Technische Universität Berlin

Realisierung eines webbasierten Score-Followers für Tasteninstrumente mit Chroma-Features

Masterarbeit

am Fachgebiet Audiokommunikation

Institut für Sprache und Kommunikation

Fakultät I Geisteswissenschaften

Technische Universität Berlin

vorgelegt von

Erik Werner

am 4. Februar 2014

Prüfer:	Prof. Dr. Stefan Weinzierl
Zweitprüfer:	Prof. Dr.-Ing. Alexander Raake

Erik Werner

[redacted]

Abstract

Subject of this master thesis is the automatic following of a music score through analysis of sound caused by playing a keyboard instrument. This thesis suggests an algorithm embedded in an e-learning application developed to learn to play a keyboard instrument. Well-known approaches based on Onset Detection and Pitch Chroma features (a.k.a. Pitch Class Profiles) are used to detect played tones and chords in order to trace their chronological sequence and follow them on a virtual musical score. The e-learning application including the Score Following algorithm are based in the web domain and therefore can be executed in modern web browsers.

Zusammenfassung

Thema dieser Masterarbeit ist das automatische Folgen einer Notenpartitur (*Score-Following*) durch Analyse von Schallereignissen, die auf einem Tasteninstrument erzeugt werden. Diese Arbeit präsentiert einen Algorithmus in einer *E-Learning*-Applikation, in deren Fokus das Erlernen eines Tasteninstruments steht. Zur Realisierung des Algorithmus werden bekannte Ansätze im Bereich der Onset-Erkennung und der Extraktion von Chroma-Features (auch *Pitch Class Profiles* genannt) aufgegriffen, um gespielte Töne und Akkorde des Klavierspielers zu erkennen, sodass deren zeitlicher Verlauf auf einer virtuellen Partitur verfolgt werden kann. Diese Anwendung inklusive des *Score-Following*-Algorithmus basiert dabei auf neuen Technologien im Webbereich und kann somit in modernen Webbrowsern ausgeführt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis **VII**

Tabellenverzeichnis **IX**

1 Vorbetrachtungen **1**

1.1 Konzept und Funktionalität von *flowkey* 2
1.2 Anforderungen 4
1.2.1 Verarbeitung polyphoner Schallereignisse 4
1.2.2 Webbasierte Implementierung für den Browser 5
1.2.3 Echtzeit-Verarbeitung 5
1.3 Überblick *Score-Following* 6
1.4 Abgrenzung zum klassischen *Score-Following* 8
1.5 Lösungsansatz 9
1.6 Zusammenfassung 10

2 Signalverarbeitung und Webtechnologien **11**

2.1 HTML5 Web Audio 11
2.2 Flash 14
2.3 Silverlight 16
2.4 Auswahl einer Technologie 17
2.5 Echtzeit-Signalverarbeitung 18
2.6 Native Sprachen vs. JavaScript 19
2.7 Zusammenfassung 20

3 Theoretische Grundlagen **21**

3.1 Fourier-Transformation 21
3.2 Chroma-Extraktion 23
3.2.1 Berechnung von Chroma-Vektoren 25
3.2.2 Ausgewählte Arbeiten 27
3.3 Onset-Erkennung 28
3.3.1 Motivation 28

3.3.2 Theorie . . . 29
3.3.2.1 Onset Features . . . 30
3.3.2.2 Peak Picking . . . 32
3.4 Zusammenfassung . . . 34

4 Implementierung des Algorithmus 35
4.1 Überblick . . . 35
4.2 Vorverarbeitung . . . 37
4.2.1 Web Audio Schnittstelle . . . 37
4.2.2 Vorverarbeitungsschritte . . . 38
4.2.3 Fast Fourier Transformation . . . 41
4.3 Onset-Erkennung . . . 42
4.3.1 Erkennungsfunktion . . . 42
4.3.2 *Peak-Picking*-Algorithmus . . . 43
4.4 Chroma-Extraktion . . . 44
4.5 Event Modell . . . 46
4.5.1 User Event . . . 46
4.5.2 Erwartetes Event . . . 46
4.6 Chroma-Schwellwert-basierte Vergleichsfunktion . . . 47
4.7 Kosinus-Ähnlichkeit-basierte Vergleichsfunktion . . . 50
4.8 Zusammenfassung . . . 52

5 Evaluierung 53
5.1 Testumgebung . . . 53
5.2 Eventerkennung . . . 54
5.2.1 Test mit Live-Samples . . . 56
5.2.1.1 Testmaterial . . . 56
5.2.1.2 Ergebnisse . . . 57
5.2.2 Test mit Midi-generierten Samples . . . 58
5.2.2.1 Testmaterial . . . 59
5.2.2.2 Ergebnisse . . . 61
5.3 Echtzeitfähigkeit . . . 64
5.4 Diskussion der Ergebnisse . . . 66
5.4.1 Live-Samples . . . 66
5.4.2 Midi-Samples . . . 68
5.4.3 Echtzeit-Test . . . 69
5.5 Sonstige Probleme . . . 69
5.5.1 *Score Follower* und *flowkey*-Player . . . 69

5.5.2 Oktavunabhängige Analyse 70
5.5.3 Sprache und Hintergrundgeräusche 70
5.6 Mögliche Verbesserungen 70
5.6.1 *Score Follower* und *flowkey*-Player 71
5.6.2 Oktavunabhängige Analyse 71
5.6.3 Andere grundlegende Ansätze 72
5.7 Zusammenfassung 73

6 Zusammenfassung und Fazit **75**
6.1 Entwicklungsprozess 75
6.2 Fazit 77

A Relevanter Code **XV**
A.1 Chroma-Berechnung XV
A.2 Spectral Flux XVI

B Wahrheitswerttabelle **XVII**

C CD-ROM mit vollständigem Quellcode **XVIII**

Abbildungsverzeichnis

1.1 Benutzeroberfläche der *flowkey*-Webapplikation mit synchronisiertem Video/Audio und Partitur 2
1.2 Grundlegendes Konzept des *Score-Following*s nach [1]: Ableich von erwarteten und tatsächlichen Events mittels eines Matching Algorithmus zum Begleiten eines Stücks 6

2.1 Beispielgraph einer Signalverarbeitungskette erzeugt mit der *Web Audio Playground*-Applikation von Chris Wilson. 12

3.1 Die Pitch-Helix, welche das Verhältnis zwischen Oktav-Intervallen modelliert. Alle Halbtöne mit diesem Verhältnis liegen in der Helix übereinander [2]. 24
3.2 Chromagramm der Aufnahme einer Piano-Version des Songs *I'm yours* von Jason Mraz 25

4.1 Flussdiagramm des *Score-Following*-Algorithmus 36
4.2 Spiegelfrequenzen (Alias-Effekte), welche durch Nichtbeachtung des Shannon-Nyquist-Theorems entstehen. Eine Tiefpassfilterung des Signals vor dem Downsampling ist deshalb nötig. 38
4.3 Frequenzgang des Tiefpassfilters mit einer Cutofffrequenz von 11025 Hz und einem Resonanzwert von 0 dB 39
4.4 Vergleich von *Spectral Flux*, *High Frequency Content* und Energie-Feature 43
4.5 Darstellung eines über die Zeit gemittelten Chroma-Vektors mit erwarteten Pitchklassen (gelb) und errechnetem Schwellwert (hellblau) 48

5.1 Benutzeroberfläche der Testumgebung mit Onset-, Chroma-, FFT- und Zeit-Monitoring 54
5.2 Richtig-positiv-Raten für Live-Samples: Vergleichsfunktionen mit über Haltezeit gemitteltem Chroma-Vektor 58
5.3 Richtig-positiv-Raten für Live-Samples: Vergleichsfunktionen mit letztem Chroma-Vektor der Haltezeit 59

5.4 Beispiele der mittels Midi generierten Akkorde: C-2-Klang, C-3-Klang, C-3+1-Klang und Cm7-4-Klang 60
5.5 Richtig-positiv-Raten für Midi-Samples: Vergleichsfunktionen mit über Haltezeit gemitteltem Chroma-Vektor 62
5.6 Richtig-positiv-Raten für Live-Samples: Vergleichsfunktionen mit letztem Chroma-Vektor der Haltezeit 63
5.7 Falsch-positiv-Raten für Midi-Samples: Varianten mit Kosinus-Ähnlichkeit-basierter Vergleichsfunktion 64
5.8 Zeitlicher Verlauf eines Tons auf einem Piano [3] 67

B.1 Wahrheitswerttabelle zum Aufstellen des booleschen Ausdruck für die Chroma-Schwellwert-basierte Vergleichsfunktion XVII

Tabellenverzeichnis

4.1 Midi-Frequenzwerte (gerundet) der ersten vollständigen Oktave auf einer Klaviatur, auf deren Grundlage alle weitere Frequenzen der Halbtöne berechnet werden. 44

5.1 Mittelwerte der Laufzeiten eines Signalblocks für alle Varianten des Algorithmus 65

Kapitel 1

Vorbetrachtungen

Thema dieser Masterarbeit ist das computergestützte automatische Folgen einer Notenpartitur. Ein mit einem Mikrofon und entsprechender Software ausgestatteter Computer analysiert dabei die Aufführung eines Musikers, um die aktuelle Position des gespielten Stücks in der vorliegenden Partitur zu markieren. Diese Disziplin wird als *Score-Following* bezeichnet. *Score-Following* ist ein Forschungszweig im Schnittpunkt der Bereiche der Signalverarbeitung, der Musikinformationsgewinnung (MIR: *Music Information Retrieval*) und der Musikwissenschaft. Weiterhin kann es als Synchronisation zwischen einem Computer und einem Musiker verstanden werden, der eine bekannte Partitur zu Grunde liegt [4] und eine synthetisierte musikalische Begleitung des Stücks durch den Computer ermöglicht.

In dieser Arbeit wird ein *Score-Following*-Algorithmus präsentiert, der in einer webbasierten *E-Learning*-Applikation namens *flowkey* eingebettet ist. Diese ermöglicht es, Klavierstücke zu erlernen, indem sie dem Nutzer ein Klavierstück audiovisuell und synchron zur Notenpartitur präsentiert und ihm die Gelegenheit gibt, die zu spielenden Töne und Akkorde Schritt für Schritt im eigenem Tempo zu erlernen.

Dieses Kapitel soll zum einen die Funktion des Algorithmus motivieren sowie einen Überblick über die Anforderungen der *flowkey*-Webapplikation geben. Es beinhaltet zudem einen Abschnitt zum Thema *Score-Following* allgemein, in dem frühere Arbeiten kurz präsentiert werden. Weiterhin wird ein eigener Lösungsansatz basierend auf den Anforderungen der *flowkey*-Applikation dargestellt. Das zweite Kapitel betrachtet verschiedene webbasierte Technologien, die für diese Arbeit in Frage kommen und stellt die geeignetste davon vor. Im anschließenden Kapitel werden die Grundlagen des hier vorgestellten Algorithmus beschrieben. Die konkrete Implementierung des Algorithmus wird anhand von Auszügen aus dem Quellcode im Kapitel 4 vorgestellt. Die beiden letzten Kapitel widmen sich jeweils der Evaluierung und der Zusammenfassung dieser Arbeit.

1.1 Konzept und Funktionalität von *flowkey*

Der hier vorgestellte Algorithmus ermöglicht es dem Benutzer der Anwendung ein Klavierstück unabhängig von dessen Tempo anhand eines Videos und einer synchronisierten Partitur zu erlernen. Dabei sind die als nächstes zu spielenden Noten sowohl im Video als auch in der Partitur hervorgehoben markiert. Die vom Nutzer erzeugten Schallereignisse werden durch das integrierte Mikrofon des (Tablet-)PCs aufgenommen und innerhalb der Anwendung analysiert. Erklingt der richtige Ton oder Akkord, zeigt die Anwendung stets den nächsten Ton oder Akkord im Video und in der Partitur an. Das Video zeigt eine Interpretation eines Stücks auf dem Klavier in der Aufsicht (siehe Abbildung 1.1).

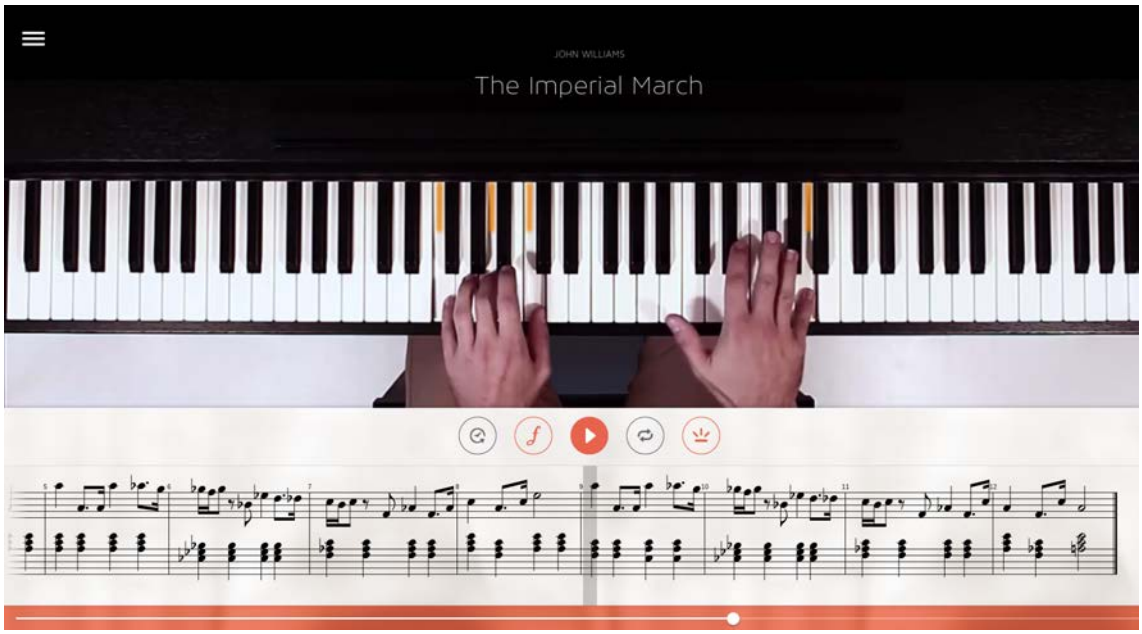

Abbildung 1.1: Benutzeroberfläche der *flowkey*-Webapplikation mit synchronisiertem Video/Audio und Partitur

flowkey[1] ist eine *E-Learning*-Applikation, die es erlaubt, Musikstücke auf einem Tasteninstrument (E-Piano oder akustisches Klavier) zu erlernen. Es arbeitet multimodal und spricht damit verschiedene Lerntypen an. Die *flowkey*-Anwendung stellt das zu erlernende Stück sowohl als Video und Audio als auch in Form einer virtuellen Partitur (siehe Abbildung 1.1) bereit, welche mit Video und Audio synchronisiert ist. So ist es möglich, die im Song erklingenden Töne auf der Partitur und im Video die entsprechende Spieltechnik nachzuvollziehen.

Die Motivation besteht weiterhin darin, das Lernen von Musikstücken anhand von Lern-Videos, wie es sie auf vielen Video-Plattformen im Internet zu finden gibt, zu erleichtern.

[1] `http://http://www.flowkey.com/de`

flowkey soll es dem Nutzer ermöglichen, individuelle Passagen im Video auszuwählen und wiederholt zu spielen, ohne dass die Passage über die Zeitleiste im Video wiederkehrend gesucht und aufgerufen werden muss. Anhand des hier vorgestellten *Score-Following*-Algorithmus kann die Anwendung dem Benutzer zuhören und Video und Partitur entsprechend synchronisieren. Dies ermöglicht es dem Benutzer, ein Klavierstück im eigenen Lerntempo zu erlernen, indem eine visuelle Lernanleitung zur Verfügung gestellt wird, die auf den Spieler wartet, statt im originalen Tempo weiterzulaufen.

Darüber hinaus wird eine hohe Benutzerfreundlichkeit sowie Gebrauchstauglichkeit angestrebt, indem dem Benutzer eine Reihe von Funktionen zur Verfügung gestellt werden, mit denen er seinen persönlichen Lernstil umsetzen kann.

Loop-Funktion: Die erste der im *flowkey*-Player enthaltenen Funktionen erlaubt das freie Selektieren eines Abschnitts in der Partitur. Wird daraufhin der Play-Button betätigt, spielt die Anwendung die Selektion in einer Schleife mit jeweils einer kleinen Pause dazwischen ab. Dies ermöglicht es dem Benutzer, das Stück in Abschnitte zu teilen und es schrittweise zu erlernen, ohne dass der Cursor stets manuell auf den Anfang einer Passage bewegt werden muss.

Leuchtende Tasten: Diese Funktion hebt die nächsten zu spielenden Tasten farbig im Video hervor. So kann der Nutzer besser erkennen, welche im Video gezeigten Finger eine Taste auf dem Klavier drücken und welche momentan passiv sind. Besonders bei schnelleren oder komplizierten Stücken erlaubt dieses Feature eine bessere Unterscheidung. Weiterhin ist es für Benutzer mit visuellem Lerntyp besonders nützlich.

Zeitlupe: Eine simple Zeitlupen-Funktion erlaubt es, Video und Partitur in der Hälfte der originalen Geschwindigkeit abspielen zu lassen. Auch dieses Feature ist vor allem bei komplizierten Stücken von Vorteil.

Warte-Funktion: Die Warte-Funktion repräsentiert den in dieser Arbeit implementierten *Score-Following*-Algorithmus. Mit ihr hört die Anwendung dem Nutzer zu und analysiert das Schallsignal über das Mikrofon. Dabei bleiben Video und Partitur bei jeder zu spielenden Note oder jedem Akkord (im Folgenden als Event bezeichnet) stehen, bis diese(r) vom Nutzer gespielt wurde. Erst dann läuft der Player zum nächsten Event weiter. Die Fehlertoleranz bei der Erkennung des erwarteten Events sollte dabei möglichst niedrig sein, um sicher zu stellen, dass der Nutzer auch wirklich den richtigen Ton oder Akkord erlernt.

1.2 Anforderungen

Aus dem *flowkey*-Konzept ergeben sich drei zentrale Anforderungen, die es während des gesamten Entwicklungsprozesses zu berücksichtigen gilt und in den folgenden Abschnitten ausführlicher diskutiert werden.

- Da ein Tasteninstrument wie das Klavier ein polyphones Instrument ist, auf dem mehrere Töne gleichzeitig erklingen können, muss der Algorithmus eine entsprechende polyphone Erkennung der Schallereignisse beinhalten.
- Die Implementierung muss auf Basis einer webbasierten Technologie erfolgen, um die Applikation im Browser auszuführen zu können.
- Der Algorithmus muss in Echtzeit laufen, um die akustischen Eingaben des Nutzers zu verarbeiten und direktes Feedback zu ermöglichen.

1.2.1 Verarbeitung polyphoner Schallereignisse

Da die *flowkey*-Applikation es ermöglicht, Klavierstücke zu erlernen, erfordert sie die Möglichkeit neben einzelnen Tönen auch Akkorde zu erkennen, die mit einer oder mehreren Händen gespielt werden. Da ein Akkord als ein simultanes Erklingen von mehreren Tönen beschrieben werden kann, reicht eine Erkennung der Grundfrequenz (*Pitch Detection*) des vom Nutzer erzeugten Audiosignals nicht aus. Durch solch eine *Pitch Detection* kann nur auf die größte Wellenlänge (die Grundfrequenz) im Signal geschlossen werden, nicht aber auf mehrere Grundfrequenzen verschiedener Töne gleichzeitig. Dies führt dazu, dass sie nur für monophone Audiosignale geeignet ist. Da das Auftreten von Akkorden aber eine polyphone Erkennung erfordert, wird in dieser Arbeit die Verwendung einer sogenannten *Pitch-Chroma*-Extraktion vorgeschlagen.

Ein *Pitch-Chroma* ist ein 12-elementiger Vektor, der die Energie eines jeden der 12 Halbtöne (C, C#, D, ..., B) innerhalb eines Zeitintervalls auf jeweils einen Wert abbildet. Die Chroma-Extraktion bildet die Basis von zahlreichen Arbeiten bezüglich Akkorderkennung [5] [6] [7]. Daneben werden sie auch in anderen Disziplinen als Zwischenschritt benutzt, etwa bei der Erkennung von Coversongs oder anderen Audio-Matching-Anwendungen [7]. Dabei gibt es verschiedene Wege, diese zu berechnen und in bestimmter Hinsicht zu modifizieren. So kann die Chroma-Extraktion entweder auf dem Spektrum einer Fourier-Transformation ausgeführt werden oder lediglich im Zeitbereich mittels einer Filterbank berechnet werden. Verschiedene Vor- und Nachbearbeitungsschritte wie

etwa geeignete Fensterung, Überlappung (Vorverarbeitung) oder geeignete Normalisierung sowie Berücksichtigung der logarithmischen Wahrnehmung von Lautstärken (Nachbearbeitung) erlauben es, die Chroma-Extraktion auf verschiedene Anforderungen hin zu optimieren.

1.2.2 Webbasierte Implementierung für den Browser

Da die *flowkey*-Applikation zunächst als Webanwendung implementiert und zur Verfügung gestellt werden soll, ist die Auswahl an Technologien auf solche beschränkt, die im Umfeld von webbasierten Plattformen angesiedelt sind. Das schließt zum großen Teil solche Technologien und Programmiersprachen aus, die bereits etabliert im Bereich der Signalverarbeitung sind. Ein Großteil der Softwaremodule im Bereich der Signalverabeitung und Musikanalyse wird in der Programmiersprache C++, einer Erweiterung von C, entwickelt. Diese kann für verschiedene Plattformen seperat kompiliert und somit optimiert werden. Solche sogenannten nativen Implementierungen resultieren in hoher Geschwindigkeit. Auch native Anwendungen für die mobile Plattform iOS werden in einer Variante der Programmiersprache C, nämlich Objective C, geschrieben. Webbasierte Technologien dagegen bieten den Vorteil, dass sie nicht auf ein bestimmtes Betriebssystem zugeschnitten sind, sondern als Laufzeitumgebung einen Webbrowser nutzen. Eine neue HTML5-Programmierschnittstelle in Form der Web Audio API wird bereits von ersten Browsern implementiert und erlaubt die webbasierte Synthetisierung und Analyse von Audiomaterial. Im Kapitel 2 wird näher auf dieses Thema eingegangen. Insbesondere werden dort die Gründe für die Verwendung der HTML5 Web Audio API erläutert.

1.2.3 Echtzeit-Verarbeitung

Der hier implementierte Algorithmus stellt ein Online-Verarbeitungssystem dar, was heißt, dass er die akustischen Signale in Echtzeit verarbeitet. Dies erfodert eine möglichst geringe Latenz des Systems. Die Zeit zwischen dem Eingang des Signals und der Antwort des Systems muss also so klein sein, dass es für den Benutzer kaum merkbar ist. Im Gegensatz zu Offline-Algorithmen, die bereits bestehende digitale Aufnahmen verarbeiten, behandeln Online-Algorithmen kontinuierliche, unbekannte und unvorhersehbare Datenströme [8]. Relativ zum aktuellen Sample oder Block aus Samples können nachfolgende Samples also nicht mit in eine Berechnung herangezogen werden, da sie schlicht noch nicht existieren. Dies resultiert in weiteren Limitierungen von Online-Algorithmen im Vergleich zu jenen, die offline arbeiten.

1.3 Überblick *Score-Following*

In diesem Abschnitt sollen einige grundlegende Konzepte erläutert werden, um eine Vorstellung von Möglichkeiten und Grenzen aktueller *Score-Following*-Technologie zu vermitteln. *Score-Following* als Synchronisierung zwischen Computer und Musiker bezüglich einer bekannten Partitur hat mittlerweile eine Geschichte von etwa 30 Jahren [4]. *Score-Following* ermöglicht es, den Zuhörern eines Konzertes die Position in der Partitur anschaulich zu machen oder erlaubt dem Musiker, der Partitur ohne Umblättern zu folgen. Ein anderes Anwendungsszenario beinhaltet das Hinzufügen eines virtuellen Musikers zu einer Live-Aufführung, welcher auch ohne ein festgelegtes Tempo in der Lage ist, eine Begleitung zu spielen.

Eine grundlegende Motivation des *Score-Following*s ist also die Idee, eine Aufführung eines menschlichen Musikers mit synthetisierten Tönen eines virtuellen Musikers zu begleiten. Dieses Problem kann auch als automatische musikalische Begleitung in Echtzeit beschrieben werden [4]. Drei grundlegende Probleme werden dabei von Dannenberg [1] beschrieben: (1) Erkennen und Verarbeiten des Inputs des Musikers, (2) Abgleich mit einem erwarteten Input aus einer Partitur und (3) Generieren der Zeitinformation, welche zur Erzeugung der musikalischen Begleitung benötigt wird. Weiterhin sollte solch ein Score-Follower eine Toleranz gegenüber Fehlern in der Solo-Aufführung bzw. in der eigenen Erkennenungsleistung mitbringen. Dannenberg beschreibt die Erkennungsleistung als Zuordnung von erwarteten Events aus der Partitur zu vom Musiker produzierten Events. Das Ergebnis der Zuordnung kann einem virtuellem Musiker übergeben werden, der etwa in Form eines Synthesizers das Stück begleiten kann (siehe Abbildung 1.2).

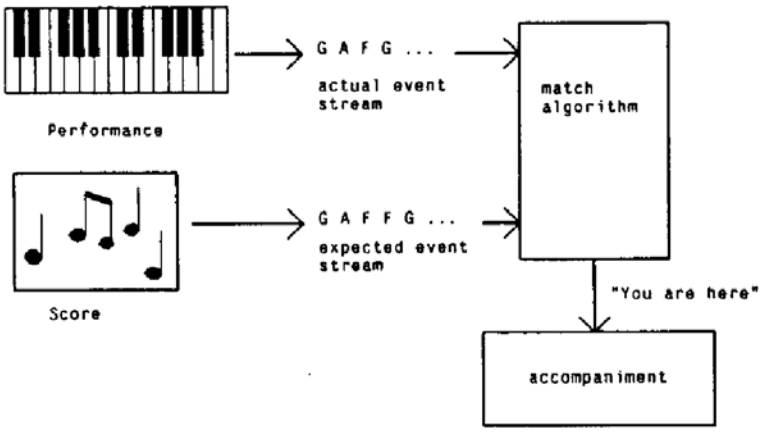

Abbildung 1.2: Grundlegendes Konzept des *Score-Following*s nach [1]: Ableich von erwarteten und tatsächlichen Events mittels eines Matching Algorithmus zum Begleiten eines Stücks

Die beste Zuordnung stellt für Dannenberg jene dar, bei der die längste gemeinsame Teilfolge zwischen den beiden Inputs erreicht wird. Er präsentiert einen Algorithmus, der nach jedem Event seitens des Musikers eine neue Spalte in einer Matrix berechnet, in der die erwarteten Partitur-Events zeilenweise und die vom Musiker erzeugten Events spaltenweise abgebildet sind. In der Matrix werden die maximal gemeinsamen Längen berechnet. Wann immer eine Zuordnung zur Folge hat, dass sich die maximale Länge erhöht, wird angenommen, dass der Musiker übereinstimmend mit einem Event in der Partitur ist, welches ihm anhand der Position in der Partitur zurückgemeldet wird. Implementiert wurde das System mit Hilfe eines 8-bit Mikrocomputers, der eine Echtzeit-Tonhöhenerkennung (monophon) für die Trompete bereitstellt und einen einstimmigen Sound als Begleitung synthetisiert. Dannenbergs Ansatz ist erfolgreich hinsichtlich der Tempounabhängigkeit des Musikers und funktioniert auch mit herausgelassenen beziehungsweise zusätzlich einfügten Noten.

Schwarz et al. schreiben 2004 in [9], dass *Score-Following* mit dem Audiosignal als direkten Input für Instrumente mit hoher Polyphonie (Piano, Orgel, Gitarre) noch schwierig zu realisieren sei. Da Instrumente wie das Piano oft einen Midi-Output bereitstellen, schlagen sie einen *Score-Following*-Algorithmus auf Basis eines Hidden Markov Modells (HMM) vor, der als Input ein akustisches Salienz-Maß verwendet, welches aus Midi-Daten des Instruments vorausberechnet wird. Ein HMM kann als Generator von Sequenzen verstanden werden [10]. Es ist ein endlicher Zustandsautomat, dessen aktueller Zustand jedoch verdeckt ist. Bei jeder Zustandsänderung von i nach j wird ein Merkmalsvektor y_t mit einer bestimmten Wahrscheinlichkeitsdichte $b_j(y_t)$ erzeugt. Der Übergang zwischen zwei Zuständen beruht dabei zudem auf einer bestimmten Wahrscheinlichkeit a_{ij}. HMMs werden so zur Beschreibung und Modellierung von Zufallsprozessen eingesetzt. Schwarz et al. [9] setzen ein 2-Level-HMM ein, welches ein *Low-Level* Notenmodell und ein *High-Level* Partiturmodell darstellt. Auch andere Arbeiten ([10]) benutzen Hidden Markov Modelle zur Umsetzung eines statistischen Ansatzes.

Die Dichtefunktionen sowie die Übergangswahrscheinlichkeiten eines HMMs müssen trainiert werden, um die Qualität des darauf basierenden Score-Followers zu optimieren. Dazu benötigt man für gewöhnlich gelabelte Sets mit Trainingsdaten. Im Fall von Schwarz et. al [9] werden verschiedene Aufnahmen bzw. generierte Aufführungen eines Stücks mit unterschiedlichen Fehlern verwendet.

In [11] präsentieren Arzt et al. einen Algorithmus für das Automatische Umblättern von Notenpartituren via *Real-Time Machine Listening*. Die Arbeit behandelt einen *Score-Following*-Algorithmus für Tasteninstrumente, der ohne das Training von HMMs, Neuralen Netzen oder dergleichen auskommen soll. Das *Score-Following* basiert auf einem

aus früheren Arbeiten bekannten *Dynamic Time Warping* (DTW) Algorithmus [12]. Die zu folgende Notenpartitur wird per Midi zu Audio synthetisiert. Auf dem synthetisierten Audiomaterial und dem ankommenden Audiosignal des Pianisten wird der DTW-Algorithmus ausgeführt. Im Vergleich zum bereits bekannten DTW-Algorithmus beschreiben Arzt et al. drei verschiedene Strategien zur Verbesserung. Die erste ist die sogenannte *Backward-Forward Strategy*. Hier wird sowohl die aktuelle Hypothese als auch die Information aus der sie konstruiert wurde benutzt, um vergangene Entscheidungen ggf. zu korrigieren und dieses Ergebnis wiederum in die Konstruktion der aktuellen Hypothese miteinfließen zu lassen. Die zweite Strategie basiert auf der Berücksichtigung von musikalischen Informationen. Damit beziehen sich die Autoren auf das Vorhandensein der Midi-Informationen, aus denen das synthetisierte Audiomaterial stammt. Besonders die darin enthaltenen Onset-Zeiten sind besonders nützlich, um abzuschätzen, ob das aktuell anliegende Audiosignal dem erwarteten Audiosignal entspricht. Die dritte Strategie besteht darin, mehrere Hypothesen gleichzeitig beizubehalten. Dies soll darauf abzielen, strukturelle Änderungen der Live-Performance im Vergleich zum synthetisierten Audiomaterial zu kompensieren, etwa wenn der Spieler ein Wiederholungszeichen ignoriert.

Während es heutzutage kein Problem mehr ist, verschiedensten Instrumenten mit monophonen Charakter zu folgen, so ist das Folgen von polyphonen Instrumenten oder gar Gruppen von Instrumenten noch immer eine Herausforderung, die die aktuellen und neuen Ansätze in dieser Disziplin jedoch lösen können. Die Arbeit von Arzt et al. zeigt auch, dass auch ohne statistische Methoden, deren Nachteil das benötigte Training ist, ein zuverlässiges *Score-Following* realisierbar ist. Weiterhin geht der Forschungsschwerpunkt dahin, *Score-Following*-Programme verlässlicher und benutzerfreundlicher zu gestalten, um sie auch ohne Anwesenheit eines Experten bei Live-Aufführungen einsetzen zu können [4].

1.4 Abgrenzung zum klassischen *Score-Following*

Im Folgenden wird erläutert, was den hier vorgestellten *Score-Following*-Algorithmus von früheren Arbeiten unterscheidet. Die in 1.3 vorgestellten Arbeiten zielen auf andere Anwendungsszenarien des *Score-Following*s ab als die *flowkey*-Webapplikation. Während die *flowkey*-Webapplikation in erster Linie eine Hilfestellung beim Lernen eines Stücks geben soll, konzentrieren sich frühere Arbeiten auf das Folgen eines Stücks, welches von einem professionellen Musiker gespielt wird. Dies soll etwa eine computergenerierte Begleitung ermöglichen. Intention des *flowkey*-Score-Followers ist es jedoch nicht, eine Begleitung zu spielen, sondern ein Feedback darüber zu geben, ob der Nutzer die richtigen

Töne trifft. Video und Partitur werden zum erklungenen Schallereignis entsprechend synchronisiert oder nicht. Dies soll das Lernen eines Klavierstücks anhand von Video und Partitur vereinfachen.

Daraus resultiert eine umgekehrte Herangehensweise hinsichtlich der Fehlertoleranz der beiden Szenarien. Während die *flowkey*-Applikation möglichst keine Fehler zulassen sollte, damit der Nutzer das Stück auch richtig lernt, sollte das Folgen einer Aufführung eines Musikers doch einen gewissen Grad an Toleranz mitbringen, um Abweichungen und zusätzlich eingefügte oder weggelassene Noten zu kompensieren.

Während frühere Arbeiten Ansätze wie das *Dynamic Time Warping* oder *Hidden Markov Modelle* nutzen, um eine Sequenz von Events zuzuordnen, betrachtet diese Arbeit jedes Event für sich selbst. Grund dafür ist wiederum das oben beschriebene Anwendungsszenario des Lernens eines Stücks, da man davon ausgehen kann, dass ein Nutzer der *flowkey*-Applikation die Warte-Funktion (siehe 1.1) nutzt, um die dargestellten Töne und Akkorde Schritt für Schritt zu lernen. Weil der Nutzer weiterhin jede beliebige Position auf der virtuellen Partitur ansteuern kann, wird stets nur das nächste erwartete Event in Bezug zur ausgewählten Position analysiert. Erklingt das entsprechende User-Event, läuft der Player zum nächsten erwarteten Event und übergibt dies dem Score-Follower für die weitere Analyse.

1.5 Lösungsansatz

Grundlage des hier vorgestellten Verfahrens bildet die Analyse des Audiosignals anhand der Extraktion von Chroma-Vektoren, auch *Pitch Class Profile* genannt. Ein Chroma-Vektor besteht aus 12 Pitchklassen, die den Anteil der Energie eines jeden Halbtons im aktuellen Signalabschnitt abbilden. Die Chroma-Berechnung erfolgt auf Basis von Kurzzeitspektren. Diese entstehen wiederum durch eine Fourier-Transformation von Signalblöcken (Short Time Fourier Transformation) aus dem Zeit- in den Frequenzbereich. Auf dieser Grundlage wird auch eine Onset-Erkennung durchgeführt, um den Zeitpunkt eines Anschlages auf dem Tasteninstrument zu bestimmen. Nach einem erkannten Onset wird ein über die Zeit gemittelter Chroma-Vektor berechnet und mit den erwarteten Noten abgeglichen. Eine nähere Betrachtung von Fourier-Transformation, Onset-Erkennung und Chroma-Extraktion wird im Kaptiel 3 gegeben. Anhand einer Vergleichsfunktion wird das Vorhandensein der erwarteten Pitchklassen im Chroma-Vektor bestimmt. Treten alle erwarteten Pitchklassen im Chroma-Vektor hinreichend auf, laufen Partitur und Video

zur nächsten Stelle weiter, an der ein oder mehrere Töne erklingen sollen. Die Umsetzung erfolgt durch die Verwendung von Webtechnologien wie der Web Audio API und der Implementierung des Algorithmus in JavaScript. Dabei wird darauf geachtet, dass die Verarbeitungsgeschwindigkeit stets so gering gehalten werden muss, dass der Algorithmus die Bedingungen einer Echtzeit-Anwendung erfüllen kann.

1.6 Zusammenfassung

In diesem Kapitel wurden die zentralen Anforderungen an diese Masterarbeit vorgestellt. Das Konzept der *flowkey*-Webapplikation resultiert erstens in der Anforderung, dass neben einzelnen Tönen auch Akkorde erkannt werden müssen. Hier ließ sich aufgrund von bereits bestehenden Arbeiten die Methodik der Chroma-Extraktion als probates Mittel zur Lösung dieses Problems umreißen. In Form der Web Audio API steht eine moderne Webtechnologie bereit, die der zweiten Anforderung eines webbasierten Algorithmus gerecht wird. Als dritte Anforderung wird die Echtzeitverarbeitung angeführt. In diesem Zusammenhang gilt es, einen möglichst effizienten Algorithmus zu implementieren und den resultierenden JavaScript-Quellcode auf Performanz zu optimieren, um die Eigenschaften eines Echtzeitsystems zu erfüllen.

Während es bei traditionellen *Score-Following*-Applikationen erwünscht ist, wenn der Algorithmus Fehler seitens des Benutzers kompensieren kann, soll der hier zu implementierende Algorithmus Fehler des Benutzers erkennen und der Partitur in diesem Fall nicht folgen. Eine weitere Eigenschaft, die sich aus dem Konzept ergibt ist, dass der Algorithmus im Unterschied zu vorherigen Arbeiten auf dem Gebiet keine längste Teilfolge von Noten bestimmen muss, sondern lediglich das aktuelle bzw. nächste erwartete Event betrachtet (vergleiche dazu 1.3).

Kapitel 2

Signalverarbeitung und Webtechnologien

Mit der immer größeren Verbreitung von leistungsfähigen Tablet-PCs und der zunehmenden Verlagerung von Anwendungen von der eigenen Festplatte hin zu Online-Diensten vergrößert sich die Möglichkeit, Dienste und Anwendungen abseits des klassischen Desktop-Computers zu nutzen. So wächst die Menge an Technologien, die es möglich machen, auf tragbaren Geräten multimediale Inhalte wiederzugeben und zu verarbeiten. Eine große Rolle spielen dabei auch webbasierte Technologien rund um den aktuellen HTML5-Standard. Auch mit älteren webbasierten Plattformen oder Programmiersprachen ist es möglich, Prozesse der Signalverarbeitung im Browser auszuführen. In diesem Kapitel sollen verschiedene Technologien, Plattformen und Programmiersprachen vorgestellt und die Auswahl einer davon für diese Arbeit begründet werden. Daneben sollen die Eigenschaften der Echtzeit-Signalverarbeitung und damit verbundene Aspekte im Kontext dieser Arbeit erläutert werden.

2.1 HTML5 Web Audio

HTML ist die Kernsprache des World Wide Web und wird vom W3C (World Wide Web Consortium) nun in seiner fünften Version spezifiziert. In erster Linie ist HTML eine Auszeichnungsprache, um Webdokumente semantisch zu beschreiben. Ursprünglich für das Erstellen von wissenschaftlichen Arbeiten konzipiert, wurde es auch für die Erstellung von anderen Dokumenten wie Webseiten, wie man sie heute kennt, adaptiert [13]. Zur Erstellung von modernen Web-Applikationen brauchte es eine lange Zeit zusätzliche Technologien, um vor allem Multimediainhalte oder 2D/3D-Graphiken darstellen zu können [14]. Die neuste Version von HTML5 soll sich diesen Problemen widmen, indem

zum Beispiel Elemente zur direkten Einbindung von Video und Audio spezifiziert und dazugehörige Programmierschnittstellen (APIs) bereitgestellt werden.

Eine dieser Programmierschnittstellen im Umfeld von HTML5 ist die Web Audio API [15], auf welche per JavaScript zugegriffen werden kann. Sie dient dazu, Audiomaterial im Browser zu verarbeiten und/oder zu synthetisieren. Die Schnittstelle erlaubt es auf eine Reihe von typischen Komponenten der Signalverarbeitung zurückzugreifen. Die Schnittstelle besitzt einen modularen Ansatz, in dem sich Komponenten (*Audio Nodes*) wie Oszillatoren, Filter, Kompressoren oder Analysewerkzeuge zu einer Signalverarbeitungskette zusammenfügen lassen. So lassen sich virtuelle Instrumente oder Effektgeräte zusammenbauen und im Browser nutzen, wie in Abbildung 2.1 beispielhaft dargestellt[1]. Zu sehen ist der Live-Input eines Mikrofons, welcher auf zwei unterschiedlichen Routen mit je einem *Biquad Filter Node* (oben: *highpass*, unten: *highshelf*) gefiltert wird. Das Signal wird anschließend jeweils durch einen *Analyser Node* geschickt, welcher dessen Kurzzeitspektrum berechnet und es sonst unverändert lässt. Beide Routen sind mit der Ausgabe (Lautsprecher-Symbol: *audiocontext.destination*) verbunden. Die Berechnungen der Ausgangssignale jedes Knotens finden in der zugrundeliegenden Implementierung des jeweiligen Browsers statt, welche typischerweise in optimiertem C/C++ oder Assembler-Code geschrieben ist. Jedoch ist auch die Verwendung von JavaScript-Code möglich, um eigene *Audio Nodes* zu erstellen [15].

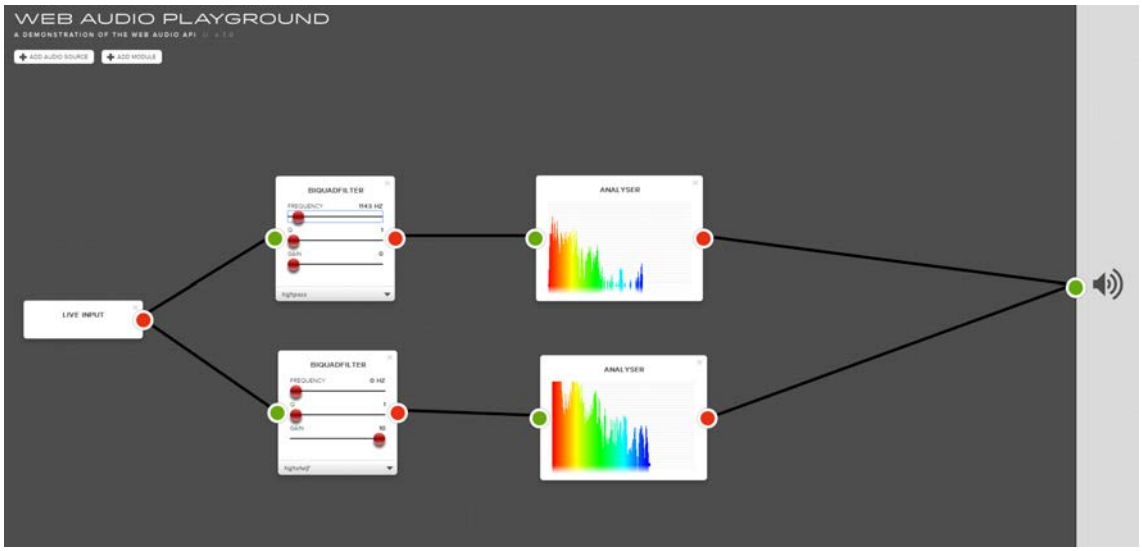

Abbildung 2.1: Beispielgraph einer Signalverarbeitungskette erzeugt mit der *Web Audio Playground*-Applikation von Chris Wilson.

[1] `http://webaudioplayground.appspot.com/`

Eine weitere Neuerung von HTML5 ist die Möglichkeit, anhand der *getUserMedia()*-Schnittstelle[2] auf das Mikrofon und/oder die Kamera des Benutzers zuzugreifen. Sie stellt einen Stream bereit, der verarbeitet werden kann. Die Schnittstelle kann eigenständig oder ergänzend zur Web Audio API verwendet werden.

Die sich noch in der Entwicklung befindende und bisher vom Chrome- und Firefox-Browser unterstützte Web Audio API bringt bereits eine Reihe von Werkzeugen mit sich, mit der es möglich ist, komplexere Anwendungen im Bereich der Signalanalyse und -Synthese umzusetzen, ohne das Rad neu erfinden zu müssen. Während die Schnittstelle auch eine *Analyser*-Komponente mitbringt, die mittels Fourier-Transformation ein Spektrum berechnet, müssen andere Arten von Analyse-Werkzeugen allerdings immer noch selbst in JavaScript implementiert werden. So basieren in dieser Arbeit benötigte Features wie die Chroma-Extraktion sowie die Onset-Erkennung auf einer eigenen Implementierung in JavaScript.

JavaScript war ursprünglich dazu gedacht, Webseiten dazu zu befähigen, kleine Änderungen an ihrer Struktur oder am Inhalt vorzunehmen. Vor einigen Jahren war JavaScript eine von vielen Webtechnologien, welche Funktionen ermöglichten, die über die von von herkömmlichen statischen Webseiten hinausgingen. Es existierten etwa noch Java-Applets[3] und Flash, deren Fähigkeiten denen von JavaScript oft noch überlegen waren. Mittlerweile gibt es zahlreiche Frameworks, Bibliotheken und APIs, die JavaScript zur relevantesten Webtechnologie machen, zumal es über viele Browser hinweg standardisiert wird [16]. JavaScript kann im Wesentlichen als eine Implementierung des ECMAScript-Standards angesehen werden. Dieser wird von der Ecma International Organisation (früher European Computer Manufacturers Association) in der ECMA-262-Spezifikation herausgegeben. Der aktuelle Standard ECMAScript 5 existiert seit 2011. Der kommende Standard wird JavaScript als Sprache noch deutlicher als in der fünften Version um- bzw. auszubauen.

Die Verwendung der HTML5-Schnittstellen macht es möglich, auf JavaScript-Bibliotheken zurückzugreifen und diese innerhalb des Algorithmus einzubinden. Diese können neben den Standard-Funktionen der Web Audio API benutzt werden, um eigene *Audio Nodes* in JavaScript zu erstellen. Folgende Bibliotheken stellen dabei potentiell nützliche Werkzeuge dar:

[2] `http://www.w3.org/TR/mediacapture-streams/`

[3] Man beachte, dass Java und JavaScript zwei völlig unterschiedliche Programmier-Sprachen sind

- *fft.js* [4] ist eine JavaScript-Implementierung einer Fast Fourier Transformation. Verwendet wird *fft.js* etwa in der JavaScript-Bibliothek *pitch.js* [5], welche zur Erkennung der Grundfrequenz von Gesang oder Musikinstrumenten dient.
- Eine Bibliothek mit einer umfangreicheren Sammlung von Werkzeugen ist *audiolib.js* [6] [7]. Neben der FFT-Funktionalität bietet sie etwa Zugriff auf verschiedene Filtertypen und einer Reihe von digitalen Effekten.
- Eine andere Sammlung von Werkzeugen der Signalverarbeitung bietet *DSP.js* [8]. Auch sie bietet FFT/DFT-Funktionalität, verschiedene Fensterfunktionen, Effekte und Filter.

Doch nicht nur auf der Client-Seite, auch serverseitig spielt JavaScript mittlerweile eine immer größere Rolle. Frameworks wie *Node.js* ermöglichen es, JavaScript-Code serverseitig auszuführen, um dynamische Webseiten zu erstellen. Bisher wurde dafür etwa die Skriptsprache PHP benutzt. Der Vorteil besteht etwa darin, dass die gleichen Software-Module sowohl server- als auch clientseitig eingesetzt werden können und so eine höhere Wiederverwendbarkeit von Quellcode ermöglicht wird. Funktionen, die auf beiden Seiten gebraucht werden, müssen nur noch in einer Sprache implementiert werden. Zudem ist der Datenaustausch zwischen Server und Client aus dem selbem Grund einfacher zu handhaben. Die *flowkey*-Applikation verwendet das Meteor-Framework, welches wiederum auf Node.js basiert. Obwohl das kein Ausschlusskriterium für die Verwendung anderer Technologien auf der Client-Seite ist, sind die genannten Vorteile von serverseitigem JavaScript ein weiterer Grund, den *Score-Following*-Algorithmus ebenfalls mit dieser Sprache zu realisieren.

2.2 Flash

Flash ist eine Plattform, die das Erstellen von (interaktiven) Animationen für den Browser ermöglicht. Es ist ein proprietäres Produkt, anfangs von Macromedia, nun von Adobe entwickelt. Dies ermöglicht unter anderem auch die Ausführung von sogenannten *Rich Internet Applications* (RIAs). RIAs sind im Wesentlichen moderne Webanwendungen, deren Design und Benutzbarkeit über das von normalen Webseiten hinausgeht. So verwenden RIAs beispielsweise Steuerungselemente wie man sie von Desktopanwendungen

[4] https://github.com/JensNockert/fft.js
[5] https://github.com/audiocogs/pitch.js
[6] http://audiolibjs.org/
[7] http://docs.audiolibjs.org/
[8] https://github.com/corbanbrook/DSP.js

kennt (*Dropdown*-Menüs, *Drag & Drop*-Funktionalität). Sie arbeiten generell nicht seitenbasiert wie HTML, sondern reagieren direkt auf Eingaben, ohne eine Seite neuzuladen. Außerdem sind RIAs oft auch in der Lage, Daten vom Server ohne Zutun des Benutzers abzufragen [17].

Adobe Flash als Plattform erlaubt es, Vektorgrafiken, Rastergrafiken und Videoclips darzustellen und zu animieren, wodurch eben jene Eigenschaften von *Rich Internet Applications* realisiert werden können. Hinzu kommt die Unterstützung von Video- und Audio-Streaming sowie die Möglichkeit, Input von Tastatur, Maus, Mikrofon und Webcam zu verarbeiten. Besonders der Zugriff auf das Mikrofon eines Gerätes macht Flash zu einer Option, wenn es um die Plattform-Auswahl für den *flowkey-Score-Following*-Algorithmus geht. Im Gegensatz zu HTML5-Technologien benötigt man zur Ausführung von Flash-Anwendungen ein Browser-Plugin, den Flash Player, der extra installiert werden muss.

Die Sprache, in der Flash-Anwendungen geschrieben werden, ist ActionScript. Sie ist eine objekt-orientierte Sprache und ein Dialekt des ECMA-Script-Standards. Mit ihr lassen sich auch komplexere Anwendungen realisieren, in der der Client etwa eine Datenbankabfrage mittels HTTP an eine Serveranwendung absetzen kann. Auf Webseiten kann ActionScript (bei installiertem Flash Player) in Form von eingebetteten swf.-Dateien benutzt werden. DSP-Bibliotheken für ActionScript existieren unter andererem in Form der *Audio Processing Library for Flash (ALF)* [18] [19]. Diese implementiert unter anderem eine FFT und diverse Features wie den *Spectral Flux* oder *Spectral Rolloff*.

Das Browser-Plugin in Form des Flash Players ist weit verbreitet. Adobe gibt an, dass 95% der Browser über den Player verfügen [20]. Flash wird aber zunehmend durch andere Standards wie HTML5 verdrängt [21]. Zunehmend wird Flash nicht mehr für die Realisierung ganzer Seiten, sondern gezielt als Gestaltungsmittel eingesetzt [20]. Ein Hauptgrund, der gegen die Flash-Technologie spricht, sind häufige Updates des Flash Players seitens Adobe, um immer wieder auftauchenden Sicherheitslücken entgegen zu wirken. Obwohl dies für den hier zu implementierenden Algorithmus kaum eine Rolle spielt, sollte dies Problem hinsichtlich der zukünftigen Veröffentlichung der *flowkey*-Applikation berücksichtigt werden.

2.3 Silverlight

Ähnlich wie Adobes Flash zielt das von Microsoft entwickelte Silverlight darauf ab, eine Plattform zur Entwicklung von *Rich Internet Applications* bereitzustellen. Gleichzeitig soll der Entwicklungsprozess von Webapplikationen deutlich schneller und kostengünstiger von statten gehen.

Silverlight steht sowohl für Windows als auch OS X bereit und unterstützt alle relevanten Browser. In Form des sogenannten Moonlight gibt es auch eine Open-Source-Implementierung, welche von Novell in Kooperation mit Microsoft entwickelt wurde und Silverlight auch auf Linux-Systemen ausführbar machen soll. Nichtsdestotrotz ist Silverlight nicht so verbreitet wie Adobes Flash [22].

Silverlight-Anwendugen können in jeder Sprache geschrieben werden, die Microsofts .NET-Framework angehören, zum Beispiel C#, *Visual Basic* oder J#, auch JavaScript wird unterstützt [23]. Eine gemeinsame Schnittmenge an Datentypen soll für einen reibungslosen Ablauf sorgen, wenn eine in einer bestimmten Sprache geschriebenen Komponente eine in einer anderen Sprache entwickelten Komponente aufruft. Dies soll in erster Linie für Flexibilität sorgen. Dies kann bei einem großen Projekt unter Umständen jedoch auch zu einem erhöhten Aufwand bei der Wartung führen. Mit Hilfe von Entwicklungsumgebungen wie Lightswitch oder VisualStudio sollen Design- und Entwicklungsprozess miteinander verschmelzen [17] und so vorrangig Business- und Multimedia-Anwendungen implementiert werden.

Für das Ausführen der Anwendungen benötigt man wie bei Flash ein Browser-Plugin. Aktuell steht Silverlight in der Version 5 bereit. Seit Version 4 unterstützt es auch den Zugriff auf Mikrofon und Webcam, was Silverlight ebenfalls zu einer möglichen Alternative für die *flowkey*-Webapplikation macht.

Aufgrund der gemischtsprachigen Programmierung bietet Silverlight auch die Möglichkeit, alle in den entsprechenden Sprachen geschriebene Bibliotheken zu nutzen, um etwa Fourier-Transformationen auszuführen oder andere Verfahren im Bereich der Signalverarbeitung zu nutzen. Das *SaluseMediaKit*[9] stellt etwa einen MP3-Dekoder mit diversen Audioeffekten für Echtzeitverarbeitung bereit. Mittlerweile rückt Microsoft von der Weiterentwicklung von Silverlight ab. Eine sechste Version des Silverlight Frameworks wird es nicht mehr geben. Die aktuelle Version wird noch bis 2021 unterstützt werden [24].

[9] `http://salusemediakit.codeplex.com/`

2.4 Auswahl einer Technologie

Für den hier vorgestellten *Score-Following*-Algorithmus werden HTML5-Technologien wie die Web Audio API benutzt. Diese Programmierschnittstelle ist explizit für Anwendungen im Audiobereich ausgelegt und bringt bereits eine Grundausstattung von virtuellen Signalverarbeitungskomponenten mit. Die Tatsache, dass diese in optimiertem maschinennahen Code implementiert sind, spricht für eine potentiell gute Performanz. Eigene Signalverarbeitungskomponenten lassen sich in JavaScript implementieren. Mit der ebenfalls zu HTML5 gehörenden *getUserMedia()*-Schnittstelle gibt es neben der Web Audio API eine Schnittstelle, die den Zugriff auf das Mikrofon des Gerätes ermöglicht. Dazu muss vom Benutzer kein zusätzliches Browser-Plugin installiert werden.

Mit Flash und Silverlight ist es ebenso möglich, auf das Mikrofon des Benutzers zuzugreifen und dessen Signal (mit entsprechenden Bibliotheken) zu verarbeiten, allerdings muss für beide Technologien zusätzliche Software in Form eines Browser-Plugins installiert werden, was ein Mehraufwand für den Nutzer darstellt. Weiterhin sind beide Technologien weniger als die Web Audio API für Anwendungen im Bereich der Signalverarbeitung ausgelegt und bedürfen regelmäßiger Updates. Die Ankündigung Microsofts, Silverlight nur noch bis 2021 zu unterstützen, macht es als Plattform umso weniger attraktiv.

Ein weiterer Grund für den Verzicht auf Flash und Silverlight ist, dass das Unternehmen Apple, welches Marktführer im Tablet-Bereich ist, Flash nicht von Haus aus in seinem mobilen Betriebssystem iOS unterstützt. Mit Hinblick auf die Optimierung der *flowkey*-Webapplikation für Apple-Tablets erscheint auch in dieser Hinsicht die Wahl von HTML5 als die Beste. Da Google als erster Browserhersteller die Web Audio API mit Chrome unterstützt, wird wohl auch die Portierung auf Googles Android-Betriebssystem auf Basis einer HTML5-Lösung einfacher zu bewerkstelligen sein als auf Basis von Flash oder Silverlight. Weiterhin werden die in der *flowkey*-Applikation verwendeten Videos ebenfalls mit HTML5 eingebettet. Auch hier wurde sich also schon die Plugin-freie Alternative zu Nutze gemacht, um die Webanwendung weniger anfällig für Sicherheitslücken und Abstürze zu machen [25].

Bisher wird die Web Audio API lediglich von Chrome und Firefox unterstützt. Da sie aber eine vom W3C spezifizierte Schnittstelle ist, ist damit zu rechnen, dass auch andere Browserhersteller wie Apple und Opera eine Implementierung in absehbarer Zeit bereitstellen werden[10].

[10] Auskunft zur Kompabilität diverser Browser mit der Web Audio API gibt etwa die Seite `http://caniuse.com/audio-api`

2.5 Echtzeit-Signalverarbeitung

Im Bereich der digitalen Signalverarbeitung kann zwischen zwei Arten von Systemen unterschieden werden: Echtzeit-Systeme und Nicht-Echtzeit-Systeme. Die dazugehörigen Algorithmen werden entsprechend auch als Online- und Offline-Algorithmen bezeichnet. Nicht-Echtzeit-Systeme und deren Offline-Algorithmen verarbeiten Signale, die bereits determiniert und digital gespeichert sind. Die Verarbeitungszeit pro Signalabschnitt kann deutlich über der zeitlichen Länge des Signalabschnitts liegen. Dagegen setzt die Verarbeitung von Signalen in Echtzeit eine bestimmte Eigenschaft des Analyse-Systems voraus: Echtzeitsysteme müssen ihre Antwort innerhalb einer strikten Zeitvorgabe liefern. Diese zeitliche Limitierung resultiert in einer maximal möglichen Samplerate. Je nach Geschwindigkeit eines Systems können mehr oder weniger Samples pro Zeiteinheit verarbeitet werden. In jedem Fall muss bei Echtzeitsystemen die Verarbeitungszeit pro Sample kleiner als das Zeitintervall zwischen zwei Samples sein. Ist dies nicht der Fall, würde es zu einer Art Datenstau kommen der stetig anwächst, da das System nicht mit der Verarbeitung hinterherkommt.

In einem Echtzeit-System, das jedes Sample einzeln verarbeitet und es dann ausgibt, sobald am Eingang ein neues Sample anliegt, besitzt eine Verzögerung, die genau einem Sampleintervall T entspricht. Das System muss die Verarbeitung innerhalb dieses Intervalls abgeschlossen haben [26]. Zusätzlich muss auch die Zeit t_0 berücksichtigt werden, die für Ein- und Ausgabe-Operationen benötigt wird. Mit der Verarbeitungszeit t_S für ein Sample ergibt sich für Echtzeit-Systeme folgende Bedingung [26]:

$$t_0 + t_S < T \tag{2.1}$$

Da ein Audiosignal bei der digitalen Analyse in Blöcken von Samples aufgeteilt wird, darf die rechnerische Verarbeitung eines Signalblocks nicht mehr Zeit in Anspruch nehmen, als ein Signalblock lang ist. Die Länge eines solchen Signalblocks gibt also die zeitliche Deadline vor. Hat ein Signalblock N Samples und ist die Zeit t_B die Verarbeitungszeit für einen Signalblock so ergibt sich:

$$t_0 + t_B < T \cdot N \tag{2.2}$$

Bei der digitalen Signalverarbeitung mit einer typischen Samplerate von 44100 Hz ($T \approx 0.02267$ ms) und einer Blocklänge von 2048 Samples ergibt sich eine zeitliche Blocklänge von rund 46 ms. In dieser Zeit (oder weniger) muss ein Signalblock verarbeitet und alle

Ein- und Ausgaben durchgeführt sein, bevor der nächste Signalblock zur Verarbeitung zur Verfügung steht. Im Gegensatz zu Algorithmen, die bereits determinierte Aufnahmen analysieren (Offline-Algorithmen), können Echtzeit-Algorithmen (Online-Algorithmen) nicht in die Zukunft blicken. Das heisst, dass Offline-Algorithmen, obwohl sie ebenfalls eine blockweise Verarbeitung durchführen, Parameter und Funktionen verwenden können, die auf der gesamten Aufnahme basieren. Oft sind Offline-Algorithmen deswegen robuster gegen verschiedene Veränderungen im Signal wie starke Peaks in der Amplitude oder gegen Rauschen.

2.6 Native Sprachen vs. JavaScript

Bei gängigen Anwendungen der Audiobearbeitungen wie digitalen Audioworkstations (DAWs) werden die Programme als native Anwendungen implementiert. Nativ meint in diesem Sinne, dass Quellcode für das jeweilige Betriebssystem angepasst werden kann und somit weniger rechnerischen Overhead verursacht wird als bei Programmen, die in höheren Programmiersprachen geschrieben und auf mehreren Systemen laufen können. Die wohl gängigste Programmiersprache für DSP-Anwendungen ist C. Ein C-Compiler kann aus C-Code Assembler-Code erzeugen, welcher wiederum in Maschinensprache übersetzt wird. Diese Maschinensprache enthält Befehle an den Prozessor als Bitmuster. Assembler-Code korrespondiert stark mit der Architektur des Prozessors, was in hoher Geschwindigkeit resultiert. Auch können Assembler-Module direkt in C/C++-Code eingebettet werden, um einzelne Operationen optimiert auszuführen [27].

Der hier implementierte Algorithmus ist webbasiert und stellt somit keine native Anwendung dar. Weder ist er für ein bestimmtes Betriebssystem noch für bestimmte Hardware optimiert. Die Programmiersprache JavaScript ist aus diesem Grund im Vergleich zu maschinennahen Sprachen als langsamer zu betrachten. Gleichzeitig werden JavaScript-Engines wie die V8-Engine von Google [11] allerdings in C++ implementiert. Diese kommt sowohl im Browser Chrome aber auch für serverseitiges JavaScript zum Einsatz.

Während der Implementierung des *Score-Following*-Algorithmus ist darauf zu achten, inwieweit sich dessen Performanz von der von nativen Anwendungen wegbewegt und ob es die Benutzbarkeit der *flowkey*-Applikation einschränkt. Gegebenfalls müssen weniger rechenintensive Verfahren gefunden werden, um die Anforderungen der Echtzeitverarbeitung erfüllen zu können.

[11] `https://code.google.com/p/v8/`

2.7 Zusammenfassung

In diesem Kapitel wurden zunächst drei verschiedene Webtechnologien betrachtet, die als Plattform für die Implementierung des *Score-Following*-Algorithmus in Frage kommen. Neben den neuesten Technologien in Form der Web Audio API und der *getUserMedia()*-Schnittstelle als HTML5-Standards, wurde auch das weit verbreitete Flash von Adobe sowie das dazu in Konkurrenz stehende Silverlight von Microsoft vorgestellt. Obwohl noch nicht so weit verbreitet wie Flash, wurde sich in dieser Arbeit für den HTML5-Standard entschieden, da dieser im Gegensatz zu den beiden anderen Technologien kein Browser-Plugin benötigt. Auch hinsichtlich der Eignung für das *Score-Following* ist die HTML5 Web Audio API gut aufgestellt, da sie eine Schnittstelle für bereits implementierte Komponenten im Bereich der Signalverarbeitung darstellt. Im zweiten Teil des Kapitels wurde erläutert, worauf es bei der Echtzeit-Signalverarbeitung ankommt. Ein kurzer Vergleich von der in HTML5 genutzten Sprache JavaScript mit nativen Sprachen wurde angestellt, um den Unterschied zwischen klassischen nativen Anwendungen und Webanwendungen hinsichtlich deren potentieller Performanz zu verdeutlichen.

Kapitel 3

Theoretische Grundlagen

Dieses Kapitel liefert einige theoretische Grundlagen bezüglich der im Rahmen dieser Arbeit implementierten *Score-Following*-Komponenten. Es werden u.a. die elementaren Komponenten der Arbeit in Form von Fourier-Transformation, Onset-Erkennung und Chroma-Extraktion mathematisch beschrieben.

3.1 Fourier-Transformation

Im Bereich der digitalen Signalverarbeitung ist die Fourier-Transformation Basis für unterschiedlichste Verfahren. Sie wird sowohl in der Telekommunikation, Medizintechnik, Geophysik oder in der Astronomie eingesetzt, um nur einige Beispiele zu nennen [28]. In dieser Arbeit stellt sie die Basis sowohl von Chroma-Extraktion als auch von Onset-Erkennung dar, weswegen im Folgenden die mathematischen Grundlagen der Fourier-Transformation geliefert werden.

Die Fourier-Transformation bildet ein Signal $x(t)$ vom Zeitbereich in den Frequenzbereich ab, erzeugt also dessen Spektrum $X(\omega)$. Dies geschieht durch Zerlegung des Signals in sinusförmige bzw. harmonisch-exponentielle Komponenten verschiedener Frequenzen [29]:

$$c_k e^{j\omega_k t}, c_k \in C \tag{3.1}$$

c_k ist dabei der Spektralwert zur Kreisfrequenz ω_k. Alle periodischen Signale können als Fourierreihe repräsentiert werden. Sie stellt eine Überlagerung aus gewichteten Sinus- oder Kosinusfunktionen dar und kann beschrieben werden mit:

$$x(t) = \sum_{-\infty}^{\infty} c_k e^{j\omega_k t}. \tag{3.2}$$

Die Grundfrequenz der Reihe bestimmt dabei ihre Frequenzauflösung. Um nicht-periodische Signale zu analysieren, lässt man die Periodendauer wachsen $T_0 \to \infty$. Equivalent dazu läuft die Grundfrequenz ω_0 gegen 0. Dies hat den Effekt, dass die ursprünglich diskrete Frequenzauflösung kontinuierlich wird mit $k\omega_0 \to \omega$. Durch die sich ergebende unendlich große Auflösung der Frequenz-Achse, vermindern sich die Koeffizienten $c_k \to 0$ [2]. Auf Basis der Gleichung 3.2 ändert sich die Fourierreihe also zur kontinuierlichen Fourier-Transformation 3.3.

$$X(\omega) = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} \mathrm{d}t \tag{3.3}$$

Andersherum ist es anhand der inversen Fourier-Transformation möglich, ein Signal vom Frequenzbereich zurück in den Zeitbereich zu transformieren, indem mit der Funktion $X(\omega)$ gewichtete sinus- und cosinusförmige Schwingungen überlagert werden, was schließlich in einem Zeitsignal resultiert [30].

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\omega) e^{j\omega t} \mathrm{d}\omega \tag{3.4}$$

Für zeitlich abgetastete (diskrete) Signale ist die Fourier-Transformation (DTFT: *Discrete-Time Fourier Transform*) wie folgt definiert:

$$X(\Omega) = \sum_{n=-\infty}^{\infty} x(n) e^{-j\Omega n} \tag{3.5}$$

Die inverse Fourier-Transformation diskreter Signale ist demnnach:

$$x(n) = \frac{1}{2\pi} \int_{2\pi} X(\Omega) e^{j\Omega n} \mathrm{d}\Omega \tag{3.6}$$

Während Gleichung 3.5 das diskrete Signal auf ein kontinuierliches Spektrum abbildet, bildet die sogenannte Diskrete Fourier Transformation (DFT: Gleichung 3.7) ein diskretes Zeitsignal auch auf ein diskretes Spektrum ab.

$$X(\Omega) = \sum_{n=0}^{M-1} x(n) e^{-j\Omega n/M}. \tag{3.7}$$

Die DFT setzt eine endliche Länge M des Signals $x(n)$ mit einer Samplingfrequenz f_s voraus. Die Anzahl der sich ergebenden Stützstellen ist ebenfalls M. Die zugehörige Frequenz der Stützstelle mit dem Index k kann mit $f_k = k \cdot \frac{f_s}{M}$ berechnet werden. Die inverse Operation der DFT ist wiederum definiert als:

$$x(n) = \frac{1}{M} \int_{n=0}^{M-1} X(\Omega) e^{j\Omega n/M} \tag{3.8}$$

Ein Sonderform der DFT ist die FFT (*Fast Fourier Transformation*). Ihr Ergebnis ist äquivalent zu dem der normalen DFT, wird jedoch schneller berechnet. Die Periodizität, die Symmetrien und die Orthogonalität der Basisfunktionen einer DFT lassen viel Raum für Verbesserungen innerhalb der arithmetischen Effizienz, was sich FFT-Algorithmen zu Nutze machen [31]. Der erste FFT-Algorithmus wurde 1965 in einer Arbeit von Cooley und Tukey in [32] vorgestellt und ist immernoch einer der am weitesten verbreitetsten FFT-Algorithmen und basiert wie die meisten FFTs auf dem Teile-und-Herrsche-Prinzip (*divide and conquer*). Er reduziert die Anzahl der benötigten Berechnungen für N Punkte von $2N^2$ auf $2N \log_2 N$ [28]. Die verschiedenen Algorithmen, welche es zur Berechnung der FFT gibt, haben gemeinsam, dass sie eine Blocklänge benötigen, die einer Potenz von 2 entspricht.

Die Block- oder FFT-Länge bestimmt Frequenzauflösung und Zeitauflösung der FFT. Die Auflösungen verhalten sich stets umgekehrt proportional zueinander. Je größer die Blocklänge, desto kleiner ist der Abstand zwischen den Frequenz-Stützstellen (Bins) der FFT und desto besser ist folglich die Frequenzauflösung. Mit großer Blocklänge reduziert sich allerdings die Zeitauflösung bei der Analyse eines Signals, da die Blocklänge der für die Berechnung des Spektrums herangezogenen Signaldauer entspricht.

3.2 Chroma-Extraktion

Chroma-basierte Audiofeatures weisen eine hohe Korrelation zur im Audiosignal enthaltenen Harmonie auf. Sie stellen ein etabliertes Werkzeug zur Verarbeitung und Analyse von Musik dar [7]. Das Verfahren besteht in der Extraktion von 12-elementigen Vektoren aus einem Audiosignal über die Zeit. Jede Dimension eines Vektors repräsentiert dabei eine der zwölf Pitchklassen (C, C#, D, D#, E, F, F#, G, G#, A, A#, B). Grund für diese Art der Zuordnung ist, dass das menschliche Ohr Töne nicht nur anhand ihrer Frequenz differenziert. Auch deren Frequenzverhältnis ist ausschlaggebend. So werden Töne, die ein Frequenzverhältnis zueinander aufweisen, welches einer Zweierpotenz entspricht, als besonders ähnlich wahrgenommen und können derselben Gruppe zugeordnet werden [2].

Solch eine Gruppe entspricht einer Pitchklasse. Bild 3.1 illustiert diesen Zusammenhang in Form einer Helix. Die z-Achse repräsentiert die Tonhöhe. X- und y-Koordinaten repräsentieren die Pitchklassen, die sich unendlich im Kreis drehen. Jene Punkte mit den gleichen x- und y-Koordinaten gehören zur gleichen Pitchklasse. Bei der Chroma-Extraktion wird die Helix bildlich beschrieben zu einem Kreis zusammengestaucht und alle auf ihr befindlichen Halbtöne werden ihrer Pitchklasse zugeordnet. Die Information der Frequenz und der zugehörigen Oktave entfällt somit.

Ergebnis der Chroma-Extraktion ist ein Vektor mit einer Dimension für jede Pitchklasse. Der Zahlenwert jeder Dimemsion entspricht dabei der Summen der Amplituden jener Signalkomponenten, deren Frequenz zur jeweiligen Pitch-Klasse gehört. Anders ausgedrückt wird die Energie einer einzelnen Signalkomponenten zu einer der 12 Pitchklassen zugeordnet [6]. Die Signalkomponenten resultieren dabei aus einer Repräsentation des Audiosignals. Die Repräsentation kann etwa aus der Kurzzeit-Fouriertransformatierten (STFT, Short Time Fourier Transformation), einer Constant-Q-Transformation oder aus dem Ergebnis der Filterung des Signals anhand einer Filterbank stammen.

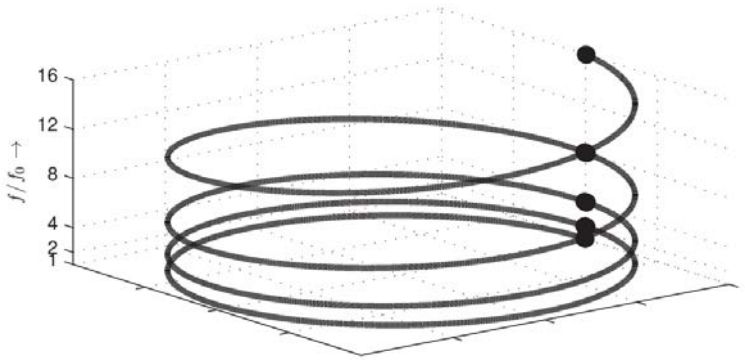

Abbildung 3.1: Die Pitch-Helix, welche das Verhältnis zwischen Oktav-Intervallen modelliert. Alle Halbtöne mit diesem Verhältnis liegen in der Helix übereinander [2].

Die Chroma-Analyse setzt eine gleichteilig temperierte Tonskala (gleichtstufige Stimmung) voraus, bei der alle zwölf Halbtonschritte einer Oktave die gleiche Größe von 100 Cent besitzen und sich auf eine Oktavenlänge von 1200 Cent aufsummieren. Viele Verfahren, die die Chroma-Analyse benutzen, resultieren in einem Chromagramm, welches eine Zeitfolge von Chroma-Vektoren darstellt (siehe 3.2).

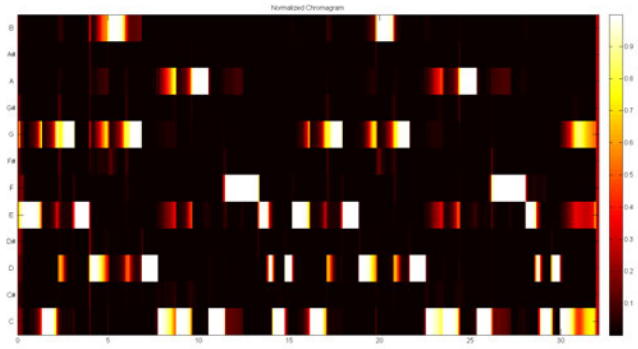

Abbildung 3.2: Chromagramm der Aufnahme einer Piano-Version des Songs *I'm yours* von Jason Mraz

Typische und beispielhafte Szenarien für die Anwendung der Chroma-Extraktion sind unter anderem:

- Audio Matching: Chroma-Vektor-Zeitreihen mehrerer Audiosignale können miteinander verglichen und aufeinander abgebildet werden. Dies kann es ermöglichen, verschiedene Versionen ein und desselben Stücks in einer Datenbank automatisch aufeinander zu referenzieren [33] [34] oder automatische Zuordnungen von Partituren und Aufnahmen zu ermöglichen [5].
- Akkorderkennung: Aus der harmonischen Zusammensetzung der Chroma-Vektor-Dimensionen lassen sich Akkorde abschätzen und einer Aufnahme zuordnen [6] [35] [36] [37]. Häufige Anwendung findet dies bereits in diversen *E-Learning*-Applikationen wie Chordify [38].
- Segmentierung: Musikstücke können auf ihre Struktur hin untersucht werden, indem in einem Chromagramm nach Selbstähnlichkeit gesucht wird. So kann zum Beispiel der am häufigsten auftretende Teil (meist der Chorus) eines Songs identifiziert werden (auch Thumbnailing genannt) [39] [40] oder ein Song in mehrere (wiederkehrende) Teile eingeteilt werden [41].

3.2.1 Berechnung von Chroma-Vektoren

Die Berechnung eines Chroma-Vektors aus dem Ergebnis einer FFT kann auf verschiedene Art und Weise vorgenommen werden. Das Grundprinzip des Mappings der FFT-Bins auf je eine Pitchklasse ist dabei immer dasselbe.

Grundlage der Chroma-Extraktion mittels FFT ist es, jene Stützstellen (Bins) der FFT zu bestimmen, deren Frequenzen am nächsten zu den Mittenfrequenzen der zur Pitchklasse zugehörigen Töne liegen. Die Mittenfrequenzen der Töne werden gemäß des MIDI-Standards berechnet, so dass eine gleichstufige Stimmung angenommen wird. Die Mittenfrequenz einer MIDI-Note p berechnet sich mit:

$$f_m(p) = 2^{\frac{p-69}{12}} \cdot 440\text{Hz}, p \in [1:120] \tag{3.9}$$

Um die zugehörige Stützstellennummer k (Bin-Nummer) einer Frequenz im Spektrum zu ermitteln, multipliziert man die Frequenz f_m mit dem Kehrwert der Frequenzauflösung der FFT und rundet das Ergebnis auf die nächste ganze Zahl (Gleichung 3.10). Dies ist nötig, da die Frequenzen der Bins nicht exakt der Mittenfrequenzen entsprechen.

$$k = [f_m \cdot \frac{K}{f_s}] \tag{3.10}$$

f_s ist dabei die dem zu analysierenden Signal zugrundeliegende Samplerate und K die FFT-Länge bzw. Blocklänge eines Signalausschnitts.

Statt lediglich eine einzelne Stützstelle für jede zu extrahierende Frequenz heranzuziehen, können auch die Werte zusätzlicher Stützstellen um die Mittenfrequenz herum addiert werden. Dieses Intervall besitzt keine absolut feste Größe sondern ist umso größer, je höher die zu analysierende Frequenz ist. Grund dafür ist der exponentiell wachsende Abstand zwischen den Mittenfrequenzen der Halbtonschritte.

Anhand der Cent-Skala kann die Beschreibung dieser Intervallverhältnisse vereinfacht werden. Sie stellt eine logarithmische Maßeinheit dar. Ein gleichstufiger Halbton wird dabei immer in 100 Schritte (Cent) geteilt. Werden neben den Stützstellen, die den Mittenfrequenzen entsprechen, auch benachbarte Stützstellen für die Chroma-Extraktion herangezogen, so sind liegen diese meist innerhalb eines Abstandes bis zu 50 Cent. Bei einem Abstand von mehr als 50 Cent würde das Intervall bereits wieder in den Bereich der benachbarten Halbtöne hineinreichen, was unerwünscht ist.

Lerch liefert in [2] eine allgemeingültige Formel zur Berechnung von Chroma-Vektoren aus einer STFT anhand der Addition der in mehreren Frequenzintervallen vorliegenden Magnitudenwerte:

$$v(j,n) = \sum_{o=o_l}^{o_u} \left(\frac{1}{k_u(o,j) - k_l(o,j) + 1} \sum_{k=k_l(o,j)}^{k_u(o,j)} |X(k,n)| \right). \tag{3.11}$$

Die Gleichung beschreibt die Berechnung des Chromawertes für eine Pitchklasse j aus einem Spektrum X . Die Variable n stellt dabei die fortlaufende Nummer des Signalblocks dar, aus der das Spektrum errechnet wird. Die Indizes o_u und o_l geben die Nummer der unteren und oberen Oktave an, welche zur Extraktion herangezogen werden. Weiterhin sind k_u und k_l jene FFT-Stützstellen, die um einen festen Cent-Betrag um die Mittenfrequenz k des aktuellen Halbtones liegen und so ein Frequenzband bzw. das Intervall von Stützstellen definieren.

3.2.2 Ausgewählte Arbeiten

Im Folgenden sollen einige ausgewählte frühere Arbeiten vorgestellt werden, in denen theoretischer Hintergrund und Anwendungsgebiete gleichermaßen vorgestellt werden und in gewisser Weise für die Implementierung in dieser Arbeit relevant sind.

Stark und Plumbley [6] implementieren ein System zur Klassifikation von Akkorden während einer Live-Performance, bei welcher sie ein Chromagramm berechnen, dessen Werte für erwartete Noten ausblenden und die verbleibende Energie minimieren. Sie benutzen eine spezielle Art der Bin-Zuordnung, bei der für jede Halbton nicht der Durchschnitt aller zugehörigen Bins gebildet wird wie in Formel 3.11, sondern lediglich das Bin mit dem maximalen Wert herangezogen wird. Äquivalent zu 3.11 lässt sich dies mit Gleichung 3.12 ausdrücken.

$$v(j,n) = \sum_{o=o_l}^{o_u} \left(\max_{k_l \leq k \leq k_u} (|X(k,n)|) \right) \tag{3.12}$$

In der Arbeit über eine eigens entwickelte Matlab-Bibliothek zur Extraktion von Chroma-Features geben Müller und Ewert [7] einen Überblick über die Verarbeitungskette vom rohen Audiosignal über eine Pitch-Repräsentation bis hin zu verschiedenen Varianten von Chromagrammen. Die Varianten unterscheiden sich hauptsächlich hinsichtlich ihrer Nachbearbeitungsschritte (Post-Processing) und in ihrer Robustheit gegen dynamische Schwankungen oder variierende Klangfarben. Im Gegensatz zu [6] arbeitet die Implementierung ausschließlich offline, also nicht in Echtzeit. So kann vor der Berechnung der Chroma-Vektoren etwa ein Kurzzeit-Mittelwert der Energie (*Short Time Mean Square Power* oder *local energy*) für jedes Teilband berechnet werden. In der Arbeit werden also verschiedene Iterationen über die Zeitreihen durchgeführt und es können sich gleichermaßen globale und lokale Signaleigenschaften zu Nutze gemacht werden. In ihrer Arbeiten heben die Autoren vor, dass für verschiedene Signalarten und Anwendungsszenarien je

eine andere Variante am geeignetsten scheint, aber keine eine allgemeine optimale Lösung darstellt.

Dannenberg beschreibt einen Audio-Matching Algorithmus [5], in der eine Technik namens *Dynamic Time Warping* benutzt wird, um das Chromagramm einer Live-Performance mit einem aus einer Midi-Datei generiertem Chromagramm abzugleichen. Da die Geschwindigkeit einer Live-Performance von der einer Midi-Datei abweicht, müssen Ereignisse, die dem Algorithmus nach in beiden Chromagrammen übereinstimmen, zeitlich neu aufeinander abgebildet werden. Dazu muss in einer Ähnlichkeitsmatrix, welche aus beiden Chromagrammen berechnet wird, jener Pfad gefunden werden, der die geringsten Kosten verursacht, wofür die Ähnlichkeitsmatrix zwei mal durchlaufen wird: zum einen um die Kosten des besten Pfades zu ermitteln und zweitens um den eigentlich Pfad zu finden [5]. Die Verwendung von Chromagrammen eignet sich zum einen für *Score-Following* für Orchester oder kleinere Ensembles. In diesem Anwendungsfall sollte allerdings der Ansatz aus [1] benutzt werden, damit Echtzeit-Verarbeitung möglich ist. Eine detailliertere Ausführung hierzu wie die Funktionsweise beider Arbeiten ineinandergreifen wird nicht gegeben. Außerdem schlägt Dannenberg vor, dass der Chroma-basierte Algorithmus auch offline für Audio-Editor-Programme genutzt werden kann, um Audio-Aufnahmen automatisch auf Partituren abzubilden und Produzenten mehr Komfort beim Schneiden und Zusammenfügen von einzelnen Takes zu geben.

3.3 Onset-Erkennung

Der hier vorgestellte *Score-Following*-Algorithmus benutzt eine Onset-Erkennung, um den Zeitpunkt von Tastenanschlägen des Benutzers zu bestimmen. In diesem Abschnitt soll der Nutzen dieser Funktion erläutert werden. Es werden einige theoretische Grundlagen geliefert und einige ausgewählte Arbeiten zu diesem Thema vorgestellt. Die letztendliche Implementierung der Onset-Erkennung wird in Abschnitt 4.3 beschrieben.

3.3.1 Motivation

Wie bereits erwähnt, ermöglicht eine Onset-Erkennung die Bestimmung des Zeitpunkt eines Tastenanschlages. Zwar würde es möglich sein, nur anhand der Berechnung der Chroma-Vektoren die vom Benutzer gespielten Töne zu analysieren und eine Schätzung der Richtigkeit dieser anzuzeigen, in dem Partitur und Video zum nächsten erwarteten Event springen. Allerdings kann dies fehleranfällig sein. Ein Szenario, dass Fehler beim

Score-Following erzeugt, ist das des Aufeinanderfolgens von gleichen Tönen oder Akkorden. Ohne Onset-Erkennung wäre es nur sehr schwer möglich, zwei oder mehr aufeinanderfolgende Events mit gleichen Tonhöhen zu unterscheiden. So würde der Score-Follower das erste von zwei gleichen Events erkennen. Klingt das erste gespielte Event allerdings noch nach und der Score-Follower erwartet das zweite gleiche Event, so wird auch dieses sofort erkannt werden. Der Score-Follower springt zum nächsten Event in der Partitur, obwohl das letzte nicht vom Benutzer gespielt wurde. Ein weiterer Vorteil der Onset-Erkennung ist es, dass sich die Rhythmik der vom Benutzer gespielten Events evaluieren lässt. So kann aufgrund der zeitlichen Relationen der gespielten Noten analysiert werden, ob der Benutzer nicht nur die richtigen Töne auf dem Klavier getroffen hat, sondern auch den vorgegebenen Rhythmus befolgt hat. Voraussetzung dafür ist natürlich sowohl ein fehlerfreies Spielen des Benutzer sowie eine fehlerfreie und genügend schnelle Erkennung seitens des Score-Followers. Im Rahmen dieser Arbeit wird das Feature der rhythmischen Beurteilung noch nicht implementiert, wird aber für künftige Weiterentwicklungen in Erwägung gezogen.

3.3.2 Theorie

Onset-Erkennung kann als Suche nach Transienten im Signal, also nach Regionen mit sehr hohen Amplituden, beschrieben werden [3]. Generell wird das Signal bzw. dessen Blöcke zunächst mit einer Erkennungsfunktion (ODF: *onset detection function*, auch *novelty function*) untersucht. Hierzu wird ein Feature, also eine bestimmte Eigenschaft aus dem Signal extrahiert, anhand dessen beobachtet werden kann, ob etwas neues passiert ist [2]. Es existieren dabei mehrere Features, die zur Onset-Erkennung geeignet sind. Aus der ODF resultiert eine Zeitreihe eines Features mit weit niedrigerer Samplerate. Diese Zeitreihe stellt einen Zwischenschritt im gesamten Erkennungsprozess dar. Dieser Zwischenschritt wird benötigt, da Musiksignale sowohl additiv und oszillierend sind und man nicht nur anhand der Ableitung des originalen Signals nach Änderungen im Signal suchen kann. Die ODF-Zeitreihe dagegen repräsentiert die lokale Struktur des originalen Signals in einer vereinfachten Form [3]. Auf der sich ergebenden Zeitreihe der Erkennungsfunktion wird für gewöhnlich ein *Peak-Picking*-Algorithmus angewendet. Das heisst, dass anhand bestimmter Kriterien Maxima in der Zeitreihe bestimmt werden, welche dann als Onset-Punkte betrachtet werden können.

3.3.2.1 Onset Features

Im Folgenden sollen exemplarisch einige gängige Features vorgestellt werden, die sich zur Implementierung einer Erkennungsfunktion (ODF) eignen.

Amplitude & Energie: Eine sehr einfache, aber im Prinzip funktionierende Herangehensweise für eine Erkennungsfunktion besteht im Verfolgen der Amplitude des Signals, da Onsets eine Erhöhung derselben mit sich bringen. Solche Funktionen werden auch als *Envelope Follower* bezeichnet. Dazu kann man den Betrag aller in einem Block befindlichen Samples mit einer Fensterfunktion w_k multiplizieren und die Ergebnisse aufsummieren:

$$E_A(n) = \frac{\sum_{k=-K/2}^{K/2-1} |x(k,n)| w_k}{K} \tag{3.13}$$

Alternativ kann auch statt der Amplitude die Energie eines Blocks berechnen werden, in dem quadriert wird [3].

$$v_E(n) = \frac{\sum_{k=-K/2}^{K/2-1} [x(k,n)]^2 w_k}{K} \tag{3.14}$$

K stellt dabei die Länge des aktuellen Signalblocks dar. Dieser ist der n-te Ausschnitt aus dem Gesamtsignal. Das Feature kann ohne vorhergehende Fourier-Transformation im Zeitbereich berechnet werden und ist einfach zu implementieren.

High Frequency Content: Betrachtet man den Anschlag einer Note als kurzes Signal mit hoher Amplitude, so wird man in dessen Spektrum einen verhältnismäßig großen Anteil an hohen Frequenzen vorfinden. Anders ausgedrückt weist ein sogenannter Transient im Frequenzbereich eine hohe Bandbreite auf. Der *High Frequency Content* berechnet für aufeinanderfolgende Spektren jeweils einen Wert, der den Anteil an hohen Frequenzen im Spektrum repräsentiert. Dazu kann das Spektrum $X(k,n)$ zugunsten von hohen Frequenzen mit einer Fensterfunktion W_k gewichtet werden [3].

$$v_{HFC}(n) = \frac{\sum_{k=0}^{K/2-1} W_k |X(k,n)|^2}{\frac{K}{2}} \tag{3.15}$$

Da der *HFC* im Frequenzbereich bestimmt wird, geht der Berechnung eine Fourier-Transformation voraus, was den Aufwand erhöht.

Spectral Flux: Der sogenannte *Spectral Flux* (oder spektrale Differenz) wird wie der *HFC* ebenfalls im Frequenzbereich berechnet. Bei der Berechnung wird sich zu Nutze gemacht, dass Transienten durch einen breitbandigen Anstieg der Energie im Spektrum sichtbar sind. Mit dem *Spectral Flux* kann die Summe der Differenzen zwischen aufeinanderfolgdenden FFT-Blöcken bestimmt werden [2]. Besitzt ein FFT-Block Frequenzanteile in einem breiteren Band als sein vorhergehender Block, ist davon auszugehen, dass innerhalb des Signals ein Onset aufgetreten ist und der *Spectral Flux* steigt.

$$v_{SF}(n) = \frac{\sqrt{\sum_{k=0}^{K/2-1} (|X(k,n)| - |X(k,n-1)|)^2}}{\frac{K}{2}} \tag{3.16}$$

$X(k,n)$ und $X(k,n-1)$ bezeichnen dabei zwei aufeinanderfolgende Signalblöcke im Frequenzbereich mit der Blocklänge K. Da bei der Onset-Erkennung nur positive Änderungen relevant sind, können alle negativen Änderungen zu 0 gesetzt werden. Dieses Verfahren wird als *Half Wave Rectification* bezeichnet und kann beschrieben werden mit:

$$HWR(x) = \frac{x + |x|}{2} \tag{3.17}$$

Will man also 3.17 auf den *Spectral Flux* anwenden erhält man:

$$v_{SF}(n) = \frac{\sqrt{\sum_{k=0}^{K/2-1} (HWR(|X(k,n)| - |X(k,n-1)|)^2}}{\frac{K}{2}} \tag{3.18}$$

Phasenableitung: Auch die Phase eines Signals kann als Feature für die Erkennungsfunktion herangezogen werden, da es höchst unwahrscheinlich ist, dass Frequenzkomponenten des Signals vor und nach einem Onset die gleiche Phase besitzen. Diese Eigenschaften lässt sich ausnutzen, um Unregelmäßigkeiten in der Phase als potentielle Onsets zu betrachten [42].

Als Grundlage dafür lässt sich zunächst die Momentanfrequenz (*instantaneous frequency*) einer Signalkomponente heranziehen, die durch die zeitliche Ableitung der 2π *unwrapped Phase* des momentanen und vorhergehenden Signalblocks berechnet wird [3].

$$f_k(n) = \left(\frac{\varphi_k(n) - \varphi_k(n-1)}{2\pi h} \right) f_s \tag{3.19}$$

Hier ist h die Hopsize zwischen zwei aufeinanderfolgenden Signalblöcken und f_s die Samplingfrequenz.

Die Änderung der Momentanfrequenz , also die zweite Ableitung der Phase nach der Zeit ergibt sich dann aus:

$$\Delta\varphi_k(n) = \varphi_k(n) - 2\varphi_k(n-1) + \varphi_k(n-2) \simeq 0 \tag{3.20}$$

Das Feature wird dann berechnet mit:

$$PD(n) = \frac{1}{N} \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} |\delta\varphi(n)| \tag{3.21}$$

3.3.2.2 Peak Picking

Nachdem mit einer Erkennungsfunktion eine Zeitreihe erstellt wurde, die meist ein Sample pro Signalblock enthält, muss auf dieser Zeitreihe ein *Peak-Picking*-Algorithmus angewendet werden. Dieser findet lokale Maxima in der Zeitreihe, die als Onset betrachtet werden. Zentrales Kriterium eines *Peak-Picking*-Algorithmus ist meist ein Schwellwert. Ein Sample der Erkennungsfunktion muss über diesen Wert liegen, um als potentieller Onset in Betracht gezogen zu werden. Es können weitere Kriterien herangezogen werden, um die Menge an irrtümlich erkannten Onsets (*false positives*) oder verpassten Onsets (*false negatives*) zu minimieren [42]. Bei jeder Anwendung im Bereich der Onset-Erkennung sollte sich daher auch die Frage gestellt werden, welche von beiden Fehlerarten mehr tolerierbar ist. Mit anderen Worten muss also entschieden werden, ob man lieber tatsächlich falsche Onsets erkennen oder lieber einige Onsets verpassen möchte.

Die Echtzeit-Onset-Erkennung bringt einige Aspekte mit sich, die es zu erwähnen gilt. Prüft man in einem Offline-Algorithmus ein ODF-Sample darauf, ob es ein lokales Maximum darstellt, so zieht man vorhergehende und nachfolgende Samples heran und prüft, ob diese kleiner sind. In einem Online-Algorithmus ist es allerdings zunächst nur möglich, vorhergehende ODF-Samples zu betrachten, da nachfolgende Samples schlicht noch nicht vorhanden sind.

Führt man jedoch eine geringe Verzögerung ein, so kann dennoch eine bestimmte Anzahl nachfolgeder ODF-Samples (meistens lediglich einer) für den *Peak-Picking*-Algorithmus herangezogen werden. Das heisst, dass im Online-Algorithmus eine Verzögerung der Länge eines Signalblocks (= 1 ODF-Sample-Intervall) eingeführt werden muss [43]. Darüber hinaus lässt sich in einem Offline-Algorithmus der Schwellwert des *Peak-Picking*-Algorithmus herausfinden, indem man in etwa das Histogramm der gesamten Zeitreihe

auswertet und Samples unter einer bestimmten Häufigkeit als potentielle Onsets heranzieht. In einem Online-Algorithmus ist dies offensichtlich nicht so trivial, da es kein determiniertes Histogramm gibt. Nichtsdestotrotz läßt sich trotzdem nach jedem neu berechnetem Sample auch ein neuer Schwellwert σ_n auf Grundlage der vergangenen Samples berechnen. Dies geschieht meist unter der Verwendung des Mittelwertes. Eine triviale Herangehensweise wäre also den Mittelwert eines sich mit bewegenden Fensters $ODF(n_m)$ zu berechnen und diesen als Schwellwert heranzuziehen, was beschrieben werden kann mit:

$$\sigma_n = \lambda \cdot mean(ODF(n_m)) + \delta \tag{3.22}$$

Es gilt $n_m \in [m-a, m+b]$. Das ODF-Fenster enthält also a Samples vor und b Samples nach dem aktuellen Sample (Echtzeit: $b = 1$). λ und δ sind frei wählbare Werte zur Skalierung. Zusätzlich wurde in verschiedenen vorangegangenen Arbeiten auch der Median zur Berechnung des Schwellwertes hinzugezogen. Durch Verwendung des Median wird eine Schwellwertfunktion weniger anfällig für starke Fluktuationen. Diese können im Schwellwert auftreten, wenn es sehr hohe Peaks in der ODF-Zeitreihe gibt. Dies kann zur Folge haben, dass weniger hohe Peaks, deren Ursprung ebenfalls in einem realen Onset haben, maskiert werden [3]. Der Median dagegen verhält sich robuster gegen starke Ausreißer. Sowohl Broussier [8] als auch Glover [43] verwenden eine Variante der Form:

$$\sigma_n = \lambda \cdot mean(ODF(n_m)) + \alpha \cdot median(ODF(n_m)) + \delta \tag{3.23}$$

Glover beschreibt den Term δ dabei mit $\delta = \omega \cdot \varphi$. φ ist dabei der höchste Peak, der bis zur Zeit von n auftrat. ω ist ein Gewichtungsfaktor, der entweder auf 0 gesetzt (bei unbegrenzter Laufzeit) oder regelmäßig aktualisiert werden kann, um dynamische Änderungen auszugleichen.

Dixon verwendet in [42] folgende Kriterien für den *Peak-Picking*-Algorithmus:

1. Der Wert das aktuellen Samples muss größer als alle anderen Samples im aktuellen Fenster sein:
$$f(n) > f(k), k \in (n-a, n+b)$$

2. Der Samplewert ist größer als der lokale Mittelwert des aktuellen Fensters:
$$f(n) > \frac{\sum_{k=n-a}^{n+b} f(k)}{a+b+1} + \delta$$

3. Der Samplewert liegt über dem Ergebnis einer Schwellwert-Funktion g_α:

$$f(n) > g_\alpha(n-1)$$

$$g_\alpha(n) = max\Big(f(n), \alpha g_\alpha(n-1) + (1-\alpha) f(n)\Big)$$

Ein Problem bei der Onset-Erkennung sind oft kleinere Peaks, die nach einem Onset auftreten. Oft liegen diese auch noch über dem Schwellwert und werden fälschlicherweise als neue Onsets interpretiert. Marolt [44] schlägt für dieses Problem eine sogenannte *Refractory* Zeit vor. In dieser Zeit werden nach einem erkannten Onset alle unmittelbar danach auftretenden Onsets ignoriert. Diese Zeit sollte zum einen so groß gewählt werden, dass alle über dem Schwellwert liegenden Nebenpeaks mit abgefangen werden. Zum anderen sollte sie aber klein genug sein, damit keine darauf folgenden tatsächlichen Onsets verpasst werden.

3.4 Zusammenfassung

Um die im folgendem Kapitel beschriebene Implementierung nachvollziehen zu können, gab dieses Kapitel die entsprechenden Grundlagen zu den dort beschriebenen Methoden. So wurde die Fourier-Transformation, welche in der Signalverarbeitung Grundlage vieler Verfahren ist, mathematisch im Wesentlichsten umrissen und kurz eine ihrer Sonderformen, die *Fast Fourier Transformation* (FFT) beschrieben. Nächster Bestandteil dieses Kapitels stellte die Chroma-Extraktion dar. Neben des allgemeinen Konzepts der Chroma-Extraktion wurden grundlegende Formeln erläutert sowie einige frühere Arbeiten vorgestellt, die Relevanz für diese Arbeit besitzen. Im Weiteren war die Disziplin der Onset-Erkennung ein wesentlicher Bestandteil dieses Kapitels. Eine Auswahl von Features, die in der Onset-Erkennung als Erkennungsfunktion zum Einsatz kommen, wurden beschrieben. Außerdem wurde auch auf das sogenannte *Peak Picking* eingegangen. Dies sind Algorithmen die Maxima in der Zeitreihe der Erkennungsfunktion finden, welche als Onsets betrachtet werden können. Beispielhaft wurde dazu ein Algorithmus von Dixon [42] vorgestellt.

Kapitel 4

Implementierung des Algorithmus

Aufbauend auf den in Kapitel 3 gegebenen Grundlagen wird in diesem Kapitel die Implementierung des *Score-Following*-Algorithmus erläutert. Es werden unter anderem die Mechanismen vorgestellt, mit der die zentralen Bestandteile der Onset-Erkennung und der Chroma-Extraktion ineinandergreifen und das Folgen einer Partitur ermöglichen. Zunächst soll ein Überblick über den *Score-Following*-Algorithmus allgemein gegeben werden. Anschließend werden alle wichtigen Bestandteile des Algorithmus vorgestellt.

4.1 Überblick

Vorbetrachtung: Im Wesentlichen basiert der *Score-Following*-Algorithmus auf den Abgleich der vom Benutzer erzeugten Schallereignissen (im Folgenden als User-Events bezeichnet) mit den aus der Partitur erwarteten Ereignissen. Ein User-Event liegt dabei in Form eines über eine bestimmte Zeit geglätteten Chroma-Vektors vor, während ein erwartetes Event als Objekt mit Eigenschaften wie enthaltenen Noten, deren Pitchklassen, Oktaven und Zeiten vorliegt. Diese erwarteten Events werden aus Midi-Daten erzeugt, welche wiederum bei der Aufnahme der in der *flowkey*-Applikation verwendeten Videos mitgeschnitten werden. Die erwarteten Events werden wie in Abbildung 4.1 dargestellt beim Initialisieren der Applikation aus einer Datenbank geladen, während die User-Events nach jedem erkannten Onset mittels einer Chroma-Exktraktion erzeugt werden.

Ablauf des Algorithmus: Der Algorithmus verarbeitet das Signal blockweise. Neben der Vorverarbeitung und einer *Short Time Fourier Transformation*, also der FFT von aufeinanderfolgenden Signalblöcken, werden Onset-Erkennung oder Chroma-Extraktion

ausgeführt. Die Chroma-Extraktion wird erst aufgenommen, wenn ein Onset im Signal erkannt wird. Die Onset-Erkennung pausiert während der Ausführung der Chroma-Extraktion, welche über eine bestimmte Zeit (im Folgenden als Haltezeit bezeichnet) stattfindet. Die Haltezeit kann entweder als Konstante vorher festgelegt oder dynamisch auf Basis des erwarteten Notenwertes berechnet werden. Alle während dieser Zeit berechneten Chroma-Vektoren werden dann in einem Puffer zwischengespeichert. Ist die Haltezeit abgelaufen, wird aus den im Puffer gespeicherten Chroma-Vektoren ein Mittelwert-Vektor berechnet. Das so entstandene User-Event wird anschließend anhand einer Vergleichsfunktion mit dem erwarteten Event verglichen. Die Übereinstimmung wird der Player-Komponente der *flowkey*-Applikation als positiv oder negativ angezeigt. Im positiven Fall läuft der Player zum nächsten erwarteten Event und gibt dies dem *Score-Following*-Algorithmus zurück. Im negativen Fall signalisiert der Player dem Nutzer, dass das erwartete Event nicht erkannt wurde und der Algorithmus wechselt wieder in den Zustand, in welchem die Onset-Erkennung läuft.

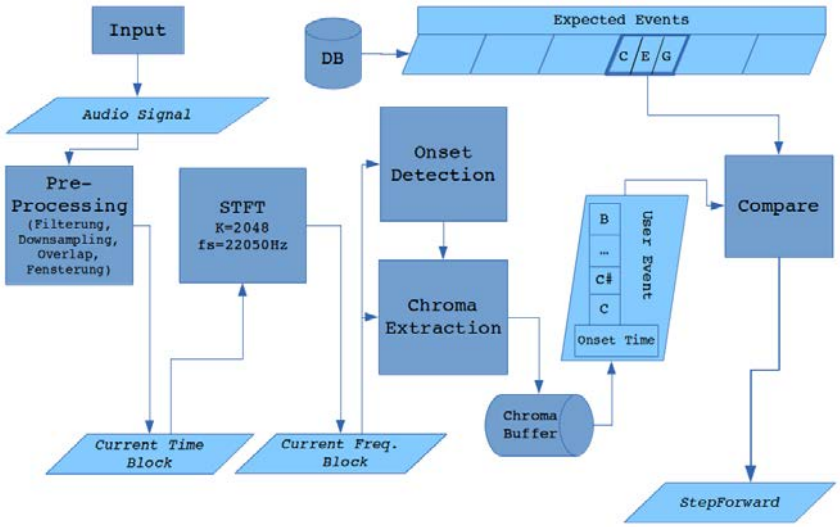

Abbildung 4.1: Flussdiagramm des *Score-Following*-Algorithmus

In den folgenden Abschnitten wird der Nutzen der Web Audio API und entsprechende Vorverarbeitungsschritte diskutiert. Entsprechend des Ablaufes im Algorithmus folgen danach Onset-Erkennung und Chroma-Extraktion. Danach wird die Modellierung von User-Events sowie erwarteten Events und im letzten Abschnitt die damit verbundenen Vergleichsfunktionen erläutert.

4.2 Vorverarbeitung

Bevor ein Signalblock der FFT übergeben wird, finden darauf einige Vorverarbeitungsschritte statt, die darauf abzielen, das Signal für die FFT zu optimieren. Die Vorverarbeitungschritte finden alle im Kontext der Web Audio API statt, da diese eine fertige Schnittstelle zur blockbasierten Verarbeitung von Signalen bereitstellt.

4.2.1 Web Audio Schnittstelle

Um Komponenten der Web Audio API zu nutzen, muss zuerst ein *Audio Context* erstellt werden. Mit diesem Objekt lassen sich alle Komponenten der Web Audio API erzeugen. Innerhalb eines *Audio Contexts* können eine oder mehrere Quellen erstellt werden, es gibt aber nur ein Ausgabeziel (*audioContext.destination*), dass auf das Soundausgabegerät (z. B. interne Lautsprecher) des jeweiligen Systems gelegt ist. Um auf das Mikrofon des Benutzergerätes zugreifen zu können und daraus eine Quelle im *Audio Context* zu erstellen, wird die *getUserMedia()*-Schnittstelle (siehe Abschnitt 2.1) verwendet. Wie beschrieben, ist dazu kein seperates Browser Plug-In nötig. Als allererster Browser unterstützt Chrome *getUserMedia()*. Zugriff darauf bekommt man über das Navigator-Objekt des Browsers:

```
1  //create audio context
2  audioContext = new AudioContext();
3
4  //request for microphone access with callback function
5  navigator.getUserMedia({audio: true, video:false}, function(stream) {
6
7    //create audiosource
8    audioSource = audioContext.createMediaStreamSource(stream);
9    audioSource.buffer = stream;
10
11   //define other web audio components like Script Processor Nodes...
12   //connect all components to a chain...
13 });
```

Neben dem Zugriff auf das Mikrofon bietet *getUserMedia* auch Zugriff auf die integrierte Kamera an, was hier jedoch nicht benötigt wird. Eine Callback-Funktion wird ausgeführt, wenn der Mikrofon-Zugriff durch den User erlaubt wird. In der Callback-Funktion, die den entsprechenden Stream übergeben bekommt, lässt sich nun die Quelle für den vorher erstellten *Audio Context* der Web Audio API erstellen. Darüber hinaus können in dieser Funktion auch alle andere benötigten Komponenten der Web Audio API definiert und anschließend miteinander verkettet werden. Eine Reihe von *Script Processor Nodes* wird

in dieser Arbeit verwendet, um Vorverarbeitungsschritte wie das *Downsampling* zu realisieren. Anhand eines *Script Processor Nodes* lassen sich eigene Komponenten für die Signalverarbeitungskette in JavaScript implementieren.

4.2.2 Vorverarbeitungsschritte

Tiefpassfilterung: Zunächst wird das zeit- und wertediskrete Signal zur Vermeidung von Alias-Effekten, die beim nachfolgenden *Downsampling* entstehen können, mit einem Tiefpassfilter bandbegrenzt. Alias-Effeke entstehen im Zuge der Nichtbeachtung des Shannon-Nyquist-Theorems. Dies wird verletzt, wenn das Signal Frequenzanteile enthält, die oberhalb der halben Samplefrequenz liegen. Beim Dowsampling ergeben die zu diesen Frequenzen zugehörigen Signalanteile neue Frequenzen im Spektrum, welche im Originalsignal nicht enthalten waren. Da sich diese Frequenzen an der halben Samplefrequenz spiegeln, werden diese Alias-Frequenzen auch als Spiegelfrequenzen bezeichnet (siehe Abbildung 4.2). Die Grenzfrequenz der Tiefpassfilterung ergibt sich zufolge des Nyquist-Shannon-Abtasttheorems auf die Hälfte der Samplefrequenz am Ausgang der *Downsampling*-Operation. Realisiert wird die Tiefpassfilterung mit einem *Biquad Filter Node* der Web Audio API. Das Interface erlaubt den Zugriff auf häufig eingesetzte Filtertypen niedrigerer Ordnung. In diesem Fall wurde ein Tiefpassfilter mit einer Grenzfrequenz von 11025 Hz eingesetzt. Der einstellbare Q-Parameter stellt bei diesem Filtertyp nicht die traditionelle Güte sondern einen Resonanzwert in Dezibel dar. In dieser Arbeit wurde ein Q-Wert von 0 dB gewählt. Der Frequenzgang des Filters kann der Abbildung 4.3 entnommen werden.

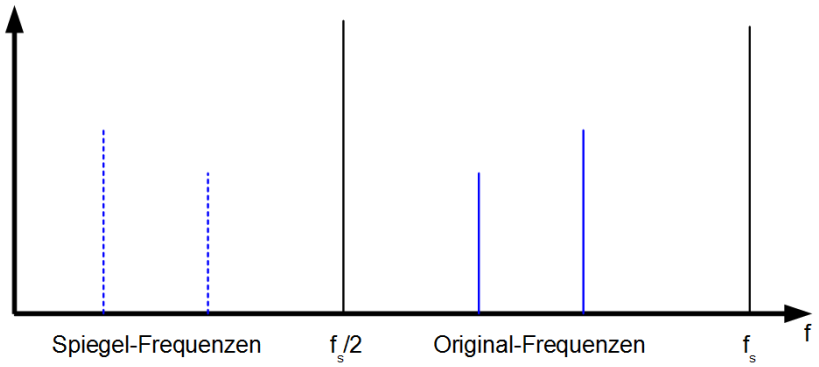

Abbildung 4.2: Spiegelfrequenzen (Alias-Effekte), welche durch Nichtbeachtung des Shannon-Nyquist-Theorems entstehen. Eine Tiefpassfilterung des Signals vor dem Downsampling ist deshalb nötig.

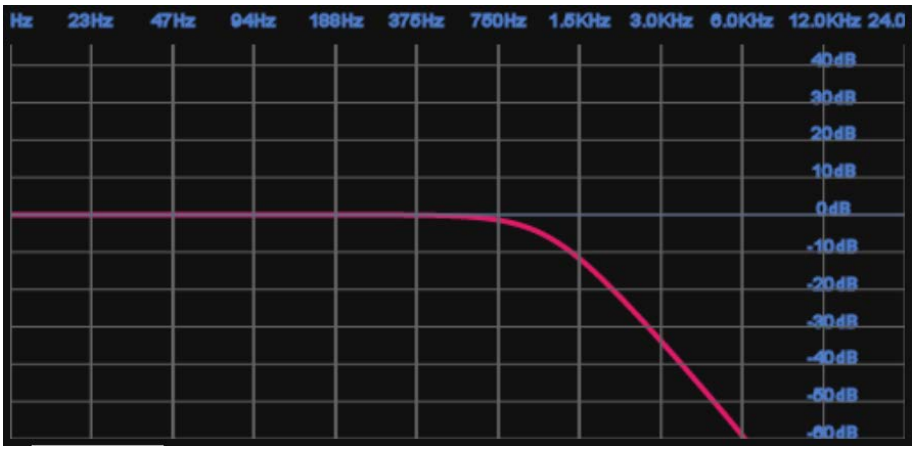

Abbildung 4.3: Frequenzgang des Tiefpassfilters mit einer Cutofffrequenz von 11025 Hz und einem Resonanzwert von 0 dB

Downsampling: Beim *Downsampling* kommt es zu einer Reduktion der Stützstellen einer Zeitreihe. Dies dient in erster Linie einer besseren Frequenzauflösung während der nachfolgenden Fourier-Transformation. *Downsampling* hat eine Reduzierung der Bandbreite des Signals zur Folge, da hohe Frequenzen nicht mehr abgebildet werden können. Die Stützstellen (Bins) der späteren FFT-Analyse konzentrieren sich also in einem kleineren Frequenz-Bereich, was die Auflösung der FFT erhöht.

Die Samplerate des *AudioContexts* wird durch die des Mikrofons bestimmt und ist zunächst nicht veränderbar. Um dennoch ein *Downsampling* durchzuführen wurde ein eigener *Script Processor Node* geschrieben, der ein *Downsampling* um einen beliebigen ganzzahligen Konvertierungsfaktor durchführt. Dieser Sonderfall des allgemein als *Resampling* bezeichneten Prozesses benötigt keine Interpolation, wie es beim *Downsampling* mit rationalen Konvertierungsfaktor der Fall ist. Stattdessen werden zur Bildung der Ausgangszeitreihe lediglich alle N Samples ($N \in \mathbb{N}$) der Eingangszeitreihe genommen. Die zwischenliegenden Samples werden verworfen.

In dieser Arbeit wurde sich für einen Konvertierungsfaktor von N=2 entschieden. Da die meisten internen Mikrofone eine Samplerate von 44100 Hz aufweisen, hat das downgesamplete Signal eine Samplerate von 22050 Hz. Aus dem Nyquist-Theorem ergibt sich so eine Bandbreite von 11025 Hz. Dies kann als angemessene Bandbreite für die Analyse von Pianomusik betrachtet werden. Die Klaviatur reicht nur bis zur Note C8, was einer Frequenz von 4186 Hz entspricht. Jedoch kann mit einer Bandbreite bis 11025 Hz noch die erste Harmonische (bei ca. 8372 Hz) dieses letzten Tons mit in die spätere Chroma-Extraktion miteinbezogen werden.

Jedem *ScriptProcessor Node* wird bei der Erstellung eine feste Blocklänge zugewiesen, die am Eingang und am Ausgang gleich ist. Da sich beim *Downsampling* die Anzahl der Samples eines Blocks reduziert, muss der Rest bis zur vollen Blocklänge mit Nullen aufgefüllt werden (*zeropadding* genannt), um die Blocklängendifferenz zwischen Eingang und Ausgang auszugleichen. Im Falle einer Blocklänge von 2048 und einem Downsample-Faktor von 2 müssen 1024 Nullen angehängt werden.

```
downsampleNode = audioContext.createScriptProcessor(blocklength,1,1);
downsampleNode.onaudioprocess = function(e){
  var downsampleNodeInput = e.inputBuffer.getChannelData(0);

  //downsampling
  var downsampledTimeBlock = applyDownsampling(downsampleNodeInput,
      downsampleFactor);

  //zeropadding
  for (var i=blocklength/downsampleFactor; i<blocklength; i++)
      downsampledTimeBlock[i] = 0;

  //set node output
  var downsampleNodeOutput = e.outputBuffer.getChannelData(0);
  downsampleNodeOutput.set(downsampledTimeBlock);
}
```

Überlappung: In einem weiteren *Script Processor Node* wird der aktuelle ankommende downgesamplete Signalblock (zeropadded) mit dem letzten zusammengefügt, indem die angehängten Nullen wieder von beiden entfernt werden und die beiden Blöcke verkettet werden. So entsteht ein Signalblock mit der ursprünglichen Blocklänge, der eine Überlappung von zwei downgesampleten Blöcken darstellt. Da jeder Signalblock im späteren Verlauf mit einer Fensterfunktion multipliziert wird, verhindert die Überlappung, dass Lücken in der Analyse entstehen und so bestimmte Ereignisse im Spektrum verloren gehen, weil die Punkte am Anfang und Ende des Signalblocks kein Gewicht in der Fourier-Transformation haben. Mit einer Überlappung werden diese Punkte bei der nächsten STFT nochmals verwendet, so dass bei einer Überlappung von 50 % das letzte Sample des Signalblocks $n-1$ zum mittleren Punkt des Signalblocks n wird.

```
overlapNode = audioContext.createScriptProcessor(blocklength,1,1);
overlapNode.onaudioprocess = function(e){
  var overlapNodeInput = e.inputBuffer.getChannelData(0);
  var secondHalf = new Float32Array(overlapNodeInput).subarray(0,
      blocklength/downsampleFactor);

  //concatenate current second half with other half from previous
      processing
  var overlapBlock = helperToolsObj.float32Concat(firstHalf, secondHalf
      );

  // set new firstHalf for next block processing (copy current
      secondHalf)
  firstHalf = secondHalf.subarray();
```

```
11
12    //set node output
13    var overlapNodeOutput = e.outputBuffer.getChannelData(0);
14    overlapNodeOutput.set(overlapBlock);
15
16  }
```

Fensterung: Die Multiplikation eines Signalsblocks im Zeitbereich mit einem bestimmten Fenstertyp reduziert Fehler in der späteren Fourier-Transformation. Diese Fehler werden als *Spectral Leakage* bezeichnet. *Spectral Leakage* tritt in allen Formen der diskreten Fourier Transformation auf. Zurückzuführen ist der Effekt auf die Tatsache, dass man für periodische Signale in der Praxis keine unendlich langen Beobachtungszeiträume anwenden kann, um sie zu analysieren. Theoretisch sind diese aber nötig, um ein sauberes Spektrum zu erhalten.

Beim *Spectral Leakage* verteilt sich die Energie einer bestimmten Frequenz f_x auf jene FFT-Stützstellen, deren Frequenz am nächsten zu f_x liegt. Das Ausblenden eines Signalblocks am Anfang und Ende mittels einer bestimmten Fensterfunktion reduziert diesen Effekt des *Spectral Leakages*. Dazu wurde ein *Script Processor Node* implementiert, der ein Hanning-Fenster sampleweise mit den Samples des am Eingang anliegenden Signalblocks multipliziert.

```
1  windowNode = audioContext.createScriptProcessor(blocklength,1,1);
2  windowNode.onaudioprocess = function(e){
3    var windowNodeInput = e.inputBuffer.getChannelData(0);
4
5    //apply window
6    var windowedBlock = applyDspWindowFunction(windowNodeInput);
7
8    //set node output
9    var windowNodeOutput = e.outputBuffer.getChannelData(0);
10   windowNodeOutput.set(windowedBlock);
11 }
```

4.2.3 Fast Fourier Transformation

Die diskrete Fourier-Transformation, die das Kurzzeitspektrum liefert, welches zur Chroma-Extraktion benötigt wird, wird anhand des FFT-Algorithmus der *DSP.js* - Bibliothek[1] berechnet. Die FFT bekommt als Input ein Signalblock mit einer Überlappung von 50 % und einer Länge von 2048 Samples, dem eine Samplerate von 22050 Hz zu Grunde liegt. Nachdem Nyquist-Theorem werden in diesem Signal Frequenzen bis zu der halben Samplerate von 11025 Hz abgebildet. Die FFT-Länge ist stets die halbe Blocklänge.

[1] https://github.com/corbanbrook/DSP.js/

Daraus ergibt sich eine Frequenzauflösung von $\frac{11025Hz}{1024bins} = 10{,}7\ \frac{Hz}{bin}$. Diese Frequenzauflösung reicht für das Klavier theoretisch nicht aus, um alle Töne bis zur untersten Oktave zuverlässig zu analysieren. Erst ab der Note G3 ist der Abstand zwischen den Grundfrequenzen zweier benachbarter Halbtöne größer als 10,7 Hz. Nichtsdestotrotz werden bei der Chroma-Analyse auch die Obertöne einer Note berücksichtigt, wodurch der resultierende Chroma-Vektor auch für Töne unterhalb von G3 ein hinreichend exaktes Ergebnis darstellen sollte.

Die FFT-Zeitauflösung beträgt zunächst $\frac{2048}{22050Hz} = 0.09288$ s. Da das resultierende Spektrum allerdings aus der Überlappung zweier Signalblocks herührt, beträgt die reale Zeitauflösung mit 0,09288 s / 2 = 0,04644 s nur die Hälfte. Jeder Block besitzt eine Überlappung von $h = 1024$ Samples (hopsize) und wurde sampleweise mit einem Hanning-Fenster multipliziert, um Fehler wie das *Spectral Leakage* zu vermeiden. Das resultierende Spektrum bildet Grundlage für die Chroma-Extraktion, welche in den weiteren Abschnitten beschrieben werden.

Das Kurzzeitspektrum, welches für die Onset-Erkennung benutzt wird, wird in einer anderen Instanz der *DSP.js*-FFT berechnet. Diese erhält als Input das unbearbeitete Zeitsignal (kein Downsampling, keine Überlappung, keine Fensterung). Dies wird ohne Umwege an der Quelle des *audioContexts*, also vom Mikrofon-Input abgerufen. Das in der Onset-Erkennung verwendende *Spectral Flux*-Feature produziert bei diesen nicht-überlappten Signalblöcken bessere Ergebnisse, da die Differenz zwischen zwei aus der STFT resultierenden Spektren größer ist, als es bei überlappten Signalblöcken der Fall ist.

4.3 Onset-Erkennung

Die Onset-Erkennung wird im Wesentlichen auf die in dieser Disziplin üblichen Schritte heruntergebrochen. Auf jedem Signalblock wird eine Erkennungsfunktion (*onset detection function, novelty function*) angewendet. Daraufhin wird auf der so entstehenden Zeitreihe mit ODF-Samples ein *Peak-Picking*-Algorithmus angewendet.

4.3.1 Erkennungsfunktion

Im Rahmen dieser Arbeit werden drei verschiedene Erkennungsfunktionen implementiert, die gegeneinander getestet werden. Neben dem *Spectral Flux* nach Gleichung 3.18 werden auch ein Amplituden-basiertes Feature nach Formel 3.14 sowie der *High Frequency Content* 3.15 implementiert.

Der *Spectral Flux* produziert dabei die besten Ergebnisse, da Peaks sich deutlich vom Grundrauschen abheben und auch in der Breite kleiner sind (siehe Abbildung 4.4). Die Implementierung der Gleichung 3.18 ist im Anhang unter A.2 zu finden.

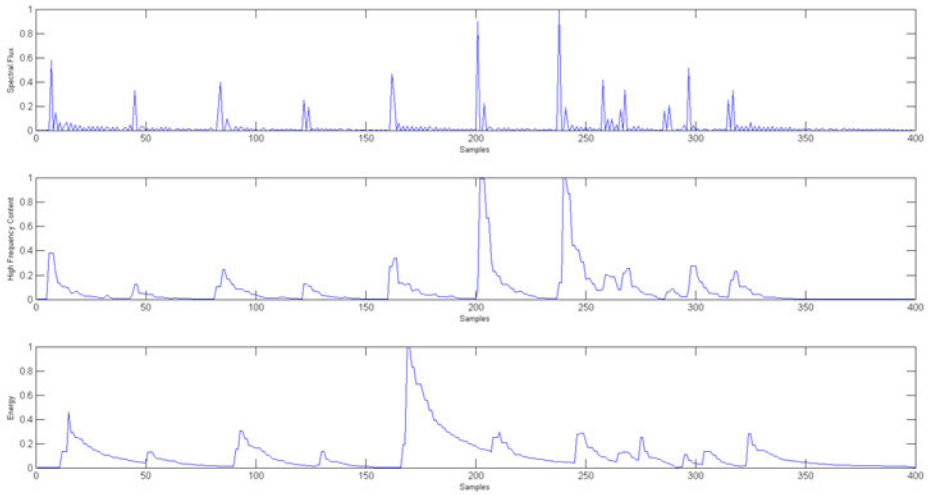

Abbildung 4.4: Vergleich von *Spectral Flux*, *High Frequency Content* und Energie-Feature

Während der Laufzeit des *Score-Following*-Algorithmus wird eine bestimmte Anzahl von berechneten ODF-Samples in einem Puffer (im Folgenden ODF-Buffer genannt) zwischengespeichert. Neu berechnete ODF-Samples werden im ODF-Buffer aufgenommen, der älteste im Buffer befindliche Wert wird dabei gelöscht. Die im ODF-Buffer gespeicherten ODF-Samples werden im *Peak-Picking*-Algorithmus verwendet.

4.3.2 *Peak-Picking*-Algorithmus

Auf den im ODF-Buffer zwischengespeicherten ODF-Samples wird der *Peak-Picking*-Algorithmus angewendet. Der Algorithmus führt wie in 3.3.2.2 beschrieben eine Verzögerung um die Länge eines Signalblocks ein, um zu bestimmen, ob ein Sample n ein lokales Maximum darstellt. Dazu wird geprüft, ob beide benachbarten Samples $n-1$ und $n+1$ kleiner als n sind.

Darüber hinaus werden Mittelwert und Median über die im ODF-Buffer enthaltenen Samples berechnet. Mit diesen Werten wird ein Schwellwert (siehe Gleichung 4.1) erstellt, der als weiteres Kriterium in den *Peak-Picking*-Algorithmus einfließt. Mittels eines Mindestschwellwertes σ_{min} kann verhindert werden, dass der Schwellwert soweit abfällt,

dass er in die Größenordnung der nur durch Rauschen produzierten ODF-Werte abfällt. Ein Faktor λ dient dazu, den Schwellwert zu justieren.

$$\sigma_n = \max((mean(ODF(n_m)) + median(ODF(n_m))) \cdot \lambda, \sigma_{min}) \tag{4.1}$$

In dieser Arbeit wurde $\lambda = 2$ gewählt. Der berechnete Schwellwert wird zusammen mit den letzten 3 ODF-Samples der *Peak-Picking*-Funktion übergeben. Wie im Quellcode dargestellt, prüft diese Funktion den Parameter $odf1$, ob er ein lokales Maximum darstellt und ob er über den Schwellwert liegt. Ist dies der Fall, wird ein *true* an den Aufrufer zurückgegeben, was dieser als Onset interpretiert.

```
1 function peakPicking(odf2, odf1, odfcur, threshold){
2   if ((odf2 < odf1) && (odf1 > odfcur) && (odf1 > threshold)){
3     return true;
4   }
5   return false;
6 }
```

4.4 Chroma-Extraktion

Wie schon eingangs beschrieben, wird die Chroma-Extraktion nach Erkennung eines Onsets gestartet. Ausgangsbasis für die Berechnung des Chroma-Vektors eines Signalblocks bildet das Kurzzeit-Spektrum der STFT (vgl. Abschnitt 4.2.3). Für jedes Kurzzeitspektrum wird ein Chroma-Vektor berechnet, indem die FFT-Bins auf einen der 12 Vektor-Elemente abgebildet wird. Die zugrundeliegenden Frequenzen der FFT-Bins, welche der Mittenfrequenzen eines Tons entsprechen, werden während der Initialisierung berechnet, indem die in Tabelle 4.4 definierten Werte rekursiv verdoppelt werden. Anhand Formel 3.10 wird die Nummer des entsprechenden FFT-Bins der Mittenfrequenz bestimmt.

Pitchklasse	C	C#	D	D#	E	F
f_0 in Hz	32.703	34.648	36.708	38.891	41.203	43.654
Pitchklasse	F#	G	G#	A	A#	B
f_0 in Hz	46.249	48.999	51.913	55.000	58.270	61.735

Tabelle 4.1: Midi-Frequenzwerte (gerundet) der ersten vollständigen Oktave auf einer Klaviatur, auf deren Grundlage alle weitere Frequenzen der Halbtöne berechnet werden.

Grundlage der Implementierung der Chroma-Extraktion ist Formel 3.11, deren Implementierung im Anhang unter A.1 zu finden ist. Im Gegensatz zu 3.11 bezieht sich der Code A.1 nicht auf eine Reihe von Chroma-Vektoren mit dem Index n, sondern auf die

Berechnung eines einzelnen Vektors. Die Funktion wird also für jeden Signalblock erneut aufgerufen.

Weiterhin beinhaltet die Funktion den boolschen Parameter *tolerance*, der angibt ob die innere Summe in 3.11 über mehrere FFT-Bins pro Halbton angewendet wird (wie in Formel 3.11) oder lediglich das mittlere FFT-Bin eines Halbtons zur Berechnung herangezogen wird. Wird etwa ein E-Piano verwendet, bei dem eine Verstimmung ausgeschlossen werden kann, reicht es für gewöhnlich, nur das Bin zur Berechnung heranzuziehen, was am nächsten zur Mittenfrequenz des Halbtons liegt. Hat man es mit einem akustischen Klavier zu tun, so sollte die Toleranz eingeschaltet werden, da von einer nicht perfekten Stimmung ausgegangen werden kann. Mit eingeschalteter Toleranz werden auch um die Mittenfrequenz liegende FFT-Bins berücksichtigt. Nach der Berechnung des Chroma-Vektors wird eine Normalisierung ausgeführt. Hier wird die euklidische Distanz benutzt.

$$v_{norm}(j,n) = \frac{v(j,n)^2}{\sqrt{\sum_{i=0}^{11} v(i,n)}} \tag{4.2}$$

Jeder normalisierte Chroma-Vektor wird in einem Buffer zwischengespeichert. Die Zwischenspeicherung ist nötig, um alle Chroma-Vektoren der Haltezeit zu mitteln. Wie in 4.1 beschrieben, wird die Chromaberechnung nach einem erkannten Onset angeschoben. Innerhalb einer Haltezeit werden Chroma-Vektoren berechnet und zwischengespeichert. Am Ende der Haltezeit wird aus allen während dieser Zeit berechneten Chroma-Vektoren ein mittlerer Chroma-Vektor berechnet. Für diese Logik wurden ein JavaScript Prototyp für ein ChromaBuffer-Objekt erzeugt. Diesem Prototyp wurden eine Reihe von Funktionen angehängt, um die eben beschriebene Logik zu implementieren:

```
1  // adds a new chromavector to the buffer, removes old
2  ChromaBuffer.prototype.addVector = function(chromaVector){...}
3
4  // sets a new holdLength
5  ChromaBuffer.prototype.setholdLength = function(timeToNextEvent,
       intervalTime){...}
6
7  // calculates averaged vector over all chromavector within the
       holdLength
8  ChromaBuffer.prototype.average = function(){...}
9
10 // checks if holdLength is reached
11 ChromaBuffer.prototype.isFull = function(){...}
12
13 // zeroing Buffer
14 ChromaBuffer.prototype.reset = function(){...}
```

Die Methode *addVector()* macht aus dem zugrundeliegenden Chroma-Array einen Buffer konstanter Größe, indem sie den ältesten Vektor entfernt bevor ein neuer Vektor aufgenommen wird. Die Anzahl der zu berechnenden Chroma-Vektoren pro Haltezeit (oder auch Bufferlänge) wird mit der Methode *setHoldLength()* berechnet. Sie wird aus der erwarteten Zeit zum nächsten Event, der Zeitauflösung der STFT sowie einem Skalierungsfaktor berechnet. Die *average()*-Methode berechnet nach Erreichen der Haltelänge einen gemittelten Chroma-Vektor. Der Ablauf der Haltezeit wird mit *isFull()* bestimmt, indem ein Zähler mit der aktuellen Haltelänge verglichen wird. Nach der Mittelung können alle Werte in allen Vektoren im Chroma-Buffer mit der *reset()*-Methode zu 0 gesetzt werden.

4.5 Event Modell

Dem Algorithmus liegt ein Modell zu Grunde, welches zwischen erwarteten Events und User-Events unterscheidet. Ähnliche Modelle wurden schon in früheren *Score-Following*-Arbeiten benutzt [1][4] und rühren daher, dass unabhängig von der Herangehensweise an das *Score-Following* stets ein Vergleich zwischen Partitur (entspricht erwarteten Events) und der Performace des Musikers (entspricht User-Events) gemacht werden muss. In diesem Abschnitt wird beschrieben, wie die Implementierung dieses Modells in dieser Arbeit aussieht.

4.5.1 User Event

Ein User Event wird als JavaScript-Objekt implementiert, welches zwei Eigenschaft beinhaltet: den Zeitpunkt des Auftretens des Events (die Onset-Zeit) und ein Chroma-Vektor, welcher aus der Mittelung mehrer Chroma-Vektoren über eine bestimmte Zeit berechnet wurde. Ein neues User Event wird nach dem Auftreten eines neuen Onsets erstellt. Die Onset-Zeit wird gesetzt und der Chroma-Vektor des User-Events wird nach Ablauf einer bestimmten Haltezeit berechnet, indem die in einem Buffer zwischengespeicherten Chroma-Vektoren gemittelt werden.

4.5.2 Erwartetes Event

Die erwarteten Events werden bei der Initialisierung des Algorithmus aus einer Datenbank als JSON-Objekt geladen. JSON (*JavaScript Object Notation*) ist ein kompaktes

Datenaustauschformat und stellt eine Teilmenge JavaScripts dar, kann aber auch unabhängig von JavaScript als Datenformat benutzt werden [45]. Ein Beispiel für ein erwartetes Event kann dem folgenden Code entnommen werden.

```
{"id":0,"x":84.94535,"t":420,"dt":14.4230625,
"notes":[{"key":76,"octave":5,"pitch":4,"status":1},
     {"key":55,"octave":3,"pitch":7,"status":1},
     {"key":52,"octave":3,"pitch":4,"status":1},
     {"key":48,"octave":3,"pitch":0,"status":1}
],
"hands":{"l":[48,52,55],"r":[76]}}
```

Ein Erwartetes Event stellt ein Objekt mit folgenden Eigenschaften dar:

- fortlaufende ID
- Position x in der virtuellen Partitur
- absolute Zeit t in Video/Audio/Midi
- Zeit dt zum nächsten Event
- erwartete Noten
- Zuordnung jeder Note zu linker oder rechter Hand

Eine Note stellt wiederum ein Objekt dar und enthält folgende Eigenschaften:

- Midi-Nummer der Klaviertaste (*key*)
- Nummer der Oktave
- Nummer der Pitchklasse (0 entspricht C)

Für den *Score-Following*-Algorithmus in der hier beschriebenen Form sind im Wesentlichen nur die Informationen über die Pitchklasse jeder Note und die Zeit bis zum nächsten erwarteten Event relevant. Der Rest der Informationen wird etwa für die Synchronisierung zwischen Video und virtueller Partitur benötigt.

4.6 Chroma-Schwellwert-basierte Vergleichsfunktion

Die Vergleichsfunktion ist ein elementarer Bestandteil des Algorithmus. Sie markiert die erwarteten Pitchklassen anhand eines Bool-Wertes entweder als positiv (*korrekt gespielt*) oder negativ. Als Eingabeparameter bekommt die Funktion das erwartete Event, den Chroma-Vektor des letzten User-Events sowie einen Chroma-Schwellwert.

Aus dem über die Zeit gemitteltem Chroma-Vektor wird ein Schwellwert für die Vergleichsfunktion berechnet. Dazu wird sowohl der Mittelwert sowie die Standard-Abweichung über alle Werte des Chroma-Vektors berechnet. Der Vergleich basiert darauf, die Töne bzw. deren Pitchklassen im erwarteten Event mit den Werten des Chroma-Vektors zu vergleichen. Liegt eine erwartete Pitchklasse über dem Chroma-Schwellwert, wird sie zunächst als positiv markiert. Abbildung 4.5 zeigt einen Chroma-Vektor und den daraus errechnenten Schwellwert. Alle erwarteten Pitchklassen liegen in diesem Fall über dem Schwellwert, was nicht immer der Fall ist. Je nachdem, ob das erwartete Event nur aus einer oder mehren Pitchklassen besteht, werden zwei verschiedene Vergleichsalgorithmen angewendet. Zunächst wird beschrieben wie mehrere erwartete Pitchklassen behandelt werden.

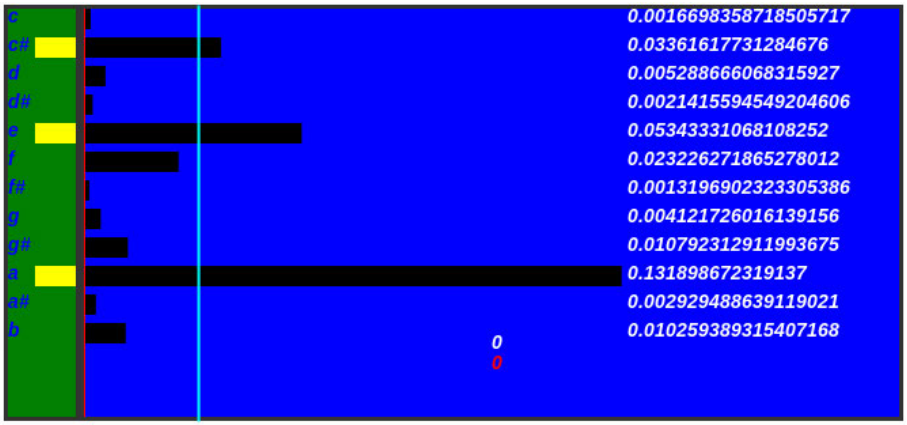

Abbildung 4.5: Darstellung eines über die Zeit gemittelten Chroma-Vektors mit erwarteten Pitchklassen (gelb) und errechnetem Schwellwert (hellblau)

Erwartetes Event mit mehreren Pitchklassen: Beim Vergleich des Chroma-Vektors mit einem erwarteten Event, welches mehrere Pitchklassen beinhaltet, wird jede Pitchklasse einzeln überprüft. Neben dem Schwellwert gibt es dabei noch andere Kriterien, die darüber entscheiden, ob ein Ton letztendlich als gespielt markiert wurde.

Diese Kriterien behandeln ein Problem bei der Extraktion der Chroma-Vektoren. Das Problem besteht darin, dass beim Erklingen eines Tons nicht nur der Wert der entsprechenden Pitchklasse im Chroma-Vektor ansteigt, sondern auch die benachbarten Pitchklassen einen Anstieg ihrer momentanen Energie erfahren. Grund hierfür kann etwa das in Abschnitt 4.2.2 beschriebene *Spectral Leakage* sein.

Solange die Pitchklassen der erwarteten Töne nicht unmittelbar benachbart sind, kann dieses Problem vernachlässigt werden. In vielen Stücken kann es jedoch vorkommen, dass

zwei verschiedene Töne, obwohl auf der Klaviatur nicht miteinander benachbart, zu Pitchklassen gehören, die direkt miteinander benachbart sind, z. B. wenn zwischen ihnen ein Intervall von einer Septime oder einer None (oder Vielfache davon) liegt (z. B. C3 und H3, D4 und D#5). So kann es vorkommen, dass ein erwarteter Ton bzw. dessen Pitchklasse als positiv markiert wird, obwohl stattdessen lediglich die benachbarte Pitchklasse erklungen ist.

Das Problem wurde gelöst, in dem während des Vergleichs für einen erwarteten Ton weitere Variablen eingeführt wurden. Eine von diesen Variablen indiziert, ob es in den erwarteten Noten eine Note gibt, deren Pitchklasse benachbart zur derzeitig untersuchten erwarteten Pitchklasse ist. Eine weitere zeigt an, ob die benachbarten Pitchklassen im Chroma-Vektor über dem Schwellwert liegen. Initialisiert werden diese zunächst alle mit *false*. Anschließend werden sie anhand zweier Abfrage mit den eigentlichen Werten belegt.

```
1  //init
2  var LB = false; //lower neighbor bigger
3  var LE = false; //lower neighbor expected
4  var UB = false; //upper neighbor bigger
5  var UE = false; //upper neighbor expected
6
7  // compare value of current expected pitch and its lower neighbor
8  if (chromaVector[lowerPitchNeighbor] > chromaVector[curExpPitch]){
9    LB = true;
10   LE = expectedNotesContains(lowerPitchNeighbor);
11 }
12 // compare value of current expected pitch and its upper neighbor
13 if (chromaVector[upperPitchNeighbor] > chromaVector[curExpPitch]){
14   UB = true;
15   UE = expectedNotesContains(upperPitchNeighbor);
16 }
```

Anhand dieser 4 Variablen wird nun entschieden, ob der erwartete Ton, dessen Energie in der Pitchklasse über dem Schwellwert liegt, letztendlich als positiv markiert bleibt oder doch als negativ gekennzeichnet wird.

Dazu wurde eine Wahrheitstabelle mit allen 2^4 möglichen Kombinationen (siehe Anhang B) erstellt. Aus den Kombinationen, deren Konjunktion (*und*) wahr ergibt, wurde ein boolescher Ausdruck erstellt, der diese Kombinationen miteinander als Disjunktion verknüpft. Dieses Vorgehen entspricht der Bildung der disjunktiven Normalform einer Booleschen Funktion. Der so entstehende boolesche Ausdruck stellt neben dem Schwellwert das zweite Kriterium dar. Werden beide erfüllt, so wird die aktuell untersuchte Pitchklasse als positiv (*korrekt gespielt*) markiert.

Erwartetes Event mit nur eine Pitchklasse: Umfasst das erwartete Event nur eine Pitchklasse, ist der Vergleich trivialer. Es muss lediglich untersucht werden, ob die erwartete Pitchklasse über dem Schwellewert liegt und ob der maximale im Chroma-Vektor enthaltene Wert dieser Pitchklasse entspricht.

```
if (chromaVector[curExpPitch] > threshold){
  if(curExpPitch != chromaVector.indexOf(chromaVector.max())){
    expectedNotes[0].playedRight = false;
  }else{
    expectedNotes[0].playedRight =  true;
  }
}else{
  expectedNotes[0].playedRight =  false;
}
```

4.7 Kosinus-Ähnlichkeit-basierte Vergleichsfunktion

Neben der in 4.6 beschriebenen Vergleichsfunktion, die im Wesentlichen auf einem Schwellwertkriterium sowie einem logischen Ausdruck basiert, wurde eine alternative Vergleichsfunktion auf Basis eines Ähnlichkeitsmaßes realisiert. Diese generiert aus einem erwarteten Event einen binären Chroma-Vektor, der eine 1 bei allen erwarteten Pitchklassen enthält und in allen anderen Pitchklassen 0 ist. Dieser binäre Chroma-Vektor wird mit dem Chroma-Vektor des User-Events verglichen. Sind die beiden Vektoren ähnlich genug, so kann der Algorithmus dem Player ein Weiterlaufen des Videos und der Partitur signalisieren.

Distanz- und Ähnlichkeitsmaße dienen beide dem Zweck, Variablen miteinander zu vergleichen und sie auf ihre Ähnlichkeit hin zu untersuchen. Während Distanzfunktionen ein Maß liefern, welches den Unterschied zweier Variablen - etwa den Abstand zweier Vektoren - quantifizieren, liefern Ähnlichkeitsfunktionen einen Wert, der die Nähe oder den Grad der Gleichheit zweier Variablen zueinander beschreibt. Letztendlich leisten beide Arten das Selbe, je nach Anwendung kann der eine oder andere Fall jedoch praktischer zu handhaben sein.

Das Kosinus-Maß ist ein Ähnlichkeitsmaß, bei dem die Länge zweier verschiedener Vektoren keinen direkten Einfluss auf das Ergebnis hat. Der Winkel der Vektoren zueinander hat dagegen einen großen Einfluss auf ihre Ähnlichkeit zueinander. Beträgt der Winkel $0°$, zeigen die Vektoren also in die gleiche Richtung, ergibt sich ein Ähnlichkeitsmaß von 1. Ein Wert von 0 bedeutet Orthogonalität, während sich der Wert -1 für genau entgegengerichtete Vektoren ergibt. Das Kosinus-Maß wird mit Hilfe des Skalarproduktes und der Vektorenbeträge berechnet [46].

$$\cos\theta = \frac{a \cdot b}{||a||||b||} \tag{4.3}$$

Interpretiert man den Kosinus als Ähnlichkeitsmaß s und schreibt die Formel aus, so entsteht:

$$s_{cos} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} b_i^2}} \tag{4.4}$$

Als Alternative zur in 4.6 beschriebenen Vergleichsfunktion wird ein weiterer Ansatz auf Basis der Kosinus-Ähnlichkeit implementiert. Der Chroma-Vektor des User-Events wird dabei mit einem aus den erwarteten Noten konstruierten binären Chroma-Vektor verglichen. Dieser binäre Vektor enthält in einer erwarteten Pitchklasse eine 1 und ist in allen anderen 0. Beide Vektoren werden einer Funktion übergeben, die ein Ähnlichkeitsmaß zwischen beiden Vektoren berechnet. In dieser Arbeit wurde die Kosinus-Ähnlichkeit nach Formel 4.4 implementiert:

```
function calcCosineSimilarity(a, b){
  var vecDotProduct=0;
  var vecMagnitudeA=0;
  var vecMagnitudeB=0;
  for (var i = a.length - 1; i >= 0; i--) {
    vecDotProduct+=a[i]*b[i];
    vecMagnitudeA+=a[i]*a[i];
    vecMagnitudeB+=b[i]*b[i];
  }
  vecMagnitudeA=Math.sqrt(vecMagnitudeA);
  vecMagnitudeB=Math.sqrt(vecMagnitudeB);
  var cosineSimilarity = vecDotProduct / (vecMagnitudeA*vecMagnitudeB);
  return cosineSimilarity;
}
```

a und *b* repräsentieren die zu vergleichenden Chroma-Vektoren. Der Index *i* stellt die aktuelle Dimension dar und läuft in diesem Fall von 11 bis einschließlich 0 (zwölf Pitchklassen).

Der Kosinus zweier Vektoren ist 1, wenn sie gleich sind. Theoretisch reicht das Maß bis -1, wenn die Vektoren genau entgegengericht sind. Da dies im Falle von Chroma-Vektoren nicht der Fall sein kann, ist die untere Grenze hier 0, was auf Orthogonalität zwischen beiden Vektoren hindeutet. Auch hier wird allerdings ein Schwellwert benötigt, um am Ende festzulegen, ob die Vektoren ähnlich genug sind, um den Player ein Weiterlaufen des Videos zu signalisieren. Initiale Versuche zeigten, dass ein Wert um $0,8$ angemessen ist. Im Vergleich zur in 4.6 beschriebenen Vergleichsfunktion ist dieser Ansatz in der Theorie zum einem weniger fehleranfällig, da das Problem der benachbarten

Töne nicht berücksichtigt werden muss. Andererseits wird mit diesem Ansatz nicht jeder erwartete Ton einzeln evaluiert, sondern der Chroma-Vektor im Gesamten. Angesichts zukünftiger Features für die *flowkey*-Applikation, die dem Nutzer detaillierte Informationen über sein Spiel bereitstellen soll, kann dies ein Nachteil darstellen. Im Rahmen dieser Arbeit soll dies zunächst jedoch ein untergeordnetes Kriterium darstellen.

4.8 Zusammenfassung

Nachdem im vorherigen Kapitel die Grundlagen für die in dieser Arbeit verwendeten Techniken gegeben wurden, präsentiert dieses Kapitel die konkrete Implementierung dieser Techniken. Zunächst wurde in einem Überblick dargestellt, wie der Algorithmus im Wesentlichen funktioniert. Hierbei sind vor allem die Vorverarbeitung, die Onset-Erkennung, die Chroma-Extraktion sowie das Event-Modell hervorzuheben. Jeder dieser Aspekte wurde konkret erläutert und mit Auszügen aus dem Quellcode ergänzt, um bestimmte Konzepte, Gedankengänge oder mathematische Angelegenheiten zu verdeutlichen.

Weiterhin wurden zwei verschiedene Vergleichsfunktionen dargestellt. Diese dienen dem Abgleich zwischen einem erwarteten Event und einem tatsächlich aufgetretenen User-Event. Die Vergleichsfunktion stellt im Endeffekt einen Klassifikator dar, der eine binäre Entscheidung darüber trifft, ob das User-Event dem erwarteten Event entspricht oder nicht. Die erste Vergleichsfunktion basiert auf einem aus dem Chroma-Vektor berechneten Schwellwert (Chroma-Schwellwert-basierte Methode), über den die Energie einer erwarteten Pitchklasse liegen muss. Die andere Vergleichsfunktion verwendet die Kosinus-Ähnlichkeit um den Chroma-Vektor des User-Events mit einem binären erwarteten Chroma-Vektor abzugleichen. Im Kapitel 5 wird untersucht, welche von beiden Methoden den besseren Klassifikator darstellt.

Kapitel 5

Evaluierung

In diesem Kapitel werden die Ergebnisse der Evaluierung des *Score-Following*-Algorithmus vorgestellt. Im Fokus der Evaluierung steht zum einem die Fähigkeit des Algorithmus, ein erklingendes Schallereignis mit dem aktuell erwarteten Ereignis zu vergleichen und eine Übereinstimmung anzuzeigen. Dabei soll untersucht werden, welche Arten von Events als korrekt (oder entgegen den erwarteten Events als inkorrekt) erkannt werden. Der zweite zu untersuchende Aspekt ist die Echtzeitfähigkeit des Systems. Dabei soll untersucht werden, ob die Verarbeitungszeit eines Signalblockes kleiner als seine entsprechende zeitliche Länge ist. In einem weiteren Abschnitt sollen andere Probleme und Erkenntnisse betrachtet werden, die bei der Benutzung der *flowkey*-Applikation zum Vorschein kamen. Zuletzt werden Möglichkeiten beschrieben, die den Algorithmus künftig verbessern können.

5.1 Testumgebung

Zur Evaluierung des hier implementierten webbasierten Algorithmus wurde eine ebenso webbasierte Testumgebung entwickelt. Diese wurde als Anwendung für das Meteor-Framework - auf welchem auch die *flowkey*-Applikation basiert - entwickelt. Die Testumgebung erlaubt es, das blockweise überlappte sowie gefensterte Signal sowohl im Zeitbereich als auch im Frequenzbereich graphisch zu beobachten. Weiterhin wird auch der momentane Chroma-Vektor sowie die berechneten Samples der Onset-Erkennungsfunktion über die Zeit dargestellt 5.1. Als Input dient entweder das Live-Signal des internen Mikrofons, ein mit der Web Audio API erzeugter Oszillator oder eine Reihe von Soundsamples, die der Testumgebung übergeben und als *Buffer Source* der Web Audio API abgespielt werden können. Der Testumgebung können zusätzlich zu den Soundsamples (die den

User-Events entsprechen) deren JSON-Repräsentation (siehe Abschnitt 4.5.2) als erwartete Events übergeben werden.

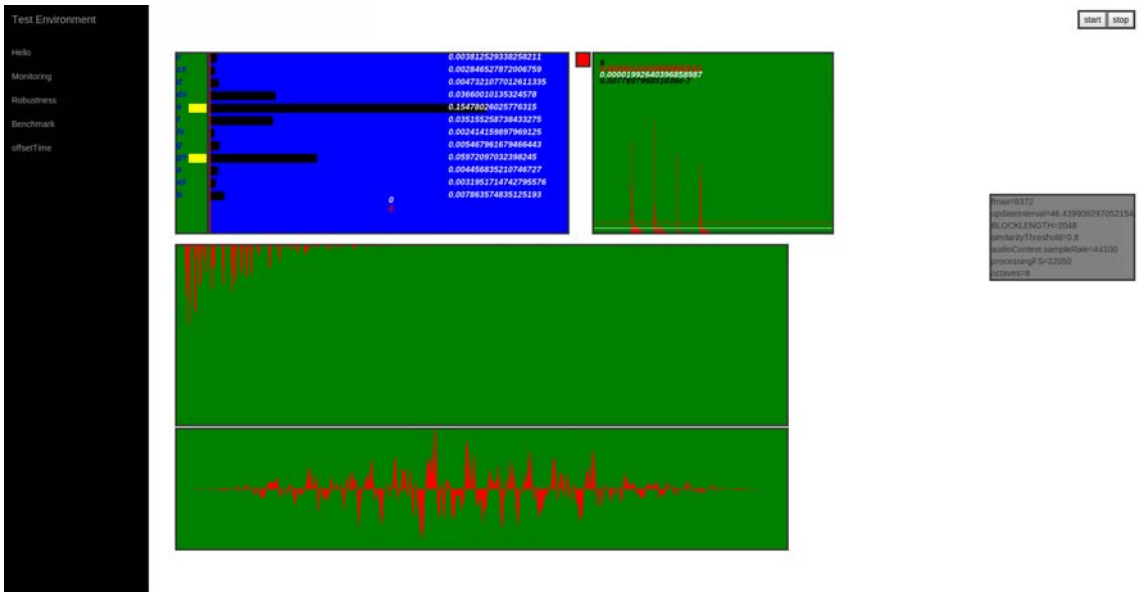

Abbildung 5.1: Benutzeroberfläche der Testumgebung mit Onset-, Chroma-, FFT- und Zeit-Monitoring

Da der *Score-Following*-Algorithmus als *Smart Package* in die Testumgebung integriert wird und so ein modulares System entsteht, ist es einfach, verschiedene Varianten des Algorithmus zu testen. Voraussetzung ist, dass alle Varianten des *Score-Following*-Algorithmus die gleichen Schnittstellen zur Testumgebung bereitstellen. In dieser Evaluierung sollen die im Kapitel 4 beschriebenen zwei Varianten der Vergleichsfunktion gegenübergestellt werden. Zwei weitere Varianten ergeben sich anhand zweier verschiedener Arten von Chroma-Vektoren, die der Vergleichsfunktion als User-Event übergeben werden.

5.2 Eventerkennung

Dieser Teil der Evaluierung widmet sich der Erkennungsleistung, indem einerseits Midigenerierte Samples und andererseits Samples einer Live-Performance untersucht werden. Die Samples der Live-Performance stammen aus dem Mitschnitt von verschiedenen Klavierstücken, die für die *flowkey*-Applikation eingespielt wurden.

Neben der Erfassung der Richtig-positive-Rate (bei Übereinstimmung von User- und erwarteten Events) wird auch die Falsch-positiv-Rate (bei Nicht-Übereinstimmung) gemessen. So wird überprüft, wieviel Prozent von User-Events irrtümlich als positiv klassifiziert

werden. Dazu werden die Midi-generierten Daten so verändert, dass Audiosamples und erwartete Events nicht übereinstimmen.

Um alle Audiosamples automatisch hintereinander abspielen zu können, wurde der *Score-Following*-Algorithmus so konfiguriert, dass mittels eines *JavaScript Buffer Loaders* alle Audiosamples geladen und sequentiell abgespielt werden können. Für jedes Audiosample existiert ein erwartetes Event. Über den Index der abgespielten Samples wird gewährleistet, dass stets das entsprechende erwartete Event zum Vergleich vorliegt.

Folgende Parameter wurden verwendet und bis auf den Onset-Schwellwert des *Peak-Picking*-Algorithmus für jeden Testlauf (aus einzelnen Samples bestehender Songausschnitt oder Midi-generierte Akkordfolge) und jede Variante des Algorithmus gleich gehalten:

- Blocklänge $b = 2048$ Samples
- Maximale zu analysierende Frequenz $f_{max} = 8372$ Hz
- Downsample-Faktor d = 2 (jedes zweite Sample wird aus dem Originalsignal herangezogen)
- Mindestlänge des Chroma-Buffers (entspricht Mindesthaltezeit) $h = 10$ Chroma-Vektoren
- Kosinusähnlichkeit Schwellwert $s_{thr} = 0{,}7$

Zur Evaluierung wurden drei Kategorien als Ergebnis eines einzelnen Event-Vergleiches definiert:

- als übereinstimmendes Event klassifiziert: Es wird ein User-Event per Onset erkannt, dessen Chroma-Vektor mit den Pitchklassen des erwarteten Events korrespondiert. In den hier gezeigten Diagrammen tritt diese Klassifizierung in Form der Richtig-positiv- oder der Falsch-positiv-Rate auf.
- als nicht übereinstimmendes Event klassifiziert: Es wird ein Event per Onset erkannt, dessen Chroma-Vektor nicht mit den Pitchklassen des erwarteten Events korrespondiert. In den hier gezeigten Diagrammen tritt diese Klassifizierung in Form der Falsch-negativ- oder der Richtig-negativ-Rate auf.
- Nicht erkannte Events: Dies stellt keine Klassifizierung im eigentlichen Sinne dar, tritt aber auf, wenn bei einem Audiosample kein Onset erkannt und somit auch kein Chroma-Vektor berechnet wird, also überhaupt kein User-Event erkannt wird. Dies kann der Fall sein, wenn der Onset-Schwellwert für den *Peak-Picking*-Algorithmus

zu hoch ist oder ein Peak nicht erkannt wird, wenn gerade noch die Haltezeit (also die Chroma-Extraktion) eines vorherigen Events läuft.

Bei initialen Versuchen konnte beobachtet werden, dass die Ergebnisse verschiedener Testdurchläufe, welche sowohl die gleiche Datengrundlage als auch identische Parameter nutzen, nicht zwingend die gleichen Resultate hervorbringen. Um dies zu kompensieren, wurde jeder Test dreimal durchgeführt und die Ergebnisse entsprechend gemittelt.

5.2.1 Test mit Live-Samples

Um Events zu testen, die nicht durch Midi generiert wurden, sondern eher den Charakter einer Live-Performance besitzen, wurden drei verschiedene Stücke aus der *flowkey*-Applikation ausgewählt und deren Audiospur in einzelne Events zerteilt, die der Testumgebung als User-Events übergeben werden.

Im Unterschied zu den Midi-generierten Events enthalten diese durch ihren Live-Charakter oft noch Signalanteile aus den vorhergehenden (noch ausklingenden) Events. Dies verfremdet den resultierenden Chroma-Vektor, was unter Umständen zu Schwierigkeiten beim Vergleich mit den erwarteten Events führt. Der Einfluss dieser Verfremdung soll hier untersucht und die verschiedenen Varianten des Algorithmus verglichen werden.

5.2.1.1 Testmaterial

Die Audiospuren dreier Videos, welche für die *flowkey*-Applikation als zu lernende Pianostücke produziert wurden, dienten als Grundlage der in diesem Test verwendeten Audiosamples. Das Audiosignal in den Videos stammt von einem E-Piano der Marke *Kawai* (Modell CN24). Grundlage der erwarteten Events bilden die bei einer Aufnahme anhand des E-Pianos mitgeschnittenen Midi-Daten, aus denen die JSON-Daten (siehe Abschnitt 4.5.2) für die Vergleichsfunktion erstellt werden. Aus der zum Zeitpunkt der Evaluierung noch begrenzten Auswahl an *flowkey* Piano-Stücken wurde je ein Auszug aus folgenden Songs ausgewählt:

- Jason Mraz, *I'm yours*, $\sim$ 65 BPM, kleinster Notenwert: Achtel-Note
- The Beatles, *Yesterday*, $\sim$ 75 BPM, kleinster Notenwert: Achtel-Note
- John Williams, *The Imperial March*, $\sim$ 115 BPM, kleinster Notenwert: Achtel-Note

5.2.1.2 Ergebnisse

Insgesamt wurden vier verschiedene Varianten des Algorithmus getestet, deren Ergebnisse im Folgenden aufgeführt werden. Während die ersten zwei Varianten der in Kapitel 4 beschriebenen Logik folgen, wurden die anderen beiden Varianten dahingehend modifiziert, dass statt des über die Haltezeit gemittelten Chroma-Vektors lediglich der letzte Chroma-Vektor als User-Event in die Vergleichsfunktion übergeben wird.

Richtig-positiv-Raten mit gemitteltem Chroma-Vektor Die ersten Messungen wurden mit dem in Kapitel 4 beschriebenem Algorithmus durchgeführt, in dem einer Vergleichsfunktion ein über die Zeit gemittelter Chroma-Vektor übergeben wird. Die beiden dort beschriebenen Vergleichsfunktionen wurden gegeneinander verglichen.

Im direkten Vergleich der Abbildungen 5.2(a) und 5.2(b) lässt sich erkennen, dass die Variante mit der in 4.7 beschriebenen alternativen Vergleichsfunktion auf Basis der Kosinus-Ähnlichkeit eine bessere Erkennungsleistung aufweist. Sie liefert im Vergleich zur Chroma-Schwellwert-basierten Vergleichsfunktion (4.6) eine größere Anzahl an übereinstimmend klassifierten (richtig-positiven) Events.

Die Richtig-positiv-Rate ist darüber hinaus stark abhängig vom untersuchten Stück. Besonders im schnellsten Stück (*Imperial March*) sinkt die Richtig-positiv-Rate auf lediglich 14,2 bwz. 21,3 %, während die Rate der nicht erkannten Events auf 44,0 bzw. 42,6 % steigt.

Richtig-positiv-Raten mit letztem Chroma-Vektor der Haltezeit Für die weiteren beiden Messungen wurde der Algorithmus dahingehend verändert, dass einer Vergleichsfunktion statt des über die Haltezeit gemittelten Chroma-Vektors lediglich der letzte Chroma-Vektor der Haltezeit übergeben wird. Theoretisch ist dieser Chroma-Vektor weniger repräsentativ für ein Event. Je nach Länge der Haltezeit kann dieser Chroma-Vektor jedoch eine deutlichere Differenz zwischen den tatsächlich erklingenden und den restlichen Pitchklassen aufweisen, was die Klassifizierung genauer macht.

Vergleicht man die beiden Diagramme 5.3(a) und 5.3(b) miteinander, so lassen sich die gleichen Beobachtungen wie schon bei den vorherigen Varianten machen. Die Kosinus-Ähnlichkeit-basierte Methode klassifiziert mehr Events korrekt als die Chroma-Schwellwert-basierte Methode.

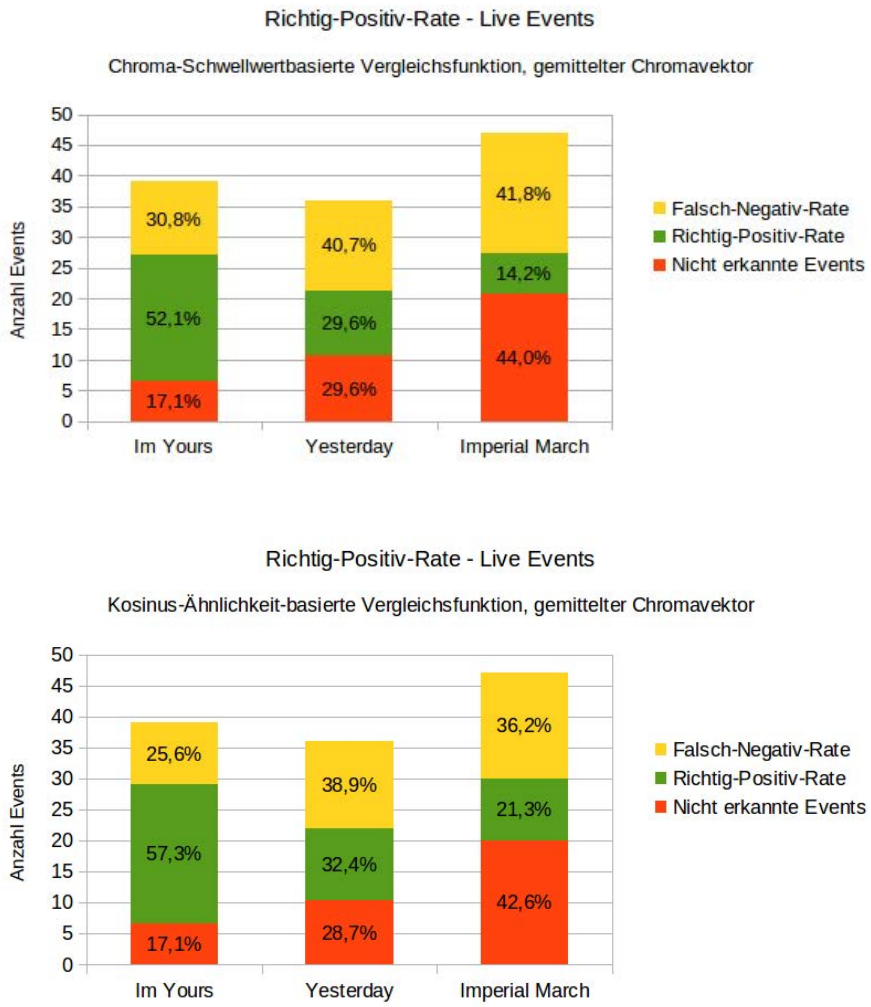

Abbildung 5.2: Richtig-positiv-Raten für Live-Samples: Vergleichsfunktionen mit über Haltezeit gemitteltem Chroma-Vektor

Weiterhin resultieren die in diesem Abschnitt betrachteten Algorithmusvarianten, die der Vergleichsfunktion den letzten Chroma-Vektor der Haltezeit übergeben, im direkten Vergleich mit 5.2(a) und 5.2(b) in besseren Richtig-positiv-Erkennungsraten.

5.2.2 Test mit Midi-generierten Samples

Während die Audiosamples der in 5.2.1 vorgestellten Testreihen auf live mitgeschnittenen Darbietungen basieren, verwenden die in diesem Abschnitt vorgestellten Testreihen ausschließlich Audiosamples, welche anhand von Midi-Daten erstellt und synthetisiert wurden. In dieser Testreihe soll neben der Richtig-positiv-Rate auch die Falsch-positiv-Raten

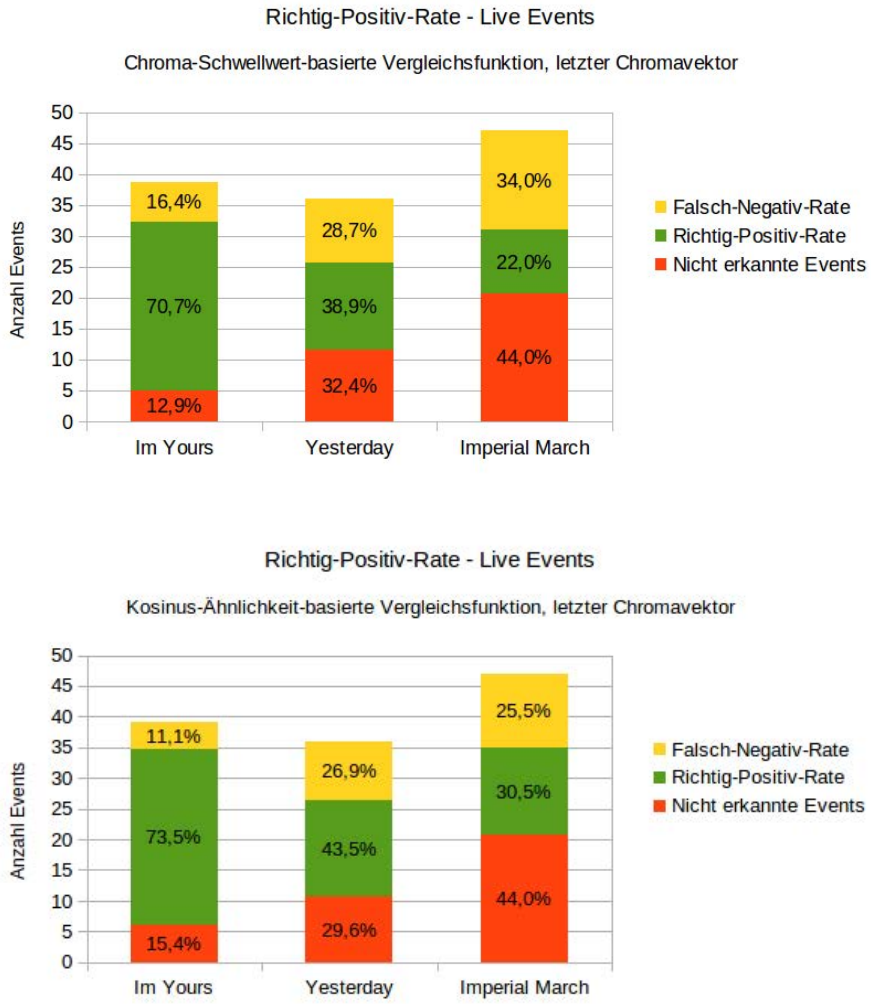

Abbildung 5.3: Richtig-positiv-Raten für Live-Samples: Vergleichsfunktionen mit letztem Chroma-Vektor der Haltezeit

der verschiedenen Varianten untersucht werden. Mit der Erfassung der Falsch-positiv-Raten kann eine Aussage darüber gemacht werden, welcher Anteil der fehlerhaften User-Events irrtümlich als positiv klassifiziert werden.

5.2.2.1 Testmaterial

Die Midi-generierten Akkorde wurden mit dem VST-Plugin *LePiano* [1] von Safwan Matni in *Ableton Live* erzeugt. *LePiano* ist ein simples VST-Instrument (VST: *Virtual Studio Technology*) mit nur einem Sample per Ton und simuliert den Klang eines Bar-Pianos.

[1] `http://www.kvraudio.com/product/le_piano_by_safwan_matni`

Die bei der Erstellung der Midi-Events verwendeten Variablen werden im Folgenden erläutert:

- Grundton: Drei verschiedene Grundtöne in Form der Töne C, E und A werden verwendet, um eine grundlegende Diversität in das Set von Audiosamples einzuführen.
- Oktave: Jeder in den Audiosamples enthaltene Akkord wird in einer niedrigen, einer mittleren und einer hohen Oktave, konkret die zweite, vierte oder sechste Oktave auf einer Standard-Klaviatur, gespielt. Es soll hierbei untersucht werden, wie sehr die Erkennungsleistung hin zu niedrigen und hohen Oktaven abnimmt.
- Akkordart: Für jeden Grundton und jede Oktave wurden vier verschiedene Akkordarten erstellt. Neben normalen Dur-2- und 3-Klängen wurden 3-Klänge mit zusätzlicher kleinen Septime, also ein (minor7)-4-Klang, benutzt. Eine weitere hier definierte Akkordart stellt ein 3-Klang dar, bei dem einer der enthaltenen Töne in einer weiteren Oktave auftritt (in dieser Arbeit als *3+1-Klang* bezeichnet). Diese Art von Akkord wird untersucht, da sie bei initialen Tests mit der Chroma-Schwellwert-basierten Vergleichsfunktion auffallend oft nicht erkannt wurde. Ein möglicher Grund ist, dass der doppelt auftretende Ton den Chroma-Vektor dominiert und die anderen beiden Töne untergehen.

Ein Beispiel für jede verwendete Akkordart kann der Abbildung 5.4 entnommen werden. Die Akkorde verwenden jeweils den Grundton C in der vierten Oktave. Insgesamt wurden mit drei Grundtönen in drei verschiedenen Oktaven mit jeweils vier Akkordarten insgesamt 36 Audiosamples anhand von Midi-Daten erstellt, welche sich anhand der oben genannten Variablen unterscheiden.
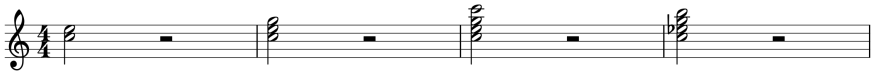
Abbildung 5.4: Beispiele der mittels Midi generierten Akkorde: C-2-Klang, C-3-Klang, C-3+1-Klang und Cm7-4-Klang

Bis hierher wurde beschrieben, wie aus den 36 verschiedenen Midisets die korrespondierenden Audiosamples zur Messung der Richtig-positiv-Rate synthetisiert werden. Die Midi-Daten werden in das *flowkey*-interne JSON-Format zur Repräsentation der erwarteten Events überführt. Die aus den Midis synthetisierten Audiosamples als User-Events stimmen soweit mit den erwarteten Events überein. Um auch zu testen, ob falsch gespielte User-Events auch als jene erkannt werden ist es nötig, entweder die User-Events oder die erwarteten Events zu verändern. In diesem Fall wurden die User-Events verändert, indem die zugrundeliegenden Midi-Daten manipuliert wurden, um fehlerhafte User-Events

zu synthetisieren. Die Fehler wurden eingebaut, indem eine der enthaltenen Noten jeweils um einen ganzen Ton nach unten oder oben verschoben wurde. Dies entspricht etwa einem häufigen Fehler bei Klavier-Anfängern, bei dem statt der eigentlichen Taste die benachbarte Taste gespielt wird. Als User-Events wurden insgesamt also 36 “korrekte“ als auch 36 “fehlerhafte“ Audiosamples synthetisiert und getestet.

5.2.2.2 Ergebnisse

Wie in 5.2.1.2 werden auch in diesem Test die vier verschiedene Varianten des Algorithmus getestet. Im Folgendem sind zunächst die Ergebnisse der beiden Varianten, die einen zeitlich gemittelten Chroma-Vektor zum Vergleich mit den erwarteten Events heranziehen, dargestellt.

Richtig-positiv-Raten mit gemitteltem Chroma-Vektor Die Abbildungen 5.5(a) und 5.5(b) zeigen die Richtig-positiv-Raten nach Akkordarten unterteilt. Zunächst lässt sich erkennen, dass die Midi-generierten Events im Gegensatz zu den Live-Events aus Abschnitt 5.2.1 stets auch als User-Events erkannt werden und es keine Ausfälle gibt. Zurückzuführen ist dies auf die bei der Erstellung der Midi-Daten konstanten Parameter (etwa für Anschlagsstärke), welche bei Live-Aufführengen stark variieren und somit eine große Dynamik hinsichtlich der Höhe der Peaks in der Onset-Erkennungsfunktion entsteht. Die Höhe der Peaks ist mit den Midi-generierten Audiosamples dagegen stets gleich, was die Onset-Erkennung durch Setzen eines festen Schwellwertes für das *Peak Picking* trivial macht.

Die höchste Richtig-positiv-Rate wird mit 66,7 % bei der Kosinus-Ähnlichkeit-basierten Vergleichsfunktion für die 3+1-Klänge erreicht (Abb. 5.5(b)). Die Chroma-Schwellwert-basierte Methode resultiert durchweg in niedrigeren Richtig-positiv-Raten, wobei sie ihr bestes Ergebnis bei den 2-Klängen (Abb. 5.5(a)) erreicht. Dies entspricht weitgehend den Beobachtungen der vorherigen Tests mit live mitgeschnittenen Audiosamples.

Richtig-positiv-Raten mit letztem Chroma-Vektor der Haltezeit Zunächst lässt sich aus den Ergebnissen in Abb. 5.6(a) und 5.6(b) erkennen, dass die Kosinus-Ähnlichkeit-basierte Vergleichsfunktion besser abschneidet als die Chroma-Schwellwert-basierte Vergleichsfunktion. Im Schnitt ist die Erkennungsrate mehr als doppelt so hoch.

Beide Algorithmusvarianten, die zum Vergleich den letzten Chroma-Vektor der Haltezeit verwenden, schneiden besser als die Varianten mit gemitteltem Chroma-Vektor ab. Bei

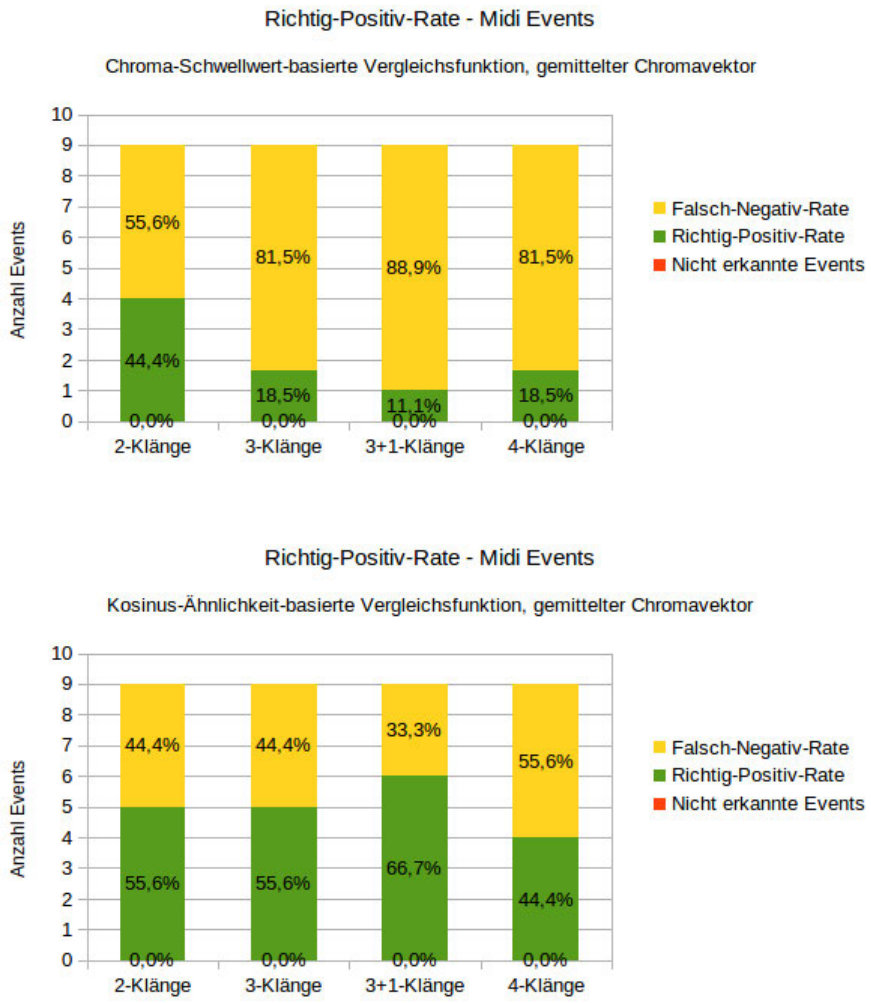

Abbildung 5.5: Richtig-positiv-Raten für Midi-Samples: Vergleichsfunktionen mit über Haltezeit gemitteltem Chroma-Vektor

allen vier Algorithmusvarianten ließ sich während des Tests beobachten, das besonders die Events in der sechsten Oktave des Pianos überdurchschnittlich oft nicht erkannt werden.

In der Chroma-Schwellwert-basierten Vergleichsfunktion werden die 3+1-Klänge wie erwartet am schlechtesten erkannt, da hier ein Ton des 3-Klangs doppelt auftritt und den Schwellwert so weit wachsen lässt, dass andere Pitchklassen ihn nicht überschreiten. Insgesamt bleibt zudem festzustellen, dass bei drei von vier Varianten die 2-Klänge am besten erkannt wurden. Nur in der Variante mit der Kosinus-Ähnlichkeit-basierten Vergleichsfunktion mit gemitteltem Chromavekor werden die 3+1-Klänge sogar am besten von allen Akkordarten erkannt.

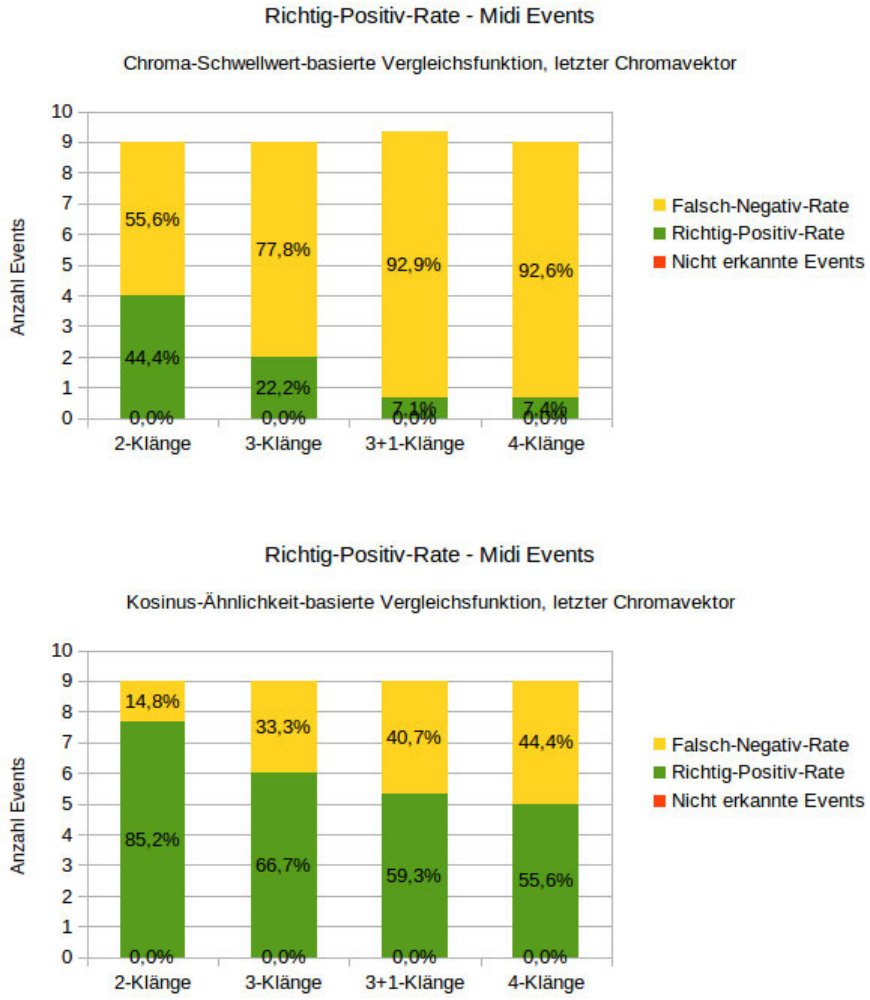

Abbildung 5.6: Richtig-positiv-Raten für Live-Samples: Vergleichsfunktionen mit letztem Chroma-Vektor der Haltezeit

Falsch-positiv-Raten Aufgrund der schlechten Erkennungsraten der Chroma-Schwellwert-basierten Vergleichsmethoden werden in diesem Abschnitt lediglich die Falsch-positiv-Raten der Algorithmus-Varianten mit der Kosinus-Ähnlichkeit-Vergleichsfunktion betrachtet.

Aus den Abbildungen 5.7(a) und 5.7(b) geht zunächst hervor, dass die Algorithmus-Variante mit gemitteltem Chroma-Vektor weniger (tatsächlich falsch gespielte) Events als positiv (also *korrekt gespielt*) klassifiziert als die Variante mit dem letzten Chroma-Vektor der Haltezeit. Vergleicht man die Abbildungen mit ihren entsprechenden Gegenstücken für die Richtig-positiv-Raten, lässt sich sagen, dass die Variante mit der besseren Richtig-positiv-Rate in Abb. 5.6(b) die schlechtere Falsch-positiv-Rate (Abb. 5.7(b)) aufweist.

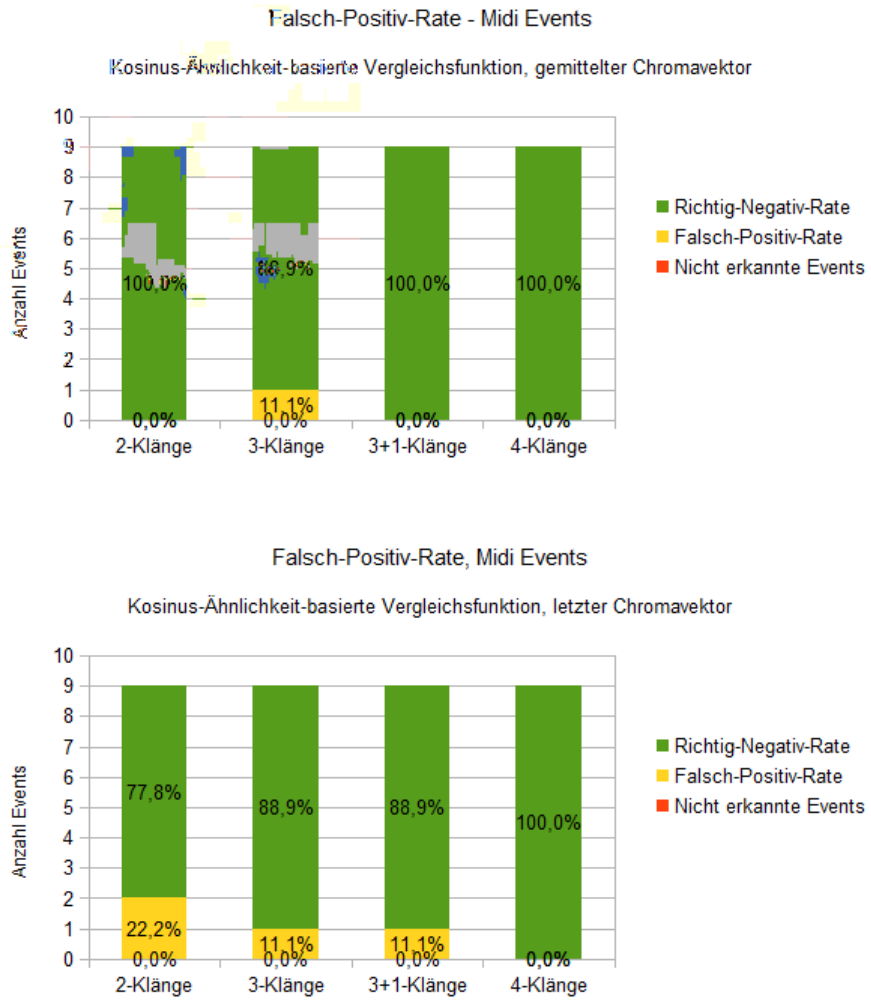

Abbildung 5.7: Falsch-positiv-Raten für Midi-Samples: Varianten mit Kosinus-Ähnlichkeit-basierter Vergleichsfunktion

5.3 Echtzeitfähigkeit

Um zu validieren, ob die benötigte Verarbeitungszeit des Algorithmus kleiner als die Blocklänge ist, werden Benchmarks zur Messung der Zeit, die der Algorithmus zur Verarbeitung eines Signalblocks benötigt, durchgeführt.

Um die Laufzeit zu messen, wurde die *now()*-Funktion des *Performance Interface* genutzt, welches der Chrome Browser implementiert. Die Funktion *performace.now()* liefert einen Zeitstempel relativ zum Start der Navigation im HTML-Dokument [47], der bis auf eine tausendste Millisekunde genau ist. Die Differenz der am Anfang und am Ende gemessener Werte bildet die Laufzeit des Algorithmus. Die Anfangszeit wird in einem für den Bench-

marktest hinzugefügten *Script Processor Node* bestimmt. Dieser wird zwischen Quelle und Tiefpassfilter geschaltet und macht nichts anderes als das Signal unverändert durch zu leiten und einen Zeitstempel für die Startzeit zu setzen. Dies *Script Processor Node* ist nötig, da der Tiefpassfilter, der eigentlich direkt an die Quelle angeschlossen ist, kein *onaudioprocess*-Event besitzt, in dessen Callback-Funktion man den Zeitstempel setzen könnte. Dieser zusätzliche *Script Processor Node* führt also eine weitere Verzögerung ein. Es wird jedoch angenommen, dass diese vernachlässigbar klein ist. Das Ende der *update()*-Funktion im Algorithmus stellt auch das Ende für die Bearbeitung eines Signalblocks dar. Hier wird die Endzeit für das Verarbeiten eines Blockes gemessen und in einem Array zwischengespeichert. Erreicht der Index die Länge des Puffers (*benchmarkTimes*), in dem die Zeitwerte gespeichert werden, so wird die durchschittliche Laufzeit berechnet und auf der Konsole des Browsers ausgegeben.

Für den Test wurde der Beginn des Stücks *I'm yours* von Jason Mraz in einer in der *flowkey*-Applikation vorhandenen Piano-Version verwendet. Es wurden die Laufzeiten von 200 Signalblöcken in einem Array zwischengespeichert, dass dem Score-Follower-Package übergeben wird. Aus diesen Werten kann der Mittelwert für die Laufzeit berechnet werden. Das graphische Monitoring über die Benutzeroberfläche der Testumgebung wurde während der Durchführung der Tests ausgeschaltet, um CPU-Zeit einzusparen.

Ein Notebook der Marke *Asus* mit der Bezeichnung *UX32VD* wurde benutzt, um die Messungen durchzuführen. Als Prozessor kommt in dem System ein *Intel Core i7* (zwei Kerne/ vier Threads, 1,9 bis 2,4 GHz) zum Einsatz. Die Größe des Arbeitsspeichers beträgt 4 GByte. Als Betriebssystem wurde *Ubuntu 13.04* (64-bit) und als Browser *Chrome 31* verwendet.

Wie man aus Tabelle 5.1 ablesen kann, unterscheiden sich die durchschnittlichen Laufzeiten der verschiedenen Algorithmusvarianten nur geringfügig. Es kann angenommen werden, dass die Unterschiede zufällig sind und nicht durch die verschiedenen Algorithmen selbst hervorgerufen werden.

	Kosinus-Ähnlichkeit	Chroma-Schwellwert
gemittelter Chroma-Vektor	2,07 ms	2,01 ms
letzter Chroma-Vektor	2,04 ms	2,11 ms

Tabelle 5.1: Mittelwerte der Laufzeiten eines Signalblocks für alle Varianten des Algorithmus

5.4 Diskussion der Ergebnisse

5.4.1 Live-Samples

Wie schon im Abschnitt 5.2.1.2 festgestellt wurde, erzielt die Kosinus-Ähnlichkeit-basierte bessere Ergebnisse als die Chroma-Schwellwert-basierte Vergleichsfunktion. Eine weitere Aussage, die man auf Grundlage der Tests machen kann ist, dass die beiden Algorithmus-Varianten, die nur den letzten Chroma-Vektor zum Vergleich heranziehen bessere Resultate als jene mit gemitteltem Chroma-Vektor hervorbringen.

Während der Tests ließ sich bei der Chroma-Schwellwert-basierten Vergleichsfunktion beobachten, dass oft mindestens eine der erwarteten Pitchklassen nicht genug Energie besaß, um den aktuellen Schwellwert ihres Chroma-Vektors zu überschreiten. Grund dafür kann etwa sein, dass die Schallenergie von vorangegangenen Events mit in den Chroma-Vektor des aktuellen User-Events einfließt und so den Schwellwert anhebt. Dies führt dazu, dass das Event als negativ bzw. nicht korrekt gespielt klassifiziert wird, was sich in der niedrigeren Richtig-positiv-Rate für die beiden Chroma-Schwellwert-basierten Varianten widerspiegelt.

Einer von zwei möglichen Gründen dafür, dass die beiden Varianten mit gemitteltem Chroma-Vektor schlechter abschneiden, könnte ebenfalls in dem Nachklingen vorangegangener Events bestehen. Der über die Haltezeit gemittelte Chroma-Vektor enthält offensichtlich mehr Signalanteile eines vergangenen User-Events, sofern es noch ausklingt. Im Gegensatz dazu besteht bei der Variante mit dem letzten Chroma-Vektor die Möglichkeit, dass das vorangegangene Event zum Ende der Haltezeit weitestgehend ausgeklungen hat. Somit fließen nur Signalanteile des aktuellen Events in den zu analysierenden Chroma-Vektor. Dies hat die bessere Richtig-positiv-Rate mit zur Folge.

Ein zweiter möglicher Grund um die unterschiedlichen Richtig-positiv-Raten bezüglich der Art des zu vergleichenden Chroma-Vektors zu erklären, kann auch mit dem zeitlichen Verlauf des aktuellen Events zu tun haben. Betrachtet man den Zeitverlauf einer einzelnen angeschlagenen Note in Abb. 5.8, so lässt sich der Verlauf idealerweise in verschiedene Phasen bzw. Zeitpunkte aufteilen. In [3] beschreiben Bello et al. die Phasen Onset, Attack, Transient und Decay. Im gemittelten Chroma-Vektor werden alle diese Phasen verarbeitet, sofern die Haltezeit lang genug ist. Besonders in der Transient-Phase treten breitbandige Signalanteile auf, die alle Pitchklassen des Chroma-Vektors ansteigen lassen. So wird der resultierende gemittelte Chroma-Vektor unter Umständen zusätzlich verfälscht. Zieht man dagegen lediglich den letzten Chroma-Vektor der Haltezeit heran, befindet der sich wahrscheinlich in der Decay-Phase. Man erhält einen Chroma-Vektor,

in dem die Differenz der auftretenden zu den nicht auftretenden Pitchklassen womöglich größer ist, was zu einer besseren Klassifizierung führt. Voraussetzung dafür ist, dass die Haltezeit so lang ist, damit die Transient-Phase zwar abgeschlossen ist, das Event aber noch laut genug erklingt.

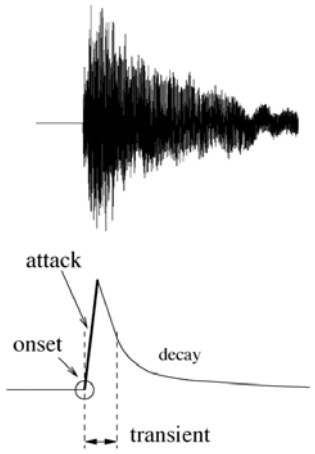

Abbildung 5.8: Zeitlicher Verlauf eines Tons auf einem Piano [3]

Für die Tests mit den Live-Samples gilt, dass die Anzahl der nicht erkannten Events in allen Varianten ähnlich hoch ist. Da die Onset-Erkennung für das bloße Erkennen eines Events verantwortlich ist und diese in beiden Varianten gleich ist, sind diese ähnlichen Ergebnisse nicht überraschend. Mögliche Ursache für die besonders im dritten Stück (*Imperial March*) extrem hohe Anzahl an nicht erkannten Events ist das vergleichsweise hohe Tempo des Stücks. Dadurch, dass eine Mindesthaltezeit festgelegt wurde, kann es bei schnellen Stücken vorkommen, dass sich die Analyse noch in der Haltezeit eines bestimmten Events befindet, während schon das nächste Event auftritt. Dadurch können Onset verloren gehen und neue Events werden nicht erkannt. Weiterhin kann es auch vorkommen, dass einzelne Audiosamples tatsächlich einen zu geringen Ausschlag in der Onset-Erkennungsfunktion (*Spectral Flux*) vorweisen und deswegen kein Onset auftritt. Weiterhin können auch falsche Onsets auftreten, da die Audiosamples per Hand im Audio-Editor zerschnitten wurden und die spektrale Kontinuität zwischen zwei Audiosamples verloren gehen kann, was zu einem Ausschlag des *Spectral Flux* an einer Stelle ohne tatsächlichen Tastenanschlag führen kann.

5.4.2 Midi-Samples

Richtig-positiv-Rate: Grundsätzlich kann als Ergebnis festgehalten werden, dass auch in der Midi-Testreihe die Kosinus-Ähnlichkeit-basierte Vergleichsfunktion besser als die Chroma-Schwellwert-basierte Methode arbeitet. Darüber hinaus lässt sich bei fast allen Diagrammen eine deutliche Tendenz hin zu besseren Richtig-positiv-Raten feststellen, je weniger einzelne Töne in einem Akkord vorhanden sind.

Zwar wurden die Erkennungsraten in den Diagrammen nur nach Akkordart und nicht nach Oktaven aufgeschlüsselt, während der Tests ließ sich jedoch beobachten, dass unter den als falsch-negativ eingestuften Events überdurchschnittlich viele waren, die in der sechsten Oktave angelegt waren. Grund dafür kann die Tatsache sein, dass diese darin auftretenden Töne kaum Obertöne besitzen. Vergleichsweise wenig Energie enthalten sie also gegenüber Tönen in tiefen oder mittleren Oktaven. Im Chroma-Vektor war bei diesen hoch-oktavigen Events kein signifikanter Ausschlag bei den erwarteten Pitchklassen zu beobachten. Stattdessen erfuhr beinahe jede Pitchklasse einen leichten, aber nur kurzweiligen Anstieg der Energie. Dies ließe sich damit interpretieren, dass diese hoch-oktavigen Events ein eher breitbandiges Spektrum besitzen, das von den Transienten beim Notenanschlag dominiert wird.

Bei den Events, die in der zweiten Oktave angelegt waren, gab es weniger Probleme, obwohl dort der Abstand der Grundfrequenzen kleiner als die Frequenzauflösung der FFT ist (vgl. Abschnitt 4.2.3). Grund dafür ist die hohe Anzahl an Obertönen, die dafür sorgt, dass die entsprechenden Pitchklassen deutlich im Chroma-Vektor hervorgehen. Versuche zeigten jedoch auch, dass Töne in der ersten Oktave zu ungenau abgebildet werden, als dass eine Kompensierung über deren Obertöne möglich ist.

Falsch-positiv-Rate: Anhand der geringen Falsch-positiv-Raten in den Abbildungen 5.7(a) und 5.7(b) lässt sich feststellen, dass der Algorithmus weitestgehend zuverlässig inkorrekt gespielte User-Events als solche erkennt. Hervorzuheben ist dabei allerdings, dass je schlechter (niedriger) die Richtig-positiv-Rate einer Algorithmus-Variante ist, desto besser (niedriger) die Falsch-positiv-Rate ist, wie man auch an den beiden Varianten mit Kosinus-Ähnlichkeit-basierter Vergleichsfunktion (5.7(a) und 5.7(b)) beobachten kann. Die beiden Varianten mit Chroma-Schwellwert-basierter Vergleichsfunktion sind nicht abgebildet. Sie produzierten jedoch Falsch-positiv-Raten von 0 bzw. nahe 0 %. Somit zeigt sich, dass eine geringe Falsch-positiv-Rate noch kein Indikator für eine zuverlässige Klassifizierung im Allgemeinen ist.

5.4.3 Echtzeit-Test

Die in Abschnitt 2.5 beschriebene Ungleichung 2.2, nach der Verarbeitungszeit eines Blocks und zusätzliche Zeit für Ein- und Ausgabeoperationen die zeitliche Länge eines Signalblocks nicht überschreiten dürfen, soll im Folgenden mit Hinsicht auf die Ergebnisse der mittleren Verarbeitungszeiten in 5.3 diskutiert werden.

Die zeitliche Länge eines Signalblocks ergibt sich aus Samplerate und Blocklänge. Geht man von einer Samplerate von 44100 Hz aus (wie es bei den meisten internen Soundkarten der Fall ist) sowie von einer Blocklänge von 2048 Samples, ergibt sich eine zeitliche Blocklänge von $\frac{2048}{44100Hz} = 0{,}04644$ s , was 46,44 ms entspricht. Die ermittelten Durchschnittszeiten von knapp über 2 ms liegen also deutlich unter der zeitlichen Länge eines Signalblocks. Sie wurden mit ausgeschaltetem Monitoring ermittelt. Das heisst, dass die graphischen Anzeigeelemente (HTML <canvas> - Elemente) der Testumgebung keine Ausgaben tätigen. Schaltet man das Monitoring hinzu, steigt die gesamte Verarbeitungszeit auf durchschnittlich 5,79 ms an. Auch dieser Wert liegt noch deutlich unter den geforderten 46 ms.

Trotz der zunächst vergleichsweise niedrigeren Werte mit viel Spielraum nach oben ist es nicht ausgeschlossen, dass auf älteren Systemen oder solchen, auf denen noch weitere Anwendungen parallel laufen, es zu Blockverarbeitungszeiten kommt, die über die zeitliche Blocklänge von 46 ms hinausgehen.

5.5 Sonstige Probleme

5.5.1 *Score Follower* und *flowkey*-Player

Bei ersten Tests der *flowkey*-Applikation stellte sich heraus, dass die Handhabung der Warte-Funktion, die den *Score-Following*-Algorithmus beinhaltet, nicht den flüssigen Charakter besitzt, der ursprünglich angestrebt wurde. Dies mag zum einen daran liegen, dass der Algorithmus noch nicht jedes mit dem erwarteten Event übereinstimmende User-Event auch als solches erkennt. Das führt dazu, dass der Benutzer bestimmte Events mehrere Male wiederholen muss, bis diese erkannt werden.

Zum anderen wird der Spielfluss jedoch auch bei trivialen Events (etwa bei einzelnen Tönen) unterbrochen, die der Algorithmus problemlos korrekt klassifizieren kann. Offensichtlich sind die Pausen des Players zwischen den Events dafür verantwortlich. Grund dafür ist dessen Funktionsweise, bei der der Player das Video weiterspielen lässt, wenn

erwartetes Event und User-Event übereinstimmen. Er pausiert daraufhin wieder kurz vor dem nächsten erwarteten Event, um auf das entsprechende User-Event zu warten. Dies unterbricht den Spielfluss. Zwar ist diese Funktionweise zunächst gerechtfertigt, weil sie das Prinzip des *Score-Following*-Algorithmus widerspiegelt, welcher ebenfalls nur ein Event abarbeitet und dann auf das nächste wartet, bedarf offensichtlich jedoch dennoch einer Verbesserung.

5.5.2 Oktavunabhängige Analyse

Eine Eigenschaft des Algorithmus, die in bestimmten Szenarien nachteilig ist, ist die Oktavunabhängigkeit der Analyse. Da bei der Chroma-Extraktion alle Töne der gleichen Pitchklasse auf einen Wert aufsummiert werden, ist es nicht ohne Weiteres möglich, die Oktave zu bestimmen, in der die einzelnen Töne des User-Events gespielt werden.

Das führt dazu, dass ein Ton in jeder beliebigen Oktave gespielt werden kann. Besteht ein erwartetes Event etwa aus einem Ton C in der vierten Oktave und der Benutzer würde ein C in der dritten Oktave spielen, würde das auftretende User-Event als positiv (*korrekt gespielt*) klassifiziert werden.

So kann es ebenfalls etwa vorkommen, dass ein erwartetes Event den selben Ton in unterschiedlichen Oktaven enthält (z. B. 3+1-Akkorde). Für eine korrekte positive Klassifizierung des User-Events (*korrekt gespielt*) würde es genügen, den Ton in nur einer der geforderten zwei Oktaven zu spielen.

5.5.3 Sprache und Hintergrundgeräusche

Die Onset-Erkennung löst unter Umständen auch User-Events aus, die nicht in einem Tastenanschlag begründet liegen. So können etwa Sprache oder Hintergrundgeräusche ein User-Event auslösen, welches in einigen Fällen auch als irrtümlich positiv klassifiziert wird.

5.6 Mögliche Verbesserungen

In diesem Abschnitt werden einige Ansätze vorgestellt, mit denen die zuvor beschriebenen Probleme wie etwa die hohe Rate an nicht erkannten Events, das noch stockende Zusammenspiel mit dem *flowkey*-Player oder die oktavunabhängige Analyse behandelt werden können.

5.6.1 *Score Follower* und *flowkey*-Player

Die in 5.5.1 beschriebene Methode lässt das im *flowkey*-Player enthaltene Video und die Partitur stets beim nächsten erwarteten Event pausieren. Für die künftige Weiterentwicklung kann auch eine Alternative in Frage kommen, bei der der Player solange Video und Partitur fortlaufen lässt, bis das erwartete Event nicht mehr mit dem letzten User-Event übereinstimmt. Eine Möglichkeit dies umzusetzen könnte die Einführung eines Timers sein. Der Timer wird gestartet, wenn ein als positiv klassifiziertes User-Event auftritt. Während der Timer läuft werden Video und Partitur weiter abgespielt. Das Ende des Timers liegt nach dem nächsten erwarteten Event. Tritt das entsprechende User-Event innerhalb des Timer-Zeitintervalls ein, wird der Timer neu gestartet, während das Video immer noch weiter läuft. Bleibt das User-Event aus, läuft der Timer ab und das Video stoppt. Auf diese Weise kann theoretisch ein flüssigerer Ablauf gewährleistet werden, da das Video nicht per se stehen bleibt, sondern nur wenn das nächste erwartete User-Event innerhalb eines bestimmten Zeitintervalls nicht eintrifft.

Zum Zeitpunkt der Fertigstellung dieser Arbeit wurde an einer weiteren Modifikation des Vergleichs-Algorithmus gearbeitet, die die Erkenntnisse aus dieser Evaluierung aufgreift. Im Wesentlichen wird dabei die Idee eines über die Zeit gemittelten Chroma-Vektors verworfen. Auch wird nicht mehr der letzte Chroma-Vektor der Haltezeit in die Vergleichsfunktion gegeben. Das Konzept, dass nach einem erkannten Onset die Chroma-Extraktion angeschoben wird, bleibt jedoch erhalten. Neu ist, dass jeder extrahierte Chroma-Vektor mit dem erwarteten Event verglichen wird. Die Vergleichsfunktion wird also pro User-Event mehrmals aufgerufen statt nur einmal. Dies geschieht solange, bis eine positive Klassifizierung vorliegt oder die maximale Analysezeit (vorher Haltezeit) erreicht ist. Erste Tests am Klavier zeigten bereits eine merkliche Verbesserung hinsichtlich der Erkennungsrate und Reaktionszeit. Grund dafür ist offensichtlich, dass nicht mehr die gesamte Haltezeit abgearbeitet werden muss, sondern (im Idealfall) nur ein Bruchteil davon. Nachteil ist ein erhöhter Rechenaufwand aufgrund des mehrmaligen Aufrufens der Vergleichsfunktion.

5.6.2 Oktavunabhängige Analyse

Ein weiteres Problem des Score-Follower-Algorithmus ist die oktavunabhängige Analyse (vgl.5.5.2). Welche Töne in welcher Oktave bzw. mit welcher genauen Frequenz gespielt werden, lässt sich theoretisch mit einem Algorithmus lösen, der mehrere Grundfrequenzen in einem Signal erkennen kann (*Multiple Pitch Detection*). Diese Disziplin stellt der-

zeit ein reges Forschungsfeld dar. Denkbar ist es, sowohl Chroma-Extraktion als auch eine *Multiple Pitch Detection* parallel laufen zu lassen und die Ergebnisse zu aggregieren.

Eine andere Möglichkeit könnte die Einführung eines Oktav-Features sein, dass etwa angibt, in welchem Frequenzbereich sich der aktuell größte Anteil an Energie befindet. Ein potentielles Feature dafür ist der sogenannte *Spectral Centroid*. Er gibt an, wo sich der Mittelpunkt der im Spektrum verteilten Energie befindet. Definiert wird er als Frequenzgewichtete Summe des Leistungsspektrum, welche mit der ungewichteten Summe normalisiert wird [2]. Allerdings liefert diese Feature lediglich einen einzigen Schwerpunkt im Spektrum. Zwei Hände am Klavier würden dagegen die Kenntnis zweier Schwerpunkte erfordern, um eine verlässliche Aussage über die Oktaven zu treffen, in der sich ein User-Event befindet.

5.6.3 Andere grundlegende Ansätze

Statt die Chroma-Vektoren aus einem mittels Fourier-Transformation erzeugten Spektrum heraus zu berechnen, kann die Chroma-Extraktion auch auf Basis einer Filterbank geschehen. Müller beschreibt in [48] etwa eine Multirate-Filterbank, die mit drei verschiedenen Samplefrequenzen (je eine für den unteren, mittleren und hohen Frequenzbereich) arbeitet. Die Filterbank kann etwa innerhalb der Web Audio API erzeugt werden. Dazu kann für jeden Ton auf der Klaviatur ein *BiquadFilterNode* der Web Audio API als Bandpassfilter erzeugt werden. Vorteil gegenüber der Fourier-Transformation, die in dieser Arbeit in JavaScript erfolgt, ist die Berechnung im optimierten C-Code der Browserimplementierung der Web Audio API. Auch wäre es möglich, die Onset-Erkennung mit den Werten der Filterbank durchzuführen statt auf Basis eines FFT-Spektrums. Die Funktion für den *Spectral Flux* würde etwa statt eines Arrays mit 1024 Stellen (Stützstellen des Spektrums) ein Array mit 96 Stellen (Anzahl der Filter) als Input bekommen. Dies hätte auch den Vorteil, dass die Onset-Erkennung besser auf die Töne eines Klaviers reagiert, statt auf das ganze Spektrum.

Eine weitere Möglichkeit, die einen grundlegend anderen Ansatz erfolgt, ist das sogenannte *Dynamic Time Warping* wie es auch schon in Abschnitt 1.3 beschrieben wurde. Ein solcher Algorithmus würde die User-Events nicht mit einzelnen theoretischen erwarteten Events abgleichen. Vielmehr wird ein User-Event bzw. dessen Chroma-Vektor auf einen bestimmten Zeitpunkt in einem Chromagramm abgebildet. Dieses Chromagramm könnte etwa aus der Audiospur des *flowkey*-Videos eines Stücks stammen. Da man bei diesem Ansatz davon ausgeht, dass ein Stück vom Musiker bereits beherrscht wird, muss

dieser entsprechend so modifiziert werden, dass inkorrekt gespielte User-Events nicht abgebildet werden.

5.7 Zusammenfassung

Evaluiert und gegeneinander verglichen wurden vier verschiedene Varianten des Algorithmus. Gegenstand der Evaluierung waren Erkennungsleistung und Performanz. Die Erkennungsleistung wurde getestet, in dem sowohl Midi-generierte als auch Live-Soundsamples den Algorithmen als User-Events übergeben wurden. Zu jedem dieser User-Events existiert ein korrespondierendes erwartetes Event. Die Anzahl der während des Vergleichs als positiv klassifizierten Events wurde für jede Algorithmus-Variante als Richtig-positiv-Rate festgehalten. Anhand fehlerbehafteten Midi-genierten Samples konnten weiterhin auch die Falsch-positiv-Raten bestimmt werden. Die Algorithmus-Variante mit Chroma-Schwellwert-basierter Vergleichsfunktion und gemitteltem Chroma-Vektor schnitt dabei am schlechtesten ab, während ihr Gegenstück (Kosinus-Ähnlichkeit-basierte Vergleichsfunktion mit letztem Chroma-Vektor der Haltezeit) hinsichtlich der Richtig-positiv-Raten die besten Resultate erzeugte. Die Anzahl an erwarteten Pitchklassen in einem erwarteten Event spielt zudem eine große Rolle. Je mehr Pitchklassen bzw. Töne ein Event besitzt, desto schwieriger ist es wiederzuerkennen.

Mit Blick auf die Performanz konnten keine signifikanten Unterschiede zwischen den Algorithmus-Varianten festgestellt werden. Auf dem in der Evaluierung eingesetzten Testsystem benötigten alle Varianten für die Verarbeitung eines 2048 Samples langen Signalblock ca. 2 ms (ohne graphisches Monitoring als Ausgabe), was deutlich unter der zeitlichen Länge eines Signalblocks von $\frac{2048}{44100Hz} = 46$ ms liegt. Zukünftige Messungen müssen noch zeigen, ob diese Zeiten auch innerhalb der fertigen *flowkey*-Applikation mit zusätzlichem Rechenaufwand durch Animationen und Video-Dekodierung erreicht werden können.

Letztendlich wurden in diesem Kapitel noch potentielle Verbesserungen für die künftige Weiterentwicklung des *Score-Followers* beschrieben. Statt einer Fourier-Transformation könnte etwa eine Filterbank zum Einsatz kommen, auf deren Basis Chroma-Extraktion und Onset-Erkennung ausgeführt werden können. Zudem wurde eine weitere Variante einer Vergleichsfunktion beschrieben, die bei ersten Tests auf dem Klavier gute Ergebnisse erzielte. Der in der Disziplin des *Score-Following* verbreitete Ansatz des *Dynamic Time Warpings* könnte ebenfalls in Zukunft in Betracht gezogen werden.

Zuletzt fehlen in dieser Evaluierung noch einige wesentliche Punkte. So wurde der Algorithmus bisher nicht auf akustischen Klavieren getestet, welche leicht verstimmt sein können. Zwar bietet die Chroma-Extraktion anhand einer Toleranz-Option die Möglichkeit, auch die benachbarten Bins einer Mittenfrequenz miteinzuziehen, was eine leichte Verstimmung kompensieren kann. Die praktische Funktionsweise dessen wurde in dieser Arbeit jedoch nicht evaluiert. Auch der Einfluss der Signalquelle war kein Untersuchungsgegenstand. So sollte noch geklärt werden, ob die in Laptop und Tablet-PCs integrierten Mikrofone signifikant schlechtere Ergebnisse als etwa Mikrofone für den Studioeinsatz liefern.

Kapitel 6

Zusammenfassung und Fazit

Dieses Kapitel gibt ausgehend von den Anforderungen der *flowkey*-Applikation nochmals einen Überblick über den gesamten Entwicklungsprozess des in dieser Arbeit präsentierten *Score-Following*-Algorithmus. Mit Hinblick auf die Ergebnisse der in Kapitel 5 beschriebenen Evaluierung wird anschließend diskutiert, ob die Anforderungen in ausreichendem Maß erfüllt wurden.

6.1 Entwicklungsprozess

Die *flowkey*-Applikation, welche eine webbasierte *E-Learning*-Anwendung für das Erlernen von Tasteninstrumenten darstellt, stellt dem Benutzer sowohl die Partitur des zu erlernenden Stücks als auch ein Lernvideo bereit. Das Video zeigt die Aufführung eines Stücks durch einen Musiker und ist in der Auf-Sicht auf das Klavier gefilmt, so dass die Spielweise des Stücks nachvollzogen werden kann. Die Partitur läuft stets synchron zum Video mit. Die Anwendung besitzt eine Warte-Funktion, die den *Score-Following*-Algorithmus einbettet. Diese Warte-Funktion pausiert das Video am aktuell erwarteten Event und erlaubt es dem Benutzer, durch Spielen der entsprechenden Töne das Video und die Partitur zum nächsten erwarteten Event weiter laufen zu lassen.

Auf Basis dieses Konzeptes wurde der *Score-Following*-Algorithmus entwickelt. Im Unterschied zum klassischen *Score-Following* gilt es, die vom Benutzer ausgelösten Schallereignisse (User-Events) auf Fehler zu überprüfen und gegenbenfalls als negativ (*inkorrekt gespielt*) zu klassifizieren. Drei zentrale Anforderungen ergeben sich aus diesem Konzept, welche in Abschnitt 1.2 näher erläutert wurden.

- Verarbeitung polyphoner Schallereignisse
- Webbasierte Implementierung für den Browser
- Echtzeit-Verarbeitung

Um das Konzept der Warte-Funktion umzusetzen, wurde ein Ansatz gewählt, der auf eine Onset-Erkennung und auf Pitch-Chroma-Extraktion basiert. Die Chroma-Extraktion ermöglicht es, die aktuelle Energie jeder der 12 Halbtöne in einem 12-elementigen Vektor zusammenzufassen. So kann ein Rückschluss auf die erklungenen Töne im User-Event gezogen werden. Die Onset-Erkennung dient dabei der Erkennung von Tastenanschlägen. Mittels einer Vergleichsfunktion wird der nach einem erkannten Tastenanschlag extrahierte Chroma-Vektor (entspricht dem User-Event) mit dem erwarteten Event verglichen. So wird er als positiv (*korrekt gespielt*) oder negativ (*inkorrekt gespielt*) klassifiziert.

Vor der Implementierung wurde eine passende webbasierte Plattform gewählt (Kapitel 2). Voraussetzung war dabei die Möglichkeit, über den Browser auf das Signal eines integrierten Mikrofons zu zugreifen. Neben der bereits etablierten Flash-Plattform von Adobe, die diese Voraussetzung erfüllt, kommen dabei auch Microsofts Silverlight-Plattform sowie Technologien rund um den neuen HTML5-Standard (Web Audio und *getUserMedia()*-API) in Frage. Während Flash und Silverlight separate Browser-Plugins benötigen, bedarf es bei HTML5 lediglich einen kompatiblen Browser, der *getUserMedia()* implementiert. Dies ist schon jetzt bzw. wird in Zukunft für alle etablierten Browser der Fall sein. Darüber hinaus bietet HTML5 mit der Web Audio API eine Schnittstelle, mit der auf grundlegende Komponenten der Signalverarbeitung zurück gegriffen werden kann. Diese Komponenten lassen sich zu einer Signalverarbeitungskette verbinden. Eigene Komponenten können dabei in Form von *Script Processor Nodes* in JavaScript implementiert werden. Aufgrund dieser Vorteile fiel die Entscheidung zugunsten der HTML5-Plattform.

Die Entwicklung begann mit der Implementierung der Chroma-Extraktion in JavaScript (Abschnitt 4.4). Das dazu benötigte Spektrum wird mittels der FFT der *DSP.js*-Bibliothek in einem *Script Processor Node* erzeugt, an welchem das Mikrofonsignal am Eingang anliegt. Mit entsprechenden Vorverarbeitungsschritten des Mikrofonsignals wie Downsampling, Überlappung und Fensterung konnte das Ergebnis der Chroma-Extraktion schrittweise verbessert werden. Anschließend wurden verschiedene Erkennungsfunktionen für die Onset-Erkennung und ein einfacher *Peak-Picking*-Algorithmus implementiert. Mittels Onset-Erkennung und darauffolgender Chroma-Extraktion lassen sich User-Events im Signal erkennen. Neben den User-Events existieren im zugrunde liegenden Event-Modell

ebenso erwartete Events, die die Partitur des zu spielenden Stücks repräsentieren (Abschnitt 4.5). Zwei verschiedene Vergleichsfunktionen wurden implementiert, die in der Lage sind, ein User-Event und ein erwartetes Event miteinander zu vergleichen. Sie klassifizieren dabei das User-Event als positiv oder negativ zum erwarteten Event. Um der Vergleichsfunktion einen über die Zeit gemittelten Chroma-Vektor zu übergeben, wurde ein Chroma-Buffer in JavaScript implementiert, der mehrere Vektoren über die Zeit puffert und einen gemittelten Vektor daraus berechnet. Die Zeit, über die gemittelt wird, wird hier als Haltezeit bezeichnet. Alternativ dazu kann der Vergleichsfunktion auch ein nichtgemittelter Vektor übergeben werden, dies ist in diesem Fall der letzte Chroma-Vektor der Haltezeit. Somit ergeben sich insgesamt vier verschiedene Varianten des Algorithmus, die hinsichtlich Erkennungsrate und Performanz evaluiert wurden.

6.2 Fazit

Die Evaluierung zeigt, dass die hier vorgestellten Varianten des Algorithmus sehr unterschiedliche Ergebnisse hinsichtlich der Erkennungsraten produzieren. Der beste der vier Algorithmen weist dabei Ergebnisse auf, die den Anforderungen für die erstmalige Einbettung in die *flowkey*-Applikation genügen. Die Ergebnisse der Evaluierung und die in Abschnitt 5.6 aufgezeigten Verbesserungsmöglichkeiten zeigen, dass der hier vorgestellte *Score-Following*-Algorithmus – obwohl grundlegend funktionsfähig – noch weiteres Potential zur Weiterentwicklung besitzt. Neben der noch zu verbessernden Erkennungsrate gilt es, die in Abschnitt 5.5 beschriebenen Probleme zu lösen.

In einem allgemeineren Kontext zeigt diese Arbeit, dass nicht nur professionelle Musiker von der Disziplin des *Score Followings* profitieren können. Eingebettet in einer E-Learning-Applikation kann das *Score Following* in einer Weise realisiert werden, die Anfängern eine Hilfestellung beim Erlenen von Klavier-Stücken gibt. Die Arbeit untermauert weiterhin, dass das *Music Information Retrieval* keine Disziplin ist, die auf native Anwendungen beschränkt ist.

Hinsichtlich der Performanz konnten alle Varianten des Algorithmus das Echtzeit-Kriterium auf dem Test-System erfüllen. Die Leistung von aktuellen Computern und Browsern reicht aus, um auch plattformunabhängige (nicht-optimierte) Anwendungen im Bereich der Echtzeit-Signalverarbeitung und der Feature-Extraktion auszuführen. Mit dem neuen HTML5-Standard entstehen zur Zeit Technologien, die die Entwicklung multimedialer Applikationen für das Web weiter vorantreiben und neue Anwendungsgebiete erschließen. So ist es mit der Web Audio API nicht nur möglich, Signale zu analysieren – wie

es in dieser Arbeit der Fall ist – sondern auch zu synthetisieren. Viele Anwendungen, die bisher nur mit nativen Programmen möglich waren (z. B. Sequencer, virtuelle Instrumente, etc.), können jetzt auch für das Web programmiert werden. Auch der hier vorgestellte *Score-Following*-Algorithmus hat von der Schnittstelle profitiert, welche auch in Zukunft Grundlage für die Weiterentwicklung des Algorithmus sein wird.

Literaturverzeichnis

[1] Dannenberg, Roger B (1984): *An on-line algorithm for real-time accompaniment*. Ann Arbor, MI: MPublishing, University of Michigan Library.

[2] Lerch, Alexander (2012): *An Introduction to Audio Content Analysis: Applications in Signal Processing and Music Informatics*. John Wiley & Sons.

[3] Bello, Juan Pablo et al. (2005): „A tutorial on onset detection in music signals." In: *Speech and Audio Processing, IEEE Transactions on*, 13(5) S. 1035–1047.

[4] Orio, Nicola; Serge Lemouton und Diemo Schwarz (2003): „Score following: state of the art and new developments." In: *Proceedings of the 2003 conference on New interfaces for musical expression*, NIME '03. Singapore, Singapore: National University of Singapore, S. 36–41. URL `http://dl.acm.org/citation.cfm?id=1085714.1085724`.

[5] Dannenberg, Roger B und Ning Hu (2003): „Polyphonic audio matching for score following and intelligent audio editors." In: *Computer Science Department* S. 507.

[6] Stark, Adam M und Mark D Plumbley (2009): *Real-time chord recognition for live performance*. Ann Arbor, MI: MPublishing, University of Michigan Library.

[7] Müller, Meinard und Sebastian Ewert (2011): „Chroma Toolbox: Matlab Implementations for Extracting Variants of Chroma-Based Audio Features." In: *ISMIR*. S. 215–220.

[8] Brossier, Paul; Juan Pablo Bello und Mark D Plumbley (2004): „Real-time temporal segmentation of note objects in music signals." In: *Proceedings of the ICMC*.

[9] Schwarz, Diemo; Nicola Orio und Norbert Schnell (2004): „Robust polyphonic midi score following with hidden markov models." In: *International Computer Music Conference*.

[10] Cano, Pedro; Alex Loscos und Jordi Bonada (1999): „Score-performance matching using HMMs." In: *Proceedings of the international computer music conference, Beijing, China*.

[11] Ghallab, M. und European Coordinating Committee for Artificial Intelligence (2008): *ECAI 2008: 18th European Conference on Artificial Intelligence, July 21-25, 2008, Patras, Greece : Including Prestigious Applications of Intelligent Systems (PAIS 2008) : Proceedings*. Frontiers in artificial intelligence and applications. IOS Press. URL http://books.google.de/books?id=3Wsn3SwEBScC.

[12] Dixon, Simon (2005): „An On-Line Time Warping Algorithm for Tracking Musical Performances." In: *IJCAI*. S. 1727–1728.

[13] Faulkner, Steve et al. (2013): *HTML5*. Candidate recommendation, W3C. http://www.w3.org/TR/2013/CR-html5-20130806/.

[14] van Kesteren, Anne (2013): *Differences from HTML4*. W3C working draft, W3C. http://www.w3.org/TR/html5-diff/.

[15] Rogers, Chris (2013): *Web Audio API*. W3C working draft, W3C. http://www.w3.org/TR/webaudio/.

[16] Koch, Stefan (2011): *JavaScript : Einführung, Programmierung und Referenz*. 6., aktualisierte und erw. aufl. Heidelberg: Heidelberg : dpunkt-verl.

[17] Pfeil, Christian (2009): *Adobe AIR: RIAs für den Desktop entwickeln Know-how für HTML/Ajax-und Flash*. Pearson Deutschland GmbH.

[18] Kim, Youngmoo (2007): „Audio Processing Library for Flash (ALF)." Webseite. Verfügbar unter http://music.ece.drexel.edu/ALF; aufgerufenam10.12.2013.

[19] Migneco, Ray et al. (2009): „An audio processing library for game development in Flash." In: *Games Innovations Conference, 2009. ICE-GIC 2009. International IEEE Consumer Electronics Society's*. IEEE, S. 201–209.

[20] Walter, Thomas 1966 (2008): *Kompendium der Web-Programmierung: dynamische Web-Sites; mit 22 Tabellen*. Berlin [u.a.]: Berlin [u.a.] : Springer.

[21] Demgen, Annika (2011): „Adobe: Steht auch der Desktop-Flash-Player vor dem Aus?" Webseite. Verfügbar unter http://www.netzwelt.de/news/89452-adobe-steht-desktop-flash-player.html; aufgerufen am 10.12.2013.

[22] Schwichtenberg, Holger (2009): „Die Wahl für das Web - ASP.NET, Ajax oder Silverlight?" Webseite. Verfügbar unter http://heise.de/-787452; aufgerufen am 10.12.2013.

[23] Schwichtenberg, Holger (2007): „Silverlight 1.0 erschienen." Webseite. Verfügbar unter `http://heise.de/-355043`; aufgerufen am 10.12.2013.

[24] Neumann, Alexander (2013): „Aufruf für ein Open-Source-Silverlight." Webseite. Verfügbar unter `http://heise.de/-2053237`; aufgerufen am 10.12.2013.

[25] Hassan, Ahmed; Jonas Gößling und Alexander Heesing (2012): „Entwicklung einer neuen Videosteuerung für eLearning Anwendungen am Beispiel Musikinstrumente." In: .

[26] Kuo, Sen M; Bob H Lee und Wenshun Tian (2006): *Real-time digital signal processing: implementations and applications*. Wiley. com.

[27] Link, Wolfgang (2003): *Assembler-Programmierung : das Praxisbuch*. 10., überarb. aufl. Poing: Poing : Franzis.

[28] Duhamel, Pierre und Martin Vetterli (1990): „Fast Fourier transforms: a tutorial review and a state of the art." In: *Signal processing*, 19(4) S. 259–299.

[29] Möller, Sebastian (2012): *Sprachkommunikation Skript*. Telekom Innovation Laboratories / TU Berlin.

[30] Weinzierl, Stefan (2008): „Kapitel 1 Grundlagen." In: *Handbuch der Audiotechnik*. Springer, S. 1–40.

[31] Brigham, E Oran (1988): „The Fast Fourier Transform and its applications." In: .

[32] Cooley, James W und John W Tukey (1965): „An algorithm for the machine calculation of complex Fourier series." In: *Mathematics of computation*, 19(90) S. 297–301.

[33] Müller, Meinard; Frank Kurth und Michael Clausen (2005): „Audio Matching via Chroma-Based Statistical Features." In: *ISMIR*, vol. 2005. S. 6th.

[34] Ewert, Sebastian; Meinard Müller und Peter Grosche (2009): „High resolution audio synchronization using chroma onset features." In: *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*. IEEE, S. 1869–1872.

[35] Oudre, Laurent; Cédric Févotte und Yves Grenier (2011): „Probabilistic template-based chord recognition." In: *Audio, Speech, and Language Processing, IEEE Transactions on*, 19(8) S. 2249–2259.

[36] Lee, Kyogu (2006): „Automatic chord recognition from audio using enhanced pitch class profile." In: *Proc. of the Intern. Computer Music Conference (ICMC), New Orleans, USA*.

[37] Oudre, Laurent; Yves Grenier und Cédric Févotte (2011): „Chord recognition by fitting rescaled chroma vectors to chord templates." In: *Audio, Speech, and Language Processing, IEEE Transactions on*, 19(7) S. 2222–2233.

[38] De Haas, W Bas; José Pedro Magalhães; Dion ten Heggeler; Gijs Bekenkamp und Tijmen Ruizendaal (2012): „Chordify: Chord transcription for the masses." In: .

[39] Goto, Masataka (2006): „A chorus section detection method for musical audio signals and its application to a music listening station." In: *Audio, Speech, and Language Processing, IEEE Transactions on*, 14(5) S. 1783–1794.

[40] Bartsch, Mark und Gregory Wakefield (2001): „"To catch a chorus: Using chroma-based representations for audio thumbnailing'"." In: *Applications of Signal Processing to Audio and Acoustics, 2001 IEEE Workshop on the*. IEEE, S. 15–18.

[41] Paulus, Jouni und Anssi Klapuri (2006): „Music structure analysis by finding repeated parts." In: *Proceedings of the 1st ACM workshop on Audio and music computing multimedia*, AMCMM '06. New York, NY, USA: ACM, S. 59–68. doi:10.1145/1178723.1178733. URL `http://doi.acm.org/10.1145/1178723.1178733`.

[42] Dixon, Simon (2006): „Onset detection revisited." In: *Proc. of the Int. Conf. on Digital Audio Effects (DAFx-06)*. S. 133–137.

[43] Glover, John; Victor Lazzarini und Joseph Timoney (2011): „Real-time detection of musical onsets with linear prediction and sinusoidal modeling." In: *EURASIP Journal on Advances in Signal Processing*, 2011(1) S. 1–13.

[44] Marolt, Matija; Alenka Kavcic und Marko Privosnik (2002): „Neural networks for note onset detection in piano music." In: *Proceedings of the International Computer Music Conference*.

[45] Crockford, D. und P. Klicman (2008): *Das Beste an JavaScript*. Yahoo! Press. O'Reilly. URL `http://books.google.de/books?id=mOQDYOM8TiYC`.

[46] Ferber, R. (2003): *Information Retrieval: Suchmodelle und Data-Mining-Verfahren für Textsammlungen und das Web*. Dpunkt.Verlag GmbH. URL `http://books.google.de/books?id=EmiOAAAACAAJ`.

[47] Mann, Jatinder (2012): *High Resolution Time*. W3c recommendation, W3C. `http://www.w3.org/TR/2012/REC-hr-time-20121217/`.

[48] Müller, Meinard (2007): *Information Retrieval for Music and Motion*. Springer.

Anhang A

Relevanter Code

A.1 Chroma-Berechnung

```
1  /**
2    @function computeStandardChroma - calculates a chroma vector from a
         spectrum B
3    @param {Float32Array} B - block in frequency domain (spectrum)
4    @param {integer} K - Blocklength
5    @param {integer} fs - sampling frequency
6    @param {integer} octaves - number of octaves to analyze
7    @param {bool} tolerance - apply chroma tolerance (default:false)
8    @returns {Array} chromaVector - 12-element vector representing
         current energy in each pitch class
9  */
10 function computeStandardChroma(B, K, fs, octaves, tolerance,
       semitonePitches){
11   var chromaVector = new Array(12);
12   for (var i=0; i<semitonePitches.length; i++){
13     if (tolerance==true){
14       var middleBin = Math.round(semitonePitches[i]*K/fs);
15       var upperBin = getBin(semitonePitches[i],
             CHROMA_UPPPER_BOUND_FACTOR, K, fs);
16       var lowerBin = getBin(semitonePitches[i],
             CHROMA_LOWER_BOUND_FACTOR, K, fs);
17       var halfToneBins;
18       var binSum;
19       if ((upperBin-lowerBin)<2){
20         halfToneBins = B[middleBin];
21         binSum = halfToneBins;
22       }
23       else{
24         halfToneBins = B.subarray(lowerBin, upperBin+1);
25         binSum = halfToneBins.sum();
26         binSum = binSum/halfToneBins.length;
27       }
28       chromaVector[i%12] += binSum;
29     }else{
30       var freqBin = Math.round(semitonePitches[i]*K/fs);
31       chromaVector[i%12] += B[freqBin];
32     }
33   }
34   return chromaVector;
```

```
35    }
36
37    function getBin(p, factor, K, fs){
38      var bin = Math.round(factor*p*K/fs);
39      return bin;
40    }
```

A.2 Spectral Flux

```
1
2  /**
3    @function spectralFluxODF - spectral difference a.k.a. spectral flux,
           onset detection function (novelty function), compares last and
         current frequency block and sums up differences in one value
4    @param {Float32Array} preBlock - last signal block frequency domain
5    @param {Float32Array} curBlock - current signal block frequency
         domain
6    @param {int} exponent - exponent=1: manhatten distance, exponent=2:
         euclidean distance
7    @param {bool} hwrFlag - half wave rectification (just positive
         changes in spectrum are considered)
8    @returns {float} sfSample - spectral flux sample
9  */
10 function spectralFluxODF(preBlock, curBlock, exponent, hwrFlag){
11
12   // Half Wave Rectification Function definition
13   function HWR(x){
14     var result = (x+Math.abs(x))/2;
15     return result;
16   }
17
18   var sfSample=0;
19   for (var i=0; i<curBlock.length; i++){
20     var curAbs = Math.abs(curBlock[i]);
21     var preAbs = Math.abs(preBlock[i]);
22     var tempSf = curAbs - preAbs;
23     if (hwrFlag==1) tempSf = HWR(tempSf);
24     tempSf = Math.pow(tempSf,exponent);
25     sfSample = sfSample + tempSf;
26   }
27   sfSample = Math.sqrt(sfSample);
28   sfSample = sfSample /(curBlock.length);
29   return sfSample;
30 }
```

Anhang B

Wahrheitswerttabelle

LB	LE	UB	UE	Ergebnis	Ausdruck
1	1	1	1	**1**	LB && LE && UB && UE
1	1	1	0	**0**	-
1	1	0	1	**1**	LB && LE && !UB && UE
1	1	0	0	**1**	LB && LE && !UB && !UE
1	0	1	1	**0**	-
1	0	1	0	**0**	-
1	0	0	1	**0**	-
1	0	0	0	**0**	-
0	1	1	1	**1**	!LB && LE && UB && UE
0	1	1	0	**0**	-
0	1	0	1	**1**	!LB && LE && !UB && UE
0	1	0	0	**1**	!LB && LE && !UB && !UE
0	0	1	1	**1**	!LB && !LE && UB && UE
0	0	1	0	**0**	-
0	0	0	1	**1**	!LB && !LE && !UB && UE
0	0	0	0	**1**	!LB && !LE && !UB && !UE

Abbildung B.1: Wahrheitswerttabelle zum Aufstellen des booleschen Ausdruck für die Chroma-Schwellwert-basierte Vergleichsfunktion

Anhang C

CD-ROM mit vollständigem Quellcode

Der Arbeit liegt eine CD mit dem vollständigem Quellcode des Algorithmus und der Testumgebung bei. Weiterhin sind auch alle in der Evaluierung verwendeten Audiosamples sowie JSON-Daten enthalten. Die webbasierte *flowkey*-Applikation kann nach Anmeldung unter `http://www.flowkey.com/de` benutzt werden.

Erklärung der Urheberschaft

Hiermit erkläre ich an Eides statt gegenüber der Fakultät I der Technischen Universität Berlin, dass die vorliegende Arbeit selbstständig und nur unter Zuhilfenahme der im Literaturverzeichnis genannten Quellen und Hilfsmittel angefertigt wurde. Alle Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind kenntlich gemacht. Ich reiche die Arbeit erstmals als Prüfungsleistung ein.

Ort, Datum Unterschrift