# Optische-Musik-Erkennung (OME)

## Erkennung von Notenschrift

# DIPLOMARBEIT

zur Erlangung des akademischen Grades
Diplom-Ingenieur
der Studienrichtung Informatik

eingereicht von

WOLFGANG EßMAYR

angefertigt am Institut für Informatik,
Abteilung für wissensbasierte Systeme
der Technischen-naturwissenschaftlichen Fakultät
an der Johannes Kepler Universität Linz

eingereicht bei a.Univ.-Prof.Dipl.Ing.Dr. Roland Wagner

Linz, im September 1994

## Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die Diplomarbeit selbständig verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe, und daß ich diese Diplomarbeit bisher weder im In- noch im Ausland in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Linz, am 7. September 1994 Wolfgang Eßmayr

## Danksagung

Danken möchte ich Herrn Dr. Josef Küng, betreuender Assistent meiner Diplomarbeit, der sich trotz großer zeitlicher Belastung immer wieder die Zeit genommen hat, mir mit Rat und Tat zur Seite zu stehen.

Mein Dank gilt auch besonders meinen Eltern Josef und Therese Eßmayr, die sich all die Jahre meiner Ausbildung immer aufopfernd für mich eingesetzt und mich finanziell und moralisch unterstützt haben.

Meiner Studienkollegin und Freundin Erika Dolnigg danke ich für die ehrliche und konstruktive Kritik, die mich immer zu Überlegungen und neuen Ideen inspiriert hat.

Für die aufwendige Arbeit des Korrekturlesens möchte ich mich noch einmal bei meinen Eltern bedanken.

**Danke!**

# Inhaltsverzeichnis


1. Einleitung ..... 1
2. Kategorisierung der Notenschriftsymbole ..... 5
    2.1. Die Notenlinien, Notenzeilen und Notensysteme ..... 5
    2.2. Die "Was"-Symbole ..... 6
    2.3. Die "Wie"-Symbole ..... 11
3. Erkennungsmethoden für Notenschriftsymbole ..... 13
    3.1. Die Erkennung der Notenlinien ..... 13
        3.1.1. Linienverfolgung ..... 13
        3.1.2. Linienwinkel und Linienstärke ..... 14
        3.1.3. Y-Projektion ..... 14
    3.2. Die Erkennung der Musiksymbole ..... 15
        3.2.1. Höhe und Breite ..... 15
        3.2.2. Musterprimitiven und zusammengesetzte Symbole ..... 15
        3.2.3. Schablonenvergleich ..... 15
        3.2.4. Projektionen ..... 16
        3.2.5. Regelbasierte Systeme ..... 16
4. Programmkonzept für die Notenschrifterkennung ..... 17
    4.1. Die Struktur eines Notenblattes ..... 19
    4.2. Die Musiksymbole ..... 22
    4.3. Die Benutzerschnittstelle ..... 24
        4.3.1. Das Hauptfenster ..... 24
        4.3.2. Das Datei-Menü ..... 24
        4.3.3. Das Erkennen-Menü ..... 25
    4.4. Phasen der Implementierung ..... 25
5. Die Implementierung von NOSE ..... 26
    5.1. Die MIDI-Spezifikation ..... 26
    5.2. Ein- und Ausgabedateien ..... 28
        5.2.1. Das Windows-Bitmap-Dateiformat ..... 28
        5.2.2. Das Standard-MIDI-Dateiformat ..... 29
    5.3. Der Erkennungsprozeß ..... 32
        5.3.1. Das Segmentieren ..... 33
        5.3.2. Das Klassifizieren ..... 46
        5.3.3. Das Interpretieren ..... 55
    5.4. Beschreibung der Klassen von NOSE ..... 58
        5.4.1. NoseModel ..... 58
        5.4.2. NoseView ..... 61
        5.4.3. ImageView ..... 63

5.4.4. Image....64
5.4.5. VIntVector....67
5.4.6. Projection....70
5.4.7. Segment....73
5.4.8. Score....79
5.4.9. Staff....82
5.4.10. Symbol....85
5.4.11. Note....87
5.4.12. Rest....91
5.4.13. Clef....93
5.4.14. MidiFile....95
5.4.15. MidiTrack....98
6. Testbericht und Ergebnisse....100
6.1. Test: DEMO.BMP....100
6.2. Test: TEST1.BMP....102
6.3. Test: TEST2.BMP....104
6.4. Test: TEST3.BMP....106
6.5. Test: TEST4.BMP....107
6.6. Test: TEST5.BMP....109
6.7. Test: TEST6.BMP....111
6.8. Test: TEST7.BMP....113
6.9. Test: TEST8.BMP....115
6.10. Test: TEST9.BMP....117
7. Zusammenfassung....118
Anhang A: Literaturverzeichnis....120

# 1. Einleitung

"Musik vergeht, sofern sie nicht vom Gedächtnis festgehalten wird, denn aufschreiben kann man sie nicht." (Bischof Isidor von Sevilla, 560-636).
Und trotzdem. Die Menschen haben schon immer versucht, ihre Musik in irgendeiner Form festzuhalten. Angefangen von der mündlichen Weitergabe von Generation zu Generation, über die ersten Formen der Buchstabennotation in der griechischen Musik um 200 vor Christus, bis zu graphischen Zeichnungen heutzutage, die den Spieler zu einer Improvisation im modernen Stil veranlassen sollen. Das Problem der Notation besteht dabei in der Komplexität der möglichen Ausdrucksformen, Spielweisen und Klangfarben, welche die Musik erst so lebendig machen. Die heutige Notenschrift hat sich in einer langen Entwicklung herausgebildet, wobei die wichtigsten Toneigenschaften (Höhe, Dauer, Lautstärke und Klangfarbe) immer genauer fixiert wurden [WZM].
Mit der Möglichkeit der analogen Aufzeichnung von Musik auf Schallplatte und Musikkassette bzw. der digitalen Speicherung auf CD, konnte man gespielte Musik nun relativ einfach festhalten und zum Hörgenuß reproduzieren. Diese rein passiven Aufzeichnungsmethoden lassen jedoch nur sehr schwierig nachträgliche Veränderungen und Analysen zu. Nicht nur deshalb, sondern auch wegen ihrer mathematischen Struktur, ist Musik ein beliebtes Feld für Computeranwendungen geworden. Es existieren bereits eine Unmenge von Programmen, mit Hilfe derer man Musik schaffen, aufzeichnen, analysieren, und in beliebiger Form weiterverarbeiten kann. Parallel dazu hat sich vor allem im elektronischen Musikbereich ein Standard entwickelt, genannt *MIDI* (Musical Instrument Digital Interface, siehe Kapitel 5.1). Dieser bereits weit verbreitete Standard bildet heutzutage trotz einiger Unzulänglichkeiten die Basis für die Kommunikation zwischen elektronischen Musik- und Tonstudiogeräten und dem Computer. Trotzdem die Aktivitäten im Bereich Musik und Computer schon sehr intensiv sind, fehlen immer noch Werkzeuge, die es ermöglichen, große Mengen von "Musikmaterial" zu erfassen und dem Computer zugänglich zu machen. Eine der möglichen Lösungen hierfür bietet die optische Musikerkennung, das Überführen von in Notenschrift gefaßter Musik in computerlesbare Form.

Betrachten wir zunächst die Ziele und die möglichen Anwendungsgebiete der optischen Musikerkennung (entnommen aus [SDIA]). Eine wichtige Klasse von Anwendungen betrifft das Editieren von Partituren:

- automatische Transkription existierender Werke in andere Instrumentierungen, z.B. Orchesterpartitur in Klavierpartitur;
- verschiedene Werke in alten Editionen lesen, um einen neuen Druck zu gestalten;
- kritische Analyse mehrerer gedruckter Versionen des gleichen Werkes;
- Transposition eines Musikstückes in eine andere Tonart;
- einzelne Instrumente aus einer gegebenen Partitur herauslösen oder zu einer Partitur zusammenfassen;
- die Notation eines neuen Musikstückes auf syntaktische oder andere Fehler prüfen;
- neue handgeschriebene Musik automatisch drucken;
- Audioversionen von geschriebener Musik erzeugen; der Computer ist dabei eine Kombination von Musiker und Instrument.

Eine andere Klasse von Anwendungen betrifft das Erstellen von Datenbanken:

- Indizes über Melodien, Themen und andere musikalische Eigenschaften erzeugen;
- musikalische Struktur und Stil analysieren;
- Musiktheorien testen;
- die Evaluierung von Algorithmen für das automatische Analysieren oder Komponieren von Musik;

Meine persönliche Motivation ergibt sich aus der Tatsache, daß ich nicht nur Informatiker sondern auch Musiker bin und diese Diplomarbeit natürlich eine willkommene Verbindung dieser beiden Gebiete darstellt.

Ein weiterer Aspekt dieser Diplomarbeit besteht darin, daß damit gewisses Neuland im Bereich der Mustererkennung betreten wird, weil bisher zwar eine Fülle von Arbeiten über die Erkennung von Schriftzeichen, jedoch nicht über die Erkennung von Notenschrift existieren. Es werden somit Analysen, Tests und Ergebnisse in einem weniger erforschten Gebiet der Mustererkennung geliefert.

Die Vision der optischen Musikerkennung ist, jegliche Form von musikalischer Notation zu erkennen, um sie für jede beliebige Anwendung zur Verfügung zu stellen. Dieses Ziel ist jedoch aufgrund der mannigfaltigen Erscheinungs- und Notationsformen und der vielseitigen Anwendungsgebiete nicht realistischer Weise erreichbar. Es müssen daher starke Spezialisierungen vorgenommen werden.

In dieser Arbeit werde ich mich völlig auf die klassische Form der Notenschrift beschränken. D.h. ein System aus Notenzeilen, bestehend aus je 5 Notenlinien und

einer Reihe von Musiksymbolen, die Auskunft über die schon angesprochenen Toneigenschaften geben.
Weiters gehe ich davon aus, daß die zu behandelnden Notenvorlagen in einem sorgfältig gescannten Zustand zur Verfügung stehen, da viele Erkennungsfehler sehr oft nur auf geringfügig rotierte Vorlagen oder auf "Schmutz" und "Löcher" im digitalen Bild zurückzuführen sind.
Handgeschriebene Notenblätter verlangen ganz spezielle Fähigkeiten vom Erkennungssystem. Jeder Autor besitzt eine spezifische Handschrift und stellt die Symbole oft höchst unterschiedlich dar bzw. erfindet sogar eigene, nicht genormte Symbole, um seinen Vorstellungen einen besseren Ausdruck zu verleihen. Das Ziel meiner Arbeit besteht jedoch darin, ein Werkzeug zu schaffen, mit dessen Hilfe man möglichst viel Information und möglichst große Mengen geschriebener Musik erfassen kann. Daher werde ich mich auf maschinengedruckte Vorlagen konzentrieren mit dem Vorteil, daß die Maschinendrucke einer gewissen Normung unterliegen. Die Notenlinien sind parallel und von gleichbleibender Stärke und die einzelnen Musiksymbole haben in verschiedenen Notenvorlagen meist die gleiche Form, sie unterscheiden sich höchstens in ihrer Größe.
In der Abbildung 1-1 ist der Ablauf eines kompletten Erkennungsvorganges graphisch dargestellt.

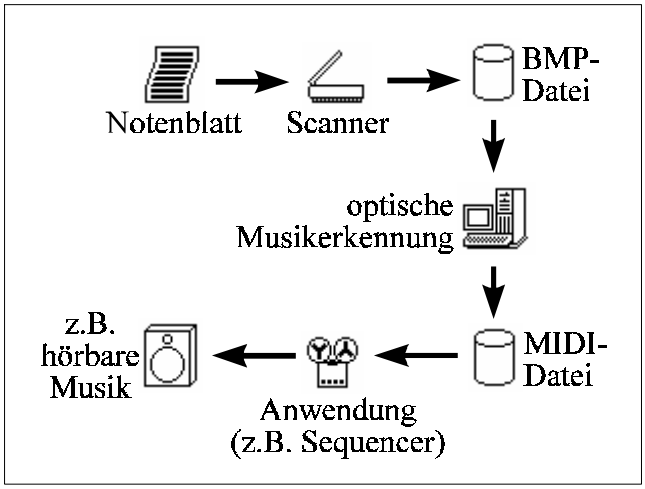


Abbildung 1-1: Erkennungsablauf

Ausgangspunkt ist ein aus den schon erwähnten Gründen maschinengedrucktes Notenblatt. Dieses wird mit Hilfe eines Flachbettscanners (kein Handscanner, weil dadurch die Qualität zu stark leidet) in ein digitales, computerlesbares Bild umgewandelt. Die meisten Scanner sind mit zugehöriger Software in der Lage, Windows-Bitmap-Dateien (BMP) zu erzeugen, die im PC-Bereich unter der graphischen Benutzeroberfläche Windows auch einen Standard darstellen. Diese BMP-

Dateien, eine genauere Beschreibung folgt im Kapitel 5.2.1, sind nun der Input für das OME-Werkzeug, dessen Entwicklung Gegenstand dieser Arbeit ist. Das Produkt des Erkennungsvorganges ist eine MIDI-Datei (siehe Kapitel 5.2.2). Diese Dateien haben den Vorteil, daß sie von einer großen Menge bereits bestehender und mit hoher Wahrscheinlichkeit auch zukünftiger Musikprogramme gelesen und verarbeitet werden können. Es ist ein im Musikcomputerbereich weit verbreiteter Standard und es gehört sozusagen zum guten Ton, daß Musikprogramme neben ihrem evt. eigenen Datenformat auch das MIDI-Dateiformat verarbeiten können. Der abschließende Schritt des Erkennungsvorganges besteht im Anwenden einer geeigneten Software (z.B. ein Sequencerprogramm), um das gewünschte Ziel (z.B. hörbare Musik) zu erreichen.

# 2. Kategorisierung der Notenschriftsymbole

Ich möchte nun eine Beschreibung der Notenschrift, nicht aus der Sicht eines Musikers, sondern unter dem Gesichtspunkt des Erkennungsvorganges vornehmen. Dabei kann man die Symbole der Notenschrift grob in 3 Kategorien einteilen: Die *Notenzeilen* (ein Koordinatensystem für die übrigen Symbole), die *"Was"-Symbole* (was gespielt wird) und die *"Wie"-Symbole* (wie es gespielt wird). Diese Kategorien unterscheiden sich grundlegend von einander, sowohl im Aussehen und in der Bedeutung der Symbole wie auch in den anzuwendenden Erkennungsmethoden.

## 2.1. Die Notenlinien, Notenzeilen und Notensysteme

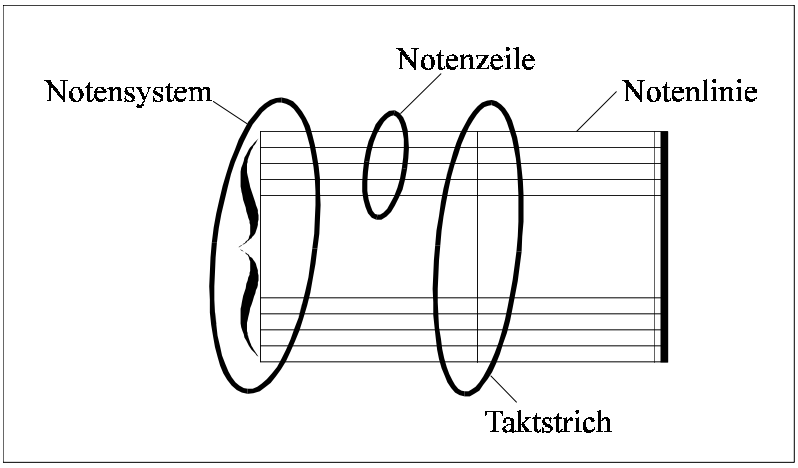


Abbildung 2-1: Notenlinie, Notenzeile und Notensystem

Notenlinien sind dünne horizontale Geraden, die in Gruppen zu 5 parallelen Linien eine Notenzeile bilden. In der x-Achse wird die zeitliche Komponente, in der y-Achse die Tonhöhe repräsentiert. Es können mehrere Notenzeilen zu einem Notensystem zusammengefaßt werden, was durch eine geschwungene Klammer am linken Rand des Notensystems und/oder mit über das ganze Notensystem durchgezogenen vertikalen Strichen (Taktstriche) symbolisiert wird und bedeutet, daß diese Notenzeilen gleichzeitig gespielt werden. Dabei entspricht jede Notenzeile einer Spur (Track) in der Terminologie der MIDI-Dateien, einem Instrument in einer Orchesterpartitur oder einer Hand in einer zweizeiligen Klavierpartitur. Da bei MIDI-Dateien das Vorhandensein mehrerer gleichzeitig gespielter Spuren explizit unterstützt wird (Format-1-Dateien, siehe Kapitel 5.2.2), darf das Erkennungssystem hier keine Einschränkungen vornehmen. Die Anzahl der Spuren, bzw. die Zahl der Notenzeilen in einem Notensystem ist beliebig und muß vom OME-Werkzeug erkannt werden. Auf einem Notenblatt befinden sich üblicherweise mehrere untereinanderliegende Notensysteme, welche zeitlich hintereinander gespielt werden. Dabei ist das erste

Notensystem oftmals etwas eingerückt dargestellt (analog zum Einrücken eines Textes bei einem Absatz), die restlichen Systeme kann man sich als horizontale Fortsetzung des ersten vorstellen. Die Notenzeilen bedeuten für die Notenschrift als zweidimensionale graphische Darstellung von Musik ein Koordinatensystem für die restlichen Musiksymbole. Man kann daraus jene Bereiche des Notenblattes ermitteln, auf denen sich überhaupt Musiksymbole befinden können. Weiters ist die Dicke und der Abstand der einzelnen Notenlinien ein Normierungsmaß für die Größe der Notenköpfe, Notenschlüsseln und der meisten übrigen Symbole. Da ich mich auf gedruckte Vorlagen konzentrieren will, kann ich davon ausgehen, daß alle Notenlinien gleichbleibend dick, geradlinig und parallel zueinander sind. Ebenso bleiben die Abstände zwischen den einzelnen Notenlinien, den Notenzeilen und auch den Notensystemen in der gesamten Druckvorlage üblicherweise konstant.

## 2.2. Die "Was"-Symbole

Die Gruppe der "Was"-Symbole drücken 3 wichtige Toneigenschaften aus: die Tonhöhe, den Zeitpunkt des Erklingens und die Dauer eines Tones. Sie legen damit fest, *was* gespielt werden soll. Diese Eigenschaften sind ohne Informationsverlust direkt in einer MIDI-Datei abzubilden und sollten daher vom Erkennungswerkzeug möglichst vollständig behandelt werden.
Zu Beginn jeder Notenzeile ist ein bestimmter Bereich für den Notenschlüssel reserviert. Es gibt eine Reihe verschiedener Notenschlüssel, die jeweils eine bestimmte Notenlinie als Referenzton markieren.

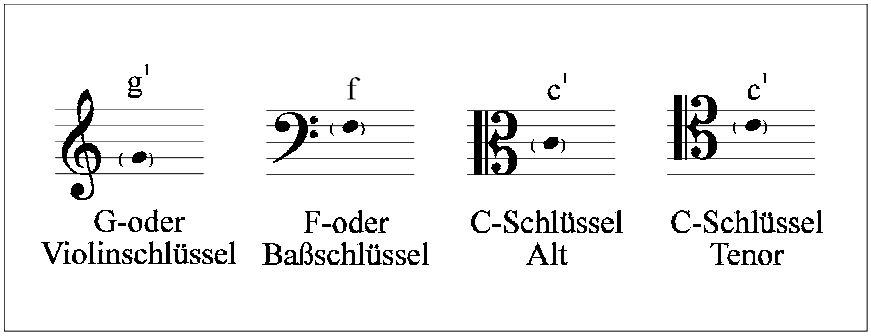


Abbildung 2-2: Beispiele für Notenschlüssel

So hat sich z.B. der Violinschlüssel (G-Schlüssel) aus dem Buchstaben G entwickelt und umschließt mit seinem geschwungenen Körper die zweite Notenlinie von unten, die den Referenzton g' (*eingestrichenes* g) darstellt. Gleiches gilt für den Baßschlüssel (F-Schlüssel), dessen Punkte das f der *Kleinen Oktave* markieren. Ich beschränke mich in dieser Arbeit aus Aufwandsgründen auf diese beiden wichtigsten und häufigsten Formen des Notenschlüssels, da eine Erweiterung keine prinzipielle Änderung des

Erkennungssystems erfordert und somit relativ leicht zu einem späteren Zeitpunkt nachgeholt werden kann. Ein Notenschlüssel gilt für eine Notenzeile, es kann also jede Zeile eines Notensystems einen eigenen Notenschlüssel aufweisen. Zusätzlich können auch Änderungen des Notenschlüssels inmitten einer Notenzeile auftreten, wobei dann der neue Schlüssel meist kleiner dargestellt wird.
Die wichtigsten und häufigsten Symbole in der Notenschrift sind die Notenköpfe, die durch gefüllte bzw. leere Ovale dargestellt werden und einen Ton symbolisieren.

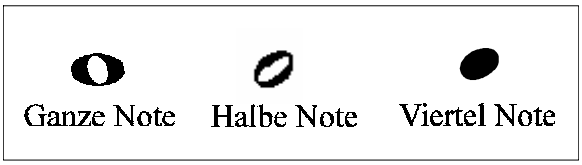


Abbildung 2-3: Notenköpfe

Vertikal untereinanderliegende, oder aus Platzgründen marginal nebeneinanderliegende Notenköpfe stellen einen Akkord dar, die Töne werden dabei gleichzeitig gespielt. Besitzen die Notenköpfe einen genügend großen horizontalen Abstand zueinander, so symbolisieren sie eine Melodie (bzw. eine Stimme), bei der die Töne nacheinander gespielt werden. Die Notenköpfe können sich entweder auf einer der fünf Notenlinien bzw. in einem der vier Zwischenräume der Notenzeile oder auf gedachten Hilfslinien bzw. in den sich daraus ergebenden Zwischenräumen über oder unterhalb der Notenzeile befinden.

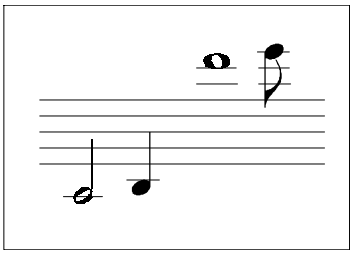

Abbildung 2-4: Hilfslinien

Diese Hilfslinien sind notwendig, um den Tonumfang einer Notenzeile zu erweitern, sie erschweren aber auch den Erkennungsprozeß, weil dadurch Überlappungen mit benachbarten Notenzeilen entstehen können. Mit Ausnahme der ganzen Note besitzt jeder Notenkopf einen Notenhals, ein dünner vertikaler Strich, der entweder rechts vom Notenkopf aufwärts, oder links davon abwärts zeigt. Alle an einem Akkord beteiligten Notenköpfe werden mit nur einem gemeinsamen Notenhals versehen. Bei den gefüllten Notenköpfen (Viertelnote) können sich sogenannte Fähnchen rechts an das Ende des Notenhalses anschließen, die abhängig von der Anzahl den Notenwert entsprechend auf eine Achtel-, Sechzehntel-, Zweiunddreißigstelnote usw. halbieren. Komplette Noten oder Akkorde kann man daher als Symbole betrachten, die aus

*Musterprimitiven* wie Notenkopf, Notenhals oder Fähnchen zusammengesetzt sind. Die Tonhöhe ergibt sich aus der vertikalen Lage in der Notenzeile, die Dauer des Tones wird durch die Form des Notenkopfes (gefüllt/leer), durch das Vorhandensein eines Notenhalses und durch die Anzahl der Fähnchen am Notenhals bestimmt.
Eine weitere Gruppe von Symbolen sind die Vorzeichen. Sie dienen zur kurzfristigen Veränderung der Tonhöhe einer Note für die Dauer eines Taktes. Das Kreuz bzw. das Doppelkreuz erhöhen den Ton um einen Halb- bzw. Ganztonschritt, B und Doppel-B erniedrigen die Tonhöhe analog dazu. Das Auflösungszeichen macht die Tonhöhenänderung mit sofortiger Wirkung rückgängig. Die Vorzeichen werden unmittelbar links vom Notenkopf der Note plaziert, die sie verändern.

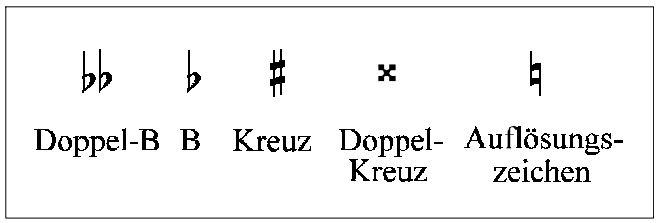


Abbildung 2-5: Vorzeichensymbole

Befindet sich rechts vom Notenkopf einer Note ein Punkt, so bedeutet dies, daß der Notenwert der Note um die Hälfte des ursprünglichen Wertes verlängert wird. Hierbei muß beim Erkennen darauf geachtet werden, daß dieser Punkt bei Noten, die sich auf einer der Linien befinden, im nächst höheren Zwischenraum dargestellt wird, um die Leserlichkeit zu erhöhen.
Das Gegenteil der Noten sind die Pausen, die ebenfalls eine Wertigkeit (Dauer) analog zu den Notenwerten besitzen. Allerdings ist hier die vertikale Position irrelevant, die Symbole werden meist vertikal zentriert in der Notenzeile dargestellt.

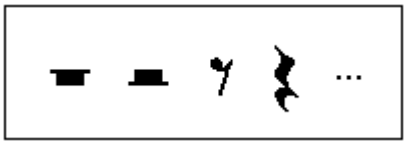

Abbildung 2-6: Pausensymbole

Am Anfang jeder Notenzeile befindet sich direkt neben dem Notenschlüssel eine gewisse Anzahl von Vorzeichen, welche die Tonart des Stückes bestimmen. Es sind nur gleichartige Vorzeichen möglich, d.h. entweder nur Kreuz- oder nur B-Symbole.

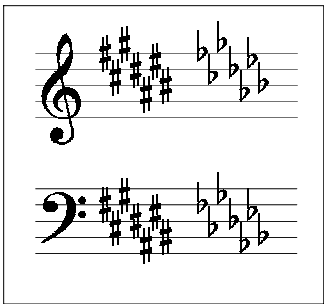

Abbildung 2-7: Tonartsymbole

Die Tonart verändert sich innerhalb eines Stückes nicht und ist auch für alle Notenzeilen eines Notensystems gleich, außer bei Partituren, die Instrumente mit verschiedenen Grundstimmungen beinhalten. Diese Vorzeichen am Anfang einer Notenzeile verändern die Tonhöhe analog zu den Vorzeichen direkt vor einer Note, jedoch für die Dauer des gesamten Stückes.
Direkt nach den Tonartsymbolen findet man im ersten Notensystem das Taktsymbol. Es besteht in den meisten Fällen aus zwei innerhalb der Notenzeile übereinander angeordneten Zahlen, wobei die untere Zahl die Auflösung als Notenwert (üblicherweise 4, 8 oder 16) und die obere die Anzahl der Schläge in einem Takt beschreibt (größer 1).

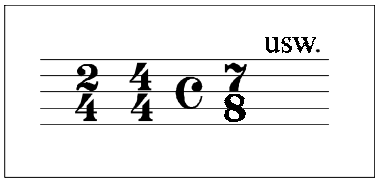


Abbildung 2-8: Taktsymbole

Alternativ zu einem 4/4-Takt kann auch ein C-Symbol geschrieben werden. Das Problem der Taktsymbole ist, daß man eine komplette Schrifterkennung implementieren müßte, um alle möglichen Taktarten zu erkennen. Dies würde aber eindeutig den Rahmen der Diplomarbeit sprengen, daher werde ich mich abhängig vom Fortschritt der Implementierung auf einige wenige Taktsymbole beschränken. Alle übrigen können entweder "händisch" eingestellt oder nachträglich in einem Anwendungsprogramm jederzeit verändert werden.
Die bisher genannten "Was"-Symbole besitzen alle eine gemeinsame Eigenschaft: sie sind größeninvariant. Dies bedeutet, daß eine evt. Änderung der Größe keine semantische Veränderung des Symbols bedeutet. Bindebögen und Balken jedoch sind parametrische (dehnbare) Symbole, die in Abhängigkeit von ihrer Größe auch eine geänderte Bedeutung besitzen.

Abbildung 2-9: Balken und Bindebögen

Die sich daraus ergebenden Informationen betreffen hauptsächlich die Mehrstimmigkeit eines Stückes, lassen sich jedoch weder in einer MIDI-Datei abbilden noch verändern sie die Informationen darüber, was gespielt wird. Balken werden häufig anstelle von Fähnchen verwendet, um zusammengehörige Noten zu gruppieren oder die Stimmen eines mehrstimmigen Stückes von einander abzuheben. Beim Erkennen muß jedoch nur die leicht unterschiedliche Form der beiden Symbole und die Tatsache beachtet werden, daß sich die Balken nur am Anfang einer Notengruppe rechts vom Notenhals befinden, sonst immer links. Bindebögen sind für das Erkennungswerkzeug nur dann von Bedeutung, wenn sie zwei bezüglich ihrer Tonhöhe gleiche Noten über eine Taktgrenze hinaus verbinden, was zum Ausdruck bringt, daß die zweite Note nicht gespielt wird und sie dafür die erste um den Notenwert der zweiten verlängert.
Es gibt noch eine Menge weiterer Symbole, Spezialformen und Ausnahmen, die ich aber aus Aufwandsgründen in dieser Arbeit nicht unterstützen kann.

## 2.3. Die "Wie"-Symbole

Für die dritte Kategorie von Symbolen gilt ähnliches wie schon bei den zuvor erwähnten Taktsymbolen. Sie treten in großen Variationen auf, bestehen sehr häufig aus Text und werden meist unter dem Gesichtspunkt der Lesbarkeit nahezu völlig frei innerhalb des Notenblattes angeordnet.

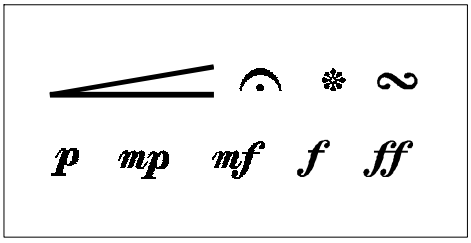


Abbildung 2-10: einige "Wie"-Symbole

Sie bestimmen das Tempo des Stückes und beschreiben die Dynamik, also Lautstärke, Anschlag und Betonung der einzelnen Noten. Mit einigen Ausnahmen sind diese Symbole nicht ohne Informationsverlust in einer MIDI-Datei abzubilden. Bei einem *Crescendo* (stärker werden) beispielsweise kann man die einzelnen Noten in der MIDI-Datei zwar ohne Probleme mit ansteigender Anschlagdynamik (Velocity) versehen, jedoch die Information, daß es sich hierbei um ein Crescendo handelt, geht prinzipiell verloren und müßte durch das Anwendungsprogramm erst wieder ermittelt werden. Weiters ist zu bemerken, daß die meisten Symbole aus dieser Gruppe sehr intuitive Angaben treffen (z.B. Allegretto = mäßig schnell), die zuerst in geeignete repräsentative Werte umgesetzt werden müssen. Ich möchte hier die wichtigsten Symbole einmal anführen.

Die *Tempoangabe* wird meist am ersten Notenblatt des Stückes links oberhalb des ersten Notensystems angebracht. Sie erfolgt in Form eines Textes, der auszugsweise folgende Werte annehmen kann:

- *Largo* = breit, sehr langsam
- *Adagio* = langsam
- *Andante* = gehend
- *Moderato* = mäßig
- *Allegretto* = mäßig schnell
- *Allegro* = schnell
- *Presto* = sehr schnell

Sehr oft werden auch Attribute wie z.B. *vivace* (lebendig) etc. hinzugefügt oder die Tempoangabe besteht aus einer Kombination der einzelnen Werte (z.B. *Allegro Moderato* statt *Allegretto*). In manchen Fällen wird zusätzlich oder ausschließlich eine Metronomangabe vorgenommen, die Tempoangabe kann dabei aber auch völlig

entfallen. Tempoänderungen werden durch die zwei Textsymbole *ritenuto* (langsamer werden) bzw. *accelerando* (schneller werden) symbolisiert. Angaben über Lautstärke bzw. Anschlagstärke werden durch folgende Symbole dargestellt:

- *pp* = *pianissimo* = sehr leise
- *p* = *piano* = leise
- *mf* = *mezzoforte* = halb stark
- *f* = *forte* = stark
- *ff* = *fortissimo* = sehr stark

Eine Änderung der Anschlagstärke erfolgt entweder texttuell mit den Symbolen *crescendo* (stärker werden) und *diminuendo* bzw. *decrescendo* (schwächer werden), oder graphisch durch eine links bzw. rechts geöffnete spitze Klammer, deren Länge die Dauer der Änderung symbolisiert.

Bei Klavierpartituren werden häufig Angaben über die Betätigung des Dämpfungspedals (Sustainpedal) gemacht, welches in gedrückter Position die Saiten eines Klaviers ausklingen läßt, sonst abdämpft.

Abschließend möchte ich noch zwei Symbole aus dieser Gruppe erwähnen: das *Staccato*- und das *Portamento*symbol. Sie werden dargestellt durch einen Punkt bzw. durch eine kleine gefüllte Pfeilspitze und befinden sich direkt oberhalb oder unterhalb einer Note. Durch das Auftreten dieser Symbole wird die Zeit, wie lange eine Note gespielt werden soll, in unterschiedlicher Weise verkürzt. Wie schon erwähnt geht diese Information aber beim Übertragen in die MIDI-Datei verloren, da aus einer kurz gespielten Note ein anderes Notensymbol mit kürzerem Notenwert entstehen würde.

Die "Wie"-Symbole sind sicherlich das schwierigste Gebiet bei der Notenschrifterkennung, weil sie oft in den Bereich der Texterkennung fallen, häufig nur sehr intuitive Werte repräsentieren und an nahezu beliebigen Stellen des Notenblattes auftreten können. Jedoch ist in vielen Anwendungsfällen der OME die Erkennung dieser Symbole gar nicht nötig, man denke an das Erstellen von Datenbanken über Melodien oder Themen bzw. an das Testen von Musiktheorien etc. Hier genügt die Erfassung all jener Symbole, die Auskunft darüber geben, *was* gespielt werden soll.

# 3. Erkennungsmethoden für Notenschriftsymbole

Ich möchte nun einige in der Literatur beschriebene Erkennungsmethoden anführen. Dabei beziehe ich mich hauptsächlich auf die Publikation [SDIA], die einen kritischen Überblick über die Analyse von Musiknotation gibt und die Arbeiten verschiedenster Autoren zusammenfaßt. Die Definition von "Musiksymbol" variiert dabei, jedoch sind sich die meisten Autoren darüber einig, daß Notenlinien Symbole mit eigener Berechtigung sind, unterschiedlich zu allen anderen Symbolen auf einem Notenblatt. Sie bilden einerseits ein Koordinatensystem, andererseits können sie die Konturen der übrigen Symbole verwischen oder gar stören. Deshalb werden die Notenlinien als erster Schritt bei der Erkennung sehr häufig eliminiert, um danach die "reinen" Musiksymbole vorzufinden.

## 3.1. Die Erkennung der Notenlinien

### 3.1.1. Linienverfolgung

Eine Möglichkeit zur Notenlinienerkennung ist die Verfolgung dünner, horizontaler Linien, die, falls sie innerhalb eines bestimmten Stärkebereiches liegen, eliminiert werden. Dabei kann jedoch eine ungewollte Fragmentierung von Symbolen entstehen, die sich auf einer Notenlinie befinden oder an eine Notenlinie tangential anschließen. Bei letzteren läßt sich diese Fragmentierung kaum mehr beseitigen, es bleiben zwei Symbole, die mit hoher Wahrscheinlichkeit zu einer Fehlinterpretation führen werden. Alternativ dazu könnte man nur jene Linien entfernen, die nicht durch andere Symbole überlagert werden. Hier bleiben natürlich Linienreste bestehen, die ebenfalls zu fehlerhafter Erkennung führen können.

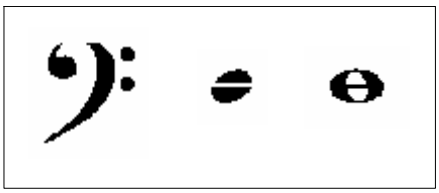

Abbildung 3-1: Fragmentierung

### 3.1.2. Linienwinkel und Linienstärke

Zum Zweck der Identifikation der Notenlinien über Linienwinkel und Linienstärke wird ein Fenster über die Vorlage bewegt um für jedes schwarze Pixel den Linienwinkel zu ermitteln. Dieser Winkel wird vom Mittelpunkt des Fensters zum am weitest entfernten schwarzen Pixel gemessen, welches so ausgewählt wird, daß der Weg dorthin keine weißen Pixel kreuzt. Der Fensterradius muß dabei für die Erkennung von Notenlinien relativ groß gewählt werden. Ist der Linienwinkel bestimmt, wird die Linienstärke gemessen, und diese zwei Maße zusammen erlauben nun die Unterscheidung zwischen horizontalen Linien und "fraglichen Pixels", die bei Überschneidungen von Linien mit Symbolen auftreten.

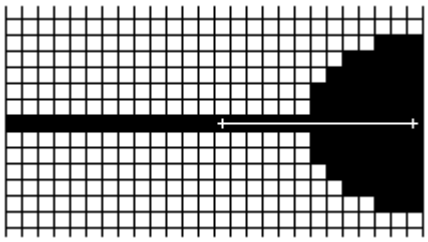

Abbildung 3-2: Bestimmung des Linienwinkels

### 3.1.3. Y-Projektion

Bei dieser Methode wird das Notenbild auf die y-Achse projiziert. Es entsteht ein Vektor, dessen i-te Komponente die Summe aller schwarzen Pixel in der i-ten Zeile des Bildes repräsentiert. Einige Autoren verwenden hierfür auch die Bezeichnung horizontale Projektion. Die Aufgabe besteht nun darin, aus diesem Vektor Gruppen von fünf Maxima zu suchen, welche die Notenlinien einer Notenzeile darstellen. Sind diese Maxima nicht eindeutig zu identifizieren (aufgrund von Hilfslinien oder Balken etc.), werden zusätzlich lokale, kleinere Y-Projektionen durchgeführt. Dabei können auch Informationen über den Mindestabstand zwischen Notenlinien in die Suche der Maxima einfließen. Diese ist eine Methode, die das Eliminieren der Notenlinien nicht voraussetzt.

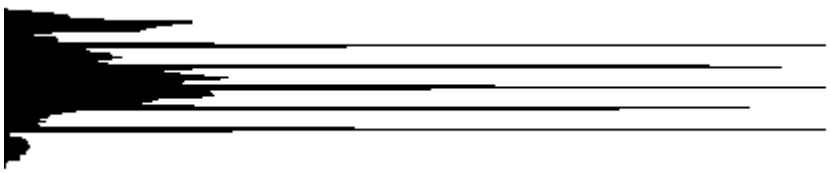

Abbildung 3-3: Y-Projektion

## 3.2. Die Erkennung der Musiksymbole

### 3.2.1. Höhe und Breite

Aufgrund der Tatsache, daß verschiedene Typen von Musiksymbolen signifikant unterschiedlich in ihrer Gesamtgröße sind, wird für jedes Symbol ein umschließendes Rechteck normiert auf den Notenlinienabstand gebildet, wodurch eine Grobklassifizierung vorgenommen werden kann. Die dadurch übriggebliebenen möglichen Symbole werden dann durch in der Literatur nicht näher genannte, genauere Analyse endgültig klassifiziert.

### 3.2.2. Musterprimitiven und zusammengesetzte Symbole

Musterprimitiven (Notenköpfe, Notenhälse, Fähnchen usw.) formen zusammengesetzte Symbole (Noten, Akkorde, über Balken verbundene Notengruppen etc.). Diese Sichtweise ermöglicht die Vereinfachung des Erkennungsvorganges durch Zerlegung der Erkennungsaufgaben. Die Primitiven werden dabei in unparametrisierte Zeichen und in parametrisierte Linien mit jeweils unterschiedlichen Erkennungsmerkmalen eingeteilt. Bestimmend für Zeichen sind:

- Masse (Ober-, Untergrenze),
- Breite (Ober-, Untergrenze),
- Höhe (Ober-, Untergrenze),
- Neigungswinkel.

Linien werden durch folgende Merkmale charakterisiert:

- prinzipielle Richtung (horizontal, vertikal),
- Winkel (erlaubte Abweichung von horizontal bzw. vertikal),
- Dicke (Ober-, Untergrenze),
- Länge (Ober-, Untergrenze),
- maximal erlaubte Größe von "Löchern".

### 3.2.3. Schablonenvergleich

Es existieren für jedes Symbol mehrere Schablonen (Vergleichsvorlagen) in verschiedenen Größen, wobei die richtige Größe durch die Normierungsparamter der Linienabstände bestimmt wird. Dabei werden die Symbole der Notenvorlage mit dem Vorrat an Schablonen verglichen, eventuelle Erkennungsfehler können durch Wissen über die Syntax der Musiknotation ausgemerzt werden.

### 3.2.4. Projektionen

Auch zum Erkennen der Musiksymbole können Projektionen herangezogen werden. Jedoch entstehen durch die Anfälligkeit für Störungen und die extreme Rotationsempfindlichkeit meist ernste Probleme bei der Erkennung. Es müssen X- und Y-Projektionen in schmalen Fenstern um das Symbol vorgenommen werden, durch charakteristische Punkte in den Projektionen bzw. durch Vergleich mit schon früher gespeicherten Projektionen wird die Klassifizierung durchgeführt. Eine X-Projektion (Projektion auf die x-Achse) über eine Notenzeile könnte etwa zum Auffinden der Musiksymbole, die sich in dieser Notenzeile befinden, dienen.

### 3.2.5. Regelbasierte Systeme

Alle bisher erwähnten Methoden verwenden bereits bestimmtes Wissen über die Musiknotation in irgend einer Form. Sei es die Existenz und die Eigenschaften von Notenlinien, -köpfen und -hälsen, die Syntax der Musiknotation etc. Bei regelbasierten Erkennungssythemen wird versucht, dieses Wissen in Form von Regeln abzuspeichern und schon bei den ersten Schritten der Symbolsegmentierung und Erkennung anzuwenden.

# 4. Programmkonzept für die Notenschrifterkennung

Das Ziel dieser Diplomarbeit ist die Entwicklung eines Programms, mit dessen Hilfe man eine Bilddatei mit der Darstellung eines Notenblattes in eine Musikdatei konvertieren kann, welche die Informationen des Notenblattes in einer Form repräsentiert, die durch entsprechende Anwendungsprogramme schnell weiterverarbeitet werden kann. Das Programm soll NOSE heißen, eine Abkürzung für den Begriff NotenSchriftErkennung und soll auf IBM kompatiblen PCs mit heute "üblicher" Ausstattung (DOS, Windows...) ablaufen. Zur Erfüllung des oben genannten Zieles würde es zwar genügen, ein Batchprogramm unter DOS zu schreiben, welches in etwa folgende Syntax aufweist:

NOSE <Bilddatei> [ <Musikdatei> ].

Der erste Parameter ist der Name der Bilddatei, danach kann man noch optional den Namen der zu erzeugenden Musikdatei angeben, entfällt dies, so wird der Name der Bilddatei mit einer veränderten Extension verwendet. Jedoch wird der Benutzer und auch der Entwickler (!) bei dieser Methode nur sehr mangelhaft visuell unterstützt. Der Erkennungsprozeß kann unter Umständen äußerst langwierig sein, und dabei hat der Benutzer keine Möglichkeit, in die Erkennung einzugreifen. Es bleibt ihm nur zu hoffen, daß am Ende das richtige Ergebnis erscheint. Aus diesem Grund und auch deshalb, weil Batchprogramme wohl kaum mehr den heutigen Bedürfnissen entsprechen, habe ich mich entschlossen, ein vollwertiges Windows-Programm zu entwickeln. Dabei möchte ich eine Klassenbibliothek mit dem Namen C++/Views (Liant Software Corp.) verwenden, die eine Fülle von Klassen zur Gestaltung graphischer Benutzerschnittstellen bereits zur Verfügung stellt, um den zusätzlichen Aufwand der Windows-Programmierung zu minimieren. Diese Klassenbibliothek bietet mehrere Vorteile:

- Compilerunabhängigkeit (Borland C++, Microsoft C++, Zortech C++...),
- Plattformunabhängigkeit (Microsoft Windows, OS/2 Presentation Manager, UNIX/Motif...),
- schlüssige Klassenhierarchie mit über 100 Objektklassen,
- SmallTalk ähnlicher Klassenbrowser usw.

Die Programmiersprache C++ ist damit vorgegeben, in der Compilerfrage tendiere ich eher zu Microsoft, weil mit Visual C++ eine meiner Meinung nach äußerst gelungene Entwicklungsumgebung angeboten wird.

Im Folgenden möchte ich die Forderungen an die Funktionalität von NOSE auflisten:

- Laden einer Bilddatei: Es existieren eine Fülle verschiedener Formate für Bilddateien, die natürlich nicht alle unterstützt werden können. Zum Anfang werde ich mich ausschließlich auf Windows-Bitmap-Dateien (siehe Kapitel 5.2.1) konzentrieren. Andere Formate müssen zuerst mit geeigneten Graphikprogrammen entsprechend konvertiert werden.
- Anzeigen der Bilddatei: Die Bilddatei wird in einem Fenster angezeigt, das horizontale und vertikale Rollbalken zum Scrollen enthält. Als spätere Erweiterung wäre eine in der Größe veränderbare (zoombare) Darstellung denkbar.
- Bearbeiten der Bilddatei: Die Bilddateien können mit NOSE nicht bearbeitet werden. Eventuelle Störungen und andere Fehler müssen mit einem geeigneten Graphikprogramm zuerst entfernt werden.
- Segmentieren der Bilddatei: Die Segmentierung der Bilddatei wird durch einen Menüpunkt gestartet. Dabei wird das Bild auf das Vorhandensein von Notenlinien, Notenzeilen, Notensysteme und Symbolsegmenten untersucht und gleichzeitig eine Datenstruktur aufgebaut, die diese Gliederungen widerspiegelt.
- Klassifizieren der Symbolsegmente: Die Klassifikation der Symbolsegmente wird durch einen Menüpunkt gestartet und ersetzt die Segmente durch klassifizierte Symbole.
- Anzeigen der erkannten Musiksymbole: Die beste Möglichkeit zur Anzeige der erkannten Musiksymbole wäre natürlich die erneute Darstellung in Form von Notenschrift, um das Original mit dem Produkt des Erkennungsvorganges direkt vergleichen zu können. Dies würde jedoch den Rahmen der Diplomarbeit bei weitem sprengen und ist auch nicht unbedingt nötig, da es bereits genügend Anwendungsprogramme gibt, mit deren Hilfe man diesen Vergleich durchführen kann. Anstelle der Notenschriftdarstellung wird aber eine verkürzte Textdarstellung neben dem betreffenden Symbol zum schnellen, direkten Vergleich angebracht.
- Bearbeiten der erkannten Musiksymbole: Für diese Version von NOSE ist ein nachträgliches Bearbeiten der erkannten Musiksymbole nicht vorgesehen. Es wäre jedoch in weiterer Folge wünschenswert, z.B. falsch erkannte Symbole zu löschen bzw. zu verändern oder nicht erkannte bzw. von NOSE nicht unterstützte Symbole nachträglich einzufügen.
- Speichern der erkannten Musiksymbole: Zu jedem Zeitpunkt können die bis dahin erkannten Musiksymbole in einer MIDI-Datei (Format 1, siehe Kapitel 5.2.2) gespeichert werden.

## 4.1. Die Struktur eines Notenblattes

Bei dieser Arbeit entspricht ja eine Bilddatei einem gescannten Notenblatt und könnte in etwa folgendes Aussehen haben:

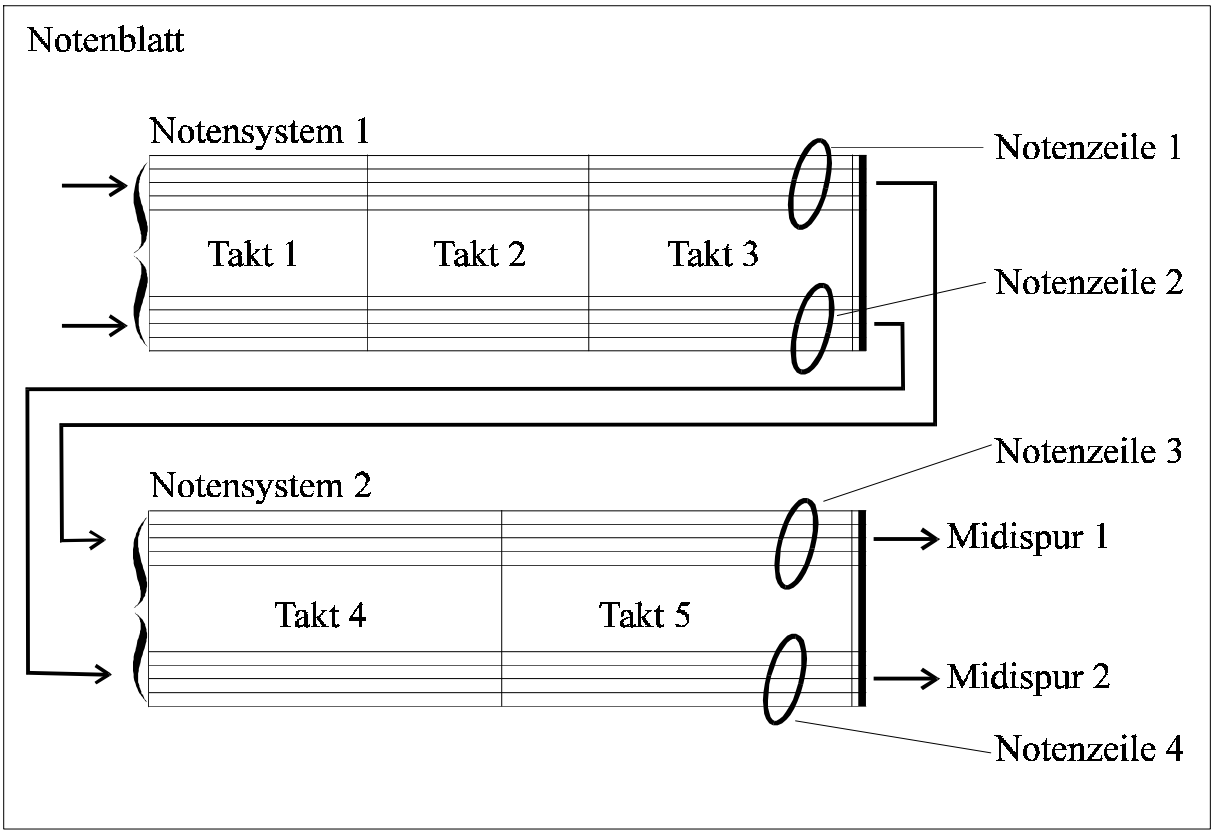


Abbildung 4-1: Beispiel für eine Bilddatei (leeres Notenblatt)

Auf dem *Notenblatt* in Abbildung 4-1 befinden sich 2 *Notensysteme* mit jeweils 2 *Notenzeilen*, die wie aus der Grafik ersichtlich zu *MIDI-Spuren* zusammengefaßt werden. Insgesamt sind also 4 *Notenzeilen* vorhanden, jedoch nur 2 *MIDI-Spuren*. Weiters besteht das erste *Notensystem* aus 3 *Takten*, das zweite aus 2, also insgesamt 5 *Takte*. Aufgrund dieser Gliederung kann man etwa folgenden Graphen angeben, welcher die Struktur des Notenblattes repräsentiert:

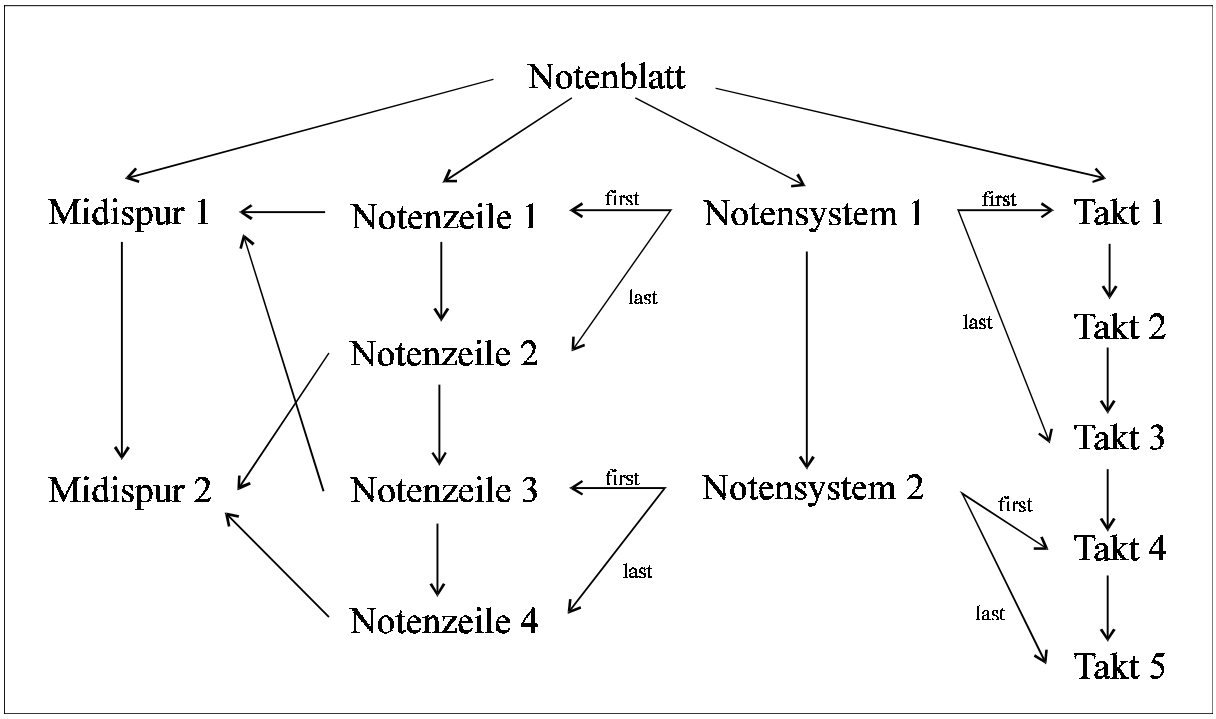


Abbildung 4-2: Struktur des Notenblattes

Die Elemente der Datenstruktur besitzen 3 verschiedenartige Merkmale:

- graphische Bestimmungsmerkmale (Positions- und Größenangaben über das jeweilige Element zur Lokalisierung in der Bilddatei),
- musikalische Bestimmungsmerkmale (Standardwerte, die im Bereich des Elementes gelten, solange sie nicht durch Musiksymbole verändert werden),
- Verweise auf andere Elemente oder auf Listen von Elementen.

Die folgende Tabelle führt alle Elemente gemeinsam mit den Verweisen an und stellt die graphischen bzw. musikalischen Bestimmungsmerkmale einander gegenüber. Die darauffolgende Abbildung dient zur Erläuterung der graphischen Merkmale.

| Element | Graph. Merkmale | Musik. Merkmale | Verweise |
|---|---|---|---|
| Notenblatt | Breite, Höhe, Liniendicke, Linienabstand | Grundtempo, -tonart und -taktart | Listen der Notensysteme, Notenzeilen, Takte und MIDI-Spuren |
| Notensystem | Bezugspunkt, Breite, Höhe | - | Teilliste der Notenzeilen und Takte |
| Notenzeile | Bezugspunkt, Breite, Höhe, Abstand zur nächsten Notenzeile | Notenschlüssel | MIDI-Spur |
| Takt | Bezugspunkt, Breite, Höhe | Taktart | |
| MIDI-Spur | - | akt. Notenschlüssel, akt. Transponierung akt. Anschlagstärke | Liste der Musiksymbole |

Abbildung 4-3: Bestimmungsmerkmale der Segmentierung

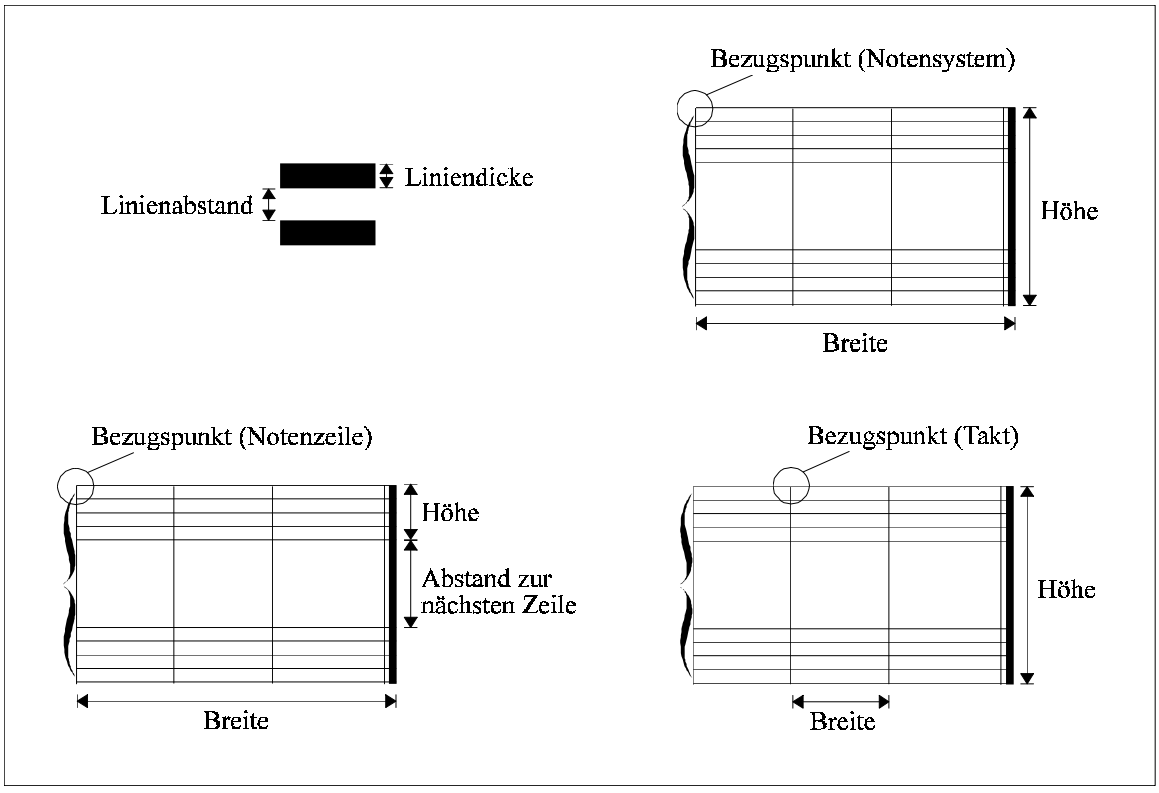


Abbildung 4-4: Graphische Bestimmungsmerkmale

## 4.2. Die Musiksymbole

Vorausschickend möchte ich bemerken, daß es natürlich wünschenswert wäre, alle Musiksymbole, die einer gewissen Normung unterliegen, zu erkennen und zu unterstützen. Dieses Vorhaben würde jedoch zwei Voraussetzungen erfordern, die für diese Diplomarbeit nicht realisierbar sind:

- eine Zeichen- bzw. Schrifterkennung für textuelle Musiksymbole und
- eine Erweiterung der Spezifikation von MIDI-Dateien um Ereignisse für jene Musiksymbole, die entweder nicht oder nur indirekt abzubilden sind.

Im Folgenden wird eine Liste der Musiksymbole angeführt, die aus dem Studium verschiedenartiger Notenschrifteditionen ([BACH1], [BACH2], [BACH3], [BACH4], [BEET1], [BEET2], [BLOCK1], [HARM], [HAY1], [HEL1], [MOZ1], [MOZ2], [MOZ3], [PIA1] und [SCHU1]) zusammengestellt wurde:

| Musiksymbol | Beispiele | Bestimmungsmerkmale |
|---|---|---|
| Noten | ω η H θ Θ εE ξΞ ρP ... | Tonhöhe, Länge |
| n-Tolen | 3 ... | Wert |
| Vorschläge | : ; ⊃ ... | Notenanzahl |
| Vorzeichen | # ⇐ β ≡ ν | Notenlinie |
| Pausen | • ⌊ ∈ ™ ⊕ ... | Länge |
| Notenschlüssel | & ? ... | |
| Transponierung | 8 ... | Wert |
| Anschlagstärke | ≠ π Π Φ ϕ ⊗... | Wert |
| Anschlagdynamik | *cresc. decresc. dim.* > ... | Wert, Länge |
| Anschlagart | . ◊ | |
| Taktänderung | χ X ... | |
| Tempodynamik | *rit. accel.* | Wert, Länge |
| Wiederholungszeichen | ] } | |
| Gruppierungen | | Notenanzahl |
| Dämpfungspedal | ϒ * | |
| Spezialsymbole | ∧ μ M ∝ T Y υ γ ... | |

Im Kapitel 2 wurde eine Kategorisierung der Musiksymbole in "Was"- und in "Wie"-Symbole vorgenommen. Die grundsätzliche Aufgabe von NOSE sehe ich in der Erkennung der "Was"-Symbole, die festlegen, welche Töne zu welchem Zeitpunkt wie lange gespielt werden. Folglich müssen daher folgende Symbole automatisch erkannt werden:

- Noten,

- Pausen,
- Vorzeichen,
- Notenschlüssel und
- Taktänderungen.

Die Verwaltung der Musiksymbole muß jedoch so ausgelegt sein, daß die restlichen sogenannten "Wie"-Symbole in weiteren, über die Diplomarbeit hinausgehenden Schritten in die Erkennung aufgenommen werden können. Im Wesentlichen wird dafür pro MIDI-Spur eine Liste angelegt, welche sämtliche Musiksymbole aufnehmen kann. Jedes Symbol erhält dabei eine Taktposition, die festlegt, wann es in Kraft tritt, eine Angabe über die zeitliche Dauer (entweder durch einen Längenwert oder durch explizite Start- bzw. Stopsymbole) und symbolspezifische Werte (siehe vorangegangene Tabelle).

## 4.3. Die Benutzerschnittstelle

### 4.3.1. Das Hauptfenster

Das Hauptfenster von NOSE dient zur Darstellung der Bilddatei und zum Anzeigen der erkannten Musiksymbole in einer noch festzulegenden, geeigneten Weise. Es stellt Scrollbalken zum Navigieren in großen Bilddateien zur Verfügung, denn ein Vergrößern oder Verkleinern der Darstellung des Bildes ist vorerst nicht vorgesehen.

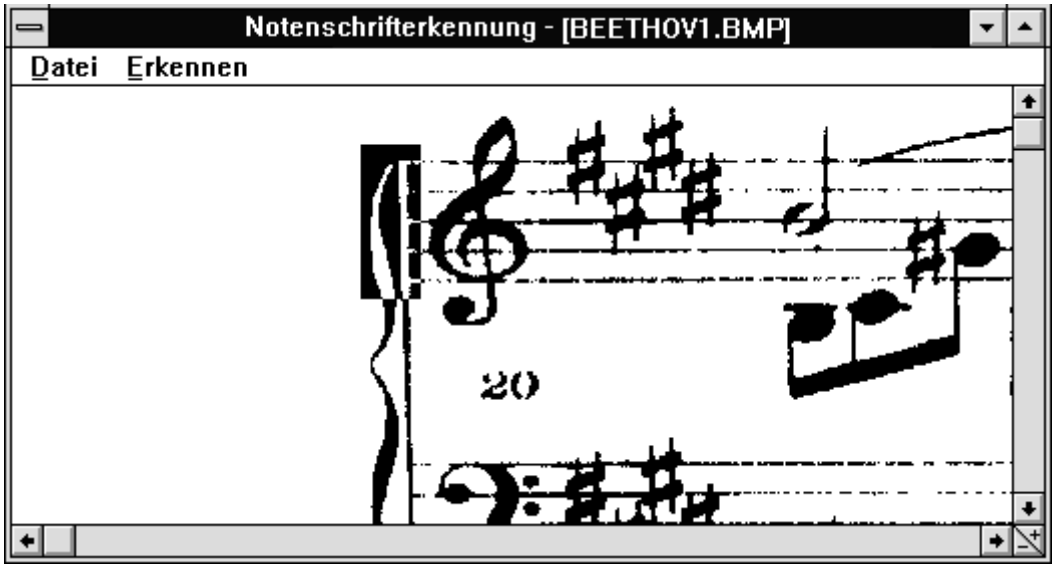


Abbildung 4-5: Illustratives Szenario von NOSE

### 4.3.2. Das Datei-Menü

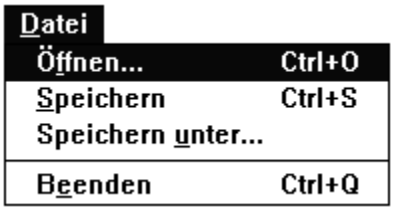


Abbildung 4-6: Das Datei-Menü

**Datei/Öffnen** läßt ein Dateiauswahlfenster erscheinen, mit dessen Hilfe die gewünschte Bilddatei geöffnet werden kann.
**Datei/Speichern** erzeugt eine MIDI-Datei von den bisher erkannten Musiksymbolen und legt sie unter dem Namen der Bilddatei jedoch mit der Extension .MID ab.
**Datei/Speichern unter** ist gleichbedeutend zu Datei/Speichern, jedoch kann der Name und der Pfad der zu erzeugenden MIDI-Datei mit Hilfe eines Dateiauswahlfensters bestimmt werden.
Mit **Datei/Beenden** verläßt man das Programm NOSE.

### 4.3.3. Das Erkennen-Menü

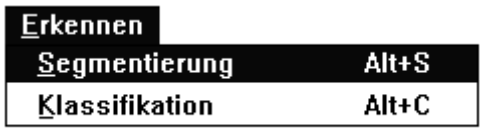


Abbildung 4-7: Das Erkennen-Menü

**Erkennen/Segmentierung** führt die Segmentierung der Bilddatei durch. Am Ende des Vorgangs gibt eine Dialogbox Auskunft über die erzielten Ergebnisse.
**Erkennen/Klassifikation** startet die Klassifikation der Symbolsegmente. Am Ende des Vorgangs gibt eine Dialogbox Auskunft über die erzielten Ergebnisse.

## 4.4. Phasen der Implementierung

Die Implementierung von NOSE wird in folgenden Schritten ablaufen:

- Erstellung eines Prototyps (keine Funktionalität, Benutzerschnittstelle);
- Laden und Anzeigen von Bitmap-Dateien;
- Segmentierung;
- Klassifikation (zuerst nur Noten)
- Erzeugen von MIDI-Dateien;
- Weitere Musiksymbole klassifizieren (Vorzeichen, Pausen, Notenschlüssel...);
- Testen;

# 5. Die Implementierung von NOSE

## 5.1. Die MIDI-Spezifikation

MIDI steht für "Musical Instrument Digital Interface". Es ist ein Standard für den Austausch von digitalen Daten im Musik- und Tonstudiobereich. Die Übertragung erfolgt seriell und asynchron, die Daten werden zeitlich hintereinander über eine Zweidrahtleitung mit 5-poligen DIN-Steckern ohne zusätzliche Synchronleitung geschickt. Nahezu alle auf dem Markt befindlichen Geräte aus dem Musik- und Tonstudiobereich verfügen über eine sogenannte MIDI-Schnittstelle, die aus drei 5-polige DIN-Buchsen mit den Bezeichnungen IN, OUT und THRU besteht. Auch Computer der unterschiedlichsten Plattformen (PC, Apple, Atari etc.) können mit einer MIDI-Schnittstelle versehen werden wodurch sich MIDI-Netzwerke zwischen Musik-, und Tonstudiogeräten und dem Computern gestalten lassen.

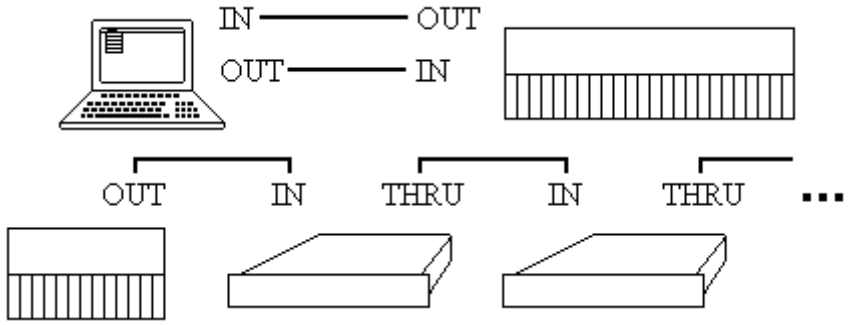


Abbildung 5-1: MIDI-Netzwerke

Die Komplexität dieser Netzwerke ist allerdings eingegrenzt durch eine maximale Kabellänge von 15 Metern zwischen zwei Geräten und durch die entstehenden Verzögerungszeiten beim Hintereinanderschalten mehrerer Geräte.
Es gibt zwei verschiedene Arten von MIDI-Nachrichten, mit denen Informationen über das Netzwerk ausgetauscht werden können:
Die *Systemnachrichen* erlauben den Transfer von großen Datenmengen (System-Exclusive-Message), z.B. der gesamte Speicherinhalt eines MIDI-Gerätes, den Austausch von systemübergreifenden Nachrichten (System-Common-Message) und die Kommunikation via Echtzeitmeldungen (System-Real-Time-Message). Diese Nachrichten werden jedoch für diese Diplomarbeit nicht gebraucht.
Die zweite Art der MIDI-Nachrichten sind die *Kanalnachrichten*, die das Spielen einer MIDI-Tastatur nachahmen. Es handelt sich dabei um kurze Echtzeitmeldungen, die aus einem Statusbyte und maximal 2 Datenbytes bestehen. Im Statusbyte enthalten ist die sogenannte Kanalnummer, eine definierte Adresse zur Steuerung eines bestimmten Gerätes. Es sind 16 Kanalnummern möglich, was eine unerfreuliche Einschränkung

darstellt wenn man bedenkt, daß ein modernes MIDI-Gerät (Synthesizer) oft schon in der Lage ist auf allen 16 Kanälen gleichzeitig zu senden bzw. zu empfangen. Daher verfügen viele MIDI-Schnittstellen über mehrere OUT-Ports, welche die Anzahl der Kanäle multiplizieren. Die höherwertige Hälfte des Statusbytes bestimmt den Status, in den das MIDI-Gerät versetzt wird (n bezeichnet die Kanalnummer):

| Status | Byte | Beschreibung |
|---|---|---|
| Note Off | 0x8n | Taste loslassen |
| Note On | 0x9n | Taste anschlagen |
| Polyphonic Key Pressure | 0xAn | erlaubt ausdrucksstarkes Spielen durch nachträgliches Drücken einer bereits angeschlagenen Taste |
| Control Change | 0xBn | Simulation des Sustainpedals eines Klaviers, Veränderung der Lautstärke etc. |
| Programm Change | 0xCn | Auswahl einer Klangfarbe (Sound) |
| Channel Pressure | 0xDn | wie Polyphonic Key Pressure jedoch monophon, d.h. gleich für die gesamte Tastatur |
| Pitch Bend Change | 0xEn | simuliert das "Ziehen" der Saiten einer Gitarre |

Zur Reduktion der Datenmenge kann das Statusbyte auch entfallen was bedeutet, daß der zuvor vorhandene Status weiter gilt (running status).
Die Kanalnachrichten werden in dieser Diplomarbeit dazu verwendet, die musikalischen Informationen, die in einem Notenblatt abgebildet sind, in computerlesbare Form zu bringen. "Was"-Symbole (siehe Kapitel 2.2) werden durch die Befehle *Note On* bzw. *Note Off* repräsentiert, "Wie"-Symbole (siehe Kapitel 2.3) können durch die restlichen Kanalnachrichten teilweise abgedeckt werden.
Die Aufgabe nach dem digitalen Analysieren des Notenblattes besteht also darin, geeignete Kanalnachrichten zu erzeugen, welche das in Notenschrift gefaßte Musikstück widerspiegeln und diese Nachrichten in zeitabhängiger Form in einer MIDI-Datei (siehe Kapitel 5.2.2) abzuspeichern.

## 5.2. Ein- und Ausgabedateien

### 5.2.1. Das Windows-Bitmap-Dateiformat

In der Planungsphase von NOSE bin ich noch davon ausgegangen, daß Bilddateien im TIFF-Format verarbeitet werden sollen. Diesbezüglich hat sich jedoch eine Änderung ergeben, die mit der verwendeten Klassenbibliothek *C++/Views* in Zusammenhang steht. Diese Bibliothek besitzt nämlich eine fertige Klasse namens *VBitMap*, welche das Laden und Darstellen (*VBitMapView*) von Windows-Bitmap-Dateien implementiert. Aus diesem programmiertechnischen Vorteil sollten sich jedoch keine Nachteile für die allgemeine Anwendbarkeit des Programms ergeben, da ja Bitmap das Windows eigene Dateiformat für Bilder ist und falls es die Scannersoftware nicht unterstützt, so ist zumindest eine Konvertierung über eines der gängigen Windows-Graphikprogramme (z.B. CorelDraw!) möglich.
Windows-Bitmap-Dateien werden in einem geräteunabhängigen Format (DIB, device-independent-bitmap) gespeichert. Jede Bitmapdatei besteht aus einem Datei-Header, einem Informations-Header, aus einer Farbentabelle und einer Liste von Bytes, welche die Bits der Bitmap (Pixel) repräsentieren. Der Datei-Header enthält dabei Informationen über den Typ, die Größe und die Gestalt der Bitmapdatei, der Informations-Header spezifiziert die Dimensionen, den Kompressionstyp und das Farbenformat der Bitmap, und die Farbentabelle enthält so viele Elemente wie es Farben in der Bitmap gibt. Die nachfolgende Liste von Bytes ist gruppiert in aufeinanderfolgenden Zeilen (scan lines) beginnend mit der letzten Zeile der Bitmap. Wieviele Bytes eine Scanlinie ausmachen hängt vom Farbenformat und von der Breite der Bitmap in Pixel ab. Es existieren folgende Farbenformate:

- monochrome: ein Bit repräsentiert ein Pixel (0 bedeutet schwarz, 1 bedeutet weiß)
- max. 16 Farben: jedes Pixel der Bitmap wird durch einen 4-bit-Index in die Farbentabelle repräsentiert
- max. 256 Farben: jedes Byte ist ein Index in die Farbentabelle
- max. $2^{24}$ Farben: Folgen von 3 Bytes repräsentieren die relative Intensität der Farben rot, grün und blau für ein Pixel.

## 5.2.2. Das Standard-MIDI-Dateiformat

Dieses Dateiformat wurde im Jahr 1988 von einer Vereinigung namens *The International MIDI Association* definiert und dient dem Zweck, zeitabhängige MIDI-Nachrichten unter verschiedenen Programmen auf der gleichen aber auch auf unterschiedlichen Computerplattformen auszutauschen. Nahezu alle am Markt befindlichen Musikprogramme (z.B. Cubase, CakeWalk etc.) verwenden zwar prinzipiell ihre eigenen Dateiformate, können aber zumindest Standard-MIDI-Dateien lesen und schreiben, wodurch im Gegensatz zu den Bildformaten ein tatsächlich weit verbreiteter Standard geschaffen wurde. MIDI-Dateien bestehen aus sogenannten *chunks* (Klotz, Klumpen), die jeweils aus einer Typbezeichnung (4-ASCII-Zeichen) und einer 32-bit Längenangabe zusammengesetzt sind, welche die Anzahl der nachfolgenden Bytes repräsentiert. Aus der folgenden Grammatik in EBNF-Schreibweise, welche durch die Angabe von Datentypen erweitert wurde, ist die Struktur von MIDI-Dateien zu erkennen (LONG steht dabei für einen 32-bit-, WORD für einen 16-bit-Wert und VLQ für Variable-Length-Quantity, siehe Kapitel 5.4.15):

```
MIDIFile    = HeaderChunk { TrackChunk }.
HeaderChunk = "MThd"
              length:       LONG    (=6)
              format:       WORD    (0, 1 oder 2)
              noTracks:     WORD
              devision:     WORD
TrackChunk  = "MTrk"
              length:       LONG
              { MTrkEvents }.
MTrkEvent   = deltaTime:    VLQ
              MIDIMessage.
MIDIMessage = ChannelMessage | SystemMessage | MetaMessage.
```

Es gibt drei verschiedene Formate für MIDI-Dateien gekennzeichnet durch den *format*-Parameter im *HeaderChunk*, die sich in der Interpretation der nachfolgenden *TrackChunks* unterscheiden. Format-0-Dateien haben nur einen TrackChunk, die anderen Formate können beliebig viele TrackChunks besitzen, die entweder in vertikal eindimensionaler Form gleichzeitig (Format-1-Dateien) oder in horizontal eindimensionaler Form nacheinander (Format-2-Dateien) zu spielen sind. Der 1. TrackChunk wird dabei üblicherweise als Mastertrack (Masterspur) bezeichnet und enthält Informationen, die das gesamte Stück betreffen (Tempo, Takt etc.).

Das Programm NOSE erzeugt Format-1-Dateien, weil alle Notenzeilen eines Notensystems gleichzeitig gespielt werden.

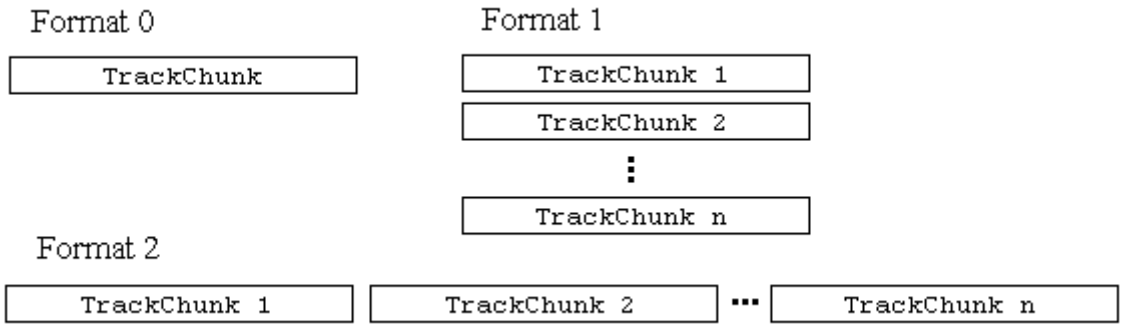


Abbildung 5-2: Formate der MIDI-Dateien

Ein Ereignis eines *TrackChunks* wird *MTrkEvent* genannt und besteht aus der Zeitangabe *deltaTime* (in Ticks) und der MIDI-Nachricht *MIDIMessage*, welche nach Ablauf dieser Zeit gesendet werden soll. Die Interpretation der Zeitangabe hängt vom Parameter *division* im *HeaderChunk* ab, der die Anzahl der Ticks für eine 1/4-Note angibt. Es können sowohl Kanalnachrichten (*ChannelMessage*) wie auch Systemnachrichten (*SystemMessage*) gespeichert werden, welche im Kapitel 5.1 beschrieben sind. Zusätzlich dazu sind jedoch auch sogenannte Metanachrichten (*MetaMessage*) erlaubt, die es ermöglichen, Texte, Copyrightnotizen, Instrumentennamen, Tempo- und Taktangaben und vieles mehr anzubringen. Anschließend ein kurzes Beispiel einer MIDI-Datei, welche mit der Klangfarbe eines Klaviers eine halbe Pause gefolgt von der halben Note E4 mit dem Tempo 120 im 4/4-Takt repräsentiert:

Header Chunk:

| | |
|---|---|
| **4D 54 68 64** | "MThd" |
| **00 00 00 06** | Länge: 6 Bytes |
| **00 01** | Format 1 |
| **00 02** | 2 Tracks |
| **00 60** | 96 Ticks/Viertelnote |

Track Chunks:

| | |
|---|---|
| **4D 54 72 6B** | "MTrk" |
| **00 00 00 14** | Länge: 20 Bytes |

| Delta-Zeit | Bytes | Kommentar |
|---|---|---|
| **00** | **FF 58 04 04 02 18 08** | 4/4-Takt,<br>24 MIDI clocks/click<br>8 32stel Noten/24 clocks |
| **00** | **FF 51 03 07 A1 20** | Tempo 120 |
| **83 00** | **FF 2F 00** | Ende des Tracks |

| | |
|---|---|
| **4D 54 72 6B** | "MTrk" |
| **00 00 00 11** | Länge: 17 Bytes |

| Delta-Zeit | Bytes | Kommentar |
|---|---|---|
| **00** | **C0 00** | Kanal 1: ProgChg 1 (Klavier) |
| **81 40** | **90 4C 50** | Kanal 1: NoteOn E4, mezzo-forte |
| **81 40** | **80 4C 40** | Kanal 1: NoteOff E4, standard |
| **00** | **FF 2F 00** | Ende des Tracks |

## 5.3. Der Erkennungsprozeß

In diesem Kapitel möchte ich die Algorithmen beschreiben, die zur Erkennung von Notenschrift im Programm NOSE verwendet wurden. Neben einer Abbildung des Source-Codes der relevanten Methoden werden hier auch die Vor- und Nachteile, die Schwächen und Stärken der angewandten Methoden beschrieben. Eine Beschreibung der Klassen von NOSE erfolgt im Kapitel 5.4., die globalen Datenstrukturen wurden im Kapitel 4.1 abgehandelt.
Ähnlich wie bei der Texterkennung (OCR) kann der Erkennungsprozeß auch hier in mehrere Schritte gegliedert werden. Der erste Schritt ist die *Segmentierung* des digitalen Bildes. Das Ziel dabei ist die Unterteilung des Bildes in verschiedene Bedeutungsbereiche, die einem Segment entsprechen. Es werden je ein Segment für die gesamte Partitur (Score), für alle Notenzeilen (Staff) und für die einzelnen Symbole (Symbol) des Notenblattes erzeugt. Diese Segmente werden in Listen gespeichert, die Partitur verwaltet eine Liste aller in ihr enthaltenen Notenzeilen und jede Notenzeile verwaltet wiederum eine Liste der auf ihr befindlichen Symbole. Im zweiten Schritt, genannt *Klassifizierung*, werden dann diese Segmente (abstrakte Symbole) mittels Vergleiche mit Mustersymbolen durch konkrete Ausprägungen, wie z.B. Noten (Note), Pausen (Rest) oder Notenschlüssel (Clef) ersetzt. Am Ende der Klassifizierung befinden sich in den Listen der Notenzeilen zum Einen konkrete Symbole, solche, die klassifiziert werden konnten, und zum Anderen abstrakte, zu keiner Symbolklasse gehörige Symbole, solche, die nicht klassifiziert werden konnten. Dazu gehören hauptsächlich die sogenannten "Wie"-Symbole (siehe Kapitel 2.3), welche in so vielen und so unterschiedlichen Erscheinungsformen auftreten, daß sie von NOSE im Rahmen dieser Diplomarbeit nicht unterstützt werden. Den dritten Schritt des Erkennungsprozesses habe ich *Interpretation* genannt. Im folgenden Beispiel ist zu sehen, was die Interpretation angewandt auf die Texterkennung in etwa bedeuten könnte: durch Segmentierung und Klassifizierung wird festgestellt, daß der Text "$2^3$" die Zeichen '2' und '3' enthält. Die Interpretation müßte nun erkennen, daß das Zeichen '3' hochgestellt und etwas kleiner geschrieben ist, was dann den mathematischen Term 2 hoch 3 ergibt. Bei der Notenschrifterkennung bedeutet die Interpretation, daß z.B. aus einem Symbol Note, bestehend aus einem gefüllten Notenkopf, einen Notenhals rechts vom Notenkopf nach oben zeigend und 2 Fähnchen am Ende des Notenhalses, die 1/16-Note E' (eingestrichenes E) wird, da sich dieses Symbol auf der untersten Notenlinie befindet und am Anfang der Notenzeile ein Violinschlüssel steht. Zu spielen ist diese Note nach der relativen Zeit von 96 Ticks, da sich zuvor eine 1/16-Pause befindet, für die Dauer von ebenfalls 96 Ticks (Dauer einer 1/16-Note). Zur Interpretation von Symbolen bei der Notenschrifterkennung ist es daher nötig, das Symbol anhand der vertikalen und horizontalen Position in der

Notenzeile, anhand der Bestimmungsstücke und aufgrund des Kontextes zu bewerten. Dabei ist der letzte Punkt im Fall von mehrstimmigen Notationen (siehe Kapitel 5.3.3) sicherlich der schwierigste.
Es folgt nun eine genauere Beschreibung der einzelnen Schritte des Erkennungsprozesses.

### 5.3.1. Das Segmentieren

Wie bereits erwähnt ist das Ziel der Segmentierung die Einteilung des digitalen Bildes in unterschiedliche Bedeutungsbereiche. Das größte Problem dabei sind die Notenlinien, denn sie stellen zwar das zentrale Koordinatensystem für die übrigen Musiksymbole der Notenschrift dar, verfälschen dabei aber gleichzeitig deren Konturen, indem sie Symbole verbinden und/oder durchlaufen. Ein auch in [CSMIA] beschriebener Ansatz eliminiert daher als ersten Schritt der Segmentierung die Notenlinien, um die restlichen Symbole dadurch leichter zu erkennen. Der Erfolg dieser Methode hängt von der Intelligenz des Eliminierungsalgorithmus ab (siehe Kapitel 3.1), denn werden dabei auch Teile der Symbole entfernt, was nahezu unvermeidbar ist, bzw. werden Teile der Notenlinien innerhalb von Symbolen nicht entfernt, so geht der Vorteil der leichteren Klassifizierung verloren. In meiner Diplomarbeit habe ich einen in der Literatur nur "erwähnten" Ansatz basierend auf Projektionen gewählt, mit den Argumenten, daß einerseits bei diesem Verfahren das Originalbild unverändert bleibt und dadurch so keine Fehler entstehen können, andererseits dieses Verfahren dadurch einmal einer genaueren Prüfung auf die Anwendbarkeit zum Segmentieren von Notenschrift unterzogen wird. Ich nenne die hier angewandte Methode *Segmentierung durch fortgesetzte X-/Y-Projektionen*, weil dabei abwechselnd X- und Y-Projektionen auf immer kleinere Bereiche des Bildes durchgeführt werden, bis daraus die gewünschten Informationen erhalten werden können. Folgende Schritte werden zur kompletten Segmentierung eines Bildes durchgeführt:

- Notenlinien finden und daraus Notenzeilen aufbauen
- horizontale Lokalisierung von Symbolen (pro Notenzeile)
- vertikale Unterteilung der Symbole (mehrere Symbole übereinander)
- Segmentgrenzen der Symbole korrigieren

All diesen Schritten ist die Durchführung je einer X- und einer Y-Projektion auf den jeweiligen Bedeutungsbereich gemeinsam. Die Analyse der Projektionen zur Gewinnung der Information ist jedoch unterschiedlich. Zunächst der Code der Methode zum Segmentieren einer Partitur (Score):

```
boolean Score::segment() {

  if ( !getImage() ) return FALSE; // Bild geladen?

  // Grenzen setzen ////////////////////////////////////////
  int w, h;
  getImage()->sizeOfImage( &w, &h );
  left( 0 ); top( 0 ); right( w ); bottom( h );

  // Notenlinien suchen ////////////////////////////////////
  findHorLines();
  if ( !getLines() ) return FALSE;

  // Notenzeilen aufbauen //////////////////////////////////
  if ( staffs.count() )
    staffs.freeContents();   // evt. schon vorhandene
                        // freigeben
  staffs.reset();

  buildStaffs(); // Notenzeilen aufbauen

  // Zeilensegmente setzen /////////////////////////////////
  setStaffBorders();

  // Zeilen nach Symbolen segmentieren /////////////////////
  DO ( staffs, Staff, st )
    st->segment();
  END

  return TRUE;
  } // segment
```

Es werden zuerst die eigenen Grenzen des Segmentes auf die Ausmaße des gesamten Bildes gesetzt. Danach werden die Notenlinien gesucht (*findHorLines*) und anschließend zu Notenzeilen gruppiert (*buildStaffs, setStaffBorders*). Zum Schluß wird noch mit der DO ... END - Konstruktion die Segmentierung jeder einzelnen Notenzeile durchgeführt.
Am folgenden Beispiel wird das Finden von Notenlinien veranschaulicht. Dieses Beispiel wird auch zur Erklärung der restlichen Schritte benutzt und stellt keine wirkliche Komposition dar, sondern lediglich eine Zusammenstellung aller Symbole, die von NOSE erkannt werden. Es handelt sich dabei um Teile von "echten", gescannten Notenvorlagen, die in einem kleinen Bild zusammengestellt wurden:

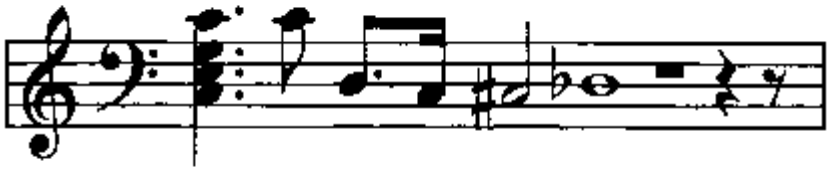

Abbildung 5-3: "konstruiertes" Notenbeispiel

Notenlinien finden:

Zum *Finden der Notenlinien* wird zunächst eine Y-Projektion auf das gesamte Bild durchgeführt, d.h. es wird ein Vektor erzeugt, dessen i-te Komponente die Anzahl der schwarzen Pixel in der i-te Zeile des Bildes enthält (siehe Kapitel 5.4.6).

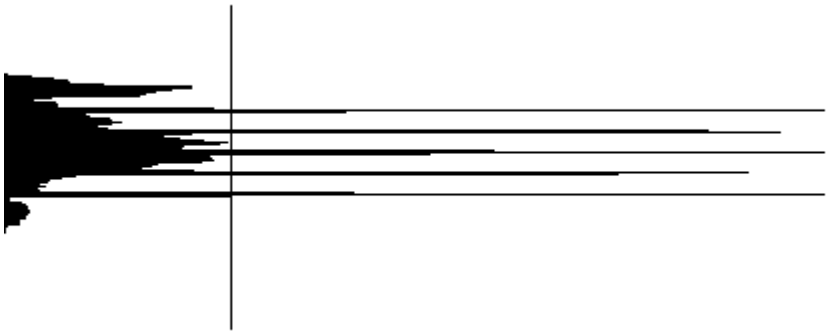

Abbildung 5-4: Y-Projektion auf das Notenbeispiel

Es ist zu erkennen, daß die lokalen Maxima der Projektion die Positionen der Notenlinien anzeigen. Der vertikale Strich in Abbildung 5.4 symbolisiert einen Grenzwert, der nur Maxima mit einem Wert größer dem des Grenzwertes als Notenlinien identifiziert. Die Schwierigkeit besteht natürlich darin, einen geeigneten Grenzwert hierfür zu berechnen. Ist er zu hoch, werden Linien "übersehen", ist er zu niedrig werden Linien sozusagen "erfunden". Beim Aufbau der Notenzeilen ist ein Algorithmus enthalten, der solche überschüssige Notenlinien erkennen und eliminieren kann, indem er die Gleichmäßigkeit der Linienabstände überprüft. Durch zahlreiche Tests an den zur Verfügung stehenden, gescannten Notenvorlagen, hat sich herausgestellt, daß der Grenzwert am günstigsten auf 1/4 der Breite der Partitur gesetzt wird. Diese relativ einfache Methode zum Finden von Notenlinien hat natürlich vor allem dann Probleme, wenn die Vorlagen nicht sorgfältig gerade, sondern leicht rotiert eingescannt wurden. In Abbildung 5-4 sehen wir ein Beispiel hierfür:

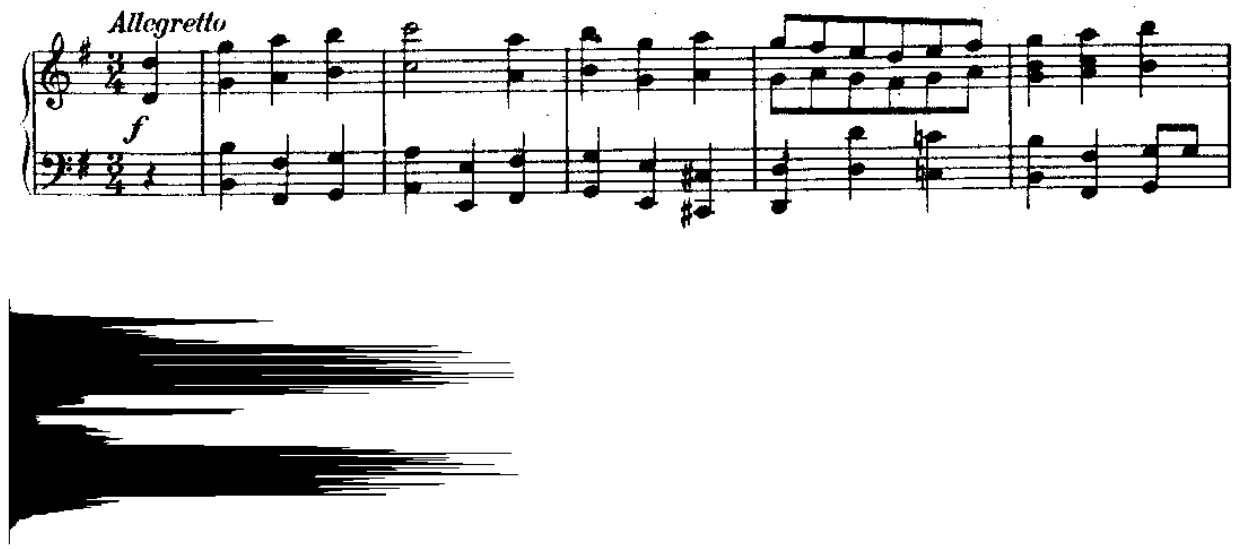


Abbildung 5-5: Beispiel für eine leicht rotierte Notenvorlage

Aus der daraus ermittelten Y-Projektion war es nicht möglich, mit dem vorhandenen Algorithmus die Notenlinien korrekt zu erkennen. Jedoch beweisen die übrigen von mir benutzten Notenvorlagen, bei denen alle samt die Notenlinien richtig erkannt wurden, daß bei genügend großer Sorgfalt beim Scannen diese Methode durchaus geeignet erscheint. Ein weiteres Problem stellen horizontale Balken bzw. eine große Anzahl von Viertelnoten oder Hilfslinien, die sich in etwa auf gleicher Höhe nebeneinander befinden, dar. Es werden dadurch zu viele Notenlinien gefunden, die aber meistens durch den schon angesprochenen Algorithmus zum Eliminieren überschüssiger Linien beseitigt werden können. Überraschend ist jedoch die Tatsache, daß die Methode bei höher werdenden Auflösungen der Scanvorlagen zunehmend versagt. Der Grund liegt daran, daß bei hohen Auflösungen einzelne Notenlinien bereits aus mehreren lokalen Maxima bestehen und die Anzahl der gefundenen Linien dadurch viel zu hoch ist. Akzeptable Auflösungen auch im Hinblick auf die Klassifizierung sind im Bereich 150 - 200% (Programm DeskScan). Nachfolgend nun der Code zum Finden von Notenlinien:

```
VIntVector *Score::findHorLines() {
  unsigned noSystems = 0;
  VRectangle r( bounds() );
  Projection *yp, *ymed;
  int threshold = round( (float)width() / 4.0 );     // 25% der Breite
  VIntVector *lns = 0;

  yp = getImage()->yProjection( r );
  if ( !yp ) return 0;

  // lokale Maxima größer threshold suchen ////////////////////////////
  lns = Segment::findHorLines( yp, threshold );
  setLines( lns );

  // Anzahl der Systeme pro Zeile ermitteln ///////////////////////////
  if ( lns ) {
    threshold = round( aveLineDistance() );
    ymed = yp->medianProjection( threshold );  // Glättung!!!
    if ( ymed ) {      // Anzahl der Systeme bestimmen
      noSystems = 0;
      ymed->setWindow( 0, 0 );
      while ( ymed->nextSeqGreater( 0 ) )
        noSystems++;
      staffsPerSystem = round( (float)lns->size() /
      (float)linesPerStaff / (float)noSystems );
      delete ymed;
      } // if
    } // if
  delete yp;
  return lns;
  } // findHorLines
```

Durch den Aufruf von *Segment::findHorLines()* wird ein Vektor erzeugt, der an der i-ten Stelle den Zeilenindex des i-ten, lokalen Maximums größer *threshold* der Projektion *yp* enthält. Dieser Vektor zeigt also die y-Positionen aller gefunden Notenlinien im untersuchten Bild an. Weiters wird auch die Anzahl der Notenzeilen pro Notensystem (*staffsPerSystem*) ermittelt, wobei zuerst die ursprüngliche Projektion geglättet (siehe Kapitel 5.4.6) und danach aufgrund der Anzahl der von 0 verschiedenen Bereiche der gewünschte Wert ermittelt wird.
Sind die Notenlinien gefunden, werden sie in 5er Gruppen zu Notenzeilen zusammengefaßt. Wie schon erwähnt wird hierbei auf gleichmäßige Abstände und auf evt. zusätzliche Linien geachtet.

```
void Score::buildStaffs() {
  Staff  *st;
  unsigned i, j, found = 0;
  float lineSpace = aveLineDistance();   // geschätzter Abstand
  float maxDiff = lineSpace / 4.0;       // max. Abweichung (25%)
  int dist;                   // tatsächlicher Abstand
  int  minLineSpace = round( lineSpace - maxDiff ),  // minimaler Abstand
       maxLineSpace = round( lineSpace + maxDiff );  // maximaler Abstand
  VIntVector  ln( linesPerStaff );       // zur Zeile gehör. Linien
  VIntVector  *lines = getLines();       // alle Linien der Partitur
  Image     *im = getImage();            // zugehöriges Bild

  if ( !im || !lines ) return; // kein Bild geladen, keine Linien gefunden

  for ( i = 0; i < lines->size(); i++ ) {

    dist = round( lineSpace );

    // Bis Abstand zw. Linie i und j >= Mindestabstand
    for ( j = i + 1; j < lines->size(); j++ ) {
      dist = lines->elem( j ) - lines->elem( i );
      if ( dist >= minLineSpace )
        break;
      } // for

    // Falls Abstand zw. Linie i und j auch <= Maximalabstand
    if ( dist <= maxLineSpace || found == linesPerStaff - 1 ) {
      ln.elem( found++ ) = lines->elem( i );
      i = j - 1;
      } // if
    else
      found = 0;

    // genug Linien gleichen Abstandes gefunden -> Zeile erzeugen
    if ( found == linesPerStaff ) {
      st = new Staff( im, ln );
      if ( !st ) break;
      staffs.add( st );
      found = 0;
      } // if

    } // for i

  } // buildStaffs
```

Die Segmentgrenzen der Notenzeilen werden durch die Methode *setStaffBorders()* gesetzt, wobei die horizontalen Grenzen denen der Partitur entsprechen, die vertikalen Grenzen zweier Notenzeilen in der Mitte zwischen der letzten und der ersten Notenlinie der beteiligten Notenzeilen gesetzt werden.

Horizontales Lokalisieren:

Nun müssen noch zur vollständigen Segmentierung der Partitur die einzelnen Symbole im Bereich der Notenzeilen gefunden werden. Der erste Schritt dazu besteht im *horizontalen Lokalisieren* der Symbolsegmente, indem pro Notenzeile eine X-Projektion (siehe Kapitel 5.4.6) durchgeführt wird und die lokalen Minima dieser Projektion als Symbolgrenzen angesehen werden. Die folgenden Abbildungen zeigen wieder das bereits erwähnte Beispiel, wobei unterschiedliche Glättungen der Projektion (abhängig von der Länge des gleitenden Medianfilters, siehe Kapitel 5.4.6) samt den Auswirkungen auf die Symbolgrenzen veranschaulicht werden.

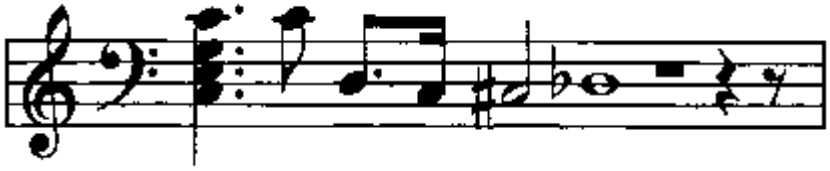

Abb. 5-6: Beispiel

Abb. 5-7: keine Glättung

Abb. 5-8: Median = 3

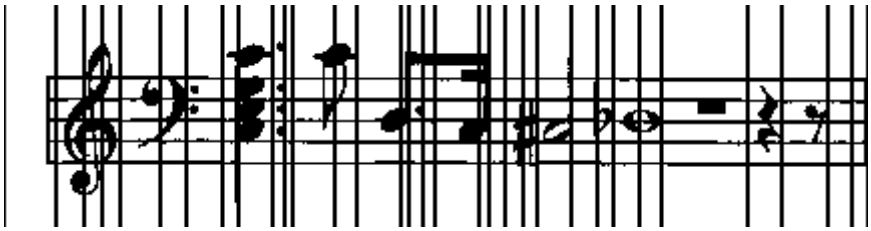

Abb. 5-9: Grenzen

Abb. 5-10: Median = 5

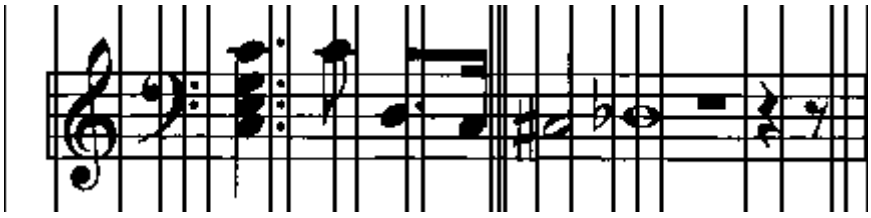

Abb. 5-11: Grenzen

Abb. 5-12: Median = 7

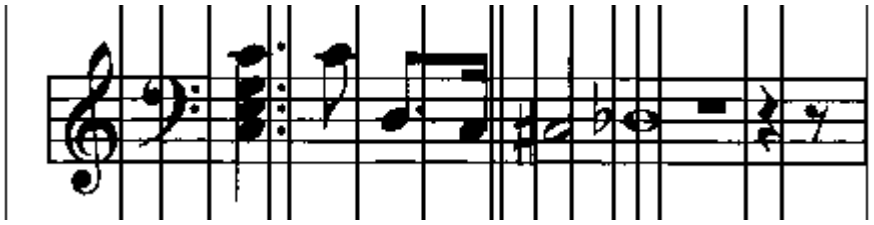

Abb. 5-13: Grenzen

Mit zunehmender Glättung wird die Anzahl der gefundenen Symbolsegmente geringer, die Symbole werden dadurch auch weniger oft geteilt. Bei zu starker Glättung kann man bei dieser Methode jedoch feststellen, daß v.a. die Note am Ende bzw. am Anfang einer Gruppe von Noten, welche mit Balken verbunden sind, regelmäßig übersehen werden. Störend ist auch die Tatsache, daß eine X-Projektion auf den Notenkopf einer ganzen Note sowie auf Viertelnoten mit Fähnchen 2 lokale Maxima (3 lokale Minima) besitzt und diese Symbole somit immer in 2 Segmente geteilt werden. Das lokale Maximum des Notenkopfes einer halben Note unterscheidet sich meist nur sehr geringfügig von den umliegenden lokalen Minima und kann bei zu starker Glättung übersehen werden. Das Problem liegt also im Finden des optimalen Glättungsgrades der X-Projektion. Durch Tests an den Notenvorlagen ([BACH2], [HARM], [PIA1]) ergab sich die günstigste Glättung bei einer Filterlänge, die der Hälfte des durchschnittlichen Notenlinienabstandes entspricht (Verhältnis zwischen Linienabstand und Größe der Notenköpfe!). Im obigen Beispiel sind das die Abbildungen 5-10 und 5-11. Hier der Code zur horizontalen Lokalisierung:

```
void Staff::horLocalize() {
  int  leftMargin,      // linker Symbolrand
       rightMargin;     // rechter Symbolrand
  float aveLineDist = aveLineDistance(); // durchschnittl. Linienabstand
  int lineDist = round( aveLineDist );   // gerundet
  int i, max;
  unsigned medianLen = round( aveLineDist / 2.0 );   // Filterlänge
  Projection  *xp,      // X-Projektion
         *xmed;         // geglättete X-Projektion
  Symbol *s;            // gefundens Symbol
  VRectangle r( bounds() );  // Grenzen der Notenzeile

  // X-Projektion durchführen ////////////////////////////////
  xp = getImage()->xProjection( r );

  // Glättung durchführen ////////////////////////////////////
  if ( !( medianLen % 2 ) ) medianLen++; // ungerade !!!
  xmed = xp->medianProjection( medianLen );
```

```
// Symbolränder ermitteln ////////////////////////////////////
max = xmed->size() - 1;
for ( i = 0; i < max; ) {

  // linken Rand setzen
  leftMargin = i;

  // vor, solange es nicht wieder fällt
  for ( ; i < max && xmed->elem( i ) <= xmed->elem( i+1 ); i++ );
  // vor, solange es nicht wieder steigt
  for ( ; i < max && xmed->elem( i ) >= xmed->elem( i+1 ); i++ );
  // zurück, solange es gleich bleibt
  for ( ; i > leftMargin && xmed->elem( i ) == xmed->elem( i-1 ); i-- );

  // rechten Rand auf i
  rightMargin = i;
  if ( leftMargin == i ) break;

  // Grenzen korrigieren
  leftMargin += left();
  rightMargin += left();

  // Symbol erzeugen
  s = new Symbol( this );
  s->left( leftMargin );
  s->right( rightMargin );
  s->top( top() );
  s->bottom( bottom() );
  symbols.add( s );

  } // for

delete xmed;
delete xp;
} // horLocalize
```

Die so gefundenen Symbole müssen in einem 2. Schritt noch *vertikal unterteilt* werden, da sich mehrere Symbole übereinander befinden können, wie bei Akkorden, bei mehrstimmiger Notation oder bei den "Wie"-Symbolen.

Vertikale Unterteilung:

Jede Notenzeile enthält bereits jetzt eine Liste von Symbolsegmenten, die nun durch Y-Projektionen evt. noch unterteilt werden. Abbildung 5-14 zeigt zunächst das Ergebnis der horizontalen Lokalisierung und Abbildung 5-15 auszugsweise die geglätteten Y-Projektionen einiger Symbole.

Abbildung 5-14: Ergebnis der horizontalen Lokalisierung

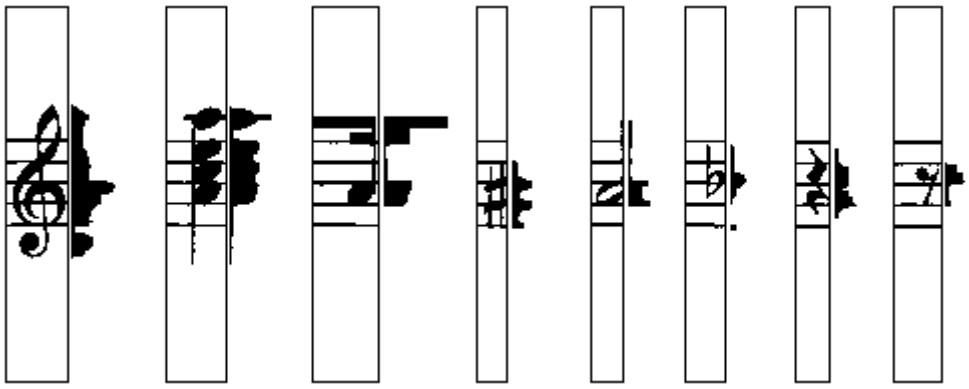

Abbildung 5-15: geglättete Y-Projektionen einiger Symbole

Der Effekt der Glättung ist in diesen Beispielen besonders gut zu sehen, weil dadurch die Notenlinien, die als einzelne "Ausreißer" betrachtet werden, aus der Projektion verschwinden.

Das Verfahren zur *vertikalen Unterteilung* der Symbolsegmente ist ähnlich der zuvor beschriebenen horizontalen Unterteilung einer Notenzeile. Im Bereich der lokalen Minima der geglätteten Y-Projektion auf ein Symbol werden die Grenzen für die entstehenden neuen Symbole gesetzt. Einziger momentaner Nutzen ist die Aufteilung von Akkorden (siehe Abbildung 5-15, 2. Symbol) in mehrere Segmente, eines für jede Note. Unerwünschter Nebeneffekt dabei ist aber, daß beinahe jedes Symbol dadurch aufgeteilt wird und so sehr viele Segmente entstehen, die durch die große Anzahl natürlich den Vorgang der Klassifizierung stark verlangsamen. Würde man diesen Schritt der vertikalen Unterteilung jedoch unterlassen, so könnten in einer späteren Erweiterung des Programms keine "Wie"-Symbole unterstützt werden. In Abbildung 5-

16 wird zuerst das Ergebnis der vertikalen Unterteilung unseres Beispiels gezeigt, danach folgt das Listing des Codes der Methode *vertSplit()*.

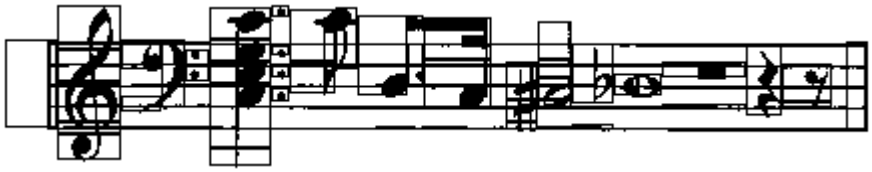

Abbildung 5-16: Ergebnis der vertikalen Unterteilung

```
void Staff::vertSplit() {
  float aveLineDist = aveLineDistance();
  int i, j;
  int  topMargin,       // neuer oberer Rand eines Symbols
       bottomMargin;    // neuer unterer Rand eines Symbols
  Symbol *s,            // altes Symbol
         *newS;         // neues Symbol
  VRectangle r;         // die Grenzen eines Symbols
  Projection  *yp,      // Y-Projektion auf ein Symbol
              *med;     // geglättete Y-Projektion
  unsigned medianLen;   // Maß für die Glättung
  unsigned pos;         // aktueller Index für Projektion

  medianlen = round( aveLineDist / 2.0 );
  if ( !( medianLen % 2 ) ) medianLen++; // ungerade

  // alle Symbole untersuchen ////////////////////////////////////
  for ( i = 0; i < symbols.count(); ) {
    j = i;    // Index des alten Symbols merken
    s = (Symbol*)symbols.idAt( i );

    // Projektion und Glättung durchführen
    r.set( Corners, s->left(), s->top(), s->right(), s->bottom() );
    yp = getImage()->yProjection( r );
    med = yp->medianProjection( medianLen );

    // lokale Minima suchen
    pos = 0;
    while ( 1 ) {
      // vor, solange es gleich bleibt
      for ( ; pos < med->size()-1 && med->elem(pos) == med->elem( pos+1 );
              pos++ );
      topMargin = pos;
      if ( !med->nextLocalMinimum( pos ) ) break;
      // zurück, solange es 0 bleibt !!!
      for ( bottomMargin = pos; bottomMargin > topMargin &&
           med->elem( bottomMargin ) == 0; bottomMargin-- );
      topMargin += top(); bottomMargin += top();
      // Symbol mit neuer oberer und unterer Grenze erzeugen
      newS = new Symbol( s );
      newS->top( topMargin ); newS->bottom( bottomMargin );
      symbols.insertAt( ++i, newS );
      } // while

    // altes Symbol löschen
    symbols.removeAt( j );
```

```
    delete s;

    delete med;
    delete yp;

    } // for

  } // vertSplit
```

Segmentgrenzen korrigieren:

Als letzter Schritt der Segmentierung werden für jedes Symbol noch einmal die Segmentgrenzen korrigiert, indem durch je eine X- und eine Y-Projektion die "leeren" Stellen (jene Stellen, an denen sich nur Notenlinien befinden) herausgefiltert werden. Hier der Code für diese Operation:

```
boolean Symbol::segment() {
  // falls Symbol keine Ausdehnung mehr hat wird es später gelöscht
  if ( !width() || !height() ) return FALSE;
  float aveLineDist = aveLineDistance();
  VRectangle r( bounds() );  // Grenzen des Symbols
  int threshold;
  unsigned medianLen;
  boolean ok = TRUE;    // FALSE, falls kein Symbol mehr vorhanden
  Projection *p, *m;    // Projektion und geglättete Projektion


  // vertikale Begrenzung ////////////////////////////////////
  p = getImage()->yProjection( r );

  medianLen = round( aveLineDist / 2.0 );
  if ( !( medianLen % 2 ) ) medianLen++; // ungerade!!!
  m = p->medianProjection( medianLen );  // Notenlinien filtern!

  threshold = round( aveLineDist / 5.0 );// ca. Liniendicke
  m->longestSeqGreater( threshold );
  if ( m->windowLen() ) {
    top( m->windowStart() + top() );
    bottom( top() + m->windowLen() - 1 );
    } // if
  else
    ok = FALSE;
  delete m;
  delete p;

  if ( !ok ) return FALSE;
```

```
// horizontale Begrenzung ////////////////////////////////////
p = getImage()->xProjection( r );

threshold = round( p->average() );// Durchschnitt der Projektion geeignet
p->longestSeqGreater( threshold );
if ( p->windowLen() ) {
   left( p->windowStart() + left() );
   right( left() + p->windowLen() - 1 );
   } // if
else
   ok = FALSE;
delete p;

return ok;
} // segment
```

Abschließend noch einige bewertende Worte über die soeben beschriebene Methode der fortgesetzten X-/Y-Projektionen zur Segmentierung von Notenschriftbildern. Diese Methode verfolgt das Ziel, jene Stellen des Bildes zu suchen, an denen mehr Information wie an anderen Stellen vorhanden ist. Durch das Suchen von Minima bzw. Maxima in den erzeugten Projektionen können Linien, Symbolgrenzen und Leerstellen gefunden werden und zwar relativ unbeeinflußt von Störungen, wie sie für die Notenschrift typisch sind (man denke an Notenlinien oder an große, segmentübergreifende Symbole wie Bindebögen und Balken). Die wirkungsvolle und dafür nötige Glättung der Projektionen wurde durch eine Medianfilterung (siehe [DIBI] und Kapitel 5.4.6) erreicht. Nachteil dieser Methode ist ihre Tendenz zur Zersplitterung der Symbole, was zu einer großen Menge an Segmenten führt, die natürlich den im nächsten Kapitel beschriebenen Vorgang der Klassifizierung entscheidend verlangsamt und häufige Doppelklassifikationen mit sich bringt. Ebenso nachteilig wirkt die Tatsache, daß die Symbolgrenzen nur relativ ungenau bestimmt werden können, was eine Grobklassifikation der Symbole aufgrund ihrer Größe und Ausdehnung, wie ich es ursprünglich geplant hatte, unmöglich macht.

## 5.3.2. Das Klassifizieren

Im Kapitel 2 wurde ein Überblick über die Symbole der Notenschrift gegeben. Ziel für diese Diplomarbeit war es ein Programm zu schreiben, das all jene Symbole erkennt, die darüber Auskunft geben, *was* gespielt werden soll ("Was"-Symbole). Zu diesem Zweck wurden 3 Symbolklassen eingeführt, die konkrete Ausprägungen der unklassifizierten Symbole darstellen. Jede dieser Klassen besitzt auch eine Typkomponente, welche die unterschiedlichen Typen ein und der selben Symbolklasse ermöglicht. Die folgende Tabelle ist eine Zusammenstellung dieser Klassen samt ihren Typen:

| deutsch. Name | engl. Name | Typen | siehe Kapitel |
|---|---|---|---|
| Note | Note | 1/4-, 1/2- und 1/1-Notenköpfe | 5.4.11 |
| Pause | Rest | 1/8- , 1/4-, 1/2- und 1/1-Pausen | 5.4.12 |
| Notenschlüssel | Clef | G- und F-Notenschlüssel (oder Violin- und Baßschlüssel) | 5.4.13 |

Die Symbolklasse *Note* ist ein aus Musterprimitiven zusammengesetztes Symbol (siehe Kapitel 3.2.2) und besitzt zusätzlich zum Typ des Notenkopfes noch 3 weitere Bestimmungsstücke:

- Vorzeichen (Kreuz, B oder keines),
- Verlängerungspunkt (ja/nein) und
- Anzahl der Fähnchen oder Balken am Notenhals.

Die Klassifikation der Symbole erfolgt durch Schablonenvergleich (siehe Kapitel 3.2.3). Es existieren dabei die folgenden 11 Schablonen (Vergleichsvorlagen, Muster):

- der 1/4-Notenkopf ( )
- der 1/2-Notenkopf ( )
- der 1/1-Notenkopf ( )
- das Vorzeichen Kreuz ()
- das Vorzeichen B ( )
- die 1/8-Pause ( )
- die 1/4-Pause ( )
- die 1/2-Pause ()
- die 1/1-Pause ( )
- der Violinschlüssel ( )
- der Baßschlüssel ( )

Es ist zu bemerken, daß einige Muster nur einen Teil des tatsächlichen Symbols abdecken. Der Grund ist darin zu suchen, daß verschiedene Notenschrifteditionen jene Symbole oft recht unterschiedlich in Schräge und Breite darstellen. Es wird daher nur

der relevante Teil des Symbols mit dem Original verglichen, was auch bei sehr großen Symbolen wie dem Violinschlüssel einen leichten Performancebonus bedeutet. Die Muster wurden zum einen Teil aus gescannten Notenblättern, zum anderen aus dem Windows True-Type-Font *MusicalSymbols* zusammengestellt und befinden sich als Windows-Bitmaps (siehe Kapitel 5.2.1) im Resourceteil des Programms NOSE.
Der erste Schritt der Klassifikation besteht also im Laden der Vergleichsmuster mit anschließender Größenkorrektur. Dabei wird angenommen, daß der aus den gefundenen Notenlinien errechnete, durchschnittliche Linienabstand (siehe Kapitel 5.4.7) der Höhe eines 1/4-Notenkopfes entspricht und daher alle Muster aufgrund dieser Tatsache in ihrer Größe angepaßt werden können. Daran anschließend werden Zeile für Zeile die Symbole durch die konkreten Ausprägungen ersetzt. Möglichen Doppelklassifikationen durch ungenaue Segmentgrenzen (siehe Kapitel 5.3.1) werden durch die Methode *deleteTwins* eliminiert. Am Schluß der Klassifikation steht die Ermittlung der Bestimmungsmerkmale bei Noten. Hier der Code für diese Operationen:

```
void Score::classify() {

  // Muster laden und in Größe anpassen ///////////////////////
  loadPatterns();

  // Symbole klassifizieren ///////////////////////////////////
  DO ( staffs, Staff, st )
    st->classify();
  END

  } // classify


void Staff::classify() {
  int    i;      // Laufvariable über alle Symbole der Zeile
  Symbol   *s;  // aktuelles Symbol
  Note   *n;     // falls Note erkannt
  HeadType ht;  // Typ des Notenkopfes
  Rest   *r;     // Falls Pause erkannt
  RestType rt;  // Typ der Pause
  Clef   *c;     // Falls Notenschlüssel erkannt
  ClefType ct;  // Typ des Notenschlüssels
  boolean  foundClef = FALSE;      // am Beginn einer Notenzeile muß ein
                                   // Notenschlüssel stehen !!!

  // Symbole durch klassifizierte Symbole ersetzten
  for ( i = 0; i < symbols.count(); ) {
    s = (Symbol*)symbols.idAt( i );

    if ( !foundClef && s->isClef( ct ) ) {     // Notenschlüssel
      c = new Clef( s, ct );
      symbols.replaceAt( i++, c );
      foundClef = TRUE;
      } // if

    else if ( foundClef && s->isNote( ht ) ) { // Note
```

```
      n = new Note( s, ht );
      symbols.replaceAt( i++, n );
      } // if

    else if ( foundClef && s->isRest( rt ) ) { // Pause
      Rest *r = new Rest( s, rt );
      symbols.replaceAt( i++, r );
      } // if

    else   // Symbol löschen
      symbols.removeAt( i );

    delete s;
    } // for

  // Doppelklassifikationen entfernen
  deleteTwins();

  // Bestimmungsstücke für Noten ermitteln
  for ( i = 0; i < symbols.count(); i++ ) {
    s = (Symbol*)symbols.idAt( i );

    if ( s->iam() == NoteCls ) {
      n = (Note*)s;
      if ( n->head() == HT_QUARTER )     // Anzahl Fähnchen/Balken
        n->setFlags();
          n->setAccidentals();                // Vorzeichen
          if ( n->head() != HT_WHOLE )
        n->setDurPoint();                // Verlängerungspunkt
      } // if

    } // for

  } // classify
```

Die Methoden *isClef*, *isNote* und *isRest* liefern den Wert *TRUE*, falls an der Stelle im Bild das entsprechende Symbol tatsächlich erkannt werden konnte. Zu bemerken ist, daß am Beginn einer Notenzeile ein Notenschlüssel stehen muß (*foundClef*), damit überhaupt nach anderen Symbolen gesucht wird. Diese Maßnahme soll die Anzahl der Schablonenvergleiche und somit auch die benötigte Zeit dafür reduzieren. Sie stellt auch keine Einschränkung der Notenschrift dar, weil hier natürlich ebenso jede Notenzeile einen Notenschlüssel an ihrem Beginn haben muß. Diese 3 Methoden führen also wie bereits erwähnt den Schablonenvergleich durch, indem sie die Methode *compare* unter Angabe eines Musters und einer Prozentzahl für die nötige Übereinstimmung aufrufen.

```
boolean Symbol::isClef( ClefType &ct ) {
  Image *c;

  c = (Image*)clefs->idAt( 0 );    // Violinschlüssel
  if ( compare( c, 70 ) ) {
    ct = CT_G;
    return TRUE;
    } // if

  c = (Image*)clefs->idAt( 1 );    // Baßschlüssel
  if ( compare( c, 75 ) ) {
    ct = CT_F;
    return TRUE;
    } // if

  return FALSE;
  } // isClef


boolean Symbol::isNote( HeadType &ht ) {
  Image *head;

  head = (Image*)noteHeads->idAt( 0 );   // 1/4 Note
  if ( compare( head, 82 ) ) {
    ht = HT_QUARTER;
    return TRUE;
    } // if

  head = (Image*)noteHeads->idAt( 2 );   // 1/1 Note
  if ( compare( head, 80 ) ) {
    ht = HT_WHOLE;
    return TRUE;
    } // if

  head = (Image*)noteHeads->idAt( 1 );   // 1/2 Note
  if ( compare( head, 70 ) ) {           // 70% ist besser
    ht = HT_HALF;
    return TRUE;
    } // if

  return FALSE;
  } // isNote


boolean Symbol::isRest( RestType &rt ) {
  Image *r;

  r = (Image*)rests->idAt( 3 );    // 1/1 Pause
  if ( compare( r, 90 ) ) {
    rt = RT_WHOLE;
    return TRUE;
    } // if

  r = (Image*)rests->idAt( 2 );    // 1/2 Pause
  if ( compare( r, 90 ) ) {
    rt = RT_HALF;
    return TRUE;
    } // if

  r = (Image*)rests->idAt( 1 );    // 1/4 Pause
```

```
  if ( compare( r, 80 ) ) {
    rt = RT_QUARTER;
    return TRUE;
    } // if

  r = (Image*)rests->idAt( 0 );     // 1/8 Pause
  if ( compare( r, 80 ) ) {
    rt = RT_EIGHTH;
    return TRUE;
    } // if

  return FALSE;
  } // isRest
```

Die günstigsten Werte für die prozentuale Übereinstimmung des Musters mit dem Original variieren und mußten einzeln durch eine Reihe von Tests bestimmt werden. Als zukünftige Erweiterung des Programms wäre denkbar, diese Prozentangaben durch eine Dialogbox vom Benutzer individuell für verschiedene Notenschrifteditionen ändern zu lassen. Die Methode *compare* hat folgenden Source-Code:

```
boolean Symbol::compare( Image *pattern, float threshold ) {
  int x, y, bestX, bestY;
  int pw, ph, wDiff, hDiff;
  int fromX, toX, fromY, toY;
  float result, best = 0;

  pattern->sizeOfImage( &pw, &ph );

  wDiff = pw - width();       // unterschiedliche Breite
  hDiff = ph - height();      // unterschiedliche Höhe

  fromX = wDiff < 0 ? left() : left() - wDiff;
  toX = wDiff < 0 ? left() - wDiff : left();

  fromY = hDiff < 0 ? top() : top() - hDiff;
  toY = hDiff < 0 ? top() - hDiff : top();

  for ( x = fromX; x <= toX+1; x++ ) {
    for ( y = fromY; y <= toY+1; y++ ) {
      result = getImage()->percentEqual( x, y, pattern );
      if ( result > best ) {
        best = result;
        bestX = x; bestY = y;
        } // if
      } // for
    } // for

  // bei Übereinstimmung Grenzen des Symbols korrigieren
  if ( best >= threshold ) {
    left( bestX ); right( bestX+pw ); top( bestY ); bottom( bestY+ph );
    return TRUE;
    } // if
  return FALSE;
  } // compare
```

Es werden mehrereVergleiche zwischen dem größennormierten Muster (*pattern*) und der Notenvorlage (*getImage()*) durchgeführt, wobei das beste Resultat mit der nötigen, prozentualen Übereinstimmung genommen wird.
Wie bereits erwähnt müssen für die Symbolklasse *Note* noch weitere Bestimmungsstücke ermittelt werden. Dies geschieht durch den Aufruf der Methoden *setFlags* (Anzahl der Fähnchen/Balken), *setAccidentals* (Vorzeichen) und *setDurPoint* (Verlängerungspunkt).

Anzahl der Fähnchen oder Balken:
Fähnchen bzw. Balken können die Dauer (den Notenwert) einer 1/4-Note verändern. Je nach Anzahl erzeugen sie 1/8-, 1/16-, 1/32-Noten usw.. Sie befinden sich am Ende des Notenhalses, der sich sowohl links oder rechts von der Note befindet, als auch nach oben oder unten zeigen kann. Erste Aufgabe besteht daher im Suchen des Notenhalses und wird durch das Messen von Lauflängen schwarzer Pixel durchgeführt (siehe Kapitel 5.4.4). Danach wird mittels einer Y-Projektion auf ein den Notenhals umschließendes Rechteck die Fähnchen bzw. Balken der Note gezählt.

```
int Note::setFlags() {
  float aveLineDist = aveLineDistance();
  int maxGap = round( aveLineDist / 3.0 );
  int runLength,
    runLengthRight,
    runLengthLeft;
  int x, y, w, h;
  int rightX, leftX;
  Image *im = getImage();
  boolean end = FALSE;
  HDC hDC = im->getHndl();
  VRectangle r;

  y = top() + height() / 2;  // vertikale Mitte der Note

  rightX = right();                // Lauflänge am rechten Rand
  while ( !im->black( hDC, rightX, y ) && --rightX > left() );
  runLengthRight = im->vertRunLength( rightX, y, maxGap );

  leftX = left();                  // Lauflänge am linken Rand
  while ( !im->black( hDC, leftX, y ) && ++leftX > right() );
  runLengthLeft = im->vertRunLength( leftX, y, maxGap );

  im->ungetHndl();

  runLength = abs( runLengthRight ) > abs( runLengthLeft ) ?
         runLengthRight : runLengthLeft;

  // falls kein Notenhals vorhanden
  if ( abs( runLength ) < height() ) return 0;

  // den Notenhals umschließendes Rechteck erzeugen
  w = round( aveLineDist );  // Linienabstand
  h = 3 * w;                 // ~ 4 Balken möglich
```

```
  if ( abs( runLengthRight ) > abs( runLengthLeft ) )
    x = rightX - w/2;              // Notenhals rechts der Note
  else
    x = leftX - w/2;               // Notenhals links der Note
  if ( runLength < 0 )
    y += runLength;                      // Notenhals nach oben
  else
    y = y + runLength - h + 1;     // Notenhals nach unten

  r.set( CornerDim, x, y, w, h );

  return noFlags = countFlags( r );
  } // setFlags


int Note::countFlags( VRectangle &r ) {
  int cnt = 0;
  float space = aveLineDistance();
  float halfSpace = space / 2.0;
  int minWidth = round( space / 5.0 );   // ~Liniendicke
  Projection *yp = getImage()->yProjection( r );

  yp->setWindow( 0, 0 );
  while ( yp->nextSeqGreater( minWidth ) )
    if ( (float)yp->windowLen() >= halfSpace )
      cnt++;

  delete yp;
  return cnt;
  } // countFlags
```

Vorzeichen:

In einem bestimmten Bereich links vom Notenkopf der Note kann sich ein Vorzeichen befinden. Dies wird durch Schablonenvergleiche mit den 2 Mustern für die Vorzeichen *Kreuz* und *B* festgestellt.

```
AccType Note::setAccidentals() {
  int lineDist = round( aveLineDistance() );
  int x, y;
  int fromX, toX;
  int patW, patH;
  Image *pattern;
  float result, best = 0;

  pattern = (Image*)accidentals->idAt( 0 );    // Kreuz
  pattern->sizeOfImage( &patW, &patH );

  toX = left() - patW;
  fromX = toX - patW;
  y = round( (float)top() - (float)( patH - height() ) / 2.0 );

  best = 0;
  for ( x = fromX; x < toX; x++ ) {
    result = getImage()->percentEqual( x, y, pattern );
    if ( result > best )
      best = result;
```

```
    } // for
  if ( best > 65 )
    return accType = AT_SHARP;

  pattern = (Image*)accidentals->idAt( 1 );    // B
  pattern->sizeOfImage( &patW, &patH );

  toX = left() - patW;
  fromX = toX - patW;
  y = round( (float)top() - (float)( patH - height() ) / 2.0 );

  best = 0;
  for ( x = fromX; x < toX; x++ ) {
    result = getImage()->percentEqual( x, y, pattern );
    if ( result > best )
      best = result;
    } // for
  if ( best > 70 )
    return accType = AT_FLAT;

  return accType = AT_NONE;
  } // setAccidentals
```

Verlängerungspunkt:

Abschließend wird wiederum durch je eine X- und eine Y-Projektion ein bestimmtes Rechteck rechts vom Notenkopf untersucht, ob sich in ihm ein Verlängerungspunkt befindet. Es muß dabei darauf geachtet werden, daß nicht Hilfslinien oder Teile anderer Symbole fälschlicherweise für Verlängerungspunkte gehalten werden. Weiters ist zur Zeit noch nicht vorgesehen, daß eine Note mehrere Verlängerungspunkte besitzen könnte, was bei "schwierigeren" Noteneditionen durchaus vorkommen kann. Hier der Code zum Finden des Verlängerungspunktes:

```
boolean Note::setDurPoint() {
  int fromX, fromY, toX, toY;
  float lineDist = aveLineDistance();
  int w = round( lineDist );
  int threshold = round( lineDist / 5.0 );     // ~ Liniendicke
  VRectangle r;
  Projection *proj;

  fromX = right() + w/3;
  fromY = top() - w/3;
  toX = fromX + w;
  toY = bottom();

  r.set( Corners, fromX, fromY, toX, toY );
  proj = getImage()->yProjection( r );
  proj->longestSeqGreater( threshold );

  durPoint = proj->windowLen() > threshold && // keine Notenlinien
          proj->windowLen() <= 2 * threshold;  // keine Vorzeichen rechts

  fromY = fromY + proj->windowStart();
  toY = fromY + proj->windowLen() - 1;
```

```
delete proj;
if ( !durPoint ) return FALSE;

r.set( Corners, fromX, fromY, toX, toY );
proj = getImage()->xProjection( r );
proj->longestSeqGreater( 0 );

durPoint = proj->windowStart() > 0 &&  // Hilfslinie der eigenen Note
       proj->windowStart() + proj->windowLen() < proj->size();
                                  // Hilfslinie der nächsten Note

delete proj;

return durPoint;
} // setDurPoint
```

Die Klassifikation der Symbole ist somit abgeschlossen. Das große Problem bei der verwendeten Methode des Schablonenvergleichs ist der Zeitfaktor, vor allem dann, wenn bei der Segmentierung schwieriger Stücke sehr viele Segmente erzeugt werden. Ebenfalls aus Gründen der Performance können nicht mehrere Muster für einen Typ einer Symbolklasse angegeben werden, was für einige Noteneditionen nötig ist, die manche Symbole schräger oder breiter als "üblich" darstellen. Zu den Vorzeichen einer Note ist zu sagen, daß wiederum aus Zeitgründen auf Doppelkreuz bzw. Doppel-B verzichtet wurde, was prinzipiell nur 2 Schablonenvergleiche weniger bedeutet. Ebenso wird das Auflösungszeichen vom Erkennungswerkzeug ignoriert, da angenommen wird, daß jede Note eines Taktes, die ein Vorzeichen braucht, auch eines besitzt, obwohl die Notation des Vorzeichens für eine Notenlinie nur für die erste Note im Takt erfolgen muß. Dies führt natürlich zwangsweise zu einigen Fehlern bei der Interpretation der erkannten Symbole. Eine relativ große Einschränkung stellt die Tatsache dar, daß nicht geprüft wird, welche Tonart das zu erkennende Stück hat, sie ist auch nicht "händisch" vom Benutzer vor der Interpretation der Symbole einstellbar, was wiederum zu einigen Fehlinterpretationen führen kann. Das Fehlen der Erkennung von Taktsymbolen ist, wie bereits im Kapitel 2.2 erklärt, kein tatsächlicher Nachteil, da der Takt eines Stückes in Anwendungsprogrammen, wie z.B. Sequencern, leicht eingestellt werden kann.

### 5.3.3. Das Interpretieren

Wie bereits erwähnt ist der Schritt des Interpretierens auf die herkömmliche Schrifterkennung (OCR) nicht unbedingt adäquat anzuwenden. Bei der Notenschrifterkennung ist er jedoch notwendig, weil die meisten Symbole abhängig von ihrer vertikalen und horizontalen Position in der Notenzeile sowie vom Kontext ihre Bedeutung ändern. Beispielsweise stellt eine 1/4-Note auf der untersten Notenlinie ein E' (eingestrichenes E) dar, befindet sich das gleiche Symbol mit identischem Aussehen jedoch auf der 2. Notenlinie von unten, so ändert sich die Tonhöhe und es wird daraus ein G' (eingestrichenes G), vorausgesetzt der Violinschlüssel ist der aktuelle Notenschlüssel der Notenzeile. Falls sich die Position wiederum des gleichen Symbols horizontal unterscheidet, so verändert sich der Zeitpunkt, wann die Note gespielt werden soll. Unter Kontextabhängigkeit meine ich z.B. den aktuellen Notenschlüssel, die aktuelle Tonart aber auch Symbole wie Tempo- oder Lautstärkenänderungen etc., obgleich diese von NOSE nicht unterstützt werden. Die Abkoppelung dieses Schrittes des Erkennungsprozesses von der Klassifikation ist auch deshalb günstig, weil dadurch die Erweiterung auf mehrstimmige Interpretation leichter durchgeführt werden kann. In der Notenschrift werden oft mehrere Stimmen in einer Notenzeile notiert, die Unterscheidung der einzelnen Stimmen erfolgt durch die Richtung der Notenhälse, durch die graphische Anordnung der Symbole und durch Gruppierungen mittels Bindebögen und Balken. In dieser Arbeit erfolgt die Interpretation der Symbole jedoch prinzipiell einstimmig, was konkret bedeutet, daß übereinanderliegende Noten an der gleichen horizontalen Position einen Akkord darstellen und nicht Einzelstimmen einer mehrstimmigen Notation sind. Sie müssen daher auch alle den gleichen Notenwert (Dauer) besitzen, ist dies nicht der Fall, so wird der Notenwert der längsten, beteiligten Note genommen. Die Aufgaben der Interpretation bestehen also darin, den Zeitpunkt des Erklingens, den Zeitpunkt des Verstummens und die Tonhöhe für jede erkannte Note zu ermitteln, das Ergebnis wird dabei in einer MIDI-Datei abgelegt.

```
void Score::interpret( MidiFile &mf ) {
  int trackCnt, // aktuelle Spurnummer
    staffCnt; // aktuelle Nummer der Notenzeile

  // Es existieren in der MIDI-Datei 'staffsPerSystem' + 1 Spuren,
  // da die Spur 0 standardmäßig als sog. Masterspur betrachtet wird.

  // für alle Spuren
  for ( trackCnt = 0; trackCnt < staffsPerSystem; trackCnt++ ) {

    // Interpretation der zur Spur gehörigen Notenzeilen
    for ( staffCnt = trackCnt; staffCnt < staffs.count();
          staffCnt += staffsPerSystem ) {
```

```
      Staff *st = (Staff*)staffs.idAt( staffCnt );
      st->interpret( mf.getTicks(), mf.track( trackCnt + 1 ) );

      } // for

    } // for

  // Ereignis für Spurende eintragen
  for ( trackCnt = 0; trackCnt < staffsPerSystem + 1; trackCnt++ ) {
    mf.track( trackCnt )->putDeltaTime( 0 );
    mf.track( trackCnt )->putEOT();
    }

  // phys. Datei erzeugen
  mf.writeAll();
  } // interpret
```

Hier der Code für die Interpretation einer Notenzeile:

```
void Staff::interpret( int ticksPerQuarterNote, MidiTrack *mt ) {
  int syCnt, i;
  Symbol *sy;
  Note   *n;
  Rest   *r;
  VOrdCollect chord;
  long deltaTicks = 0;
  Clef *gClef = new Clef();  // Violinschlüssel
  Clef *actClef = gClef;
  if ( !gClef ) return;

  for ( syCnt = 0; syCnt < symbols.count(); ) {

    sy = (Symbol*)symbols.idAt( syCnt );

    if ( sy->iam() == RestCls ) {
      r = (Rest*)sy;
      deltaTicks += r->noTicks( ticksPerQuarterNote );
      syCnt++;
      } // if

    else if ( sy->iam() == ClefCls ) {
      actClef = (Clef*)sy;
      syCnt++;
      } // if

    else if ( sy->iam() == NoteCls ) {

      // Akkord aufbauen ////////////////////////////////////////
      n = (Note*)sy;
      chord.reset();
      chord.add( n );

      for ( syCnt++; syCnt < symbols.count(); syCnt++ ) {

        sy = (Symbol*)symbols.idAt( syCnt );

        if ( n->inChord( sy ) ) {
          if ( sy->iam() == NoteCls )
```

```
            chord.add( (Note*)sy );
          } // if
        else
          break;

        } // for

      // Note On/Off schreiben ////////////////////////////////////
      long maxTicks = 0, ticks;

      mt->putDeltaTime( deltaTicks );
      deltaTicks = 0;

      for ( i = 0; i < chord.count(); i++ ) {
        n = (Note*)chord.idAt( i );
        mt->putNoteOn( n->keyNr( actClef ), 100 );
        ticks = n->noTicks( ticksPerQuarterNote );
        if ( ticks > maxTicks ) maxTicks = ticks;
        if ( i < chord.count() - 1 )
          mt->putDeltaTime( 0 );
        else
          mt->putDeltaTime( maxTicks );
        } // for

         i = 0;
      DO ( chord, Note, n )
        mt->putNoteOff( n->keyNr( actClef ), 100 );
        if ( i++ < chord.count() - 1 )
          mt->putDeltaTime( 0 );
      END

      } // if
    else
      syCnt++;

    } // for

  delete gClef;
  } // interpret
```

Die 3 für die Interpretation wichtigen Methoden eines Symbols heißen *noTicks*, welche die zeitliche Dauer eines Symbols in Ticks (siehe Kapitel 5.2.2) berechnet, *inChord*, welche den Wert TRUE liefert, falls sich das betrachtete Symbol im gleichen Akkord befindet und *keyNr* (für Noten), welche abhängig von einem Notenschlüssel die MIDI-Tastennummer (siehe Kapitel 5.1) ermittelt. Bei der letzteren Methode zeigten sich manchmal Probleme, da beim Scannen v.a. am Seitenrand oft Verzerrungen entstehen, wodurch natürlich auch die Notenlinien in diesem Bereich verzerrt sind und dadurch die Tonhöhe nicht korrekt ermittelt werden kann. Dieser Mangel könnte behoben werden, da jedes Symbol eine eigene Liste für die vertikalen Positionen der Notenlinien führt und es dadurch möglich ist, diese pro Symbol noch einmal exakter zu ermitteln.

## 5.4. Beschreibung der Klassen von NOSE

### 5.4.1. NoseModel

*Oberklassen:*

**VObject**

*Unterklassen:*

**Keine**

*benutzte Klassen:*

**Image, Score**

*Header-Datei:*

**nosemodl.h**

*Beschreibung:*

Die Klasse **NoseModel** repräsentiert das Modell der Applikation im MVC-Paradigma. Sie enthält das zu bearbeitende Bild (**theImage**), welches die reine Pixelinformation darstellt und das bearbeitete, "verstandene" Bild (**theScore**, die Partitur), welches die Ergebnisse des Erkennungsprozesses beinhaltet.

***Komponenten:***

**Image *theImage;**

Zeiger auf das Bild (reine Pixelinformation).

**Score *theScore;**

Zeiger auf die Partitur, das "verstandene" Bild mit den Ergebnissen des Erkennungsprozesses.

***Methoden:***

**Konstruktor**

**NoseModel()**
erzeugt das Modell der Applikation NOSE. **theImage** und **theScore** sind zunächst NULL.

**~NoseModel()**
gibt den Speicher für **theImage** und **theScore** wieder frei.

**Zugriff auf das Bild**

**Image *getImage()**
liefert einen Zeiger auf das Bild.

**Image *loadImage( VPathString &fname )**
wählt aufgrund der Dateierweiterung von **fname** die geeignete Methode zum Laden des gewünschten Bildes aus (zur Zeit nur Windows-Bitmaps *.BMP)

**Zugriff auf die Partitur**

**Score *getScore()**
liefert einen Zeiger auf die Partitur.

**Segmentierung**

**int segment()**
segmentiert das Bild **theImage**. Als Ergebnis wird eine Partitur **theScore** erzeugt und mit Informationen gefüllt.

**Klassifizierung**

**void classify()**
klassifiziert die Symbole von **theScore**.

### Interpretation

**boolean interpret( VPathString &fname )**

interpretiert die Symbole von **theScore** und speichert sie als MIDI-Datei unter dem Namen **fname** mit der Erweiterung MID ab.

### Segmente laden/speichern

**boolean loadSegments( VPathString &fname )**

lädt, falls zur aktuellen Datei bereits eine Partitur (**Score**) bestanden hat diese in die Komponente **theScore**.

**void saveSegments( VPathString &fname )**

speichert **theScore** als persistentes Objekt, kann mittels der Methode **loadSegments** wieder geladen werden.

## 5.4.2. NoseView

*Oberklassen:*

**VAppView**

*Unterklassen:*

**Keine**

*benutzte Klassen:*

**VPathString, ImageView**

*Header-Datei:*

**noseview.h**

*Beschreibung:*

Die Klasse **NoseView** stellt das Windows-Hauptfenster der Applikation dar. Sie besitzt eine Unterview auf das bearbeitete Bild, ein Menü und Methoden zum Ausführen der gewählten Menüpunkte.

***Komponenten:***

**ImageView *iv;**

Verbindung zur Unterview der Applikation.

**VPathString fname;**

Dateiname des bearbeiteten Bildes.

**Konstruktoren**

**NoseView( id aModel, char *fileName )**

erzeugt das Windows-Hauptfenster und die Unterview **ImageView** und führt alle nötigen Initialisierungen durch. **aModel** ist dabei ein Querverweis auf das Modell der Applikation (MVC-Paradigma). Falls **fileName** nicht **NULL** ist wird die zugehörige Datei geladen.

## Menüpunkte

**boolean openImage()**

führt einen Windows-Standard-Dialog zum Auswählen eines Dateinamens und öffnet die unter diesem Namen gespeicherte Datei.

**boolean openImage( VPathString &full )**

öffnet die Datei mit dem Namen **full**.

**void segment()**

veranlaßt die Segmentierung des Bildes.

**void classify()**

veranlaßt die Klassifikation der bei der Segmentierung gefundenen Symbole.

**void saveMidi()**

veranlaßt die Interpretation der Ergebnisse des Erkennungsprozesses um sie als MIDI-Datei unter dem aktuelle Dateinamen mit der Erweiterung MID abzuspeichern.

**void saveMidiAs()**

führt zuerst einen Windows-Standard-Dialog zum Auswählen eines Dateinamens und veranlaßt dann die Interpretation der Ergebnisse des Erkennungsprozesses um sie als MIDI-Datei unter dem ausgewählten Dateinamen abzuspeichern.

**void quitNose()**

beendet die Applikation.

### 5.4.3. ImageView

*Oberklassen:*

**VBitMapView**

*Unterklassen:*

**Keine**

*benutzte Klassen:*

**NoseModel, Image, Score, Staff, Symbol, Note, Rest, Clef, VPen, VPort**

*Header-Datei:*

**imagevw.h**

*Beschreibung:*

Die Klasse **ImageView** übernimmt die Darstellung des Bildes (gescannte Notenvorlage), die Darstellung der Ergebnisse des Erkennungsprozesses und ermöglicht das Scrollen bei Bildern, die größer als der vorhandene Raum des Fensters sind.

***Methoden:***

**boolean paint()**

zeichnet zuerst den Inhalt des Bildes (gescannte Notenvorlage) und darüber die Ergebnisse des Erkennungsprozesses. Nach der Segmentierung sind das die Notenlinien (Farbe grün) und die Segmente der Notenzeilen und Symbole (Farbe rot). Nach der Klassifizierung wird in blauer Farbe rechts vom Symbol eine kurze Textbeschreibung des selben dargestellt. Dabei stehen die Buchstaben **n** für **Note** (Note), **r** für **Rest** (Pause), **c** für **Clef** (Notenschlüssel), **w** für **whole** (ganze Note/Pause), **h** für **half** (halbe Note/Pause), **q** für **quarter** (viertel Note/Pause), **#** für das Vorzeichen **Kreuz**, **b** für das Vorzeichen **B**, **G** für den **Violinschlüssel**, **F** für den **Baßschlüssel**, **p** für **point** (Verlängerungspunkt einer Note) und eine Zahl für die Anzahl der Fähnchen bzw. Balken bei Noten und Pausen.

### 5.4.4. Image

*Oberklassen:*

**VBitMap**

*Unterklassen:*

**Keine**

*benutzte Klassen:*

**VPathString, Projection**

*Header-Datei:*

**IMAGE.h**

*Beschreibung:*

Die Klasse **Image** stellt eine Erweiterung der Standard-Bibliotheks-Klasse von C++/Views **VBitMap** dar. Objekte dieser Klasse (Bilder) werden sowohl für die Notenvorlage (gescanntes Notenblatt) als auch für die Vergleichsmuster, die im Resourceteil des Programms gespeichert sind, erzeugt. Die zusätzliche Funktionalität im Vergleich zu **VBitMap** besteht im Laden von Bildern aus dem Resourceteil des Programms (**loadBitMap**), im wahlfreien Zugriff auf die Pixel des Bildes (**black**), im Erzeugen von Projektionen auf das Bild (**xProjection**, **yProjection**), in Lauflängenuntersuchungen (**vertRunLength**), im Vergleich zweier Bilder (**percentEqual**) und im Verkleinern bzw. Vergrößern des Bildes (**stretch**).

***Komponenten:***

**handle hBits;**

Handle auf die Bits des Bildes, ein Pixel entspricht dabei einem Bit. **hBits** wird durch die Methoden **lock** und **unlock** gesperrt bzw. entsperrt.

### Private Methoden:

**BYTE huge *lock( int lineNr )**

sperrt die Bits des Bildes (**hBits**) zum nachfolgendem, wahlfreien Zugriff und liefert einen Zeiger auf den Speicherbereich der Zeile **lineNr**. Dem Aufruf von **lock** muß der Aufruf der Methode **unlock** folgen.

**void unlock()**

entsperrt die Bits des Bildes (**hBits**).

**boolean black( BYTE huge *lpLine, int x )**

ermöglicht den direkten Zugriff auf die x-te Spalte der Zeile **lpLine**. **lpLine** muß zuvor durch die Methode **lock** ermittelt und abschließend durch **unlock** wieder freigegeben werden.

**boolean copyBits()**

wird von der Methode **loadBitMap** aufgerufen und kopiert die Pixel des Bildes auf die Komponente **hBits**. Dabei findet eine Transformation der bildschirmabhängigen Pixel auf Monochrom-Pixel statt (1 Bit = 1 Pixel). Nach dieser Methode ist der wahlfreie, schnelle Zugriff auf die Pixel des Bildes über die private Methode **black** möglich.

### Laden eines Bildes

**boolean loadBitMap( char *fname, boolean rsc )**

ermöglicht das Laden beliebiger Bitmaps. Ist **rsc TRUE**, wird die Bitmap mit dem Namen **fname** vom Resourceteil des Programms geladen (für Vergleichsmuster).

### Wahlfreier Zugriff

**boolean black( HDC hDC, int x, int y )**

liefert **TRUE**, falls das Pixel an der Stelle (**x**,**y**) schwarz ist. Das Handle auf den Device-Kontext (**hDC**) wird durch die geerbte Methode **getHndl** ermittelt (nachfolgender Aufruf von **ungetHndl**). Diese Methode eignet sich jedoch nicht zum Zugriff auf große Mengen von Pixel, da sie zu langsam ist (die X/Y-Projektionen verwenden die private Methode **black**).

### Lauflängenuntersuchung

**int vertRunLength( int xstart, int ystart, int maxGap )**

ermittelt die längste, vertikale Lauflänge (Anzahl der aufeinanderfolgenden schwarzen Pixel) ausgehend vom Punkt (**xstart**, **ystart**). Ein positiver Wert bedeutet die Richtung abwärts, ein negativer aufwärts. Es werden Löcher von maximal **maxGap** Pixel zugelassen. Weiters ist es möglich, daß die nachfolgenden Pixel nicht unmittelbar aufeinanderfolgen, sondern um 1 nach links bzw. rechts versetzt sind.

### Projektionen

**Projection *xProjection()**

führt eine X-Projektion auf das gesamte Bild durch.

**Projection *xProjection( VRectangle &r )**

führt eine X-Projektion auf das Bild, eingeschränkt auf den Bereich des Rechtecks **r** durch.

**Projection *yProjection()**

führt eine Y-Projektion auf das gesamte Bild durch.

**Projection *yProjection( VRectangle &r )**

führt eine Y-Projektion auf das Bild, eingeschränkt auf den Bereich des Rechtecks **r** durch.

### Vergleich

**float percentEqual( int x, int y, Image *pattern )**

ermittelt die prozentuale Übereinstimmung des Bildes mit dem Vergleichsmuster **pattern**, wenn man das Muster auf die Position (**x**,**y**) des Bildes legt.

### Größe verändern

**boolean stretch( int newWidth, int newHeight )**

vergrößert bzw. verkleinert das Bild auf die Ausmaße **newWidth** und **newHeight**. (Dient zur Anpassung der Größe der Vergleichsmuster).

### 5.4.5. VIntVector

*Oberklassen:*

**VObject**

*Unterklassen:*

**Projection**

*benutzte Klassen:*

**Keine**

*Header-Datei:*

**intvec.h**

*Beschreibung:*

Die Klasse VIntVector implementiert einen Vektor von Integer-Zahlen mit zusätzlicher Funktionalität zum Sortieren (**sort**) und zur Berechnung einiger Maßzahlen (**maximum**, **indexOfMax**, **average** und **median**). Die Indizierung der Elemente des Vektors erfolgt mit Prüfung der Feldgrenzen.

***Komponenten:***

**int *vec;**

dynamisch angelegter Speicher für die Elemente des Vektors.

**unsigned sz;**

Größe des Vektors (Indizes von 0 bis **sz** - 1)

***Methoden:***

**Konstruktoren**

**VIntVector()**
legt einen Vektor mit der Defaultgröße 1 an.

**VIntVector( unsigned size )**
legt einen Vektor der Größe **size** an. Die Elemente des Vektors werden mit dem Wert 0 initialisiert.

**VIntVector( VIntVector &v )**
Erstellt eine Kopie des Vektors **v**.

**Zugriff auf Komponenten**

**int &elem( unsigned i )**
liefert das i-te Element des Vektors. **i** wird dabei MODULO der Größe des Vektors genommen (Indexprüfung).

**int size()**
liefert die Größe des Vektors

**Duplizieren**

**VIntVector *duplicate()**
erzeugt eine Kopie des gesamten Vektors.

**VIntVector *duplicate( unsigned start, unsigned len )**
erzeugt eine Kopie des Vektors der Länge **len** beginnend mit dem Element **start**.

**Sortieren**

**void sort()**
sortiert die Elemente des Vektors in aufsteigender Reihenfolge.

## Berechnungen

**int maximum()**

liefert den größten Wert des Vektors.

**unsigned indexOfMax()**

liefert den Index des letzten, größten Wertes des Vektors.

**float average()**

berechnet das arithmetische Mittel der Elemente des Vektors.

**float median()**

berechnet den Median des Vektors, eine Maßzahl, die im Gegensatz zum arithmetischen Mittel einzelne Ausreißer (untypische hohe/niedrige Werte) ignoriert. Die Berechnung ist einfach. Aus einer sortierten Version des Vektors wird, bei ungerader Größe, das mittlere Element, bei gerader Größe der Mittelwert der beiden mittleren Elemente zurückgegeben. Bsp.: v = 13,11,14,11,145,15,12. (145 ist ein eindeutiger Ausreißer). v->sort() = 11,11,12,13,14,15,145. v->median() = 13, v->average() = 31.57.

## 5.4.6. Projection

*Oberklassen:*

**VIntVector**

*Unterklassen:*

**Keine**

*benutzte Klassen:*

**Keine**

*Header-Datei:*

**projectn.h**

*Beschreibung:*

Die Klasse **Projection** ist abgeleitet von **VIntVector** und dient zur Handhabung von X-/Y-Projektionen. Solche Projektionen werden von der Klasse **Image** erzeugt. Sie stellen einen Integervektor dar, dessen i-te Komponente die Summe der schwarzen Pixel der i-te Spalte (X-Projektion) bzw. der i-te Zeile (Y-Projektion) des Bildes (**Image**) enthält. Durch die Komponente **wStart** und **wLen** können "Fenster" über die Projektion gelegt werden, aus deren Position und Länge wichtige Informationen gewonnen werden.

***Komponenten:***

**unsigned wStart;**

Startindex des Fensters der Projektion.

**unsigned wLen;**

Länge des Fensters der Projektion.

### *Methoden:*

#### Konstruktoren

**Projection( unsigned s )**
legt eine Projektion der Größe **s** an und setzt das Fenster auf Maximalgröße.

#### Duplizieren

**Projection *duplicate()**
erzeugt eine Kopie der Projektion.

#### Fenster

**boolean window()**
liefert **TRUE**, falls die Länge des Fensters größer 0 ist.

**unsigned windowStart()**
liefert den Startindex des Fensters der Projektion.

**unsigned windowLen()**
liefert die Länge des Fensters der Projektion.

**void setWindow( unsigned start, unsigned len )**
setzt Startindex und Länge des Fensters der Projektion.

**boolean nextSeqGreater( int value )**
setzt die Fenstergrenzen auf die nächste Folge von Werten > **value** nach dem aktuellen Fenster. Typisches Codestück: p->setWindow(0,0); while(p->nextSeqGreater(value)) ...

**unsigned longestSeqGreater( int value )**
setzt die Fenstergrenzen auf die längste Folge von Werten > **value**.

### Lokale Minima/Maxima

**boolean nextLocalMinimum( unsigned &pos )**

setzt **pos** auf das nächste lokale Minimum der Projektion. Falls kein Minimum mehr nach **pos** vorhanden ist, wird **FALSE** zurückgegeben, sonst **TRUE**.

**boolean nextLocalMaximum( unsigned &pos )**

setzt **pos** auf das nächste lokale Maximum der Projektion. Falls kein Maximum mehr nach **pos** vorhanden ist, wird **FALSE** zurückgegeben, sonst **TRUE**.

### Median-Projektion

**Projection *medianProjection( unsigned k )**

erzeugt eine Kopie der Projektion, bei der alle Werte i auf den Median der Teilfolge beginnend mit **i**-**k**/2 und der Länge **k** gesetzt werden. Dies bewirkt ein mehr oder weniger starkes Ignorieren von Ausreißern (untypisch hohe/niedrige Werte) der Projektion, abhängig von der Länge der betrachteten Teilfolge (**k**).

### 5.4.7. Segment

*Oberklassen:*

**VObject**

*Unterklassen:*

**Score, Staff, Symbol**

*benutzte Klassen:*

**Image**

*Header-Datei:*

**segment.h**

*Beschreibung:*

Ein Notenblatt läßt sich in mehrere Bereiche unterschiedlicher Bedeutung aufteilen (segmentieren), diese Bereiche werden **Segmente** genannt. Ein **Segment** ist ein rechteckiger Ausschnitt eines Bildes (**theImage**) und besitzt alle Informationen, die zur weiteren Segmentierung bzw. zur Klassifizierung notwendig sind. Alle weiteren Spezialisierungen werden durch Klassen dargestellt, die von **Segment** abgeleitet sind und können daher auf die gleiche Information zugreifen.

***Komponenten:***

**Image *theImage;**

Zeiger auf das dem Segment zugeordnete Bild (gescanntes Notenblatt).

**VLocation lu;**

linke, oberer Ecke des Segmentes, bezogen auf **theImage**.

**VLocation rd;**

rechte, untere Ecke des Segmentes, bezogen auf **theImage**.

**VIntVector *staffLines;**

Liste mit den vertikalen Positionen der Notenlinien. Stellt das Segment eine Partitur (Score) dar, so kann diese Liste auch länger als **linesPerStaff** (üblicherweise 5) sein, weil sie dann alle im Bild gefundenen Notenlinien enthält.

**unsigned linesPerStaff;**

bestimmt die Anzahl der Notenlinien, die eine Notenzeile ausmachen.

**Muster**

**VOrdCollect *noteHeads;**

Liste der Muster aller möglichen Notenköpfe (Indizes: 0=1/4-Notenkopf, 1=1/2-Notenkopf und 2=1/1-Notenkopf).

**VOrdCollect *accidentals;**

Liste der Muster aller möglichen Vorzeichen (Indizes: 0=Kreuz und 1=B).

**VOrdCollect *rests;**

Liste der Muster aller möglichen Pausensymbole (Indizes: 0=1/8-Pause, 1=1/4-Pause, 2=1/2-Pause und 3=1/1-Pause).

**VOrdCollect *clefs;**

Liste der Muster aller möglichen Notenschlüssel (Indizes: 0=Violinschlüssel und 1=Baßschlüssel)

### *Methoden:*

#### Konstruktoren

**Segment()**

erzeugt standardmäßig ein leeres Segment ohne Bildzuordnung. Achtung: alle Komponenten des Segmentes müssen händisch eingestellt werden!!!

**Segment( Segment *s )**

erzeugt eine Kopie des Segmentes **s**. Nur die Notenlinien (**staffLines**) werden tief kopiert, bei alle übrigen Komponenten werden nur die Zeiger gesetzt (flache Kopie).

**~Segment()**

gibt den Speicher für die Liste der Notenlinien (**staffLines**) wieder frei.

#### Runden

**int round( float x )**

Methode zum Runden einer float-Zahl.

#### Bildzuordnung

**Image *getImage()**

liefert den Zeiger auf das dem Segment zugeordnete Bild.

**void setImage( Image *i )**

ordnet dem Segment das Bild **i** zu.

#### Dimensionen abfragen

**int left()**

liefert die X-Koordinate der linken, oberen Ecke des Segmentes.

**int top()**

liefert die Y-Koordinate der linken, oberen Ecke des Segmentes.

**int right()**

liefert die X-Koordinate der rechten, unteren Ecke des Segmentes.

**int bottom()**
liefert die Y-Koordinate der rechten, unteren Ecke des Segmentes.

**VRectangle bounds()**
liefert die Grenzen des Segmentes in der Form eines Rechteckes.

**int width()**
liefert die Breite des Segmentes.

**int height()**
liefert die Höhe des Segmentes.

### Dimensionen setzen

**void left( int val )**
setzt die X-Koordinate der linken, oberen Ecke des Segmentes.

**void top( int val )**
setzt die Y-Koordinate der linken, oberen Ecke des Segmentes.

**void right( int val )**
setzt die X-Koordinate der rechten, unteren Ecke des Segmentes.

**void bottom( int val )**
setzt die Y-Koordinate der rechten, unteren Ecke des Segmentes.

### Notenlinien erhalten/setzen

**VIntVector *getLines()**

liefert einen Zeiger auf die Liste der Notenlinien des Segmentes.

**void setLines( VIntVector *lns )**

ordnet dem Segment eine neue Liste von Notenlinien (**lns**) zu. Bereits vorhandene Notenlinien werden freigegeben.

### Notenlinienabstand und -dicke

**float aveLineDistance()**

berechnet den durchschnittlichen Abstand zwischen den Notenlinien des Segmentes. Enthält das Segment mehr als **linesPerStaff** Notenlinien, dann repräsentiert es eine Partitur (Score), und der Linienabstand wird nicht durch das arithmetische Mittel, sondern durch den Median berechnet, da der Abstand der letzten Notenlinie einer Notenzeile zur ersten Linie der nächsten Zeile viel größer ist und den Durchschnitt somit unproportional erhöhen würde. Enthält das Segment noch keine Notenlinien (Zustand vor der Segmentierung), so wird der Linienabstand aus der Größe des zugeordneten Bildes geschätzt.

**float aveLineWidth()**

die Liniendicke wird auf ein fünftel des durchschnittliche Linienabstandes geschätzt.

### Position der Notenlinien

**int lineNo( unsigned i )**

liefert die Y-Position der i-ten Notenlinie des Segmentes (i wird dabei MODULO der Anzahl der Notenlinien genommen).

**int firstLine()**

liefert die Y-Position der ersten Notenlinien des Segmentes.

**int lastLine()**

liefert die Y-Position der letzten Notenlinien des Segmentes.

## Notenlinien suchen

**VIntVector *findHorLines( Projection *yp, int threshold )**

untersucht die Y-Projektion **yp** auf das Vorhandensein horizontaler Linien, indem alle lokalen Maxima von **yp**, deren Wert größer **threshold** ist, in eine Liste aufgenommen werden und ein Zeiger darauf zurückgegeben wird.

## 5.4.8. Score

*Oberklassen:*

**Segment**

*Unterklassen:*

**Keine**

*benutzte Klassen:*

**MidiFile**

*Header-Datei:*

**SCORE.h**

*Beschreibung:*

Die Klasse **Score** (Partitur) repräsentiert nach durchgeführter Segmentierung (Methode **segment**) und Klassifizierung (Methode **classify**) das "verstandene", erkannte Bild eines Notenblattes. Sie führt dabei eine Liste aller Notenlinien (**staffLines**), eine Liste der vorhandenen Notenzeilen (**staffs**) und zur Interpretation (Methode **interpret**) die Anzahl der Notenzeilen pro Notensystem (**staffsPerSystem**). Weiters besitzt **Score** eine Reihe von privaten Methoden zur Aufteilung der Segmentierung in einzelne, leicht verständliche Schritte. Mit der Methode **loadPatterns** werden die für die Klassifizierung notwendigen Vergleichsmuster geladen.

**Private Methoden:**

**VIntVector *findHorLines()**

wird von der Methode **segment** aufgerufen und liefert eine Liste der horizontalen Linien im Score (= vertikalen Positionen der Notenlinien). Ebenso wird die Anzahl der Notenzeilen pro Notensystem bestimmt (**staffsPerSystem**).

**void buildStaffs()**

wird von der Methode **segment** aufgerufen und erzeugt aus jeweils **linesPerStaff** Notenlinien eine Notenzeile, die danach an die Liste der Notenzeilen (**staffs**) angehängt wird. Dabei werden die Abstände zwischen den Notenlinien überprüft, um evt. falsch erkannte Notenlinien ignorieren zu können.

**void setStaffBorders()**

wird von der Methode **segment** aufgerufen und setzt die Segmentgrenzen für jede Notenzeile der Liste **staffs**. Die Grenze zwischen zwei Notenzeilen wird beim Mittelpunkt des Abstandes der ersten zur letzten Notenlinie der beteiligten Notenzeilen gesetzt. Die linken und rechten Grenzen werden auf die Werte von Score gestellt.

**void resizePattern( Image *pat, float factor )**

verändert die Breite und Höhe des Vergleichsmusters **pat** um **factor**.

### Öffentliche Variablen:

**unsigned staffsPerSystem;**

zeigt an, wieviele Notenzeilen zu einem Notensystem zusammengefaßt werden. Dies entspricht der Anzahl der zu erzeugenden MIDI-Spuren - 1 bei der Interpretation.

**VOrdCollect staffs;**

Liste der gefundenen Notenzeilen (wird beim Segmentieren erzeugt).

### Muster

**void loadPatterns()**

lädt alle vorhandenen Vergleichsmuster und paßt sie anhand des durchschnittlichen Linienabstandes in ihrer Größe an.

**void freePatterns()**

gibt alle Vergleichsmuster wieder frei.

### Konstruktoren

**Score( Image *im )**

erzeugt ein Score-Segment auf das gesamte Bild **im**. Die Liste der Notenzeilen ist leer und es wird zunächst angenommen, daß ein Notensystem auch nur eine Notenzeile (**staffsPerSystem**=1) enthält. Durch den Aufruf der Methode **segment** wird das Bild **im** segmentiert, die Liste der Notenlinien erzeugt und die Liste der Notenzeilen aufgebaut. Beim Aufruf der Methode **classify** werden die Vergleichsmuster geladen und die Klassifizierung der Symbole durchgeführt.

**~Score()**

gibt die Liste der Notenzeile und die Vergleichsmuster frei.

### Segmentierung

**boolean segment()**

führt die Segmentierung der gesamten Partitur (**Score**) durch. Erster Schritt ist das Finden der Notenlinien. Aus jeweils **linesPerStaff** gefundenen Linien wird eine Notenzeile aufgebaut und in die Liste der Notenzeilen (**staffs**) aufgenommen. Anschließend wird jede Notenzeile noch weiter segmentiert (siehe Klasse **Staff**).

### Klassifizierung

**void classify()**

führt die Klassifizierung der Symbole der Partitur (**Score**) durch (ruft für jeden Notenzeile die Methode **classify** auf. Zuvor werden sämtliche Vergleichsmuster geladen und in der Größe angepaßt.

### Interpretation

**void interpret( MidiFile &mf )**

interpretiert die klassifizierten Symbole der Partitur und schreibt die resultierenden MIDI-Ereignisse in die MIDI-Datei **mf**. Die Spur 0 der MIDI-Datei wird dabei als Masterspur verwendet, auf die Spuren 1 bis **staffsPerSystem** werden die Musiksymbole geschrieben (Definition von MIDI-Dateien!).

### 5.4.9. Staff

*Oberklassen:*

**Segment**

*Unterklassen:*

**Keine**

*benutzte Klassen:*

**VIntVector, MidiTrack**

*Header-Datei:*

**STAFF.h**

*Beschreibung:*

Die Klasse **Staff** (Notenzeile) repräsentiert nach durchgeführter Segmentierung (Methode **segment**) und Klassifizierung (Methode **classify**) das "verstandene", erkannte Bild einer Notenzeile. Sie führt dabei eine Liste von **linesPerStaff** Notenlinien (**staffLines**) und eine Liste der gefundenen Symbole (**symbols**). Nach der Segmentierung befinden sich in dieser Liste nur Instanzen der Klasse **Symbol**. Durch das Klassifizieren werden diese Symbole danach durch konkrete Symbole (**Note**, **Rest** oder **Clef**) ersetzt. Weiters besitzt **Staff** eine Reihe von privaten Methoden zur Aufteilung der Segmentierung in einzelne, leicht verständliche Schritte. Mit der Methode **deleteTwins** können doppelt erkannte Symbole gelöscht werden.

**Private Methoden:**

**void horLocalize()**

wird von der Methode **segment** aufgerufen. Mittels einer X-Projektion auf den Bereich der gesamten Notenzeile wird festgestellt, wo sich Symbole befinden bzw. wo Leerstellen sind. Dieser horizontalen Lokalisierung der Symbole folgt eine vertikale Aufteilung (**vertSplit**), da sich auch mehrere Symbole an der gleichen horizontalen Position der Notenzeile befinden können.

**void vertSplit()**

wird von der Methode **segment** nach der horizontalen Lokalisierung (**horLocalize**) aufgerufen. Dabei wird auf jedes bereits gefundene Symbol eine Y-Projektion durchgeführt. Falls das Symbol aus mehreren übereinander liegenden Symbolen besteht, so wird es entsprechend aufgeteilt.

**void deleteTwins()**

löscht doppelte Symbole in der Liste **symbols**. Dies kann nötig sein, wenn bei der Segmentierung ein Symbol in 2 oder mehrere Teile zerlegt wurde, jedoch bei der Klassifizierung jeder dieser Teile wieder als das gleiche Symbol erkannt wurde.

### Öffentliche Variablen:

**VOrdCollect symbols;**

Liste der gefundenen Symbol (wird beim Segmentieren erzeugt). Die Symbole werden beim Klassifizieren durch konkrete Ausprägungen (**Note**, **Rest** oder **Clef**) ersetzt.

### Konstruktoren

**Staff( Image *im, VIntVector &lines )**

erzeugt eine Notenzeile auf dem Bild **im** mit den Notenlinien **lines**. Die Liste der Symbole (**symbols**) ist zunächst leer.

**~Staff()**

gibt alle erkannten Symbole wieder frei.

### Segmentierung

**void segment()**

führt die Segmentierung einer Notenzeile durch. Es werden dabei in zwei Schritten, der horizontalen Lokalisierung (**horLocalize**) und der vertikalen Aufteilung (**vertSplit**), Instanzen der Klasse **Symbol** erzeugt und in der Liste **symbols** aufgenommen. Diese Symbole werden dann bei der Klassifizierung durch Ableitungen der Klasse **Symbol** (**Note**, **Rest** oder **Clef**) ersetzt.

## Klassifizierung

**void classify()**

führt die Klassifizierung der Symbole der Notenzeile durch. Das erste Symbol muß ein Notenschlüssel sein (Violin- bzw. Baßschlüssel), andernfalls werden alle Symbole ignoriert. Danach werden Noten- oder Pausensymbole erwartet. Die Klassifizierung findet durch Mustervergleich mit Vorlagen für Notenköpfe (1/4, 1/2 oder 1/1) und für Pausen (1/8, 1/4, 1/2, oder 1/1) statt. Doppelt erkannte Symbole werden danach durch die Methode **deleteTwins** eliminiert. Anschließend werden für alle erkannten Noten die Bestimmungsstücke (Vorzeichen, Fähnchen/Balken...) ermittelt.

## Interpretation

**void interpret( int ticksPerQuarterNote, MidiTrack *mt )**

interpretiert die erkannten Symbole der Liste **symbols** und schreibt das Ergebnis in die MIDI-Spur **mt**. Mehrstimmige Notationen werden zur Zeit noch nicht unterstützt.

## 5.4.10. Symbol

*Oberklassen:*

**Segment**

*Unterklassen:*

**Clef, Note, Rest**

*benutzte Klassen:*

**Keine**

*Header-Datei:*

**SYMBOL.h**

*Beschreibung:*

Die Klasse **Symbol** ist eine Unterklasse von **Segment**. Objekte dieser Klasse werden beim Segmentieren des Notenblattes erzeugt um später bei der Klassifizierung der Symbole durch Objekte der von **Symbol** abgeleiteten Klassen ersetzt zu werden (klassifizierte Symbole). Diese Klasse enthält auch eine Methode zum Mustervergleich (**compare**).

***Methoden:***

**Konstruktoren**

**Symbol( Segment *s )**

erzeugt ein Symbol auf der Basis des Segmentes **s**. Alle Eigenschaften (Grenzen, Positionen der Notenlinien etc.) werden dabei von **s** übernommen.

**Segmentierung**

**boolean segment()**

dient zur "genaueren" Bestimmung der Symbolgrenzen. Durch X- bzw. Y-Projektionen werden die leeren Stellen des Symbols, an denen sich "nur" Notenlinien befinden, entdeckt und dabei die Grenzen entsprechend korrigiert.

## Klassifizierung

**boolean compare( Image *pattern, float threshold )**

führt einen Mustervergleich zwischen dem Symbol und dem Muster **pattern** durch und liefert **TRUE**, falls die prozentuale Übereinstimmung >= dem Grenzwert **threshold** ist. Das Muster wird dabei über dem Symbol bewegt, damit die beste Übereinstimmung erzielt werden kann. Im Erfolgsfall werden die Grenzen des Symbols entsprechend korrigiert.

**boolean isNote( HeadType &ht )**

liefert **TRUE**, falls das Bild des Symbols einen der 3 Notenköpfe darstellt (1/1-, 1/2- oder 1/4-Notenkopf). **ht** enthält dann den entsprechenden Typ. Konnte keine Übereinstimmung festgestellt werden, ist **FALSE** der Rückgabewert.

**boolean isRest( RestType &rt )**

liefert **TRUE**, falls das Bild des Symbols eine der 4 Pausentypen darstellt (1/1-, 1/2-, 1/4 oder 1/8-Pause). **rt** enthält dann den entsprechenden Typ. Konnte keine Übereinstimmung festgestellt werden, ist **FALSE** der Rückgabewert.

**boolean isClef( ClefType &ct )**

liefert **TRUE**, falls das Bild des Symbols einen der 2 Notenschlüssel darstellt (Violin- oder Baßschlüssel). **ct** enthält dann den entsprechenden Typ. Konnte keine Übereinstimmung festgestellt werden, ist **FALSE** der Rückgabewert.

## 5.4.11. Note

*Oberklassen:*

**Symbol**

*Unterklassen:*

**Keine**

*benutzte Klassen:*

**Clef**

*Header-Datei:*

**note.h**

*Beschreibung:*

Die Klasse **Note** ist eine Unterklasse von **Symbol** und wird beim Klassifizieren erzeugt, wenn eine Note erkannt wurde. Es existieren 3 verschiedene Typen (headType), 1/1-, 1/2-, und 1/4-Noten, die sich in der Erscheinungsform des Notenkopfes grundlegend unterscheiden. Bei 1/4-Noten gibt **noFlags** darüber Auskunft, wieviele Fähnchen bzw. Balken die Note besitzt. **durPoint** zeigt an, ob sich rechts von der Note ein Verlängerungspunkt befindet (**TRUE**). Besitzt die Note ein Vorzeichen, so wird der Typ des Vorzeichens in **accType** angezeigt, sonst steht hier der Wert **AT_NONE**.

***Komponenten:***

**HeadType headType;**

bestimmt den Typ des Notenkopfes der Note. Mögliche Werte sind: **HT_WHOLE** (1/1-Note), **HT_HALF** (1/2-Note) und **HT_QUARTER** (1/4-Note).

**int noFlags;**

enthält die Anzahl der Fähnchen bzw. Balken der Note (nur relevant beim Typ 1/4-Note, bei allen anderen Typen ist der Wert 0). Der Notenwert wird durch das Vorhandensein von Fähnchen/Balken entsprechend verkürzt.

**boolean durPoint;**

ist der Wert **TRUE**, so befindet sich ein Verlängerungspunkt rechts neben der Note. Der Notenwert wird dadurch entsprechend verlängert.

**AccType accType;**

zeigt an, ob sich links von der Note ein Vorzeichen befindet und falls ja, welches. Mögliche Werte für **accType** sind: **AT_NONE** (kein Vorzeichen), **AT_SHARP** (Kreuz) und **AT_FLAT** (B).

**Private Methoden:**

**int countFlags( VRectangle &r )**

ermittelt die Anzahl der Fähnchen bzw. Balken durch eine Y-Projektion auf das Rechteck **r**, welches den Notenhals der Note umschließt.

***Methoden:***

**Konstruktoren**

**Note()**

erzeugt standardmäßig eine 1/4-Note ohne Vorzeichen sowie ohne Verlängerungspunkt.

**Note( Symbol *s, HeadType ht )**

erzeugt eine Note ohne Vorzeichen sowie ohne Verlängerungspunkt mit einem Notenkopf vom Typ **ht** auf der Basis des Symbols **s**. Alle Eigenschaften (Grenzen, Positionen der Notenlinien etc.) werden dabei vom Symbol **s** übernommen. Die tatsächlichen Bestimmungsmerkmale der Note werden durch die Methoden **setFlags**, **setAccidentals** und **setDurPoint** ermittelt.

### Abfrage der Bestimmungsmerkmale

**HeadType head()**
liefert den Typ des Notenkopfes der Note.

**int flags()**
liefert die Anzahl der Fähnchen.

**boolean point()**
**TRUE**, falls die Note einen Verlängerungspunkt besitzt, sonst **FALSE**.

**AccType accidental()**
liefert den Typ des Vorzeichens, falls die Note ein solches besitzt, sonst **AT_NONE**.

**long noTicks( long ticksPerQuarterNote )**
berechnet die Anzahl der Ticks der Note (relativ zur Anzahl der Ticks einer 1/4-Note) unter Berücksichtigung der Bestimmungsmerkmale **headType**, **noFlags** und **durPoint**. Bsp.: angenommen eine 1/4-Note würde 96 Ticks dauern, dann dauert eine 1/1-Note 384 Ticks (96*4), eine 1/2-Note 192 Ticks (96*2) usw.

### Bestimmungsmerkmale ermitteln

**int setFlags()**
ermittelt die Anzahl der Fähnchen bzw. Balken der Note. Zuerst wird mittels Lauflängenuntersuchungen die Position (links/rechts vom Notenkopf), die Richtung (nach oben/unten) und die Länge des Notenhalses der Note bestimmt. Danach wird ein Rechteck, welches den Notenhals der Note umschließt, erzeugt und der privaten Methode **countFlags** zur Bearbeitung übergeben. Dabei wird die Komponente **noFlags** gesetzt.

**boolean setDurPoint()**
stellt mittels X- und Y-Projektion fest, ob sich innerhalb eines bestimmten Bereiches rechts vom Notenkopf ein Verlängerungspunkt befindet und setzt die Komponente **durPoint** entsprechend auf **TRUE** bzw. **FALSE**.

**AccType setAccidentals()**

stellt fest, ob sich innerhalb eines bestimmten Bereiches links vom Notenkopf ein Vorzeichen befindet. Wenn ja, wird mittels Mustervergleich der Typ des Vorzeichens bestimmt und die Komponente **accType** entsprechend gesetzt.

### Interpretieren der Note

**int keyNr( Clef *c )**

errechnet aus der vertikalen Position der Note im Bezug auf die Notenlinien die MIDI-Tastennummer (60 entspricht dem C3 unter Berücksichtigung des Notenschlüssels **c** und des evt. vorhandenen Vorzeichens).

**boolean inChord( Symbol *s )**

liefert **TRUE**, falls die linken Grenzen der Note und des Symbols **s** nicht weiter als 1/3 des durchschnittlichen Linienabstandes von einander entfernt sind, was bedeutet, daß das Symbol **s** zum gleichen Akkord gehört.

## 5.4.12. Rest

*Oberklassen:*

**Symbol**

*Unterklassen:*

**Keine**

*benutzte Klassen:*

**Keine**

*Header-Datei:*

**rest.h**

*Beschreibung:*

Die Klasse **Rest** ist eine Unterklasse von **Symbol** und wird beim Klassifizieren erzeugt, wenn eine Pause erkannt wurde. Es existieren 4 verschiedene Typen (restType), 1/1-, 1/2-, 1/4- und 1/8-Pausen, die sich in ihrer Erscheinungsform grundlegend unterscheiden. Bei 1/8-Pausen gibt **noFlags** darüber Auskunft, wieviele Fähnchen die Pause besitzt.

***Komponenten:***

**RestType restType;**

bestimmt den Typ der Pause. Mögliche Werte sind: **RT_WHOLE** (1/1-Pause), **RT_HALF** (1/2-Pause), **RT_QUARTER** (1/4-Pause) und **RT_EIGTH** (1/8-Pause).

**int noFlags;**

enthält die Anzahl der Fähnchen der Pause (nur relevant beim Typ 1/8-Pause, bei allen anderen Typen ist der Wert 0). Der Notenwert der Pause wird dabei durch das Vorhandensein von Fähnchen entsprechend verkürzt.

### *Methoden:*

### Konstruktoren

**Rest()**

erzeugt standardmäßig eine 1/4-Pause

**Rest( Symbol *s, RestType rt )**

erzeugt eine Pause vom Typ **rt** auf der Basis des Symbols **s**. Alle Eigenschaften (Grenzen, Positionen der Notenlinien etc.) werden dabei vom Symbols **s** übernommen.

### Bestimmungsmerkmale

**RestType rest()**

liefert den Typ der Pause.

**int flags()**

liefert die Anzahl der Fähnchen.

**long noTicks( long ticksPerQuarterNote )**

berechnet die Anzahl der Ticks der Pause (relativ zur Anzahl der Ticks einer 1/4-Pause) unter Berücksichtigung der Bestimmungsmerkmale **restType** und **noFlags**. Bsp.: angenommen eine 1/4-Pause würde 96 Ticks dauern, dann dauert eine 1/1-Pause 384 Ticks (96*4), eine 1/2-Pause 192 Ticks (96*2) usw.

## 5.4.13. Clef

*Oberklassen:*

**Symbol**

*Unterklassen:*

**Keine**

*benutzte Klassen:*

**Keine**

*Header-Datei:*

**clef.h**

*Beschreibung:*

Die Klasse **Clef** ist eine Unterklasse von **Symbol** und wird beim Klassifizieren erzeugt, wenn ein Notenschlüssel erkannt wurde. Es existieren 2 verschiedene Typen (clefType), der Violinschlüssel (G-Schlüssel) und der Baßschlüssel (F-Schlüssel), die sich in ihrer Erscheinungsform grundlegend unterscheiden.

***Komponenten:***

**ClefType clefType;**

bestimmt den Typ des Notenschlüssels. Mögliche Werte sind: **CT_G** (Violinschlüssel) und **CT_F** (Baßschlüssel).

**int whiteKeys[ 7 ];**

enthält die MIDI-Tastennummern der weißen Tasten einer Oktave, anhängig vom Typ des Notenschlüssels (dient zur Berechnung der MIDI-Tastennummer einer Note).

***Methoden:***

**Konstruktoren**

**Clef()**

erzeugt standardmäßig einen Violinschlüssel.

**Clef( Symbol *s, ClefType ct )**

erzeugt einen Notenschlüssel vom Typ **ct** auf der Basis des Symbols **s**. Alle Eigenschaften (Grenzen, Positionen der Notenlinien etc.) werden dabei vom Symbol **s** übernommen.

**Bestimmungsmerkmale**

**ClefType clef()**

liefert den Typ des Notenschlüssel.

## 5.4.14. MidiFile

*Oberklassen:*

**VFile**

*Unterklassen:*

**Keine**

*benutzte Klassen:*

**MidiTrack**

*Header-Datei:*

**midifile.h**

*Beschreibung:*

Die Klasse **MidiFile** realisiert das Erzeugen und Speichern von Dateien im Standard-MIDI-File-Format. Zuerst wird eine MIDI-Datei mit gewünschtem Format, der Anzahl der Spuren und der gewünschten Auflösung erzeugt. Danach werden die Spuren (**MidiTrack**) mit MIDI-Ereignissen gefüllt und abschließend wird das Ganze durch die Methode **writeAll** gespeichert.

***Komponenten:***

**int format;**

es existieren 3 Formate: Format 0 bedeutet, daß die MIDI-Datei nur eine Spur besitzen kann, Format 1 erlaubt beliebig viele Spuren (**noTracks**), die gleichzeitig gespielt werden und Format 2 erlaubt ebenfalls beliebig viele Spuren, jedoch werden diese zeitlich hintereinander gespielt.

**int ticksPerQuarterNote;**

bestimmt die Auflösung der Noten der MIDI-Datei in **ticks**.

**VOrdCollect tracks;**

Liste der Spuren der MIDI-Datei (Länge = **noTracks**).

**int noTracks;**
Anzahl der Spuren der MIDI-Datei (1 bei Format-0-Dateien).

### Private Methoden:

**void writeHeader()**
schreibt den Header einer Standard-MIDI-Datei.

**void writeTrack( MidiTrack *mt )**
schreibt die MIDI-Ereignisse der Spur **mt**.

### *Methoden:*

### Konstruktoren

**MidiFile()**
Erzeugt eine MIDI-Datei im Format 0, mit einer Spur und der Auflösung 96 Ticks pro 1/4-Note.

**MidiFile( int f, int nt, int ticks )**
Erzeugt eine MIDI-Datei im Format **f**, mit **nt** Spuren und der Auflösung **ticks** pro 1/4-Note.

**~MidiFile()**
gibt alle Spuren der MIDI-Datei wieder frei.

### Zugriff auf Komponenten

**int getTicks()**
liefert die Auflösung der Noten der MIDI-Datei in **ticks**.

**MidiTrack *track( unsigned nr )**
liefert einen Zeiger auf die Spur **nr** der MIDI-Datei. **nr** wird dabei MODULO der Anzahl der Spuren (**noTracks**) genommen.

## MIDI-Datei speichern

**boolean writeAll()**

schreibt zuerst den Header (**writeHeader**) und dann alle vorhanden Spuren (**writeTrack**).

## 5.4.15. MidiTrack

*Oberklassen:*

**VStream**

*Unterklassen:*

**Keine**

*benutzte Klassen:*

**Keine**

*Header-Datei:*

**miditrck.h**

*Beschreibung:*

Die Klasse **MidiTrack** stellt eine Spur einer MIDI-Datei (**MidiFile**) dar. Sie besitzt einen Basis-MIDI-Kanal und kann beliebig viele MIDI-Ereignisse, bestehend aus einer Delta-Zeitangabe und einer MIDI-Nachricht (z.B NoteOn), enthalten. Die MIDI-Spur muß mit der MIDI-Nachricht End-Of-Track (Spurende) abgeschlossen werden.

***Komponenten:***

**int midiChannel;**

jede Spur besitzt einen Basis-MIDI-Kanal (Werte 0 bis 15), obwohl einzelne MIDI-Ereignisse einen vom Basiskanal unterschiedlichen MIDI-Kanal haben können.

**Private Methoden:**

**void putVLQ( long value )**

Einige Werte werden bei MIDI-Dateien in der Form von Variable-Length-Quantities repräsentiert. Alle Bytes haben 7 signifikante Bits (Bit 0 bis 6) und Bit 7 gesetzt (1) ausgenommen dem letzten Byte der Folge, welches Bit 7 gelöscht (0) hat. Bei Werten zwischen 0 und 127 ist das somit genau 1 Byte, Werte darüber müssen dann auf 2 oder mehrere Bytes aufgeteilt werden.

***Methoden:***

### Konstruktoren

**MidiTrack()**

Erzeugt eine MIDI-Spur mit dem Basis-MIDI-Kanal 0.

**MidiTrack( int channel )**

Erzeugt eine MIDI-Spur mit dem Basis-MIDI-Kanal **channel** Dabei wird **channel** MODULO 16 genommen (es sind nur 16 verschiedene MIDI-Kanäle möglich.

### Delta-Zeit

**void putDeltaTime( long deltaTime )**

Ein MIDI-Ereignis besteht immer aus einer Zeitangabe (**putDeltaTime**) in ticks und einer darauffolgenden MIDI-Nachricht (NoteOn, NoteOff, EOT...).

### MIDI-Ereignisse

**void putNoteOn( int keyNr, int velocity )**

schreibt die MIDI-Nachricht Taste drücken (NoteOn) mit der Tastennummer **keyNr** und der Anschlagstärke **velocity** in die MIDI-Spur.

**void putNoteOff( int keyNr, int velocity )**

schreibt die MIDI-Nachricht Taste loslassen (NoteOff) mit der Tastennummer **keyNr** und der "Release Velocity" **velocity** in die MIDI-Spur.

**void putEOT()**

schreibt die MIDI-Nachricht End-Of-Track (Spurende). Nachfolgende MIDI-Ereignisse werden ignoriert.

# 6. Testbericht und Ergebnisse

Abschließend möchte ich nun einen Bericht über die Tests mit dem Programm NOSE und die daraus resultierenden Ergebnisse bringen. Zu diesem Zweck habe ich 10 Bilder mit Notenschrift untersucht, die größtenteils mit dem Flachbettscanner HP ScanJet 2P unter dem Programm DESKSCAN mit unterschiedlichen Auflösungen (100%, 150% und 200%) eingescannt wurden. Eines davon hat den Namen DEMO.BMP und ist das "konstruierte" Beispiel, welches auch im Kapitel 5.3 zu Demonstrationszwecken verwendet wurde. Die übrigen Bilder mit den Namen TEST1.BMP bis TEST9.BMP stellen "echte" Kompositionen bzw. taktweise Auszüge davon dar, wobei letztere sich auf einen bestimmten Typ von Musiksymbolen konzentrieren. Sämtliche Tests wurden ebenso wie die Entwicklung des Programms auf einem PC 486/33 mit 8MB Hauptspeicher durchgeführt.

## 6.1. Test: DEMO.BMP

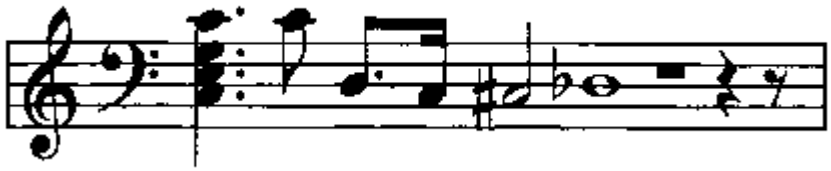

Abbildung 6-1: DEMO.BMP

Ergebnisse der Segmentierung:

| | |
|---|---|
| Notenlinien | 5 |
| Notenzeilen | 1 |
| Spuren | 1 |
| Segmente | 40 |
| Zeit | 4" |

Ergebnisse der Klassifizierung:

| | |
|---|---|
| Noten | 8 |
| Pausen | 4 |
| Notenschlüssel | 1 |
| Zeit | 58" |

Kommentar:

Der 2. Notenschlüssel (Baßschlüssel) wurde ignoriert, da für NOSE nur ein Notenschlüssel pro Notenzeile zulässig ist. Dafür wurde in einem Teil des Baßschlüssels fälschlicherweise eine 1/8 Pause erkannt. Die unterste Note des Akkordes konnte nicht klassifiziert werden, da der nötige Grenzwert von 82% Übereinstimmung knapp unterschritten wurde. Alle übrigen Symbole einschließlich Verlängerungspunkte, Balken, Fähnchen und Vorzeichen wurden korrekt klassifiziert und auch interpretiert.

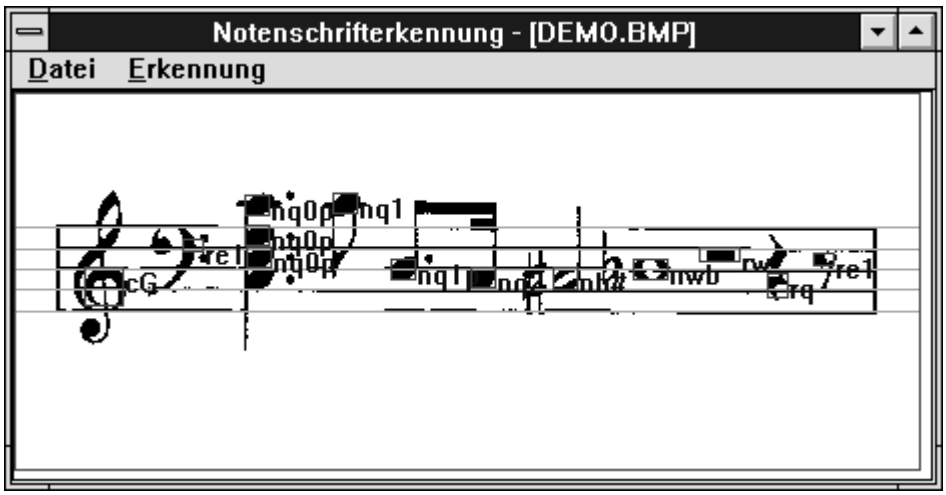


Abbildung 6-2: DEMO.BMP erkannt.

## 6.2. Test: TEST1.BMP

(Auszug aus BACH2, S. 43 "Giguetta", 200% Auflösung)

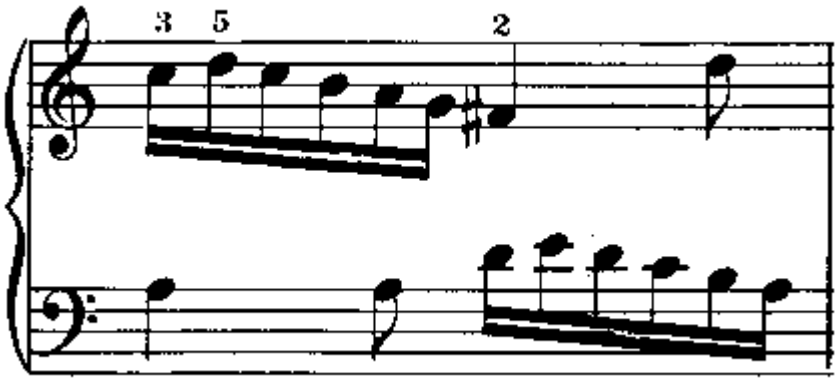


Abbildung 6-3: TEST1.BMP

Ergebnisse der Segmentierung:

| | |
|---|---|
| Notenlinien | 10 |
| Notenzeilen | 2 |
| Spuren | 2 |
| Segmente | 72 |
| Zeit | 8" |

Ergebnisse der Klassifizierung:

| | |
|---|---|
| Noten | 16 |
| Pausen | 0 |
| Notenschlüssel | 2 |
| Zeit | 4'03" |

Kommentar:

Das Bild wurde korrekt erkannt.

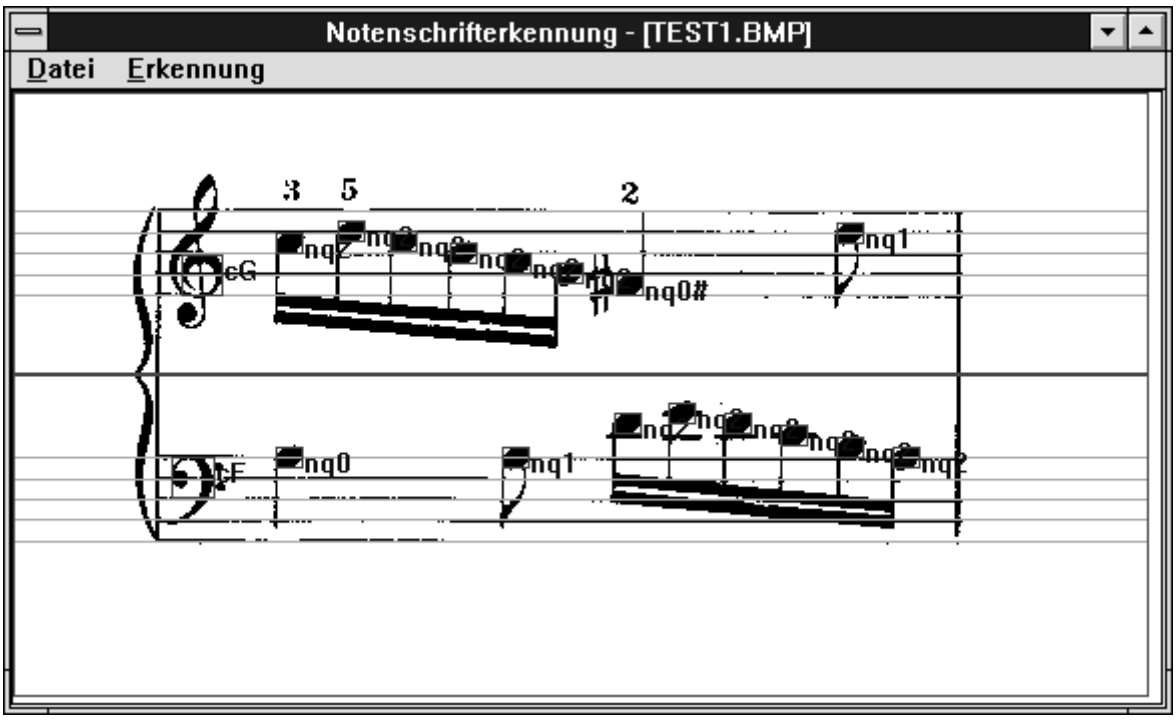


Abbildung 6-4: TEST1.BMP erkannt.

## 6.3. Test: TEST2.BMP

(Auszug aus BACH2, S. 43 "Giguetta", 150% Auflösung)

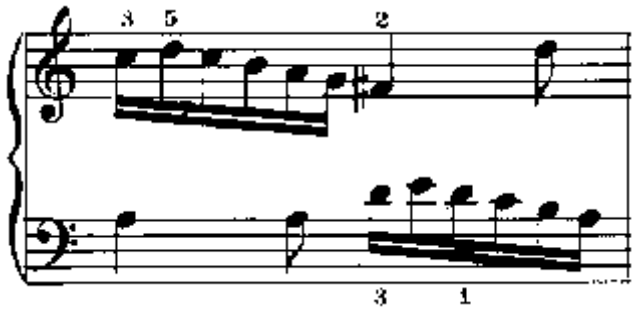

Abbildung 6-5: TEST2.BMP

Ergebnisse der Segmentierung:

| | |
|---|---|
| Notenlinien | 10 |
| Notenzeilen | 2 |
| Spuren | 2 |
| Segmente | 63 |
| Zeit | 5" |

Ergebnisse der Klassifizierung:

| | |
|---|---|
| Noten | 8 |
| Pausen | 0 |
| Notenschlüssel | 1 |
| Zeit | 56" |

Kommentar:

In der 1. Notenzeile konnte der Notenschlüssel nicht erkannt werden, als Folge davon wurde kein einziges Symbol in dieser Zeile klassifiziert. In der 2. Notenzeile wurden alle Symbole korrekt klassifiziert, jedoch sind aufgrund der geringen Auflösung 2 Notenhälse schon in der Vorlage nicht vorhanden, was falsche Notenwerte bewirkt. Die Tonhöhe der Note auf der 2. Hilfslinie wurde zu hoch interpretiert.

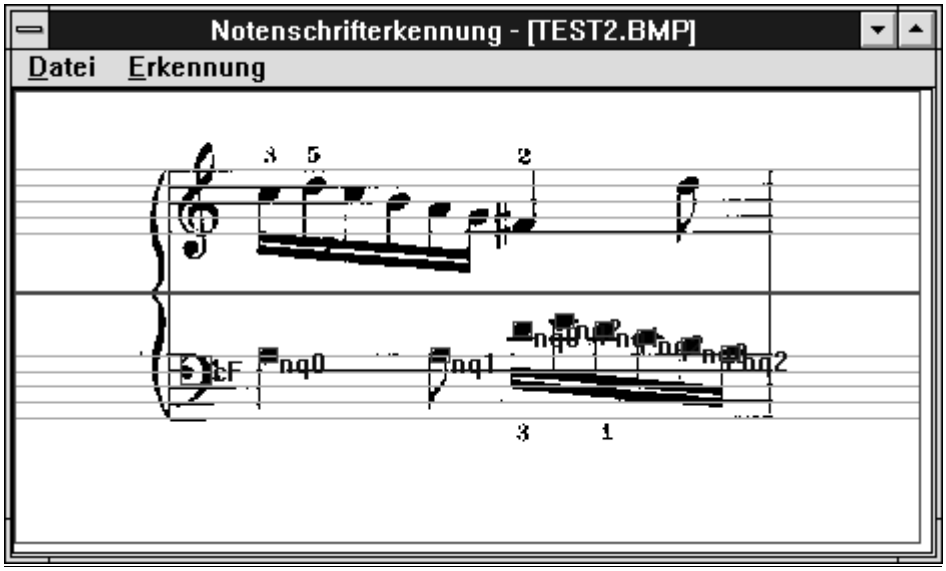


Abbildung 6-6: TEST2.BMP erkannt

## 6.4. Test: TEST3.BMP

(Zusammenstellung aller möglichen Pausentypen aus diversen Notenvorlagen)

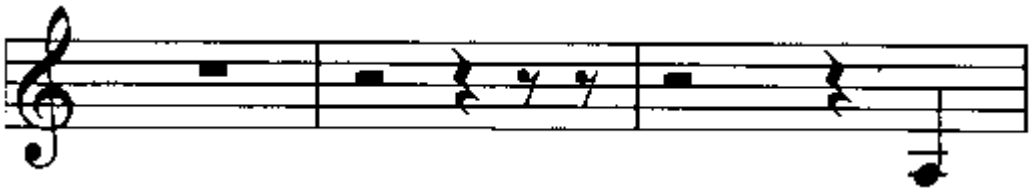

Abbildung 6-7: TEST3.BMP

Ergebnisse der Segmentierung:

| | |
|---|---|
| Notenlinien | 5 |
| Notenzeilen | 1 |
| Spuren | 1 |
| Segmente | 25 |
| Zeit | 5" |

Ergebnisse der Klassifizierung:

| | |
|---|---|
| Noten | 1 |
| Pausen | 7 |
| Notenschlüssel | 1 |
| Zeit | 38" |

Kommentar:

Das Bild wurde korrekt erkannt.

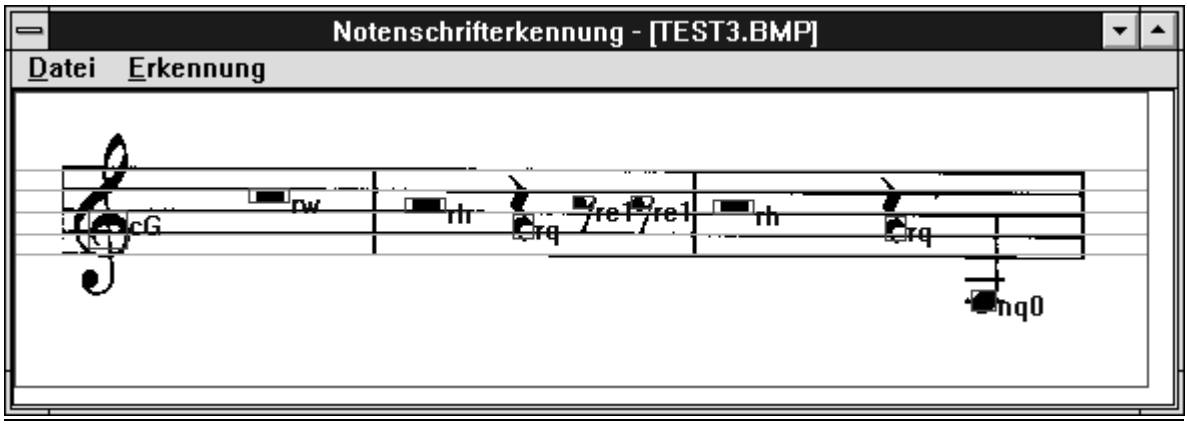


Abbildung 6-8: TEST3.BMP erkannt

## 6.5. Test: TEST4.BMP

(Auszug aus BEET2, S.249 Op. 27 No. 2, Auflösung 150 DPI)

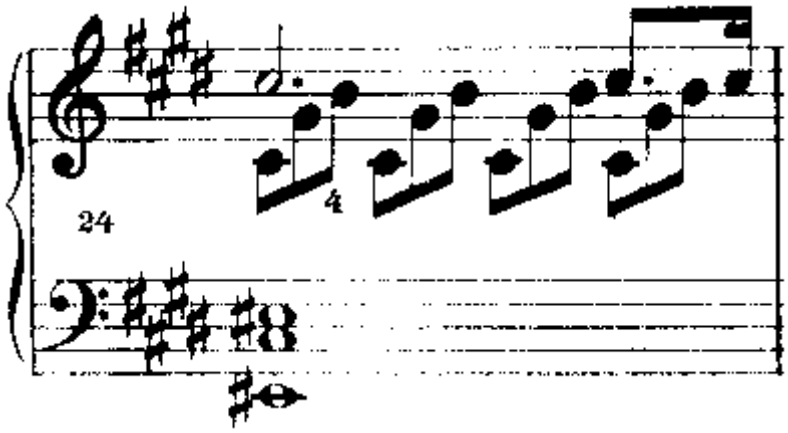


Abbildung 6-9: TEST4.BMP

Ergebnisse der Segmentierung:

| | |
|---|---|
| Notenlinien | 11 |
| Notenzeilen | 2 |
| Spuren | 2 |
| Segmente | 101 |
| Zeit | 7" |

Ergebnisse der Klassifizierung:

| | |
|---|---|
| Noten | 0 |
| Pausen | 1 |
| Notenschlüssel | 1 |
| Zeit | 7'01" |

Kommentar:

Dieses Bild wurde im Gegensatz zu den anderen mit einem schlechteren Scanner erzeugt. Die Auflösung beträgt 150 DPI (entspricht in etwa 200%) und die Einstellung des Kontrasts ist zu hoch. Alle Symbole ausgenommen der Notenlinien sind zu fett. Das Erkennungsergebnis ist daher denkbar schlecht. Da der Notenschlüssel in der 1. Notenzeile nicht erkannt werden konnte, wurde keine der Noten in der 1. Zeile klassifiziert. Alle Noten der 2. Zeile sind größer als "üblich", ein Vorzeichen des Tonartsymboles wurde für eine 1/4-Pause gehalten.

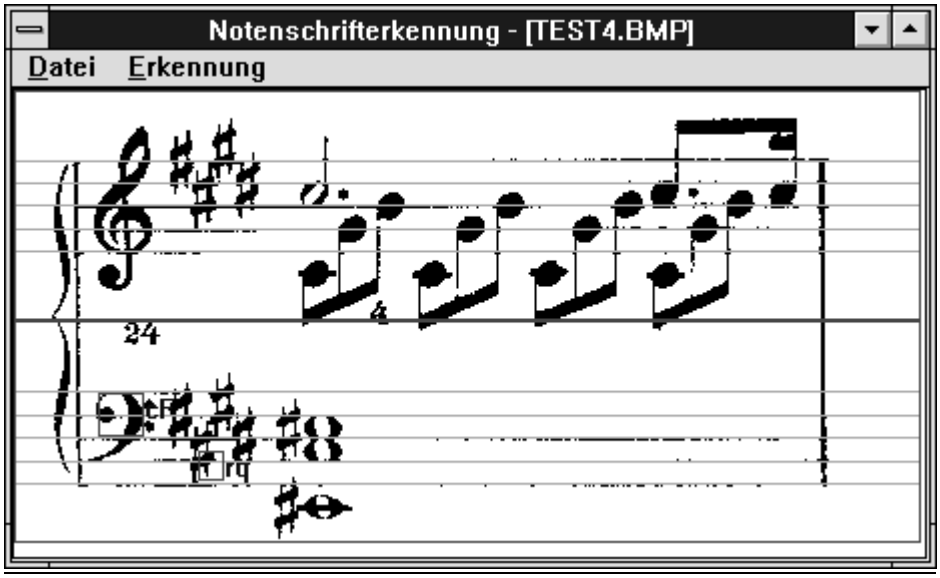


Abbildung 6-10: TEST4.BMP "erkannt"

## 6.6. Test: TEST5.BMP

(Auszug aus HARM, S. 21 "Fughette", 200% Auflösung)

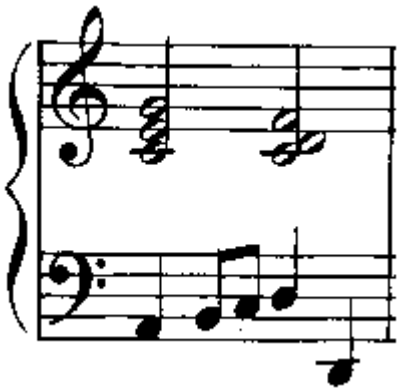

Abbildung 6-11: TEST5.BMP

Ergebnisse der Segmentierung:

| | |
|---|---|
| Notenlinien | 9 |
| Notenzeilen | 1 |
| Spuren | 2 |
| Segmente | 43 |
| Zeit | 5" |

Ergebnisse der Klassifizierung:

| | |
|---|---|
| Noten | 9 |
| Pausen | 0 |
| Notenschlüssel | 1 |
| Zeit | 1'55" |

Kommentar:

Es konnten nur 9 von 10 Notenlinien gefunden werden, da das Bild ziemlich schräg eingescannt wurde. Deshalb existiert auch nur eine Notenzeile, die Noten der oberen Zeile werden betrachtet, wie wenn sie sich auf gedachten Hilfslinien der unteren Notenzeile befänden und daher auch falsch in der Tonhöhe interpretiert. Die 2 halben Noten in der Mitte der Akkorde konnten nicht klassifiziert werden.

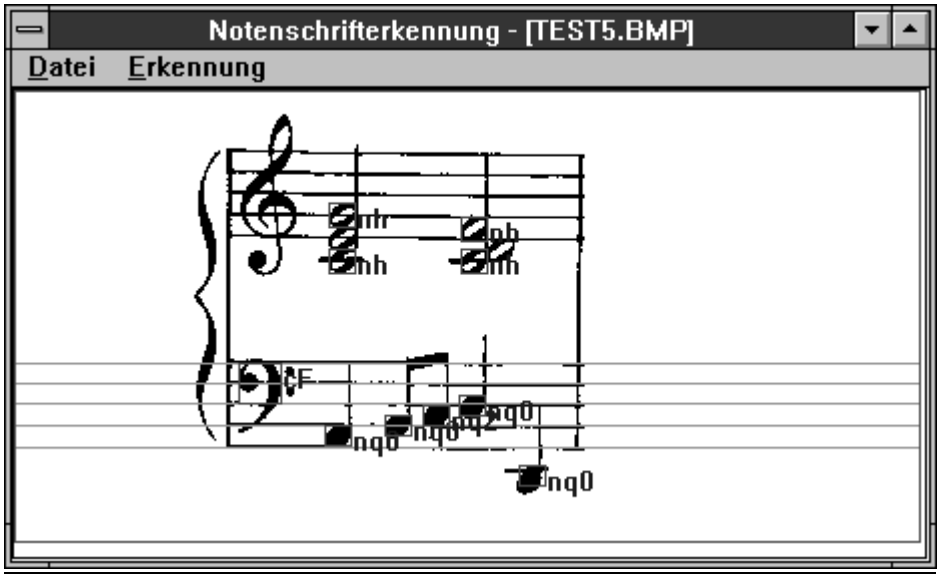


Abbildung 6-12: TEST5.BMP erkannt

## 6.7. Test: TEST6.BMP

(BACH2, S. 43 "Giguette", 200% Auflösung)

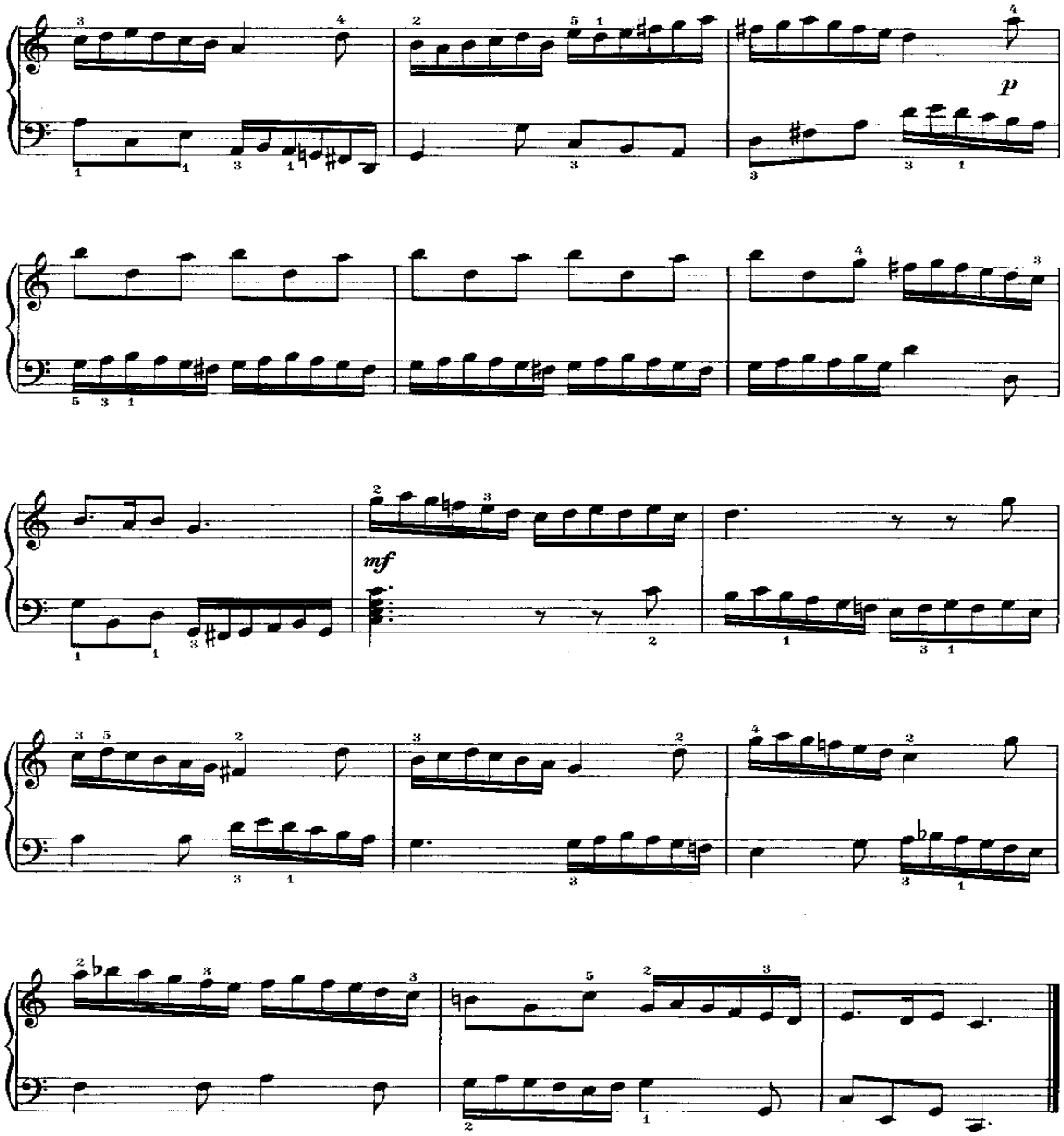


Abbildung 6-13: TEST6.BMP, ganze Seite

Ergebnisse der Segmentierung:

| | NOSE | tatsächlich |
|---|---|---|
| Notenlinien | 55 | 50 |
| Notenzeilen | 10 | 10 |
| Spuren | 2 | 2 |
| Segmente | 956 | 272 |
| Zeit | 1'57" | - |

Ergebnisse der Klassifizierung:

| | NOSE | tatsächlich |
|---|---|---|
| Noten | 239 | 236 |
| Pausen | 40 | 4 |
| Notenschlüssel | 10 | 10 |
| Zeit | 29'22" | - |

Kommentar:

Bei ganzen Notenblättern steigt mit zunehmender Schwierigkeit (Anzahl der Segmente) die benötigte Zeit für die Klassifizierung ins Unerträgliche. Auch ist die gewählte Methode zur Klassifizierung (Schablonenvergleich) nur bedingt geeignet, was man an der Anzahl der erkannten Pausen sieht, welche die tatsächliche Anzahl um den Faktor 10 übersteigt. Der Grund dafür ist darin zu suchen, daß viele Bestandteile von Noten (Balken, Fähnchen) aber auch Kreuzungspunkte von Notenlinien mit Taktstrichen oder Notenhälsen für Pausen gehalten werden. Falls sich Symbole örtlich zu nah beieinander befinden (Vorzeichen), so kommt es ebenfalls regelmäßig zu Fehlklassifikationen.

## 6.8. Test: TEST7.BMP

(HARM, S. 6 "Menuett", 200% Auflösung)

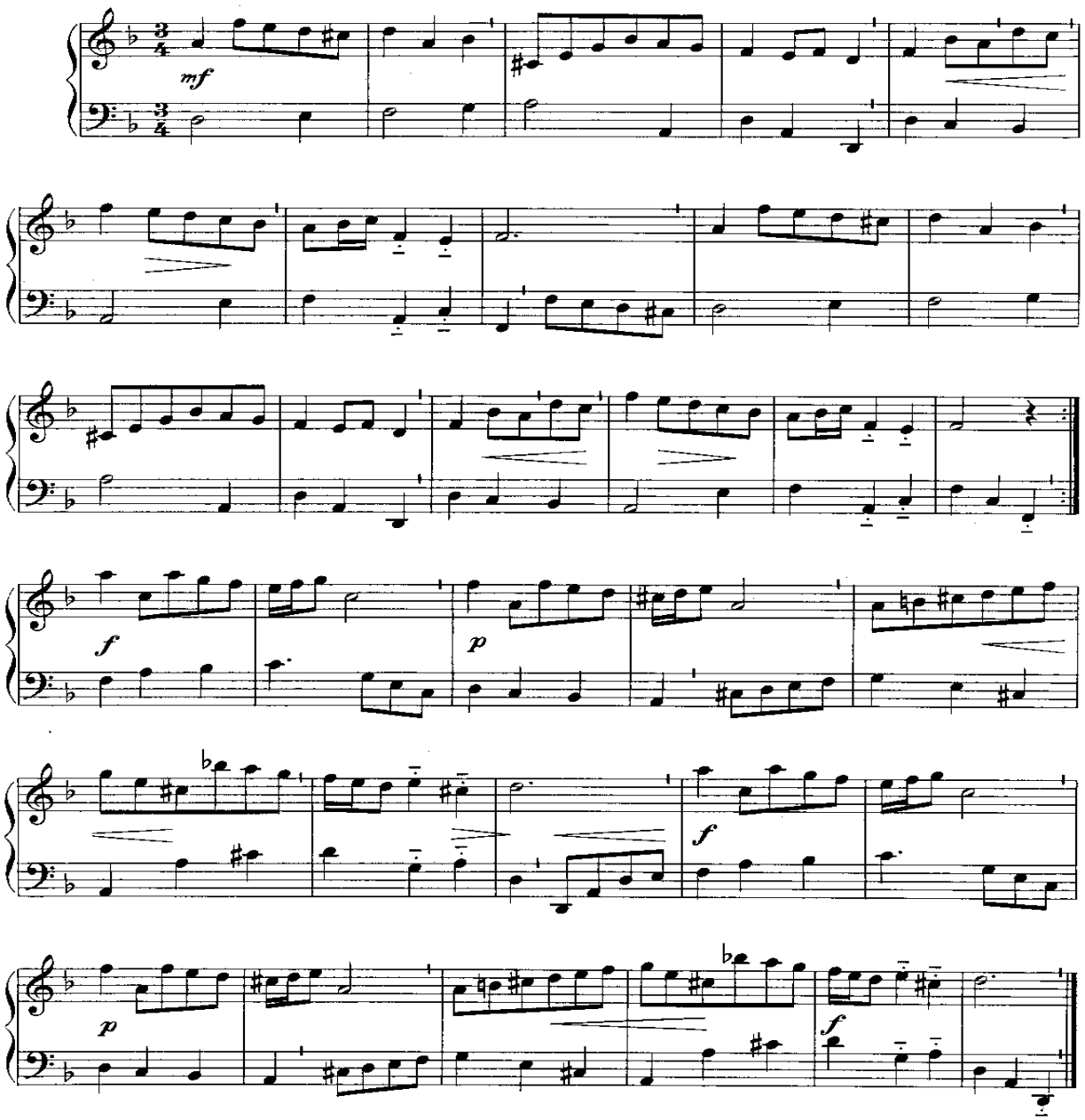

Abbildung 6-14: TEST7.BMP, ganze Seite

Ergebnisse der Segmentierung:

| | NOSE | tatsächlich |
|---|---|---|
| Notenlinien | 65 | 60 |
| Notenzeilen | 12 | 12 |
| Spuren | 2 | 2 |
| Segmente | 896 | 275 |
| Zeit | 2'02" | - |

Ergebnisse der Klassifizierung:

| | NOSE | tatsächlich |
|---|---|---|
| Noten | 239 | 238 |
| Pausen | 3 | 1 |
| Notenschlüssel | 12 | 12 |
| Zeit | 42'11" | - |

## 6.9. Test: TEST8.BMP

(PIA1, S. 9 "Rondo", 100% Auflösung)

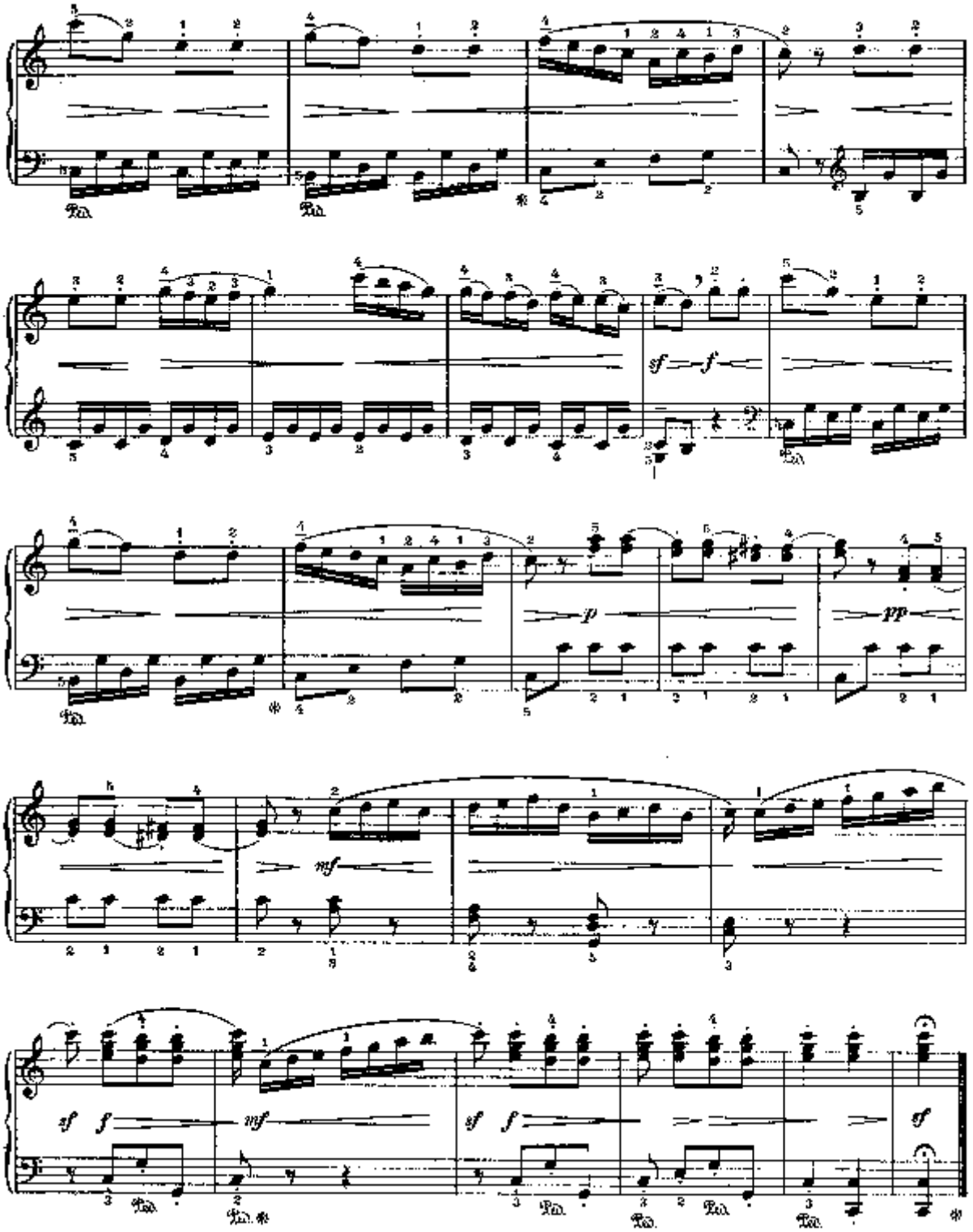

Abbildung 6-15: TEST8.BMP, ganze Seite

Ergebnisse der Segmentierung:

| | NOSE | tatsächlich |
|---|---|---|
| Notenlinien | 62 | 50 |
| Notenzeilen | 10 | 10 |
| Spuren | 2 | 2 |
| Segmente | 1087 | 304 |
| Zeit | 43" | - |

Ergebnisse der Klassifizierung:

| | NOSE | tatsächlich |
|---|---|---|
| Noten | 293 | 272 |
| Pausen | 165 | 16 |
| Notenschlüssel | 10 | 12 |
| Zeit | 5'02" | - |

Kommentar:
Eine Auflösung von 100% ist für die Klassifizierung zu wenig, die verwendete Methode zum Segmentieren ist relativ unabhängig von der Auflösung.

## 6.10. Test: TEST9.BMP

(PIA1, S. 9 "Rondo", das gleiche Notenblatt wie bei TEST8, jedoch mit einer Auflösung von 200%)

Ergebnisse der Segmentierung:

| | NOSE | tatsächlich |
|---|---|---|
| Notenlinien | 64 | 50 |
| Notenzeilen | 10 | 10 |
| Spuren | 3 | 2 |
| Segmente | 1723 | 304 |
| Zeit | 2'20" | - |

Ergebnisse der Klassifizierung:

| | NOSE | tatsächlich |
|---|---|---|
| Noten | 260 | 272 |
| Pausen | 90 | 16 |
| Notenschlüssel | 9 | 12 |
| Zeit | 1h 12'24" | - |

Kommentar:

Den Rekord für die Dauer der Klassifikation hält dieses Bild, geschlagene 1 1/4 Stunden verstrichen, bis das Ergebnis feststand.

# 7. Zusammenfassung

Das Ergebnis dieser Diplomarbeit ist ein Programmsystem mit dem Namen NOSE (NOtenSchriftErkennung), welches digitale Bildinformation in digitale Musikinformation konvertiert. Dabei werden Notenblätter im Windows-Bitmap-Dateiformat verarbeitet und als Ergebnis Standard-MIDI-Dateien erzeugt. Die Realisierung wurde unter Windows mit der Programmiersprache C++ durchgeführt. wobei besonderes Augenmerk auf die Erweiterbarkeit des Programms gelegt wurde, was sich einerseits durch die Verwendung einer Standard-Klassenbibliothek (C++/Views) für die Benutzeroberfläche, andererseits durch die sehr allgemein gehaltenen Datenstrukturen ausdrückt.

Dieses Dokument stellt eine Sammlung von Erfahrungen in einem bisher wenig dokumentierten Bereich der Mustererkennung, nämlich die Erkennung von Notenschrift, dar. Es enthält die komplette Beschreibung der gewählten Mustererkennungsverfahren, die Dokumentation der Klassen, eine Auflistung der wichtigsten am Erkennungsprozeß beteiligten Methoden und einen Bericht über die Testphase anhand von 10 ausgewählten Beispielnotationen.

Weiters werden alle bei der Entwicklung gewonnenen Ergebnisse dokumentiert, bewertet und kommentiert sowie Vergleiche mit der Schriftzeichenerkennung bezüglich des Erkennungsprozesses und der angewandten Methoden gezogen.

Nachfolgend die wichtigsten Ergebnisse der Diplomarbeit:

- Zur Segmentierung des Bildes wurde ein neuer Ansatz gewählt: *Segmentierung durch fortgesetzte X-/Y-Projektionen*. Der Vorteil dieses Verfahrens liegt darin, daß dabei ein aufwendiges Entfernen der Notenlinien ebensowenig wie digitale Filterungen des Notenblattes nötig sind. Die vertikalen Positionen der Notenlinien sowie die umschließenden Rechtecke der vermeintlichen Musiksymbole werden durch verschiedene Analysen von Projektionen auf das Bild oder auf Teilbereiche davon bestimmt und in einer geeigneten Datenstruktur abgelegt.

- Die Klassifikation der segmentierten Symbole erfolgt durch pixelweise *Vergleiche mit größennormierten Mustern*. Dabei stellt sich heraus, daß der Zeitaufwand bei zunehmender Komplexität der Notation überdurchschnittlich stark steigt. In diesem Bereich der Erkennung ist sicherlich noch Bedarf und Raum für alternative Verfahren und Methoden gegeben.

- Im Gegensatz zur Schriftzeichenerkennung wird bei der Erkennung von Notenschrift zusätzlich zur Segmentierung und Klassifikation ein weiterer Schritt eingeführt: die *Interpretation*. Sie ist einerseits nötig, da Musiksymbole, v.a. Noten, abhängig von ihrer vertikalen bzw. horizontalen Position in der Notenzeile ihre Bedeutung (Tonhöhe, Zeitpunkt des Erklingens etc.) ändern. Andererseits bringt sie auch den Vorteil mit sich, daß zukünftige Erweiterungen bzw. Änderungen, wie z.B. die mehrstimmige Interpretation oder die Abbildung der erkannten Musiksymbole auf andere Dateiformate etc., dadurch erheblich erleichtert werden.

# Anhang A: Literaturverzeichnis


[ARPM] Automatic Recognition of Printed Music
N.O. Carter, R.A. Bacon
Springer-Verlag, 1992 (Artikel in [SDIA])

[ARSTMN] Automatic Recognition of Several Types of Music Notation
T. Itagaki, M. Isogai, S. Hashimoto, S. Ohteru
Springer-Verlag, 1992 (Artikel in [SDIA])

[BACH1] J.S. Bach: Orgelwerke (Band I - IV)
F.C. Griepenkerl, F. Roitzsch
Edition Peters

[BACH2] Johann Sebastian Bach: Ausgewählte Werke (Piano)
K. Herrmann (Neuausgabe)
Edition Schott, 1960

[BACH3] Johann Sebastian Bach: Französische Suiten
J. Röntgen (Edition)
Universal-Edition

[BACH4] Bach: Das Wohltemperierte Klavier (Band II)
C. Czerny (Revidiert)
Universal-Edition, 1943

[BEET1] Ludwig van Beethoven: Leichte Klavierwerke
EFI-Ton-Verlag

[BEET2] Beethoven: Klaviersonaten (Band I, II)
H. Schenker, E. Ratz (Edition)
Universal-Edition, 1945

[BLOCK1] Schule und Spielbuch für die Blockflöte in C (Teil 1)
Rudolf Schäfer
Musikverlag Doblinger, 1974

[CSMIA] A Critical Survey of Music Image Analysis
D. Blostein, H.S. Baird
Springer-Verlag, 1992 (Artikel in [SDIA])

[DIBI] Digitale Bildverarbeitung, Grundlagen und Anwendungen
P. Haberäcker
Carl Hanser Verlag München Wien, 1987

[HARM] Harmonium-Buch J. Bonset (Edition)
Edition Schott, 1937

[HAY1] Joseph Haydn: Klaviersonaten (Band I)
W. Rauch (Reviediert)
Universal-Edition, 1947

[HEL1] Stephen Heller: Fünfundzwanig Etüden
J.V. Wöss (Edition)
Universal-Edition, 1959

[MBB] Das Moderne Bildungsbuch (Band II)
S. Hohl, I. Steinsiek (Redaktion)
Verlagsgruppe Bertelsmann, 1976

[MIDISPEZ] Die MIDI-Spezifikation
Peter Kaminski

[MOZ1] Mozart: Klaviersonaten (Band I)
K.H. Füssl, H. Scholz (Edition)
Wiener Urtext Edition, 1973

[MOZ2] Wolfgang Amadeus Mozart: Auswahlband für Klavier zu 2 Händen
Verlag Ludwig Doblinger, 1946

[MOZ3] Wolfgang Amadeus Mozart: Eine Kleine Nachtmusik
Verlag Ludwig Doblinger, 1946

[NNREMS] Neural Networks for the Recognition of Engraved Musical Scores
Ph. Martin, C. Bellissant
World Scientific, 1992 (Artikel in [NNVPR])

[NNVPR] Neural Networks in Vision and Pattern Recognition
J. Skrzypek, W. Karplus (Edition)
World Scientific, 1992

[PIA1] Von Bach bis Schumann H. Protiwinsky (Bearbeitung)
Bosworth Edition

[RSPPM] A Recognition System for Printed Piano Music Using Musical Knowledge and Constraints
H. Kato, S. Inokuchi
Springer-Verlag, 1992 (Artikel in [SDIA])

[SCHU1] Robert Schumann: Kinderszenen Op. 15
F. Goebls (Edition)
Wiener Urtext Edition, 1973

[SDIA] Structured Document Image Analysis
H.S. Baird, H. Bunke, K. Yamamoto (Edition)
Springer-Verlag, 1992

[SMF] Standard MIDI Files 1.0
The International MIDI Association, 1988

[VASK] Verfahren zur automatischen Schriftzeichenerkennung
A. Regl
Dissertation, Univerität Linz, 1985

[WZM] Wege zu Musik (Band I)
W. Knapp
Edition Heidelberg, 1981

# Lebenslauf

| | | |
|---|---|---|
| **Name:** | Wolfgang Eßmayr | |
| **geboren:** | 15. März 1968 in Linz, OÖ | |
| **Wohnort:** | A-4040 Linz, Nißlstr. 21 | |
| **Staatsbürgerschaft:** | Österreich | |
| **Eltern:** | Josef Eßmayr, geb. 26. Jänner 1928 | |
| | Therese Eßmayr, geb. Ettinger, 15. Juli 1934 | |
| **Schulbildung:** | 1974 - 1978 | Volksschule Froschberg, Linz |
| | 1978 - 1986 | Neusprachliches Gymnasium Kollegium Aloisianum, Linz |
| | 16. Juni 1986 | Matura |
| **Bundesheer:** | 1. Oktober 1986 - 1. März 1987 | |
| **Studium:** | seit 1987 | Studium der Informatik an der Johannes Kepler Universität Linz |
| **Praxis im Studium:** | 1988 - 1989 | Fa. ROSENBAUER, Erstellung von CAD-Konstruktionszeichnungen |
| | 1991 | VOEST-ALPINE Steinel, Vereinheitlichung der Debitorenstammsätze |
| | 1992 - 1993 | VOEST-ALPINE Industrieanlagenbau, Mitarbeit bei der Erstellung von Leitsystemen, Software, Graphikbildern und Strukturierungen |