```

4.12 Laden

Durch Deserialisierung können zuvor abgespeicherte Daten wieder eingelesen werden:

```
fileinputstream_text = new FileInputStream("object.sigmaii");
bufferedinputstream_text = new bufferedInputStream(fileinputstream_text);
objectinputstream_text = new ObjectInputStream(bufferedinputstream_text);
objectinputstream_text.writeObject(text_components);
objectinputstream_text.close();
```

Abbildung 4.22: Menüpunkt: Laden

Gespeicherte Soziale Netze vom Dateityp **.sigmaii** können zur Anzeige oder zur weiteren Bearbeitung wieder geladen werden. Hierzu gibt es den Menüpunkt „Laden“, der den Funktionsaufruf

file_load()

auslöst. Dieser Menüpunkt ist aktiviert, bis die erste Komponente geladen wurde.
Der Aufruf des Menüpunkts öffnet ein Dialogfeld zur Auswahl der gewünschten Datei:

```
loadfile = new FileDialog(this,"Datei öffnen",FileDialog.LOAD);
loadfile.setVisible(true);
directory = loadfile.getDirectory();
filename_withsuffix = loadfile.getFile();
```

Hierzu wird die Hauptdatei der Komponenten im Hauptverzeichnis (in unserem obigen Beispiel **d:\sigmaii\socialnet1**) gewählt. Die zugehörigen Dateien werden automatisch geöffnet und die gespeicherten Daten nacheinander eingelesen. Am Beispiel der vorher abgespeicherten Textdatei sieht das folgendermaßen aus:

1. Datei wählen, die geöffnet werden soll:

```
loadfile = new FileDialog(this,"Datei öffnen",FileDialog.LOAD);
loadfile.setVisible(true);
directory = loadfile.getDirectory();
filename_withsuffix = loadfile.getFile();
int position = filename_withsuffix.indexOf(".sigmaii");
filename = filename_withsuffix.substring(0,position);
```

   In unserem obigen Beispiel haben wir die Datei unter

   d:\SIGMAii\socialnet1.sigmaii

   gespeichert. Diese wählen wir aus:

   - directory: d:\SIGMAii\
   - filename_withsuffix: socialnet1.sigmaii
   - filename: socialnet1

2. Da unsere Textdatei im Unterverzeichnis **socialnet1** gespeichert wurde, muss hier das Verzeichnis entsprechend geändert werden:

   directory = directory + filename + File.separatorChar;

3. Dann wird der Dateiname auf den Namen der Textdatei geändert:

   filename_withsuffix = filename + "_text.sigmaii";

4. Schließlich wird die Textdatei deserialisiert:

```
fileinputstream_text = new FileInputStream(directory + filename_withsuffix);
bufferedinputstream_text = new BufferedInputStream(fileinputstream_text);
objectinputstream_text = new ObjectInputStream(bufferedinputstream_text);
```

5. Die Daten (auch leere Datenstrings) werden wiederum in einem Array gespeichert. Die Inhalte nicht leerer Datenfelder werden in die entsprechenden Textfelder der Komponenten geschrieben:

```
text_components =(String[][])objectinputstream_text.readObject();
objectinputstream_text.close();

text = Integer.toString(id_component);
text_components[id_component][0] = text;
textfield = new JTextField(text);
add(textfield);
add(components2);
textfields.add(textfield);
textfields.get(id_component).setSize(400,20);
```

```
textfields.get(id_component).setLocation(0,0);
final int width = this.components.getWidth(id_component);
final int height = this.components.getHeight(id_component);
x = this.components.getX(id_component);
y = this.components.getY(id_component);
textfields.get(id_component).setSize(width, 20);
textfields.get(id_component).setLocation(x, y+height);
textfields.get(id_component).setVisible(false);


if (!textfields_empty[id_component][0]){
    text = text_components[id_component][0];
    textfields.get(id_component).setText(text);
    textfields.get(id_component).setVisible(true);

    this.repaint();

    textfields_empty[id_component][0] = false;
    edit_text[id_component][0] = true;

    textfield.setLocation(x, y+components.getHeight(id_component));
    textfields.get(id_component).setSize(components.getWidth(id_component), 20);
}
```

4.13 Drucken

Der Menüpunkt „Drucken“ ruft die Funktion

file_print()

auf.

Abbildung 4.23: Menüpunkt: Drucken

Die eigentliche Druckfunktion wird in Java durch das Paket print() (java.awt.print) bereitgestellt. Hier sieht das folgendermaßen aus:

```
toolkit = Toolkit.getDefaultToolkit();
printjob = toolkit.getPrintJob( new Frame(), "", null );
if (printjob != null ){
    Graphics g = printjob.getGraphics();
    printComponents( g );
    g.dispose();
    printjob.end();
}
```

Kapitel V – Zusammenfassung & Ausblick

5.1 Zusammenfassung

Diese Diplomarbeit macht die vielseitigen Verwendungsmöglichkeiten der Wiimote deutlich.

Mit der von mir erstellten *SIGMAii-API* ist ein Java-Programmierer in der Lage, die Wiimote nach vertrauten Entwurfsmustern am Computer zu nutzen, da sie sich an das Maus-Event-Modell anlehnt und darüber hinaus die umfangreichen Funktionen der Wiimote zur Verfügung stellt.

Die *SIGMAii-API* macht es nun leicht, die Wiimote neuen PC-Anwendungen zu erschließen, wie ich am Beispiel der hier implementierten Anwendung *SIGMAii-SocialNet* erstmals zeigen konnte.

Mit *SIGMAii-SocialNet* können zum Beispiel CRC-Karten kollaborativ am PC verwaltet und ein Netzwerk aufgebaut werden. Die Anwendung ermöglicht es, neue Objekte, mit Grafiken aus Dateien oder vom Scanner, frei zu platzieren. Die Objekte können in ihrer Größe, Ausrichtung und Position frei verändert und den Objekten frei editierbare Textfelder hinzugefügt werden. Abhängigkeiten zwischen den Objekten werden mittels Kanten dargestellt, wofür mehrere Farben zwecks Differenzierung zur Auswahl stehen.

5.2 Ausblick

In dieser Anwendung fanden bislang lediglich die Listener **WiiListener()** und **WiiMotionListener()** Verwendung, die, alleine genutzt, die Funktionen der Wiimote auf die einer Maus einschränken. Die Listener, die sich im speziellen mit der Drehung und Beschleunigung der Wiimote beschäftigen, wurden hier nicht verwendet.

Zurzeit beschäftigt sich Herr Glesener von der Universität Trier mit einem Gestenerkennungsprojekt. Dieses soll in naher Zukunft in die SIGMAii-API eingebunden werden, was dem Anwender weitere interessante Möglichkeiten offen legt. Die Einschränkung auf die Benutzung der Tasten der Wiimote würde entfallen, und könnte durch vorher festgelegte Bewegungen der Wiimote (Gesten) Aktionen auslösen.

Folgende intuitive Gesten und deren Aktionen wären zum Beispiel denkbar:

- Wurf nach hinten: Löschen der Komponente
- Kreisbewegung gegen den Uhrzeigersinn: Vergrößern der Komponente
- Kreisbewegung im Uhrzeigersinn: Verkleinern der Komponente

Zum Zeitpunkt der Fertigstellung dieser Diplomarbeit war dieses Projekt leider noch nicht weit genug fortgeschritten. Dennoch werden solche Funktionalitäten in Zukunft die Bedienbarkeit einer Anwendung erheblich bereichern.

Dass es vielen Benutzern meiner API schon jetzt möglich ist, durch die Listener eigene Methoden hinzuzufügen, die auf die entsprechenden Bewegungen bzw. Beschleunigungen reagieren, wird die Entwicklung in nächster Zeit sicher vorantreiben.

Da Nintendo bereits zahlreiches Zubehör zur Wii-Konsole auf den Markt gebracht hat, liegt eine Implementierung neuer Funktionen in die API auf der Hand. Zum Beispiel wäre eine Drehung von Objekten auf dem Monitor, unter Verwendung des Wii Balance Boards, durch Gewichtsverlagerung denkbar. So könnten dreidimensionale Objekte durch die freie Drehung vollständig dargestellt werden. Dies zeigten bereits de Haan et al. in ihrem Artikel “Using the Wii Balance Board™ as a low-cost VR interaction device.” [de Haan 08].

Durch schon jetzt verfügbares und zukünftiges Nintendo-Zubehör kann der Funktionsumfang der SIGMAii-API bereichert werden. Weitere und vor allem einfachere Arten der Bedienung für Computer-Anwender direkt durch die Wiimote und deren Zubehör werden möglich.

Die gezeigten Beispiele zeigen deutlich, dass meine Erstellung der SIGMAii-API und der zugehörigen Anwendung nur ein erster Schritt sein kann und dass die Beschäftigung mit dem Thema Wiimote noch zahlreiche Entwicklungsmöglichkeiten bietet, die die bisher gebräuchliche Computer-Bedienung wohl gravierend verändern wird.

Literaturverzeichnis

[ADXL330] http://www.analog.com/en/mems-and-sensors/imems-accelerometers/adxl330/products/product.html

letzter Zugriff: 14.12.2008

[BCM2042] http://www.broadcom.com/products/Bluetooth/Bluetooth-RF-Silicon-and-Software-Solutions/BCM2042

letzter Zugriff: 14.12.2008

[Begole] Bo Begole, David W. McDonald (Eds.):

“Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work”

CSCW 2008, San Diego, CA, USA, November 8-12, 2008. ACM 2008, ISBN 978-1-60558-007-4

[CSCW] http://wwwfgcscw.in.tum.de/

letzter Zugriff: 14.12.2008

[Cunningham 89] Kent Beck and Ward Cunningham:

„A Laboratory for Teaching Object-Oriented Thinking”

in: Norman K. Meyrowitz (Ed.):

“Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89)”

Oktober 1-6, 1989, New Orleans, Louisiana, Proceedings. SIGPLAN Notices 24(10), S. 1-6, Oktober 1989

http://c2.com/doc/oopsla94.html

letzter Zugriff: 14.12.2008

[Daum 07] Berthold Daum: „Java 6“

Addison-Wesley, München; Auflage: 1 (2007)

[de Haan 08] Gerwin de Haan, Eric J. Griffith, Frits H. Post:

“Using the Wii Balance Board™ as a low-cost VR interaction device.”

in: ACM, Steven Feiner, Daniel Thalmann, Pascal Guitton, Bernd Fröhlich, Ernst Kruijff, Martin Hachet:

„Proceedings of the ACM Symposium on Virtual Reality Software and Technology”

VRST 2008, Bordeaux, France, October 27-29, S. 289-290, 2008

[Diehl 08] Stephan Diehl: Vorlesung: Fortgeschrittene Softwaretechnik Wintersemester 2008/09, Universität Trier

[GlovePie] http://carl.kenner.googlepages.com/glovepie_download
letzter Zugriff: 14.12.2008

[LeeJ] http://www.cs.cmu.edu/~johnny/projects/wii/
letzter Zugriff: 14.12.2008

[LeeH 08] Hyun-Jean Lee, Hyungsin Kim, Gaurav Gupta, Ali Mazalek:
„WiiArts: creating collaborative art experience with Wii Remote interaction"
in: ACM, Albrecht Schmidt, Hans Gellersen, Elise van den Hoven, Ali Mazalek, Paul Holleis, Nicolas Villar:
„Proceedings of the 2nd International Conference on Tangible and Embedded Interaction 2008"
Bonn, Germany, February 18-20, S. 33-36, 2008

[Microsoft] http://www.microsoft.com/downloads/details.aspx?FamilyID=A137998B-E8D6-4FFF-B805-2798D2C6E41D&mg_id=10122&displaylang=en
letzter Zugriff: 14.12.2008

[PixArt] http://www.pixart.com.tw/investor.asp?sort=4&learname=level03
letzter Zugriff: 14.12.2008

[Poppinga 07] Benjamin Poppinga:
„Beschleunigungsbasierte 3D-Gestenerkennung mit dem Wii-Controller"
Individual Project at University of Oldenburg, Germany, September 2007

[Schlömer 08] Thomas Schlömer, Benjamin Poppinga, Niels Henze, Susanne Boll:
„Gesture recognition with a Wii controller"
in: ACM, Albrecht Schmidt, Hans Gellersen, Elise van den Hoven, Ali Mazalek, Paul Holleis, Nicolas Villar:
„Proceedings of the 2nd International Conference on Tangible and Embedded Interaction 2008"
Bonn, Germany, February 18-20, S. 11-14, 2008

[Schou 07] Torben Schou, Henry J. Gardner:

„A Wii remote, a game engine, five sensor bars and a virtual reality theatre”

in: ACM, Bruce Thomas:

“Proceedings of the 2007 Australasian Computer-Human Interaction”

Conference, OZCHI 2007, Adelaide, Australia, November 28-30, S. 231-234, 2007

[Schou 08] Torben Schou, Henry J. Gardner:

„A surround interface using the Wii controller with multiple sensor bars.”

in: ACM, Steven Feiner, Daniel Thalmann, Pascal Guitton, Bernd Fröhlich, Ernst Kruijff, Martin Hachet:

„Proceedings of the ACM Symposium on Virtual Reality Software and Technology”

VRST 2008, Bordeaux, France, October 27-29, S. 287-288, 2008

[SUN] `http://java.sun.com/j2se/1.5.0/docs/api/overview-summary.html`

`letzter Zugriff: 14.12.2008`

[Ullenboom 2007] Christian Ullenboom: „Java ist auch eine Insel“

7., aktualisierte Auflage, geb., mit DVD (November 2007), 1.492 S., Galileo Computing, ISBN 978-3-8362-1146-8

`http://www.sws.bfh.ch/~amrhein/Swing/javainsel7/javainsel_07_002.htm`

`letzter Zugriff: 14.12.2008`

[wiiuse] `http://www.wiiuse.net/`

`letzter Zugriff: 14.12.2008`

[WiiuseJ] `http://code.google.com/p/wiiusej/`

`letzter Zugriff: 14.12.2008`

[Wunderworks] `http://www.wunderworks.com/public/home.aspx`

`letzter Zugriff: 14.12.2008`