A survey of computer uses in music

Maria N Russell

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation

https://digitalscholarship.unlv.edu/rtds/3345

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
A SURVEY OF COMPUTER USES IN MUSIC

by

Maria N. Russell

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
August 1997

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The Thesis of Maria N. Russell for the degree of Master of Science in Computer Science is approved.

Chairperson, Evangelos A. Yfantis, Ph.D.

Examinining Committee, John T. Minor, Ph.D.

Examinining Committee, Lawrence L. Larmore, Ph.D.

Graduate Faculty Representative, Ebrahim Salehi, Ph.D.

Dean of the Graduate College, Ronald W. Smith, Ph.D.

University of Nevada, Las Vegas
August 1997
ABSTRACT

This thesis covers research into the mathematical basis inherent in music including review of projects related to optical character recognition (OCR) of musical symbols. Research was done about fractals creating new pieces by assigning pitches to numbers. Existing musical pieces can be taken apart and reassembled creating new ideas for composers. Musical notation understanding is covered and its requirement for the recognition of a music sheet by the computer for editing and reproduction purposes is explained. The first phase of a musical OCR was created in this thesis with the recognition of staff lines on a good quality image. Modifications will need to be made to take care of noise and tilted images that may result from scanning.
# TABLE OF CONTENTS

ABSTRACT ................................................................. iii
LIST OF CHARTS ......................................................... vi
LIST OF FIGURES ........................................................ vii
CHAPTER 1 INTRODUCTION .......................................... 1
CHAPTER 2 MATHEMATICAL MUSIC .............................. 3
   Artificial Intelligence Program ................................. 3
   Human Speech Recognition ................................. 5
   Rhythm Perception ................................................. 7
   Music Analysis ..................................................... 9
   Music in Fractals .................................................. 11
CHAPTER 3 MUSIC STORAGE ......................................... 13
   Musical Instrument Digital Interface ...................... 13
   Digital Audio ....................................................... 16
   Digital Audio Performance Issues ......................... 18
   Predicting the Size of Digital Audio ....................... 19
CHAPTER 4 FILE FORMATS .......................................... 22
   BMP File Format .................................................. 23
CHAPTER 5 MUSICAL NOTATION CONSIDERATIONS ........ 27
   Music Notation Analysis ....................................... 28
   MusicEase ......................................................... 31
CHAPTER 6 RECOGNITION SOFTWARE .......................... 35
   Optical Character Recognition (OCR) ................. 35
   Musical OCR ...................................................... 35
   Applications and Common Methods ....................... 37
CHAPTER 7 OPTICAL MUSIC RECOGNITION (OMR) ......... 39
   Neural Network Method ....................................... 40
   The DoReMIDI System ......................................... 43
   DoReMIDI++ ..................................................... 46
   Recognition by Grammar ...................................... 47

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
LIST OF CHARTS

Chart 1  File Size at 11.025 KHz Sample Rate ...................................................... 20
Chart 2  File Size at 22.05 KHz Sample Rate ...................................................... 21
Chart 3  File Size at 44.1 KHz Sample Rate ...................................................... 21
Chart 4  Time Relation for Various Notes Depending on Time Signature ................. 29
LIST OF FIGURES

Figure 1 First Scanned Image Tested by Program .................................................. 78
Figure 2 Second Scanned Image Tested ............................................................... 79
Figure 3 Tilted Image ....................................................................................... 80
Figure 4 Music Book Image also Slightly Tilted .................................................. 81
Figure 5 Single Sheet of Music Scanned ............................................................... 82
Figure 6 Output File ......................................................................................... 83
CHAPTER 1

INTRODUCTION

Computer Science has been involved in many fields of research. Music is no exception. Artificial Intelligence programs have been written that create music from scratch by using mathematical formulas or fractals as well as programs that take an existing composition of music and develop a new piece by rearranging portions of the original. With the knowledge of musical analysis, composers can use these types of programs to aid in the composition of new and innovative songs.

MIDI and Digital Audio has provided options to storing sound. Digital Audio allows for the storing of the actual musical input much in the same way a tape recorder would. Because these create huge output files within a few minutes of recording, MIDI can be used to provide an alternative method. By only storing information that is needed to recreate the sound, MIDI is able to store music in much less space. This provides a type of compression that was used by products such as MIDISCAN and PIANOSCAN. These products are able to read in a scanned image of sheet music and store information needed by MIDI to reproduce the song using a limited amount of disk space. This also allows for editing and changes in the song before reprinting by first incorporating the file in a musical editing program. The techniques of Optical Character Recognition were expanded to musical notation for such products.
Many research teams have made progress in the field of Musical Optical Character Recognition. To understand the difficulty and complexity of such projects, a study of music analysis is required. Most of the work done so far in this area only recognizes a limited amount of notation which is hoped to be expanded in future attempts. First recognizing the basics of sheet music notations such as staffs, clefs, notes and rests, a recognition system can then be easily extended to include more detailed information such as dynamic markings, staccatos, and phrasing.

This thesis is a product of research and example. Several attempts are made at recognizing staff lines on scanned pieces of sheet music. Since the scanned images consisted of shades of grey, a cut off point had to be found to distinguish between what should be considered black pixels or white pixels. Various hex values were changed and tested as well as the number of blackish pixels in a row to achieve the best results. By only storing the information that is needed to reproduce the sheet of music, it is hoped that a significant amount of compression can be achieved. The appendix contains the programs used to read in a BMP file and to pick out the staff lines.
CHAPTER 2

MATHEMATICAL MUSIC

Hidden patterns in music have been investigated by mathematicians ever since Pythagoras made numerical sense of octaves, fifths, and fourths. Over the centuries, computers have played a vital role in the extension of these discoveries to mathematically composed music. Perception of speech and rhythm combined with the analysis of musical notation have helped to preserve communication and developed the field of computer generated music.

Artificial Intelligence Program

One popular method randomly generates notes and then sorts through them using an artificial-intelligence program. This program contains a set of general rules embodying the programmer's vision of what makes music musical. Eric Iverson of New Mexico State University provides a different approach. Artificial life is used to tear an existing piece of music apart, digest it, and assemble the pieces into a new composition. Artificial life covers a wide range of computer programs that create simple "automata," which are given a few rules to describe their interaction. In 1986 theoreticians at the Santa Fe Institute found that an artificial-life program could simulate a living metabolism. Chemicals acting as enzymes could digest other molecules and reassemble them in complicated cycles. Considering the musical notes as a chain of
chemicals, Iverson realized that an artificial-life program could be
designed that would metabolize a piece of music and create a new one. This
music-metabolizing program was called Metamuse. When fed a piece of music,
usually no more than a page long, Metamuse randomly extracts a string of four
notes. Using that string as a digestive enzyme, it seeks out the same sequence
of notes in the rest of the composition cutting the composition in two in the
middle of that sequence. The enzyme reproduces itself, and both copies search
for more matches. Whenever an enzyme reproduces, the chance of it mutating
(having one or more of its notes changed) exists. This process continues until
the entire composition has been reduced into fragments roughly the size of the
enzymes themselves. Metamuse then proceeds in reverse with the fragments
reassembling themselves, each one looking for two other fragments to stitch
together end-to-end. The sequence AABB, for instance, may link DFAA and
BBDC into DFAABBDCA. Once again a sequence that catalyzes a reaction
copies itself, and once again is subject to mutation. Since the fragments are not
cut too small and the enzymes are allowed to copy themselves, Metamuse
preserves patterns in the music while introducing novelty through the allowance
of mutation and by fragments being stitched together in a different order. The
program contains no rules that give the computer an idea of what constitutes
good music, so any style of music can be used. "About a third of the time you
get something you really like, and two-thirds of the time it goes off into the
ozone," Iverson says. Given a snippet of a Bach prelude or a Dizzy Gillespie
solo, Metamuse creates structure in the music and preserves, at least to some
extent, the spirit of the original although the regurgitated Bach has a twentieth-
century flare. Ultimately, Metamuse might be used to generate ideas for
composers. Such software already exists in the artificial-intelligence format, but
is limited to a particular style of music (Zimmer, 1993).
Human Speech Recognition

A research group at the Technical University of Vienna has taken a unique approach in trying to better understand how humans process speech. Features of speech signals have been previously characterized by using Fourier analysis to identify the most prominent frequencies in the recorded speech samples. This technique, however, does not capture the highly aperiodic short structures contained in speech. The research group in Vienna instead starts with mathematical descriptions of sound waveforms and listeners are asked to identify them. Comparing these mathematically defined sounds, a better understanding of how humans process speech is obtained. They suggest that the time differences between peaks may be more important than the frequency spectrum in terms of human speech recognition.

Two practical problems were faced in carrying out this research; playing arbitrary sounds for the listeners and the generation of those sounds from mathematical descriptions. Since the custom software requirements for building a digital analog converter were daunting, a Sound-Blaster card, providing CD-quality mono and stereo outputs and supporting several standard data formats, was used. The widely supported WAV file format was used to store and manipulate the synthesized sounds and several software tools were developed to easily create and manipulate WAV-format sound files.

The system, "Wave Generation Language" (WAVE-GL for short), was developed to create new sounds from mathematical descriptions. WAVE-GL is a two-layer programming system. The inner layer specifies a mathematical function that the outer layer uses when creating a sound file. The outer layer selects the sampling rate, normalizes the output amplitude, and selects the left or right channel for stereo output. Data is written to the file as it is generated, allowing the length of the output file to be limited only by disk space availability.
By repeatedly using DEFFUNC to define a new formula and GENERATE TONE to create a sound using that new formula, a variety of sounds can be added to the file. WAVE-GL is implemented in C++. WAVEGEN and FUNCGEN uses scanners and parsers that were built with FLEX++ and BISON++ generating them as C++ classes.

In WAVEGEN, every statement is executed as it is read whereas FUNCGEN is a bit more sophisticated. When DEFFUNC is executed by WAVEGEN, the inclosed source is interpreted and stored as a parse tree, so that the function source need not be reread and recompiled. The FUNCGEN object contains two symbol tables; one being a simple array that holds variables with single-character names and the other using a hash structure. Variables with single-character names are not declared and can be preset before the GENERATE TONE statement appears. Those stored in the hash table must be declared before use. The next version of WAVE-GL will treat formulas as functions so that complex descriptions can be decomposed. A parse-tree optimizer is also in the works.

The next step looks toward the investigation of distorting a speech signal to the max while still remaining understandable. These investigations help in better understanding how the brain decodes sounds, and provides insight into better compression of speech. Software tools will be expanded such as a planned WAV-TOOLBOX including a variety of filters, sound editing tools, and a tool to graphically display WAV files. Based on the portable WAV sound file format and easily-obtainable hardware, many researchers should find this software system useful (Lutter, Müller-Wernhart, Ramharter, Rattay, and Slowik, 1996).
Rhythm Perception

A model of the human perception of rhythm, based on Lerdahl and Jackendoff's A Generative Theory of Tonal Music, was simulated by computer and called GTSIM. Being a rule-based model with a neural network component, GTSIM simulates the metric and grouping stages of Lerdahl and Jackendoff's theory. Parallel processing is simulated using a rule-based production system operating left-to-right without backtracking. GTSIM identifies the key of the piece of music and produces hierarchical representations of the metric and grouping structure according to Lerdahl and Jackendoff's theory. An attempt to formalize the intuitions of a human listener about classical Western tonal music is achieved by GTSIM.

The program's input is a symbolic representation of what a listener immediately hears in music - pitch, note duration, and dynamic level. Information, such as key, time signature, or placement of bar lines, which must all be derived from listening are not considered on input. This limitation categorizes their work into that of perception rather than score-reading. The internal representation of the music consists of a list of nodes corresponding to successive events in the score called the Notochord. The Notochord is linked to a matrix of note nodes, each representing a note in the score as well as to the grouping and meter hierarchies.

A modular rule-based production system is the core of GTSIM which uses a Hearsay II blackboard type of architecture. Three basic components exist in a blackboard system:

1) a global data structure called the blackboard;
2) a set of Knowledge Sources (KSs), each of which encapsulates knowledge about some aspect of the problem;
3) A control structure that monitors and controls which Knowledge Sources are active.

The Notochord is the blackboard containing links to information about the analysis of the piece.

Metric analysis is a matter of constructing a legal metric hierarchy and one that most closely fits the music. Two psychological models of meter extraction have been developed. One model, first implemented in a computer program known as BEATS, has been integrated as a KS in GTSIM. A family of grids is generated by this model constituting a well-formed metric hierarchy. BEATS operates in a single pass based on metric hypothesis testing with a minimum of backtracking. A duration's experience yields a hypothesis that this duration is a level in the metric hierarchy, which will by future events be either confirmed or rejected. Rules contained in BEATS create new metric levels from existing ones.

Our second model of meter extraction, BeatNet, is a network model producing an indication of the most active metric levels at any point in the piece of music. BeatNet has its foundation on the idea that every level in a metric hierarchy corresponds to the duration and phase of a single note or small group of adjacent notes in the piece. Therefore by determining the durations of notes and note groups, a correct metric hierarchy can be constructed. BeatNet attempts to identify the durations found in the score belonging to the meter. While BEATS directly implements GTTM's well-formedness rules, BeatNet contains no knowledge, rules, expectations, or goals. Instead, its response is determined upon the global interaction of simple local processes in complex, hard-to-predict ways.

Both meter and grouping can be affected by the perceived key of a piece. Generally, strong beats in the metric structure fall on the important pitches of the
key. Implementing the pitch grouping rule requires that the key of the piece of music be determined. Counting the frequency of the notes of the chromatic scale, the most common notes generally identify the key of the piece. Implemented by a simple network, a modified version of this counting algorithm is incorporated into GTSIM as a KS. Each note of the chromatic scale is represented by a "pitch-node" connected to "chord-nodes" that receives input from just three pitch-nodes. Three sets of chord-nodes representing the tonic, subdominant, and dominant chords of a particular key are connected to "key-nodes". Pitch-nodes are activated, as the analysis of a piece progresses, to the degree of the occurrence of their notes in the input. These pitch-nodes activate the chord-nodes, which in turn activate key-nodes. This network successfully identifies the tonality of a large number of pieces of music. For example given the piece Auld Lang Syne, it correctly guesses the key on the first note. The piece is in the key of F, with the first note being C, the fifth degree of the scale, an important note occurring in both the tonic and dominant chords of the key of F. The particular choice of weights made F the most plausible key which later was confirmed. In many other cases, only a few notes are needed before the network correctly identifies the key. Modulations can also be distinguished.

Currently an expansion of the model is being developed to recognize parallelism in music by encoding musical sequences and constructing a hierarchy. To recognize parallelism in music, a discrimination net approach as well as alphabet encoding schemes are being explored (Jones, Scarborough, and Miller, 1993).

Music Analysis

Programming of music analysis is described next using an object oriented approach. Different structural analysis range from simple melodic analysis,
through harmonic analysis to cluster analysis used for analysis of modern music. The programming language SIMULA was developed between the period 1975 to 1979 as an object oriented programming environment for structural analysis of music and named MUSIKUS. An approach, by Lande and Vollsnes (1995), to the analysis of computer-aided music has been concerned with the question of musical style. So far they have dealt with music from the Western tradition and folk music. This research dealt with examining compositional practice and the "laws" of musical construction, eventually being able to describe aspects of style. Every work can not be analyzed in one definitive way and constructing a program which will perform a complete analysis is not realistic. Rather, the computer and the programs are "preanalytical" tools which make it possible to work systematically through a vast database of information.

Research consists of scores entered into a large database normally by playing them through a MIDI-keyboard to a sequencer program, where the data are stored. The recorded music is then translated to a dedicated code for music analysis. The MUSIKUS System was designed to provide an easy and reliable programming environment suitable for music analysis programs. Such a system's success depends on the versatility of the music model, the way of viewing music and capturing the structural properties, on which the programming environment is based.

The music in the MUSIKUS system is represented in a special purpose code called MUSIKODE, which was developed in 1974. Being a fairly general code for music representation, this representation uses translators to and from MIDI, making it possible to proofread or to prooflisten the music. In addition, MUSIKODE organizes coded music in a simple database and the MUSIKUS system contains specially developed objects for reading this code. The user-
dialogue and database search is hidden in the MUSIKODE-object (Lande and Vollnes, 1995).

Music in Fractals

Musical results provided by fractals have become one of the most exciting new fields of music research.

Music has been studied mathematically dating back to ancient Greece when musical intervals were expressed as numeric proportions by the Pythagoreans. Joseph Schillinger, in the 1920's and 1930's, began to use mathematics to develop a scientification of music. His research was later published in a twelve book work called "The Schillinger System of Musical Composition". The theory of rhythm was dealt with in the first book. As a way of finding simple rhythms, a new wave was formed combining the two waves of different periodicities. Distributive powers provided another means of producing rhythmical patterns. By taking a series of fractions that add to one in the form \((a + b)\) and squaring it, a new pattern is created. Schillinger also squared and cubed higher order polynomials in a similar manner leading to richer rhythmic patterns (Degazio, 1997).

In the mid-1970's, Richard Voss and John Clarke performed a more general mathematical study of music. Instead of studying the written structure of music, the actual audio physical sound of the music was studied. Several recordings of music and speech were analyzed by Voss and Clarke (1975), first examining the spectral density of the audio signal itself. The quantity called the audio power of music was proportional to the power delivered to the speakers rather than the voltage and was examined. An interesting result occurred as the audio power seemed to display \(1/f\) behavior. Midway between white noise \((1/f^0)\) and Brownian noise \((1/f^2)\), \(1/f\) has been found in other phenomena as a
spectral density such as electronic flicker noise, sunspot activity, the wobbling of the Earth's axis as well as the flood levels of the river Nile. The 1/f behavior held for completely different kinds of music. Voss and Clarke (1975) analyzed Bach's First Brandenburg Concerto, a recording of Scott Joplin piano rags, as well as recordings from a rock radio station, classical station and a news and talk station; all of which demonstrated the 1/f behavior.
CHAPTER 3

MUSIC STORAGE

Two different ways of creating and recording sound on the computer are MIDI and Digital Audio. MIDI actually records a series of commands such as keystrokes made and various instruments selected. Playing back the file causes the sound card or MIDI synthesizer to re-create the music by replaying these commands. Providing different commands changes the outcome of the song. Digital Audio is comparable to recording with a tape deck. Actual sounds are recorded, not commands, and stored on the computer's hard disk (instead of on audio tape). The advantage of Digital Audio is that the limitation on the sounds of a sound card or synthesizer can create does not exist. Anything that a microphone can pick up, from vocals to an electric guitar can be recorded using Digital Audio (Digital Orchestrator Plus Reference Guide, 1996).

Musical Instrument Digital Interface

Musical Instrument Digital Interface (MIDI), is a system for encoding, sending and receiving electronic messages which control MIDI devices. A MIDI device, therefore, is anything that generates or responds to MIDI messages. Common MIDI devices are electronic musical synthesizers (synths), keyboards, and drum machines. A computer with its sound card can be programmed to
respond to MIDI commands and specialized devices, such as theatrical lighting systems, can be controlled. MIDI has been standardized by widely accepted rules introduced in the early 1980's by the MIDI Manufacturers Association. These rules specify how MIDI data should be encoded and sent among MIDI devices as well as govern hardware issues such as designing of cables and connectors used by MIDI devices. A series of MIDI commands, called a sequence, can be organized in a deliberate way to instruct a MIDI synthesizer to play a musical passage. A sequencer is a computer software program or hardware device that creates and edits sequences. Sequencer software allows the computer to record, edit, or play MIDI sequences and save them as files on a disk, which then instructs a synthesizer to play the music. Through the use of a single MIDI sequence, a number of MIDI devices can be controlled; an entire synthesized orchestra for instance. Certain procedures are required to keep things organized. An instruction assigning a command to a MIDI port is included in each MIDI command to ensure that the correct device is reached. If a command is assigned to port 1, only the MIDI device attached to port 1 receives the command. One of 16 MIDI channels is also assigned to each command routed to a given port. A MIDI device can receive incoming signals on all channels, but respond only to selected channels. A synth, however, can "tune in" more than one channel at a time. Each channel is assigned a patch or instruction telling the synthesizer which particular type of instrumental sound to generate for that channel. So, assigning an oboe patch to port 2, channel 4, creates an oboe sound for any note directed to port 2, channel 4 until the patch assignment is changed. Two kinds of events can be triggered by MIDI commands: note events and non-note events. A note event instructs the synthesizer to play a particular pitch at a precisely defined time much in the same way as pressing a key on the piano. The command can convey such
musical nuances as how abruptly or hard a piano key is struck, held down, or released. However, not all MIDI devices are designed to respond to this type of information. Non-note events (sometimes called "controller events" or simply "controllers") manage other functions of the MIDI device such as adjusting the volume or assigning a new patch to one of the channels. MIDI files contain a set of instructions that tell a synthesizer which sounds to make and when to make them. In this sense, a MIDI device can be thought of as a player piano with the perforated paper roll that control it being a MIDI sequence. A piano will play middle C provided that one of the holes on the paper roll instructed it to do so. The ultimate result, however, depends on the quality and condition of the piano. If the middle C string is tuned to E-flat, then E-flat will be heard. Since synthesizers can simulate many different instruments, the possibility of hearing a flute sound instead of a piano, for instance, is great.

A balance between standardization and versatility was attempted by the MIDI specification upon it's first release. MIDI manufacturers, software programmers, and composers agreed on certain basic commands, but other parts of the specification were less rigidly defined and led to the emergence of several fundamental problems:

**File Format** - Each MIDI software program developed its own way of storing sequences on computer disks. A MIDI file, therefore, could only be played by the program that created it.

**Patches** - The commands that tell the synthesizer which patches (instrument sounds) to use are really just numbers. Standardizing a numbering scheme was developed with patch numbers running from 0 to 127. Governing which sound went with which number, however, was not regulated. Consequently, a violin part on one synthesizer might result in a trumpet sound when played on another.
Capabilities of the Synthesizer- All hardware has inherent limits including a limited number of sounds that a synthesizer can make and play at once. These capabilities vary widely. A sequence created on a relatively powerful synthesizer that can simultaneously generate many notes and instruments, can overwhelm a lesser synthesizer.

As the market grew, problems multiplied. Patch and drum maps varied from one synthesizer manufacturer to another, or even among different models from the same manufacturer.

More fully defined conventions are being established to help composers create device-independent MIDI files. The MIDI Manufacturing Association has offered a specification called General MIDI which standardizes, among other things, instrument and drum sounds. Composers now have the ability to tailor their arrangements to the synthesizers differing capabilities due to the division of synthesizers into two types: Base-Level and Extended-Level by the Multimedia PC Marketing Council (MPC). Adopting a new standard will still leave incompatibility problems for most of the MIDI devices and files already in use. Microsoft created the MIDI Mapper as an interim solution. This software utility for Windows translates MIDI data as it is played, allowing newer files to work with older synthesizers. The MIDI Mapper has recently been replaced with the MIDI Configuration utility upon the release of Windows 95. As new MIDI standards emerge, the Configuration utility will be updated.

Digital Audio

Digital Audio records sound instead of a sequence of commands as in MIDI. The hard drive is the storage medium and serves much the same function as tape in a conventional tape recorder. Sound, occurring in nature as a series of waves, must be converted to a series of numbers that in turn can be read by a
computer. To accomplish this, a technique called Pulse Code Modulation, or PCM is used by most PC-based Digital Audio hardware. Incoming electrical signals in the form of waves (analog signals), such as those from a microphone, are turned into numbers by a circuit called an Analog to Digital Converter (ADC) and saved in computer memory. These numbers can be saved in a file, and manipulated as easily as text in a word processor. When you play the file, a Digital to Analog Converter (DAC) converts the numbers back into analog electrical signals. These signals are then amplified by audio equipment and sent to speakers or headphones.

Wave files, identified by the .WAV file extension, are the most commonly used Digital Audio files by Windows. Another widely supported type is the .VOC file originally developed by Creative Labs for their Sound Blaster products. Among Digital Audio files, various data formats exist each having four defining characteristics:

**Sample Rate**- To record sound, a sample (a discrete instant in the sound wave) must be selected and measured by the ADC and its amplitude (a measure of its loudness) stored as a number in the computer. This process is rapidly repeated in order to capture a sustained interval of sound. The frequency with which the sampling process occurs is called the sample rate which is generally expressed in Kilohertz, or thousands of cycles per second. A typical sampling of a Digital Audio recording might be at the rate of 11,025 Hz (samples per second), or 11.025 KHz. Higher sample rates produce better sound fidelity in general, but can generate so much data that it outpaces the processor of the computer or overruns the capacity of its hard disk.

**Bit Length**- Each sample has an amplitude expressed as a number and, stored in a computer, as binary numbers. The numbers of binary digits...
that make up each sample is called the bit resolution (also called "bit
length" or "sample size"). Eight bits can represent a range of values from
0 to 255, while sixteen bits can represent values from 0 to 65,535. 16 bits
has become the standard for most Digital Audio programs. Larger bit
lengths allows Digital Audio data to be measured more precisely. As an
illustration, imagine two tape measures both 10 feet long, but one marked
with 255 divisions, the other with 65,535. Both may be accurate, but the
latter is more precise. Better sound fidelity results from larger bit lengths
at the expense of processing more data.

Compression- To speed processing of large amounts of data, data
compression is used by many sound cards to reduce the size of the data
files. Before storing the data, some of it is removed and artificially
restored on playback. Compressed data cannot be edited, however,
exporting recorded data as a WAVE file allows the use of compression
software on the resulting data. The system on which the file will be
played must contain the necessary hardware and software to work with
the compression format selected.

Mono or Stereo- Requiring twice as much data as its mono equivalent, a
stereo Digital Audio file consists of two sets of data working together, one
assigned to the left channel, the other to the right. The issues of
computer system speed and hard disk storage capacity are faced
whenever working with stereo files.

Digital Audio Performance Issues

Sound quality depends on the sample rate and bit length and therefore
results in a tradeoff between these and the file size. For example, two minutes
of 16 bits per sample, one track recording at 11.025 KHz requires about 2.5
Megabytes of hard drive space whereas 44.1 KHz sample rate requires about 11 Megabytes (see charts 1, 2, and 3). Some problems may occur due to limitations of the computer system such as Digital Audio tracks at high resolutions that are too long may not successfully record or sound may break up on playback.

Predicting the Size of Digital Audio

Four things about the file is needed in order to predict the approximate amount of disk storage space a Digital Audio file will require.

B-Bits  The length of a single sample
C-Number of Channels  1 for mono, 2 for stereo
T-Time  The length of recording, in seconds
R-Sample Rate in Hertz ( samples per second )

The following equation will predict Digital Audio file size,

\[(B/8)xCxTxR=\text{File size, in bytes}\]

For example, placing the following values into the formula:

\[B=16 \text{ (bits)}\]
\[C=1 \text{ (Mono)}\]
\[T=14 \text{ (seconds)}\]
\[R=22,050 \text{ (samples per second)}\]

and solving the equation gives:

\[(16/8)x1x14x22,050=617,400 \text{ bytes}\]

To convert this answer from bytes to kilobytes, divide the result by 1024.

\[617,400/1024 = 602.929 \text{ or about 603KB}\]

This answer is still an approximation. A file header must be added to the file, which adds bytes, and, of course, you have to multiply by the number of tracks you wish to record.
The charts 1, 2 and 3 show approximate file sizes (in megabytes) generated when recording 16-bit Digital Audio to different sample rates.

Note that for No. of Tracks:

1 track = 1 mono track
2 tracks = 1 stereo track
3 tracks = 1 mono track and 1 stereo track

**CHART 1 File Size at 11.025 KHz Sample Rate**

![File Size at 11.025 KHz Sample Rate](image)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHART 2  File Size at 22.05 KHz Sample Rate

CHART 3  File Size at 44.1 KHz Sample Rate
CHAPTER 4

FILE FORMATS

Two types of graphic files exist: vector and raster. Vector graphics describe shapes mathematically relying on a language to draw a graphic, whereas bitmap (raster) files describe shapes as a pattern of dots or pixels. Converting vector graphics into raster graphics is achieved by processing the vector information as if the output was saved and printed as a raster graphic. Consequently, vector files are not tied to a particular resolution and raster images of any size can be generated from vector files. Unfortunately, converting a raster graphic into a vector graphic isn't as simple. The computer interprets the patterns of pixels in the raster image as shapes and then writes the directions to draw them. Some tracing programs can do this conversion but with less than perfect results. By including the raster information in the vector graphic, a raster graphic can be embedded since a vector graphic is simply a description of how an image is to be drawn. A shape is created and given a fill-the raster graphic. For example, saving a Photoshop file as an Encapsulated PostScript format (EPS) embeds a raster graphic inside a vector graphic. However since a raster graphic can't contain vector information, a vector graphic can never be embedded in a raster graphic. Converting the vector image to raster information first and then consolidating the two becomes the only way of
combining a vector graphic with a raster graphic (Chamberlain, 1995).

BMP File Format

The BMP file format is the Microsoft Windows Bitmap format for Device Independent Bitmaps (DIBs) containing images with 1, 4, 8, or 24 bits per pixel. Two incompatible versions of this format exist, one introduced with OS/2 version 1.x and another introduced with Windows 3.0. The Windows 3.0 format being newer and more common, is the one described here. The BMP format supports run-length encoding (RLE) for 4 and 8 bit per pixel images, but is rarely used. For optimal file access, the file is padded in such a way as to be evenly divisible by 4 bytes. It follows the Intel byte ordering; that is least significant bit first. Being popular on Intel based personal computers, the BMP format is supported by Microsoft applications and optimized for Intel processors access. A fairly simple format and high availability on other systems makes BMP a good choice for new applications. The BMP is composed of three parts: The header, followed by the color map, then the pixel data. With a fixed size of 54 Bytes, the header contains the following information, the most important being the first and second bytes containing 'B' and 'M' in that order. The 18'th byte holds the image's width then the length both 4 byte double words. The number of bits per pixel, either 1 bit, 4 bit, (older machines) 8 bit (pseudocolor), or 24 bit (true color), fills the words contained at the 28'th byte. At the 46'th byte is a 4 byte double word telling the number of colors in the color table. A value of zero represents true color with the pixel data beginning after the 54'th byte. Four unsigned characters representing the blue, green, and red components specify the BMP palette entries; the fourth byte being reserved. The RGB specification's odd ordering and reserved byte allows proper ordering of the loaded entries into
an integer on Intel processors. The image contains a palette entry for each

color; the important colors represented first (Meyer, 1997).

The following shows the form of a typical BMP file:

BITMAPFILEHEADER bmfh;
BITMAPINFOHEADER bmih;
RGBQUAD aColors[ ];
BYTE aBitmapBits[ ];

The bitmap-file header, defined as a BITMAPFILEHEADER structure, contains
information about the type, size, and layout of a device-independent bitmap file.
The bitmap-information header, BITMAPINFOHEADER structure, provides such
specifications as the bitmap's dimensions, compression type, and color format.
The color table, depicted as an array of RGBQUAD structures, contains the
same number of elements as colors in the bitmap. For bitmaps with 24 color
bits, the color table is not present since each pixel is represented by 24-bit red-
green-blue (RGB) values in the actual bitmap data area. The colors in the table
appear in order of importance so that a display driver can render a bitmap on a
device that may not have the ability to display as many colors as presented in
the bitmap. The importance of the colors can be determined by the
biClrImportant member of the BITMAPINFOHEADER structure provided the DIB
is in Windows versions 3.0 or later format. The BITMAPINFO structure acts as a
combined bitmap-information header and color table. An array of BYTE values
representing consecutive rows, or "scan lines," of the bitmap immediately follow
the color table. Each scan line consists of consecutive bytes representing the
pixels in the scan line, in left-to-right order. A scan line can be represented by
any number of bytes depending on the color format and the width, in pixels, of
the bitmap. A zero-padding of the scan line may be necessary to ensure ending
on a 32-bit boundary. Consequently, segment boundaries can appear anywhere

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
in the bitmap. In contrast to most graphics formats the scan lines in the bitmap are stored from bottom up, meaning the pixels in the lower-left corner of the bitmap are represented by the first byte in the array and the last byte represents the pixels in the upper-right corner. The biBitCount member of the BITMAPINFOHEADER structure determines the number of bits defining each pixel and the maximum number of colors in the bitmap. Any of the following values can be used:

1. **Monochrome bitmap and a color table that contains two entries.** A pixel is represented by each bit in the bitmap array. A clear bit indicates a pixel displaying the color of the first entry in the color table. A set bit dictates that the pixel's color is that of the second entry in the table.

2. **Bitmap contains a maximum of 16 colors.** Each pixel being represented by a 4-bit index into the color table. For example, a value of 0x1F in the first byte in the bitmap, represents two pixels. The first pixel contains the color in the second table entry, while the second pixel contains the color in the sixteenth table entry.

3. **Bitmap has a maximum of 256 colors.** A 1-byte index into the color table is used for representing each pixel in the bitmap. For example, if the first byte in the bitmap contains 0x1F, the first pixel has the color of the thirty-second table entry.

4. **Bitmap has a maximum of 2^24 colors.** The bmiColors (or bmciColors) member is NULL, and the relative intensities of red, green, and blue, respectively, for each pixel is represented by a 3-byte sequence in the bitmap array.

The number of color indexes in the color table that are actually used by the bitmap is specified by the biClrUsed member of the BITMAPINFOHEADER structure. A value of zero in the biClrUsed member signifies that the bitmap
uses the maximum number of colors corresponding to the value of the biBitCount member. An alternative form of bitmap file uses the BITMAPCOREINFO, BITMAPCOREHEADER, and RGBTRIPLE structures (The Graphics File Formats, 1997).
CHAPTER 5

MUSICAL NOTATION CONSIDERATIONS

Producing publication-quality scores can be handled by music-score editing programs that contain not only notation capabilities but many powerful automation features as well. Users, however, must understand the positioning of many notational elements such as deciding whether to place a slur above or below the affected notes or if a tuplet should appear between the slur and the notes or outside it. Unfortunately, rules that a computer can use for notating music does not exist. Books, for the most part, describe these rules through examples. Notational software has little problem dealing with individual rules, but the interaction between the rules becomes difficult to capture. Common music notation is not unique. Old notations are adapted to new uses and new notations are often invented as needed by composers and publishers. National dialects of music notation exists as well as many different levels of notational complexity. Thus, a single recognition system capable of recognizing all music notation may not be possible to create. Varying in orientation, appearance and positioning, music symbols are much less regular than the characters in printed text. Further complicating the recognition process are adjacent and overlapping symbol placements (Blostein and Baird, 1992).
Music Notation Analysis

To obtain a readable description of music, music notation uses different parameters such as pitch and duration which are determined by the position and shape of note symbols or additional symbols and words to express tempo, dynamics, expression, etc. A note is one of the fundamental symbols in music notation with its vertical position representing its pitch, and the shape corresponding to its duration. A note contains building components such as different head shapes, stem and flags or beams. The stem of a note head can be placed depending on the direction of the stem being either upward (right side) or downwards (left side), whereas the flags are always positioned at the right side of a stem. Beamed notes are used to replace groups of flagged notes for clearer readability. A variety of rests exists that corresponds to each note value. Other symbols give additional information about pitch, tempo, volume or articulation.

The note's duration is illustrated by a note head being hollow or solid, containing a vertical stem, perhaps ending with a curved flag or not, or the presence of one or two dots following the note increasing its duration. Rests indicate places of silence and come in different shapes depending on their durations. Basic durations are expressed as fractions of a whole note, which is represented by a hollow oval with no stem. Smaller grace notes are considered to have no duration and should be played relatively fast. Chords contain a number of notes sounded simultaneously occupying the same horizontal position on a staff, while having different vertical positions.

Five parallel lines constitute a staff on which note heads are positioned either on the staff lines or between two consecutive staff lines to determine the relative pitch of the associated note. In an orchestral score, each staff generally indicates the part for a different instrument such as first violin, second violin,
viola, and so on. Keyboard instruments such as a piano generally use two separate staffs to represent the parts for each hand. A written note illustrated as an oval note head uses its vertical staff position and an optional accidental sign to identify the pitch it represents. Extra leger lines may be required if the note head represents a pitch outside the staff's range. Vertical bar lines subdivide a staff into units called measures.

At the beginning of each staff is a clef sign, either bass or treble for piano music, which gives a basic interpretation of pitches. Following the clef is the key signature which indicates the key of the piece of music and tells which notes are automatically sharped or flatted if any. The time signature is located at the right side of the clef and key signature and is represented by a nominator (telling how many beats to a measure) and denominator (describing which type of note receives one beat) (See Chart 4). Several voices, whether representing a human voice or a single instrument, may be printed together on one staff. Staves meant to be played in parallel are grouped together by a continuous left hand bar line.

| Time Signature Number indicating which notes gets 1 beat: |
|-------------|---|---|---|---|
| 2           | 4 | 8 | 16 |
| 1/2         | .25 | .5 | 1  |
| 1/4         | .125 | .25 | .5 | 1  |
| 1/8         | .0625 | .125 | .25 | .5 | 1  |
| 1/16        | .03125 | .0625 | .125 | .25 | .5 | 1  |
| Number of beats or fraction of a beat each note or equivalent rest gets. |

CHART 4 Time Relation for Various Notes Depending on Time Signature
A score is made up of a set of such systems with additional information about the composer, title of composition, page number, etc. printed on the heading of the first page (Baumann, 1995). Associated with the notes are lyrics or words that the performer will sing at the pitch and duration specified by the note. A score will usually hyphenate lyrics centering the syllables under the notes with which they are associated.

A tie, represented by a curved line, connects two notes of the same pitch across metric boundaries. Measures are created by dividing each staff with vertical bars. The sum of the durations of the notes and rests in each measure is the same for all measures with the same time signature. Abbreviations such as "p" and "crec" and horizontal wedges specify dynamics.

Besides the basics of staves, notes, and rests, a score can also indicate how the notes should be played by a performer. Upper-voice notation appears above the staff, while lower-voice notation usually appears below the staff. To convey to the performer a better feel of the rhythm, flagged notes are often joined by one or more horizontal beams. The groupings depend on the durations of the notes, their placement within the measure, and the meter. A downward stem is placed on a note if its note head lies above the middle staff line. An upward stem on one of the notes in a group indicates that the beam runs above the grouping, otherwise it appears below. Beams are half as wide as the space from one staff line to the next, and its slant should follow the general trend of the grouped note heads, but at a shallower angle. The beam becomes horizontal if the note heads lie closer together than normal due to tight spacing.

Articulation signs generally lie between the slur and the affected note head. To shorten a note, a period is placed above or below a note indicating staccato. Similarly, a dash indicates tenuto, a lengthened note. To represent a percussive accent, the characters resembling greater-than signs are used.
When the starting or ending note of a slur has a percussive accent, the slur should generally appear between the note and the accent mark. Three notes can be indicated to be played in the time of two. Called triplets, a small numeral "3" and an optional bracket is placed above or below the notes. This concept, generically called a tuplet, extends to any number of notes. Following the rules for slurs, tuplet brackets extend over or under a group of notes. The tuplet bracket can be omitted if the notes are fully beamed, that is the start and end of the beam correspond with those of the tuplet. As with slurs and beams, tuplet brackets should slant in accordance with the general slant of the note heads.

MusicEase

MusicEase, a music editing program, was introduced to help speed up the notation process allowing copyist to make fewer decisions or enter less data. Applying constraints to notational information that the user enters, MusicEase correctly arranges the graphical elements automatically.

MusicEase calculates the complex interaction of notational elements. Basically, it is a WYSIWYG editor with a menu system for novice users and keyboard shortcuts for advanced users containing such features as zooming and page preview of a modern word processor. It also imports and exports standard Musical Instrument Digital Interface (MIDI) files. It was written using muLisp, a MS-DOS version of Lisp. MusicEase automatically creates leger lines when the user enters a note beyond the staff's range. By applying a set of constraints, MusicEase handles the interaction of the various musical elements.

Items are positioned by MusicEase in accordance to their order in the following list:

1. Note heads
2. Beams and stem directions and lengths
3. Articulation symbols that, if a slur is present, generally lie between the slur and their associated note heads
4. Slurs
5. Articulation symbols that, if a slur is present, generally lie on the side of the slur away from their associated note heads.
6. Tuplets
7. Phrase marks
8. Crescendo wedges
9. Overlays

The items affect each other depending on their order in the list. For example, slur positions depend on the positions of simultaneously occurring beams. The positions of tuplets depend on those of slurs, but the reverse is not true. The order in which the user enters the items does not effect the outcome. If a staff contains beams, slurs, and tuplets, MusicEase calculates the positions of beams first, then slurs, then tuplets. The user may specify beams and tuplets and then later enter slurs. MusicEase will correctly position the beams and tuplets and then when the slurs are entered make room by shifting the tuplets.

Updating information a program maintains about the staff positions of notational elements is done before redisplaying a staff. It saves the x and y coordinates for a note head so that positioning anything with respect to the note head follows these points. The program first determines the correct side, computes an initial length, before finally determining the stem position. The program aligns the bottom of the stem with the point on the right side of the note head for stems that go up. Otherwise for stems that go down, the point on the left side marks the position for the top of the stem. To display beams and other notational elements whose positions depend on the stem end, MusicEase uses
the stem's end point coordinates. The y coordinate is changed accordingly if the stem needs to be lengthened or shortened to connect to a beam. In the case of a chord, MusicEase saves the coordinates for all note heads and uses the appropriate coordinates when attaching a stem. It handles other elements similarly. To position more complex elements such as slurs and phrase marks, MusicEase uses Bezier splines to generate the curves and ties that must curve around intervening elements. The program first makes an initial guess at the shape taking into account the starting and ending notes and whether the curve goes above or below the notes. The Bezier spline is specified by extending the two interior points if the curve does not pass around all intervening elements. The new curve is then checked for passage around all intervening elements. This process continues until it eventually succeeds or the curvature becomes too round, forcing the program to alter one or both end points.

MusicEase also applies constraints to a number of symbols that can occur simultaneously as well as casting off and justification. Adding notational elements automatically repositions other elements. The breaking of staves or systems so that all measures have approximately the same density after justification is the process known as casting off. For instance, two measures that contain exactly the same elements should have the same horizontal spacing after justification. Casting off also encompasses setting page breaks to minimize the disturbance caused by turning pages. The horizontal spacing of system elements are stretched or shrunk by justification in order to extend from the left margin to the right margin. Basically, spacing is logarithmically proportional to the durations of the notes. However, certain elements, such as accidentals and lyric syllables, can force a note to require more space. Similarly ties, for instance, have a minimum length, which could force the space between two notes to increase.
Being in use for several years now, MusicEase's word-processing paradigm and constraint-based approach seem to work reasonably well for many routine, uncomplicated, notational tasks. Simpler pieces require little or no overridding of the automatic formatting relieving naive users from handling most notational details and therefore letting them quickly produce publication-quality scores without copyist knowledge. More complex notational tasks, however, are currently beyond MusicEase's capabilities. For example, the occurrence of grace notes may require some user intervention. Further research will provide improved results using more thorough specifications of existing constraints. However, adequately addressing all forms and periods of music notation will require many more constraints (Rader, 1996).
CHAPTER 6

RECOGNITION SOFTWARE

Optical Character Recognition (OCR)

Optical character recognition (OCR) software reads in a scanner generated bitmap "image" of a page of text and interprets the series of scanner samples into individual letterforms using complex pattern-matching algorithms. A page of (mostly) accurate text results. OCR software may be confused by dot-matrix or fax input text and works best with un kerned, amply ledged body text. Some high-end OCR applications boast of 99 percent accuracy indicating that a typical page of text (2,000 to 3,000 characters) may have twenty to thirty errors that will need to be fixed manually or with a spelling checker. Obtaining an OCR with learning capability gives better results over time using your specific combination of scanner and input text style (Sullivan, 1995).

Musical OCR

Christopher Newell (1997), a graduate of Berkeley with a minor in music applied his knowledge of OCR toward the development of a musical OCR. Several years of optical character recognition research led to establishment of Musitek Corporation. Under Dr. Wladyslaw Homenda, a development team was formed at CPZH, a division of ZH Computer in Warsaw, Poland releasing 35
MIDISCAN Version 1.0 in August of 1993. The development of Musitek's Music Notation Recognition technology has been approximated to investing thirty-five Ph.D man-years of research (Newell, 1997).

The lately released Version 2.5 of MIDISCAN automatically converts printed sheet music into standard MIDI files for playback through a sound card or synthesizer. MIDISCAN has the ability to quickly and accurately capture piano, vocal/piano arrangements, solo pieces, part and ensemble scores recognizing note pitches, note and rest durations, chords, ties, accidentals, bar lines, clefs, key and time signatures with 90 - 98% accuracy. The time taken to input notes becomes significantly reduced by MIDISCAN as compared to manual keyboard methods.

MIDISCAN 2.5 gives the ability to operate a scanner, process, edit, playback, and reprint music. MIDISCAN scans multiple pages of sheet music, "recognizes" the notation and displays the reconstructed score in standard notation. The powerful editing environment makes cleanup easy. The converted MIDI file can be played back and then modified by assigning instruments and even transposing. MIDISCAN will preview, crop and scan your sheet music when used with any TWAIN-compatible scanner with the ability to process and save these scanned images as PCX, BMP or any TIF file. PianoScan, a complemented version of MIDISCAN, quickly puts piano, organ, solo or duet scores into your computer but is limited to processing music with a maximum of two staves per system. PianoScan creates up to two separate MIDI tracks where as MIDISCAN creates up to 16 MIDI tracks (PianoScan and MIDISCAN, 1997).
Applications and Common Methods

A variety of reasons exists for the creation of automatic recognition of machine-printed music. This leads to various technical goals. For example, extracting pitch, duration, and simultaneity of all notes may be sufficient for the analysis of musical style. Producing parts from a score becomes more difficult since the notes as well as all musical symbols must be recognized and associated correctly with voices. Editing of scores for reprinting, revision, and preparation of performance materials has become one of the important applications for recognition methods. Collecting databases has emerged as another application. Music-description languages that uses alphanumeric entry is common but slow and prone to errors. Instead, music editors with graphical user interfaces are used to reduce errors and for quicker entry. This is especially true if pitch and rhythm information can be entered directly through MIDI input devices. Recognizing music from audio input has seen some success with monophonic music. However, extending this to include polyphonic music becomes difficult.

Often thresholding is the first operation in a music recognition system. This converts a gray-scale image into a binary image whereas noise reduction is sometimes considered with other forms of preprocessing. Choosing a threshold-level is not critical since the majority of image points are not near the threshold.

Music notation centers around the existence of staff lines that provide a vertical coordinate system for pitches, and a horizontal direction for the timing. The spacing of the staff becomes useful both for symbol recognition and interpretation. A common approach used by most music recognition systems recognizes the staff lines as one of the first steps, since most musical symbols are disguised by the existence of these horizontal lines through the symbols.
Noise and distortion often complicates the identification of the staff lines (Blostein and Baird, 1992).
CHAPTER 7

OPTICAL MUSIC RECOGNITION (OMR)

Optical Music Recognition requires the knowledge of musical notation which is governed by a set of rules. With the high density of music information involved in complex scores, segmentation problems associated with document analysis cannot be overcome with existing systems.

Many research activities over the last two decades have been concerned with the topic of optical music reading. Whereas the recognition of printed scores as well as handwritten music notation is still in the research process, considerable progress has been made with the first commercial product introduced in 1993. Recognition systems attempt to transform a presented score into a symbolic representation to be used as input to diverse applications. Application-specific codes such as MIDI (acoustic output) or SCORE (notation) are then generated using these music representation languages as intermediate formats. The following are the stages involved in a typical OMR system: scanning, preprocessing, staff line identification/removal, notational symbol classification, contextual postprocessing and generation of a symbolic representation.

The main difficulty with optical music recognition (OMR) as opposed to other fields of document analysis is the connection of all musical objects by staff
lines. Similar to other fields, the information is presented in two dimensions.
The existence of syntactic rules for the music notation gives OMR an advantage.
Current recognition systems do not take full advantage of this fact. Systems
cannot overcome difficulties associated with complex scores such as
decomposition problems of touching objects that should not be touching.
Complex scores contain a high density of information and are polyphonic with
different voices on a single line.

Staff lines are required to be detected so that logically disconnected
objects would not be seen as a single object. Most authors erase the staff lines
to make an initial decomposition of objects and then the note's pitches are
calculated later in the processing phases.

Research of recognition methods have been active since the 1970's with
systems not only extracting the notes, but also recognizing other symbols. Good
recognition results were obtained even for complex scores. Based on the staff
position, a horizontal search method was mainstream with respect to the head
extraction. This method, however, suffers from the influence of staff distortion
and extracting a head written out of the staff regions becomes difficult due to the
possibility of wrongly positioned search lines. Traditional ways dealt with note
recognition using the complex if then rule and subtle adjustments of parameters
through experiments in order to recognize the combination of the extracted
head, stem and flag or beam composing the note. Tedious work motivated
Miyao and Nakano (1995) to adopt the neural network using a three-layer back
propagation model.

Neural Network Method

A simple method of extraction is proposed for head candidate regions.
which are searched based on the stem positions, and identification of note heads using the neural network. Utilizing the back propagation model, the network is able to learn the note formation rule.

An extraction is started by converting a music score into a black-and-white image on an A4 sheet. Setting the scanner’s resolution to 400dpi results in an image size of about 1.9 Mbytes for an A4 sheet. The staves and bar lines decide the position and size of other symbols in a music score and therefore were extracted at the first stage. Since scanned images might contain staff lines that are skewed or disconnected, Piece-wise Linear Hough Transform (PLHT) was used for the extraction of the staff lines. Skew angle of the staves can be known as a side effect of PLHT. The average interval between the staff lines $f_{\text{int}}$ and average width of the staff lines $f_w$ are then calculated and used in the symbol extraction. Since a bar in a piano score, consists of upper and lower staves, the extraction of the bar line is done by taking the projection of the region vertically. The process was applied for every bar after the stave regions were divided. The staff lines existence increases the search points in the head and stem extraction process, and are therefore eliminated using the traditional method.

The vertical straight line attached to a note is called the stem which is longer than a threshold. The vertical projection profile is used over the two staff regions with an extra space for upper and lower staves in order to determine the top of the vertical line. The black pixels are then traced vertically down to extract vertical straight lines as stem candidates. To take the slight slant and the intermittence of the stem into consideration, a certain width was added to both sides of the traced line, and the lines were connected provided their gap was less than a certain value. Next, the possible area for note heads were extracted along the extracted stems. A template matching method based on mesh
features was used to extract black and white head candidates (an ellipse filled with black and white pixels, respectively) by running vertically to both sides of each stem candidate.

The system was tested on thirteen piano scores ranging in difficulties from beginners to middle or high grade level with the following results. The number of oversighted stems were 6 whereas 1,041 stems were excessively extracted and 2,814 of the stems extracted were correctly extracted. Black note heads were oversighted twice and excessively extracted 29 times while 3,876 of them were extracted correctly. Oversighted white heads numbered 1 and 228 were excessively extracted while 144 were extracted correctly. The number of heads connected to oversighted stems were not included in the number of extracted heads.

Quick and accurate recognition of stem and head positions in a score can be obtained using a robust method which can execute the task using the neural network as opposed to the complex if-then rules based on the musical theory and the subtle adjustment of parameters. As a result, high recognition rates of 99.0% and 99.2% for the stems and the note heads, respectively are obtained. Unfortunately, the network was unable to learn the note formation rule completely, even though it had the ability to accurately recognize the heads in actual score images. This system has the advantage of being able to recognize symbols connected with other symbols analyzing a printed piano score ten or more times as fast as the manual input. Only piano scores were used in the experiments with future plans to recognize different types of scores and other symbols utilizing the information on the extracted stem and head positions (Miyao and Nakano, 1995).
The DoReMIDI System

Researchers, composers, and musicians have become attracted to the study of music in fields such as computer science. The basic research is prompted by the rich syntax of musical notation and its subtle semantics. Recognizing and interpreting musical notation becomes of great importance when dealing with multimedia systems or musical databases. A scanner and recognition system for printed music as opposed to input by a MIDI keyboard provide a more comfortable way for automatic data acquisition. Recognition within individual bar units is made possible through various information such as pitch and beat. Remarkable results were achieved using a top-down strategy developed by Kato and Inokuchi to serve as a basis for Baumann's and Dengel's (1992) system described next.

Transforming printed music that has been recognized and interpreted into MIDI becomes the task of generating electronic music automatically. DoReMIDI (Document Recognition of printed scores and MIDI generation) is a system created to recognize printed music and generate musical output. Originally printed piano music is transformed into MIDI representation by using Artificial Intelligence (AI) techniques and pattern recognition. An optical scanner in line art mode at 200 dpi resolution is used to obtain an electronic representation of printed music. The scanner provides a bitmap of black and white pixels that represents the paper copy. The system currently recognizes simple piano music for two hands with monophony in both voices using a reduced set of musical symbols. Staff lines connect individual symbols and therefore are considered as one connected image component. Two aspects have to be taken into account when eliminating the staff lines: (1) positions of the exact locations of the staff lines and (2) avoidance of remaining pixel clusters when deleting staff lines and deletion of pictorial information belonging to musical symbols. After all staff lines...
within the bar units are deleted, a technique called puzzletree encoding is used
to isolate the remaining musical information within the image data.

Puzzletree encoding decomposes the image recursively, adapting to the
various structures. Adjacent blocks of similar height or width is created from the
decomposition of the original image. Subdivisions are continually made until
each block consist of similarly colored pixels. The image is searched for areas
containing pixels of the same color in order to determine the positions where
decomposition should occur. A number of image blocks results which can later
be arranged to reconstruct the original image much in the same way as a puzzle.
The decomposition ends when entirely white blocks can no longer be obtained
resulting in the isolation of the blocks containing musical symbols.

Image objects can be preclassified by considering the difference in the
musical symbol’s dimensions. A decision tree classifier is used for this
preclassification of the image objects resulting in 14 classes of symbols. The
classifier is a binary tree with the proportions of the image objects resulting in a
binary decision that is represented by each node in the tree. To reach a leaf in
the tree which results in a musical class being assigned to an image object, an
average of three or four decisions are necessary. In general for each object, two
or three and no more than five alternative hypotheses are generated. The
distance between two staff lines is used so that a preclassification independent
of notation style may be obtained. This measurement is further used to modify
the dimensions of the rectangles circumscribing the isolated musical symbols.
Symbols with similar dimensions such as half/whole rest and whole note or half
and quarter note, are classified into one class. To guarantee the right musical
classification for a set of hypotheses, non-disjunctive features are used to
characterize the classes since uncertainties exist in the physical segmentation.

After recognizing the individual musical symbols, they are transformed,
using sufficient information such as the pitch and duration of the note, into an intermediate format. A sequence of NOTE-EVENTs are generated for both the right and left hand with the transition consisting of several steps. First multiple notes connected by beams, where the stems' position are located by vertical image projection, are split into single notes. The thickness of the beam can therefore be determined to help identify the note's duration. The pitch of the note is found by locating the position of its head for each corresponding stem. This is done by tracking the image columns capturing the stems from the ends opposite to the beams. The position of the head is indicated by the first black pixel allowing for components of the symbol layer to be taken to determine the NOTE-EVENTs. Two respective MIDI-EVENTs are generated for each class NOTE-EVENT instance. A pattern of three bytes make up a MIDI-EVENT to describe pitch, velocity, and such information as note on/off, after touch, and control change. The first of the two MIDI-EVENTs begins playing a sound at the specified pitch whereas the second MIDI-EVENT stops the acoustic output after a variable duration. Rests generates a specified time without sound. The process is concluded when all MIDI-EVENTs are stored within a MIDI file.

The recognition rate of the various symbols varies between 80 and 100%. This range is partially due to pieces of the symbols being detected by the simple algorithm for staff line detection and removed as noise. Fragmentation causes unexpected results when staff lines are touched by symbols such as when half and whole notes fill out the staff spaces or when the staff lines are touched by sharps or flats. Future work may better handle these problems through the use of improved staff line removal or techniques that will enable the assembling of symbol fragments. Enriching the grammar will further the process by allowing complex notations that for instance might contain changing clef or key signatures (Baumann and Dengel, 1992).
DoReMIDI++

The DOREMIDI system which was developed in 1992 to recognize monophonic piano music in printed form has been recently modified using C++ to allow for the recognition of multiple score layouts as well as polyphonic scores. The entire OMR system is called DOREMIDI++ which was developed by modifying H. Bunke’s attributed programmed graph grammar for the recognition of circuit diagrams with H. Fahmy’s music notation recognition method.

The initial image is transformed into a graph structure to which production rules are applied leading to the desired representation. Two-dimensionality is the advantage that graph grammars have over the traditional string grammars, which becomes important if pictures, maps, diagrams, etc are postprocessed contextually by the grammars. Blostein and Fahmy adapted Bunkes recognition approach since music notation can be considered as a two-dimensional input picture. An Attributed Programmed Graph Grammar (APGG) is used to interpret the results of the symbol recognizer for music notation. Three nodes maximum are restricted on both sides of a production in order to simplify the graph-matching problem that matches the left-hand side of a production with part of the initial graph. Edges are then considered as attributes of the corresponding nodes as opposed to separate objects. Seven subsets divide the productions into different types allowing for complex operations.

The bowing and discontinuity of staff lines and note stems, symbols that are broken or touching, as well as the presence of noise and skew are a few of the reasons that often cause low-level recognition problems. Fragmentation of symbols due to staff line removal create additional problems. These can all cause the symbol recognizer to produce incorrect hypotheses. The transformation identifies incorrect hypotheses as being false provided the musical syntax allows this (Baumann, 1995).
Recognition by Grammar

Syntactic rules can be formalized by using a grammar to represent the context and creating a new method of recognition. A discrepancy occurs when an object is described by an existing grammar and the way it is recognized when dealing with touching objects. To deal with objects that touch, such as accidentals that touch a note head, the use of an operator is proposed in order to modify the way an image is parsed by the grammar. Fujinaga observed that music notation grammar is context-free and LL(k) thus making it possible to formalize music writing rules that govern complex scores through the use of a defined grammar.

Complex scores contain a high density of symbols often resulting in connections between musical objects that syntactically should not touch. This causes some problems with segmentation.

Broken objects may result from initial documents of poor quality. Such objects are usually small such as tuplets's figures. Scanning also produces some noise. However, broken objects are most frequently caused by the initial removal of the staff lines in the recognition process.

Constructs and symbolics divide the objects in a score into two categories. The group of constructs consist of simple stemmed and beamed notes composed of segments such as stems, beams, and note heads. A set of rules for construction that apply to these elements are also included in this group. Symbolics consisting of clefs, accidentals, etc. can be considered as characters and therefore recognized using optical character recognition (OCR) techniques. Information on a musical score can also be divided into two levels; a physical one and a logical one. Forming and adjusting the way notes as well as their attributes such as accidentals, accents, and so forth on a score correspond to the physical level. The syntactic way that notes are used in
written music correspond to the logical level. Independent of the beaming, these
notes contain attributes associated with an individual note such as pitch, value,
accidental, etc. Taking these two levels into consideration, the grammar
requires a two level structure with a graphical one corresponding to the physical
level, and a syntactic one corresponding to the logical level.

Allowing for symbols to touch a construct whether at the segment level
(stem or beam) or at head level (note head or rest head), the system must first
segment them correctly before being able to recognize these symbols. In order
to do this, the segments and heads of the constructs must be identified and
removed from the image thus allowing for a correct extraction of the symbols.
The detector makes this segmentation possible since it can extract segments
even ones connected to other objects. Low level recognition is possible even if
the object has been segmented incorrectly due to the localized segment
information. For a construct, the grammatical rule must recognize both symbols
and segments. All note grammar (noteGr) must be recognized first before the
beam of a beamed Note. To accomplish this, the stem (a segment), head, and
possible accidental (a symbol) must be recognized for each noteGr. A problem
occurs when the accidental touches the beam. Separation by removing
recognized segments is not possible since the beam has not been identified yet
due to the fact that all noteGr have not been completely recognized. Separating
the segment's and head's description in the construct from the attributes of the
notes becomes one solution of this problem. The construct rules for the
segments and heads and the rules for the attributes should be applied
separately in the recognition process. Identifying which attribute is associated
with which note would be difficult and separating this knowledge becomes
essential so that a scaleable system can be created and rules adapted and
changed without difficulty. The discrepancy between an object's description and
its recognition method used prevents this separation. Because of this, most systems try to use only a limited knowledge incorporated directly in the program and not separated from it. A Delay operator is proposed to help overcome this problem. It should be incorporated to modify the operational part of the parser while keeping the grammar structured and the actual description of the rule unchanged.

A grammar defined for full scores, which includes a single staff consisting of different voices, chords on a voice, accents, tuplets, pause, octave, dynamic markings, phrasing slurs, rhythmic slurs and appoggiatura, may be utilized. Abbreviations, ornaments (except appoggiatura) and lyrics have not yet been included. The initial grammar contains a subset defined to recognize full polyphonic scores with clefs, key and time signature, half, quarter, single eighth notes and beamed notes, accidentals, dots after a note, rests, bar lines, with the Delay operator and pitch processing. Besides being able to recognize full scores, the system checks for the correct number of beats in a bar (per voice) according to the time signature and for the vertical alignment of notes in a system. After running these two checks, the system is able to detect most errors and even suggest corrections (Coüasnon and Camillerapp, 1995).

Most optical music recognition systems incorporate some musical rules in the recognition process, but they are selective and not formalized. A grammar may formalize some rules for the limited purpose of verification, error correction or final pitch calculation. Only simple scores are used and the graphical rules are not embedded. A formalization, including the graphical level, is proposed by Andronico in which a "natural" grammar as well as a set of "bidimensional" grammars are used. The system currently only recognizes simple and monophonic scores. Eventually, the complete recognition process is hoped to be enabled by the parser to use the music context.
Locating and correcting decomposition errors can be achieved by describing a prior knowledge through a grammar. Deciding if a symbol is decomposed correctly, the system can then continue the decomposition if necessary. The decomposition of the connected components making up the symbols are then presented to a classifier in the form of an array of pixels. The classifier determines whether a symbol is decomposed correctly or not by rejecting an entry not belonging to a learned class.

Utilizing grammar allows for the inclusion of handwritten scores. Handwritten scores contain the possibility that elements such as stems, heads, or beams may be separated. While the grammar gives no indication of connectedness between elements, the position operators introduce the idea of closeness.

To enable a context of maximum integration, Coûasnon and Camillerapp (1995) formalizes all the recognition rules thus producing a coherent system. In contrast to most systems, the syntax here is used to control the recognition process and thus a more reliable labelling is produced as opposed to just checking the labelling. This system produces an accurate decomposition at the image level therefore obtaining a good recognition whereas most grammatical methods only work at a high level.

Separating the parser, the operating part of the system, from the grammar, in this case the definition of musical rules, becomes an advantage of a formalized grammar. The rules can then be easily modified or the system adapted to any kind of structured document. A new recognition system can then be created by simply defining a new grammar keeping the parser unchanged. Unresolved problems, such as reconstruction of broken objects, or decomposition of touching objects as described by Blostein and Carter as well
as the scaling up problem can be solved by this system (Coüasnon and Camillerapp, 1994).

Recognition by Genetic Algorithm (GA)

Yoda, Yamamoto and Yamada (1994) use an original image and a target image as a procedure for recognizing musical notes. No models or preliminary knowledge is used to extract simple shapes, instead a hierarchical mathematical morphology procedure, such as erosion and dilation operations, is used with search procedures being done by genetic algorithms (GA). These procedures of morphology are constructed through the use of a GA parallel searching function and the introduction of era. The operations of erosion and dilation are used to automatically transform the original image into a presented target image that was created manually by deleting unnecessary pixels.

The genetic algorithm (GA), based on biological evolution, searches for an optimal procedure satisfying this objective. It is a parallel searching method for probabilistic searching, learning and optimization which considers the four basic operations of erosion and dilation as genes in GA, and the processing sequence of the image as a chromosome. A chromosome is searched for using an algorithm of GA, however, by using only four kinds of operations satisfactory results may not be obtained. To resolve this problem, three cases are created for classification of more complex original and target images. The processed image shows differences from the target image in the first case while the second case contains over extraction pixels and under extraction pixels are contained in the third case. Searching continues to the end of the process in the first case. The second and third cases are resolved by introducing a concept of era being a period of evolution for a sequence of operations or chromosome. Unsatisfactory results produces a new era. Extracting basic musical shapes was the main
application of this method, however, total analysis of musical notations may be obtained by extending the fundamental operations of erosion and dilation with isotropic neighbors (Yoda, Yamamoto, and Yamada, 1994).

Wabot 2 Robot

Japanese research teams have been involved with more recent work such as the WABOT 2 keyboard playing robot at the Waseda University. It's vision system uses mask-matching to enable it to read nursery song sheets. A recognition system for keyboard music has also been undertaken by a team at Osaka University. Problems arising from producing a score from a soundtrack and sentiment extraction are resolved by this system. Other research includes recognizing music symbols using projection profiles being done at the University of Ottawa and providing a low-cost solution to the automatic input problem for publisher and engravers' use at the University College Cardiff. Other short term projects in this area include works by Tojo, Roach ([1982]), and Tatem ([1988]), Mahoney ([1982]), Martin ([1987]), and Tonnesland ([1986]).

In the early 1980's the Wabot-2 robot was developed in Japan as an impressive keyboard-playing robot. It has the capability among other things to interpret images taken of sheet music placed on a music stand through its vision system. After the sheet music is read and interpreted, the robot proceeds to play the piece. Achieving very fast image interpretation, the Wabot design can interpret one page of music in approximately 10 to 15 seconds. This high speed is made possible by using special purpose hardware and parallel processing achieving a recognition rate of nearly 100% for simple scores. Three-part organ scores that contain relatively simple notation are used. Three staves are involved; one for the right hand, one for the left hand and the bottom staff for the feet. A two-level hierarchy is established to recognize the musical symbols; the
upper level being implemented by the hardware and the lower level in software. The upper level consists of staff lines, note heads and bar lines occurring in many places in the image. Hardware-implemented template-matching is used in searching for these. Symbols with locations constrained by the recognition results for the upper level symbols are included in the lower level and found using software implemented localized search. Rests, stems, flags, repeat signs, staccato and marcato marks, accidentals, prolongation dots, clefs and time signatures are examples of such lower level symbols. Incorrect matches may result from using template matching to detect filled note-heads which is later eliminated by using knowledge about the syntax of music notation (Blostein and Baird, 1992).

Other Researchers and Their Work

Several techniques exist for detecting and removing staff lines, as well as alternate analysis techniques involving staff-line identification without removal. Detecting the staff lines help in normalizing the image, determining the score geometry, and restricting the musical symbols search area. Five equally spaced lines constitute a staff. Projection methods have been used by various authors to identify staff lines, some operating with the staff lines remaining in the image. Fujinaga uses a Y projection for the entire page of music to locate the staves. Five or more peaks clustered together are accepted as a staff with extra peaks resulting from ledger lines, horizontal beams, or skew.

Classifying musical symbols becomes the next major task once the staff lines have been successfully identified and/or removed. Many methods exists that apply to this problem.

Pruslin and Prerau at the Massachusetts Institute of Technology began research in the late 1960s and early 1970s into what would become the
beginning work on automatic recognition of printed music. Hardware limitations for acquiring and manipulation images restricted the research, however by using low-pass filtering and contour tracing, some progress was achieved. Pruslin and Prerau both addressed the removal of staff lines in sheet music images in their respected PhD theses at MIT. Pruslin first eliminated all thin horizontal and vertical lines, including bare staff-line sections and stems from the music image leaving only isolated symbols, such as note heads and beams. These remaining symbols are then recognized using contour-tracing methods to describe connected binary image regions. This method results in music symbols that are distorted or erased. A “fragmentation and assemblage” method is described by Prerau to deal with staff lines and music symbol isolation. Prerau initially classifies a subset of music symbols with the use of their relative size. Horizontal overlap does not always occur and thus symbols that merge with staff lines would be disconnected by Prerau’s method.

As an alternate method, Andronico and Ciampa attempts to remove only the staff lines in sections where they are not crossed by symbols. Line removal is divided into two types by Mahoney that of real lines or bare staff line sections and that of ideal lines or complete staff lines. Real line removal only removes that part of the line not overlapping any other symbols. By establishing an allowed thickness range for the line and removing only the portions of the line that satisfy this condition, this can be achieved. Most common for removing staff lines is the real line removal as opposed to ideal line removal which removes the entire line. Adjacent symbols can be split by using this method such as separating note heads occupying adjacent spaces. Pattern primitives such as note heads, stems, beams and flags are extracted by Mahoney’s method to be combined later to form notes, chords, and beamed note sequences called composite symbols. By dividing symbols into these two groups, the recognition
task is subdivided and made simplified. Lines, dots, and characters are the pattern primitives utilized by Mahoney. Ranges of values for parameters further classify the primitives.

Projections have been used by various researchers in the task of recognizing symbols. A microcomputer-based music recognition system described by Lee and Choi recognizes staff lines, bar lines, and finally notes including chords and rests through the use of projection methods. Problems with lighting as well as with the image method create noise problems especially near the border of the image which is reduced through preprocessing. A Y projection is used to find the staff lines and an image area is formed that contains only the staff nucleus. Next bar lines are found and notes recognized by using X and Y projections. Pitch and duration are also recognized, however, the method is rotation-sensitive failing on a tilted image.

Both X and Y projections are used by Fujinaga, Alphonce and Pennycook for segmentation as well as for symbol recognition. Although these projections only give an approximation of the symbol’s size and shape, they are enough to identify many of the music symbols because of the variation in shape and size of the music symbols. Using an X projection of the staff usually leaves off symbols protruding above or below the staff and becomes difficult to analyze with the interference of expression marks, measure numbers, and lyrics. Fortunately, these methods remain sufficient when locating individual musical symbols. The vertical extent of the symbol is determined by a Y projection and these bounds are later used when taking another X projection. This system has only been used to analyze monophonic music.

Knowledge about music notation, such as its syntax, existence and properties of staff lines, note stems and note heads, are used in part by all of the previously mentioned systems. Roach and Tatem proposes a rule-based system
to represent such information. Starting with the earliest steps of symbol
segmentation and recognition, this information should be applied. Their theory
has been developed into a prototype system for recognizing music notation that
is hand-written. Circular blobs (for note heads that are filled in), circles (for note
heads that are not filled in), horizontal lines, non-horizontal line segments, and
arcs are the primitives that the system recognizes. Hand-written images make
detecting note heads extremely difficult often leaving a general-purpose blob
detector fooled. Constrained locations are therefore searched for note heads
locating vertical lines as possible note stems. At the ends of each potential
stem, a test for wide spots is administered by measuring for thickness. If a wide
spot is encountered, a note head is deemed to exist provided its circularity
measure is greater than some threshold. As of most recently, this system makes
no comparisons to techniques already in use by recognition systems that
incorporate the knowledge of music notation.

Kato and Inokuchi describe a sophisticated recognition system for printed
piano music which presents challenging problems in image pattern matching as
well as semantic analysis. A top-down approach is used, recognizing music one
measure at a time and handling both simple and complex notation. Interpreting
simple, monophonic notations uniquely requires the use of simple rules.
Complex notations consisting of higher symbol density, more connections,
overlaps and complicated placement of symbols, may be ambiguous, requiring
musical knowledge for their interpretation. Devising a single method for
recognizing all symbols becomes difficult due to their difference in size and
position, frequency of appearance, importance, and so on. Because of this, a
variety of recognition methods may be necessary requiring a flexible control
structure. A common working memory operates to allow a collection of
processing modules used by Kato and Inokuchi to communicate. Five levels of
abstraction are used to represent information about the current bar of music in
the working memory. The pixel image is contained in the first layer whereas
primitives such as stems, note heads, beams, flags, accidentals, duration dots,
and rests are included in the second layer. Combinations of the primitives from
the second layer such as notes and rests are contained in the third layer of
abstraction. Each symbol's meaning, pitch and duration of a note, make up the
fourth layer. Time-order combinations from the fourth layer form possible
interpretations of the entire bar which are stored in the fifth layer. Four
processing modules exist each containing one or more recognition and
verification units. They are as follows: (1) primitive extraction, (2) symbol
synthesis, (3) symbol recognition, and (4) semantic analysis. Recognition
proceeds one measure at a time detecting staff and bar lines in the
preprocessing stage. Next, subimages containing single measures such as staff
line elimination, recognition of attributive symbols (clefs, key signatures and time
signatures), and recognition of note symbols are processed. Finally, symbols
spanning measure boundaries are recognized in the postprocessing stage. The
partial results from each measure are then combined to form an interpretation of
the final image. Recognition results of Kato and Inokuchi's recognition system
exceed 90% correctness on beginner's piano music and good recognition results
on complicated piano music (Blostein and Baird, 1992).

Projection profiles have been proposed for use by Nakamura and
Fujinaga. Recognition of the type and position of each symbol is done by
extracting an important feature of the shape and position that exist at a certain
point in the pattern of a symbol through the use of horizontal or vertical
projection. These methods contain the advantage of simplicity, but may create a
difficult recognition process if connected symbols or ones drawn in vertical
alignment exist. Presently only simple monophonic music notation such as that found in children's songs are recognized by these methods.

The recognition method proposed by Tojo classifies the symbols into groups depending on the rectangle shape circumscribed with symbols and the symbol's discrimination from the structure analysis. This method requires preprocessing for staff line elimination as well as the segmentation of symbols.

A high-speed recognition system has been developed by Matsushima for real time musical performance through the use of a robot. Symbols are detected by the hardware in about 10 seconds, but the system is unable to handle complex notation. Difficulties that arise when recognizing music notations are various. Connections between symbols can be ignored in simple notations due to the low density of symbols. However, in complex notations where there is a high density of symbols, connection, overlap and containment of symbols become plentiful. This feature as well as overlapping with staff lines creates a difficulty when segmenting symbols. These difficulties have created problems for existing systems when dealing with complex notations. Matsushima uses the knowledge and constraints of music notation to overcome these difficulties in a recognition system for the printed notation of piano music.

The difficulty in recognizing all symbols by a specific method is due to the difference of size, position, frequency of appearance and importance, and so on of the symbols in music notations. Since there are various methods of pattern recognition available, each symbol may utilize a specific recognition method that applies. A very useful approach to recognition methods is the top-down approach. Recognizing the bar as bar units is a successful method for interpreting most symbols. Two task are involved with the recognition process; a pattern processing task and a semantic analysis task. Overlapping of symbols, breaks of thin lines, and ink spots that were unexpected are examples of
unexpected states dealt with in the pattern processing task. Using the top-down approach, pattern processing units are prepared with the knowledge and constraints of music notations for each symbol. The combination of primitives form musical notes with varying shapes simple to complex. Musical notes are dealt with in the primitive level by the pattern processing task. The results from the pattern processing task are interpreted and translated by the semantic analysis task into a playable musical data format. Contradictions are rejected by this task and requests are made for results from the pattern processing task to resolve the conflict. An optical scanner is used after estimating the staff lines and image's width into bar-units to apply the staff lines and bar lines detection to the notation image in the preprocessing stage. Those symbols, though few, whose recognition becomes difficult using bar-unit and unification processing are applied a recognition process in postprocessing.

Staff lines restrict the position and size of the symbols and are therefore the core of music notation. The time series of musical data is symbolically represented by music notation dividing the staff by bar lines into the constant number of the beat making bar-unit processing reasonable. Detecting the staff lines and bar lines divide the music notation into bar-units in the preprocessing stage. Since the size of the staff is unknown, the distance between as well as the average width of the staff have to be estimated first. The size of the symbols is determined by the distance between the staff lines and thresholds are later set using the width of the staff lines to determine the image's quality.

Staff lines are detected by first scanning the image vertically on 10 lines separating the image into equal areas. The estimated distance and width of the staff lines found in the above process are used by each scan to estimate the staff lines' height. The height of the staff lines is chosen from among these estimations by accepting the one with highest frequency. Horizontal and vertical
projections are made after eliminating short components in the horizontal
directions. The horizontal projection produces an accurate height of the staff
line whereas the vertical projection gives the accurate edge point. If the edge
point is not obtained in this area of the image, the preceding steps are repeated
on the next area by moving in the horizontal direction.

Since bar lines are drawn from the upper to lower staff, they are detected
by placing rectangular masks around the upper, lower staffs and around the area
between the two staves. In the vertical direction of each area, short components
are eliminated and an extraction of the hypotheses of bar lines is achieved.
Identical positions of three hypotheses in each area represents the existence of
a bar line. Checking the distance from neighbor bar lines distinguishes them as
being single or double bars. Repeat marks are double bars with the presence of
dots in the second and third space. Each bar unit is recognized separately by
eliminating staff lines, recognizing attributive symbols, and finally recognizing
note symbols.

Even though staff lines are crucial in the positioning of all symbols, they
are usually eliminated since the occurrence of overlapping may disturb the
recognition process. This system recognizes two types of clefs the G clef and
the F clef by the size of the rectangle circumscribing it. Sharps, flats and
naturals are included in the key signature and two numbers are used to indicate
the time signature. In this system, note symbols are considered either as notes,
rests, or accidentals. The recognition of these symbols may become difficult for
a number of reasons. First segmentation of notes are restricted by connections,
overlaps, and containments of the notes with other symbols. Some symbol's
size and shape are not kept constant as well as fragmentation occurring with the
elimination of staff lines due to their involvement with some symbols. Other
difficulties include breaks in the symbols, ambiguity of the symbols and the
semantic relationship between symbols as well as difficulties in the pattern signal processing and semantic analysis. A top-down approach using bar-unit recognition is adopted to deal with such difficulties. Simple beginner music achieves a recognition rate of over 90% with expert music accomplishing a little less. This system does not take into account all possible symbols leaving room for future study. Without human assistance, the automatic recognition system takes too much time for recognizing the notation. To make the system more practical, processing time will need to be reduced by using a simpler recognition method and intelligent human interface (Kato and Inokuchi, 1992).

Noise tolerance and rotation of the original image are considered in the system presented by Carter and Bacon (1992). The line adjacency graph is transformed and becomes the basis for the use of a structural decomposition technique.

A structural decomposition technique becomes the basis for their recognition system by using an original transformation of the line adjacency graph. A scanner at 300 dots per inch resolution, A4 image size and automatic thresholding provide an image of approximately 1Mbyte of binary data per page. Rendering staff lines by means of forming a horizontal histogram of black pixels becomes impractical when skewing, bowing, fragmentation, and obscurenness due to superimposed musical symbols exist. Therefore, this system makes two passes over the binary image and an original transformation of the line adjacency graph. In the first pass, the image is run-length encoded, orienting the continuous runs of black pixels or segments vertically. The transformed line adjacency graph is constructed in the second pass forming nodes or sections by examining the run length encoded data and collecting segments together where appropriate. The structural decomposition technique provides several advantages including significant data reduction so that the original bitmap image
is not needed in subsequent processing and the breakdown of the image remains consistent even if rotation or skewing of the original exists. Information about the maximum and minimum x and y values, the area, length/average thickness, various connections, and a least-squares fit line through the midpoints of constituent segments are stored in the data structure. Hopefully, this system will successfully handle two or more voices per staff and eventually cope with chords and multiple staff systems.

Object recognition is achieved by deriving a number of models in conjunction with a limited set of parameters such as recognizing a dot or staccatissimo marking as a single section bounded by a rectangle of appropriate maximum side length and aspect ratio. The staccatissimo's triangular shape is eventually identified through further testing. More complicated symbols containing permutations of pitch and rhythm and optional ledger lines requires its own processing routine. Beamed groups, for example, are examined by using the transformed line adjacency graph with horizontally oriented segments to detect the presence of vertical lines. Note stems are identified as those sections with high aspect ratio whereas stem fragments formed from thin sections are identified through the use of a collinearity test. Note head sections, including multiple ones clustered together, are searched for by examining both sides of each note stem. To keep the relevant pitch values, local staff line sections are found and each note head's or cluster's vertical extent is established. To identify the rhythm value of each note, the end of each note stem is measured for its thickness. Objects not dependent on the staff for information, such as dots, are harder to fully recognize. After a dot is discovered, its association whether note or rest has to be identified and role distinguished between rhythm augmentation and staccato marking. Similarly, a slur indicating phrasing may extend beyond
the horizontal extremes of the staff even though it is associated with only two
notes.

Carter and Bacon's (1992) current system outputs an ASCII file
compatible with the SCORE desktop music publishing package. Five
categories; pitch, rhythm, marking, beams and slurs are represented. A letter
and octave number identify the pitch, a number representing the denominator of
the fractional representation of the rhythm such as 8 for an eighth note tells the
rhythm and note numbers are stored in the last three categories indicating the
notes to which they apply. In order to obtain a comprehensive music recognition
system, a larger vocabulary of symbols will need to be included as well as the
ability to process multiple voices on one staff and the recognition of overlapping
or touching symbols. To attain this, an algorithm will need to be developed
which would separate a composite object into parts that could be recognized by
existing techniques. Text of various fonts and point sizes will need to be
isolated and then given to existing character recognition software, before finally
reassembling the text and music. Their existing system achieves significant
tolerance to noise and rotation however extreme cases of fragmentation will
require new techniques. To successfully reconstruct these fragmented symbols,
spatial relationships between portions of the symbol will need to be established
through use of graph structures and cluster analysis (Carter and Bacon, 1992).

An automatic score recognition system has been developed by Itagaki,
Isogai, Hashimoto and Ohteru (1992) as part of "An Integrated Music Information
System: PSB-er". "PSB-er" stands for "Performance Score and Braille music
system for Education and Research." Many important forms of artistic
representation can be viewed as universal languages such as music
performance, printed score, musical braille and dance. Analogue information is
used to represent dance movement in a two dimensional graphic image notation.
Braille score, on the other hand, is a sequence of braille characters represented in one dimensional digital form. Piano scores from beginners to a Chopin Etude to popular songs were used achieving nearly 100% recognition rate on beginner's scores, and 85% to 95% otherwise. Staff lines are detected first obtaining candidates at a number of positions since they may not lie horizontally. Using this information, the image's inclination is corrected and normalized and the threshold for pattern discrimination set. The image is then divided into individual staffs so that note head and bar detection can be performed on each area. Symbols, such as the clefs, time and key signatures, are detected followed by note heads which are checked for attributes such as dots, tenuto and staccato marks as well as for four types of flags. The recognition results are generated and symbols analyzed through the use of the musical syntax and staff location.

The "Automatic Translation System of Printed Music to Braille" translates printed music including almost all piano symbols into braille music within a few seconds and analyzes a complex braille score in about two CPU minutes. This system not only has an application in direct braille music acquisition but also for translation verification. Future developments include the recognition of song and orchestral scores overcoming the problems of musical symbols and lyrics distinction. The obvious solution would include the capabilities of character recognition within the score recognizer (Itagaki, Isogai, Hashimoto, and Ohteru, 1992).

Music analysis, point-of-sale printing of scores, creation of braille scores and production of new editions based on the computer are all applications that could be enhanced by the availability of a system that automatically recognizes printed music. Carter (1992) describes a score-reading system with significant tolerance of size, font variation, notation and noise in the source images. The
system is being tested on William Walton's music piece called Facade. The recognition system is based on the structural decomposition technique used to locate pairs of isolated black regions containing the properties of length being between three and five times the inter-staveline spacing and aspect ratio greater than three.

An original transformation of the line adjacency graph becomes the basis of the structural decomposition technique with two passes over the image being made. First, run-length encoding is used vertically on continuous runs of black pixels or segments. The transformed line adjacency graph is built in the second pass by combining adjacent segments to form sections which become the nodes of the graph. Noise, considered as being sections with area \( \leq \) five pixels, is removed and sections containing high aspect ratio are seen as possible stave line fragments or filaments and are isolated. Exposed portions of the stave are located by grouping horizontally-overlapping, roughly equispaced and parallel filaments into sets of five. After all such occurrences are found, they are appropriately linked across the page. All other stave line sections are found by either interpolation between fragments or extrapolation beyond the horizontal extremes of each group of fragments. The above stave line-finding technique achieves satisfactory tolerance of scale variation and rotation with performance on a variety of images working reasonably well.

The transformed line adjacency graph is then reorganized after all stave line sections are found so that each section's connections can be categorized into stave line and non-stave line. Objects are then formed from these connected non-stave line sections.

Models are used for comparison to identify various objects. Next, the object's overall size, aspect ratio and thickness classifies it into various categories such as dot, tenuto or staccatissimo marking. Future projects will
incorporate existing text recognition software so that lyrics and markings of expression can easily be recognized. Various syntax checks would also make it possible to check for the presence of the correct number of beats in a bar before being passed to the output data file (Carter, 1992).
CHAPTER 8

COMPRESSION

Image files consume huge amounts of disk space and thus compression schemes are utilized. Two forms of compression schemes, Loss Less and Lossy, are currently used in reducing the large size of graphic files; the most common used scheme being the Loss Less. An alternative way of looking at compression of sound files would be to consider storing them as MIDI files. A file requiring more than 88,000 bytes can be reduced to 75 bytes by storing it as a MIDI file as opposed to Digital Audio (Aikin, 1997). Typical examples of compression files that may be used for music are run-length encoding and LZW.

Loss Less Compression

Loss Less compression results in no loss of information during compression. Instead of encoding each pixel value as in an uncompressed image, Loss Less compression encodes areas of pixels containing the same value. This scheme results in a moderate compression rate and ensures a decompressed file identical to the original. Eliminating redundant data using mathematical algorithms, loss less compression results in significant compression and reasonable performance. Examples of loss less compression are Photoshop native format, TIFF 6.0, and compression utilities such as Stuffit.
and PKZIP. However, files having very little redundant data can actually increase in size as a result of compression using a loss less scheme. Loss less compression is often used when archiving a file for backup.

Lossy Compression

Greater image compression can be obtained through lossy compression even though this scheme actually loses information when saving and compressing a file. Currently, this scheme is only used in the JPEG file format. Fortunately, Lossy compression achieves a compression rate up to ten or more times greater than the Loss Less compression. Furthermore, the retrieval of an uncompressed image displays little or no degradation despite the fact that the compression scheme loses information. This is due to the fact that the information lost is beyond the range of human vision; i.e. graphic images often represent more colors than the eye can perceive or that printers can print. For photographic images of 24 bits or higher, Lossy compression works well, but, for any computer-created graphic file where a loss in information can result in a corrupted file, this scheme is not recommended (Chamberlain, 1995). The trade-off becomes the loss of data during the compression process. In theory, not all data in an image is important and consequently expendable. A file compressed using a lossy scheme can usually be detected by examining flat color areas near the border of transitional colors. Depending on the original and also on how much compression you specify (high, medium or low), the effect can be subtle or irritating. Usually, results are often gratifying compressing an image using JPEG from 3MB to 300K for instance. By applying JPEG compression in the Quality = Good mode, a reduced file size was able to retain nearly the same image quality as without compression. Greatly reducing the file size, an image can be sent via modem with a transfer time of one-tenth the time it would have
taken to send the same file uncompressed. Lossy compression has become very popular with multimedia professionals (Sullivan, 1995).

Run-length Encoded (RLE)

Compression of a bitmap reduces the disk and memory storage required. Run-length encoded (RLE) formats for compressing bitmaps that use 4 bits per pixel and 8 bits per pixel is supported by Windows versions 3.0 and later. When the biCompression member of the BITMAPINFOHEADER structure is set to BI_RLE8, the DIB is compressed using a run-length encoded format for a 256-color bitmap. This format uses two modes: encoded mode and absolute mode, both of which can occur anywhere throughout a single bitmap. In encoded mode, a unit of information consists of two bytes. The first byte specifies the number of consecutive pixels to be drawn using the color index contained in the second byte. To denote an end of line, end of the bitmap, or a delta, the first byte of the pair can be set to zero to indicate an escape. The escape is interpreted depending on the value of the second byte of the pair, which ranges from 0x00 through 0x02. Following are the meanings of the escape values that can be used in the second byte:

0 End of line.
1 End of bitmap.
2 Delta. The two bytes following the escape contain unsigned values indicating the horizontal and vertical offsets of the next pixel from the current position.

Absolute mode is established by setting the first byte in the pair to zero and the second byte to a value between 0x03 and 0xFF. The second byte represents the number of bytes that follow, each containing the color index of a single pixel. A word boundary must align each run. Following is an example of an 8-bit RLE:
bitmap (the two-digit hexadecimal values in the second column represent a color index for a single pixel):

<table>
<thead>
<tr>
<th>Compressed data</th>
<th>Expanded data</th>
</tr>
</thead>
<tbody>
<tr>
<td>03 04</td>
<td>04 04 04</td>
</tr>
<tr>
<td>05 06</td>
<td>06 06 06 06 06</td>
</tr>
<tr>
<td>00 03 45 56 67 00</td>
<td>45 56 67 00</td>
</tr>
<tr>
<td>02 78</td>
<td>78 78</td>
</tr>
<tr>
<td>00 02 05 01</td>
<td>Move 5 right and 1 down</td>
</tr>
<tr>
<td>00 00</td>
<td>End of line</td>
</tr>
<tr>
<td>09 1E</td>
<td>1E 1E 1E 1E 1E 1E 1E 1E</td>
</tr>
<tr>
<td>00 01</td>
<td>End of RLE bitmap</td>
</tr>
</tbody>
</table>

Setting the biCompression member of the BITMAPINFOHEADER structure to BI_RLE4 compresses the DIB using a run-length encoded format for a 16-color bitmap. This format also uses the encoded and absolute modes. In encoded mode, a unit of information consists of two bytes; the first byte containing the number of pixels to be drawn using the color indexes in the second byte. Two color indexes are contained in the second byte, one in its high-order nibble (that is, its low-order 4 bits) and one in its low-order nibble. The color specified by the high-order nibble is used in drawing the first pixel while the second pixel is drawn using the color in the low-order nibble, the third is drawn with the color in the high-order nibble, and so on, until all the pixels specified by the first byte have been drawn. Setting the first byte of the pair to zero indicates an escape which denotes the end of a line, the end of the bitmap, or a delta. Depending on the value of the second byte of the pair, the escape can be interpreted in different ways. In encoded mode, the second byte has a value ranging from 0x00 through 0x02. These values mean the same as for a DIB with 8 bits per pixel. The first byte, in absolute mode, contains zero, the second contains the
number of color indexes that follow, and subsequent bytes contain color indexes in their high- and low-order nibbles, one color index for each pixel. Each run must be aligned on a word boundary (Meyer, 1997).

Run-length encoding is a very simple, intuitive, and fairly effective compression scheme for bitmapped graphics. RLE basically tries to detect repeating pixels on each row of an image and output them as a pixel count and pixel value pair, instead of outputting the repeated pixels individually. For stipple patterns or scanned images, which do not have repeating pixels in rows, the encoding may actually be larger after RLE encoding. Despite this limitation, RLE is very good for other types of images, and is supported by the BMP as well as many others. Run-length encoding scans all of the original data in rows and converts it into count byte-data group pairs. Two types of groupings may be done. The first is a run, which is a group of repeating pixel values encoded as a count of pixels followed by the pixel value. The second grouping type is a sequence, which is a group of non-repeating bytes. A sequence is encoded as a count of pixels followed by the pixel values. To make the logic simpler, all runs and sequences may not extend past the end of a scan line. A run of n bytes is given the count 1 - n, while a sequence of j bytes is given the count j - 1 to distinguish between the two. The example below shows the RLE encoding for a series of bytes that has five occurrences of the byte A and then the sequence BCDE. The run is encoded as the count -4 followed by the value A, and the sequence is encoded as the count 3 followed by the values BCDE.

Original: AAAAAABCDE

Encoded: -4A3BCDE

RLE encoding is more complex than decoding, as is common for many compression schemes. Any input stream can be encoded in many different ways, making the optimal encoding difficult to determine. For this reason, using
an intelligent but simple encoder is preferred over a fully optimal one (Meyer, 1997).

LZW Compression

First published in 1977 by J. Ziv and A. Limpel, and later refined by T. Welsh, LZW compression became popularized due to its use in the GIF file format and high compression obtained on virtually all images. The goal of LZW compression is to replace repeating input strings with n-bit codes by generating a string table on the fly, which is a mapping between pixel values and compression codes. As the encoder processes the data, the string table is built. The decoder can then reconstruct the string table as it processes the compressed data depending on the encoding method used. This differs from other compression algorithms, such as Huffman coding, where the lookup table needs to be included with the compressed data. Since images contain common groupings of pixels, LZW scans the image data encoding as large a grouping of pixels as possible with an encoding from the string table, placing unrecognized groupings into the string table so they can be compressed on later occurrences. For an image with n-bit pixel values, it uses compression codes that are n + 1 bits or larger. The size of the compression code limits the size of the string table, however, a smaller compression code helps gain larger amounts of compression. For example, a common arrangement is 12 bit compression codes for 8 bit per pixel images, which allows for 4096 entries in the string table. If the encoder runs out of space in the string table, the traditional LZW encoder must abort and try again with a larger compression code size. The string table is initialized at the start of compressing or decompressing to contain all the possible single pixel values in the image. The encoder then loads pixel values from the source image until it has accumulated a string that is not in the string
table. The compression code for the string minus the newly added pixel value is placed in the string table to be recognized later. The process continues until the entire image has been processed. The basic LZW algorithm is extended by the GIF file format to add two additional roots to the string table: a clear code and an end of input code. When the string table is full, a clear code is used causing the string table to be re-initialized with only the roots. The end of input code is used as a convenience for the decompressor, as it simplifies the logic (Meyer, 1997).
CHAPTER 9

CREATING A MUSICAL OCR

Researchers have investigated various ways of creating a musical OCR system and for numerous reasons. The main concern in this thesis was to find an effective algorithm that would eventually enable the storage of music books on the computer for reproduction when requested. Since image files consume a tremendous amount of disk space, some form of compression will need to be considered, utilizing OCR techniques to convert the image files into usable data.

The Process

To obtain images of music, five pieces of music ranging from beginners to intermediate level were scanned using a Hewlett Packard Scanjet -5p scanner saving them as tiff files and then converting them into the BMP format. Four of the images were taken from music books whereas one was an individual sheet. The next step was to read the BMP file. David Charlap's (1995) files were utilized to read in the images and print the RGB values of the pixels into an output file. By modifying, his "test.c" file, the staff lines were sought as the first piece of musical notation to be identified. The hypothesis is that once a staff line is detected it's associated row and column can be saved in a file which can later be read by another program to reproduce the lines. Since the images contain
shades of gray, a hexadecimal value was chosen as a threshold to indicate between what should be considered black and white areas. To help determine this threshold value, an image with black lines and several shades of gray was created to find what hexadecimal value gave the best results. When setting the hex value at 0x90, all gray areas were detected whereas setting the hex value to 0x50 only detected the black lines. A line ideally should consist of successive blackish pixels, but with the fragmentations associated with scanning, a value also had to be picked to determine when enough blackish pixels in a row could be considered a line assuming the image was scanned with lines horizontal on the page.

Results

First a hexadecimal value of 0x90 was chosen as the threshold between what would be considered black and white pixels, and a value m of 150 or greater consecutive blackish pixels in a row would indicate an existence of a line. The file "notes1.bmp" (see Figure 1) was tested first with only an indication of the last four lines of the last staff line being indicated. The hexadecimal value was lowered to eliminate the whiter shades of gray and checked with various m values ranging from 88 to 150. None of the results were agreeable for staff line detection, either getting an indication of too many staff lines or not enough staff lines. The problem would eventually be associated to the scanned image containing staff lines that had shown through from the back of the page. "Notes2.bmp" (see Figure 2) was then considered a better candidate for testing since "notes3.bmp" and "notes4.bmp" (see Figures 3 and 4) both were slightly tilted when scanned and would not be able to be detected using this method without further modifications. Using a hex value of 0x50 and m >= 150, "notes2.bmp" (see Figure 2) was checked for staff lines resulting in no detection.
This gave the indication that either lighter shades of gray were part of the lines or that the lines had breaks in them and did not consist of at least 150 consecutive pixels. The hexadecimal value was then raised to 0x90 keeping the m value the same. This resulted in one staff line being detected at the beginning of the image and four equally spaced staff lines being detected at the end of the image. With a beginning indication of staff line recognition, the hex value was kept constant and the m value was lowered to 100 to take into consideration any fragmentation or breaks that may occur in the lines. These values resulted in an indication of more lines, but not enough and some that seemed to be too thick. This may be the result of several lines being recognized as one thick line due to extra dots on the page along with possible overlapping of other symbols. The values were then changed to 0x9f and m >=75 resulting in an indication of too many staff lines. The values of 0x88 and m >= 100 were finally settled on as giving the best results and these were then tested on the image "music.bmp" (see Figure 5) since this image had no indication of staff lines being scanned in from the back of the page as did the other images. First the program was run producing an output file that demonstrated how many blackish pixels in a row were detected given the above threshold values. A modified version of this, only showing the rows that actually indicated a possible existence of a line, can be seen in Appendix B. A near final version of the output file can be seen in Figure 6 indicating the image's dimensions along with the row's number that contained at least 100 blackish pixels in a row and the column number indicating where the last such pixel was detected. By comparing the two files, mainly the number of pixels with the ending column number, an indication of what part of the staff line was detected as well as an approximation of where the staff lines begin and end can be determined. This gave very promising and consistent results indicating
five staff lines averaging three rows thick and five rows apart and four staffs total
with approximately 169 rows between each.
FIGURE 1 First Scanned Image Tested by Program. (Staff Lines From Back of Page were Picked Up When Scanned Resulting in Detection Problems.)
FIGURE 2 Second Scanned Image Tested. (Best Candidate From Among the Images Taken From Music Books.)
Minuet

Andante grazioso

Wolfgang Amadeus Mozart

FIGURE 3 Tilted Image. (Image not Flat Enough to get a Straight Image From Music Book.)
FIGURE 4 Music Book Image also Slightly Tilted.
Let The Love Lights In
by G. V. Russell

Hear our children, Trusting voices moves faith. Guide

a child's world. Families value their fate. See a

small world through the eyes of love. Children need friends.

Gentle hearts touched, sharing language of love. Friends

FIGURE 5 Single Sheet of Music Scanned. (Produced Best Results From Algorithm.)
<table>
<thead>
<tr>
<th>Row</th>
<th>Col</th>
<th>Row</th>
<th>Col</th>
<th>Row</th>
<th>Col</th>
</tr>
</thead>
<tbody>
<tr>
<td>125</td>
<td>263</td>
<td>126</td>
<td>529</td>
<td>127</td>
<td>651</td>
</tr>
<tr>
<td>128</td>
<td>651</td>
<td>133</td>
<td>267</td>
<td>134</td>
<td>548</td>
</tr>
<tr>
<td>135</td>
<td>651</td>
<td>140</td>
<td>145</td>
<td>141</td>
<td>403</td>
</tr>
<tr>
<td>142</td>
<td>651</td>
<td>143</td>
<td>651</td>
<td>149</td>
<td>403</td>
</tr>
<tr>
<td>150</td>
<td>651</td>
<td>151</td>
<td>651</td>
<td>156</td>
<td>150</td>
</tr>
<tr>
<td>157</td>
<td>469</td>
<td>158</td>
<td>651</td>
<td>159</td>
<td>651</td>
</tr>
<tr>
<td>328</td>
<td>310</td>
<td>329</td>
<td>568</td>
<td>330</td>
<td>651</td>
</tr>
<tr>
<td>336</td>
<td>303</td>
<td>337</td>
<td>545</td>
<td>338</td>
<td>651</td>
</tr>
<tr>
<td>344</td>
<td>423</td>
<td>345</td>
<td>651</td>
<td>346</td>
<td>651</td>
</tr>
<tr>
<td>352</td>
<td>370</td>
<td>353</td>
<td>651</td>
<td>354</td>
<td>651</td>
</tr>
<tr>
<td>359</td>
<td>197</td>
<td>360</td>
<td>527</td>
<td>361</td>
<td>651</td>
</tr>
<tr>
<td>362</td>
<td>651</td>
<td>531</td>
<td>253</td>
<td>532</td>
<td>568</td>
</tr>
<tr>
<td>533</td>
<td>652</td>
<td>539</td>
<td>202</td>
<td>540</td>
<td>578</td>
</tr>
<tr>
<td>541</td>
<td>652</td>
<td>547</td>
<td>310</td>
<td>548</td>
<td>648</td>
</tr>
<tr>
<td>549</td>
<td>652</td>
<td>555</td>
<td>290</td>
<td>556</td>
<td>651</td>
</tr>
<tr>
<td>557</td>
<td>652</td>
<td>563</td>
<td>383</td>
<td>564</td>
<td>651</td>
</tr>
<tr>
<td>565</td>
<td>652</td>
<td>735</td>
<td>361</td>
<td>736</td>
<td>651</td>
</tr>
<tr>
<td>737</td>
<td>652</td>
<td>743</td>
<td>436</td>
<td>744</td>
<td>652</td>
</tr>
<tr>
<td>745</td>
<td>652</td>
<td>751</td>
<td>389</td>
<td>752</td>
<td>652</td>
</tr>
<tr>
<td>753</td>
<td>652</td>
<td>759</td>
<td>560</td>
<td>760</td>
<td>652</td>
</tr>
<tr>
<td>767</td>
<td>505</td>
<td>768</td>
<td>652</td>
<td>769</td>
<td>652</td>
</tr>
</tbody>
</table>

**FIGURE 6** Output File. (music.bmp Showing Row and Column Number Only for the Rows Containing 100 or More Black Pixels in a Row.)
CHAPTER 10

CONCLUSION

This thesis and the research provide advance insight into some of the various applications dealing with the fields of computers and music. The application of OCR techniques to music, for instance, will open a wide range of opportunities from providing a vast amount of music for storage on disk to being able to play and or edit a piece of music. While attempting to create such a musical OCR program several problems were encountered that will need to be taken into account for future improvement.

Various considerations such as fragmentations and tilted images need to be identified in order to obtain a more versatile musical OCR. By beginning with staff line detection, the program can eventually be extended to include more musical notation such as the notes, clefs, and phrase markings. Existing OCR systems could also be included so that lyrics as well as other text may be recognized.

Obtaining images from music books created problems as well. As shown in this thesis, images can easily come out tilted especially if a thick book is used. Because of physical limitations, it is difficult to provide an image flat enough on the scanner to obtain an image that is not tilted or curved at the ends when using books of music. The problem of the scanner picking up print from the back
of the page also exist when scanning from books.

The scanning quality should also be a concern so that noise and fragmentation may be minimized. By reducing the noise on an image, a higher recognition rate may be achieved. Dots produced by noise may occur beside, above, or below notes resulting in an incorrect recognition. Modifications to the program would be extremely difficult to create one that would be able to distinguish musical notation dots, such as staccato marks, from noise if such dots happen to occur in correctly positioned places. A comparison of the original image with the scanned one would need to be implemented to distinguish between "true" dots and noise.

Symbols that overlap or touch, such as the staff lines and notes, or accidentals touching notes, need to be considered. Most existing programs remove the symbols that have been recognized but this may cause fragmentations in the notes for example. This seems, however, the most plausible solution. Ledger lines provide another complication since they exist away from the regular staff lines. The number of lines on a note outside the staff region will need to be recognized as well as if the line goes through the note or rest below it in order to determine the pitch of the note.

Through the use of musical notation analysis knowledge, a complete OCR system that successfully recognizes most existing music may be obtained. Extensive open problems remain such as system organization and image noise. Scaling up a working prototype into a complete system becomes another difficulty. For example, a manageable grammar organization containing 30 production rules may become unmanageable when hundreds more are added. The difficulty in extending a system that presently analyzes monophonic music to one that analyzes polyphonic music becomes hard to predict. Methods that interpret music notation even when affected by image noise or symbol...
fragmentation due to poor print quality is essential in providing reliable optical score reading. OCR techniques become most useful with music in such applications as score editing and sound reproduction of music images.

If the application is to store compressed forms of music books and sheets on the computer for later printing of the images then image compression techniques such as RLE-encoding would be more productive. By choosing a threshold value of what should be considered black and white, one could simply store the number of blackish or whitish pixels in a row which would be used for later reproduction of the image. This will more likely produce greater compression on most images and problems due to overlapping of symbols and pitch identification would no longer exist.

Other fields of computer generated music such as fractal music further connects the fields of music and computer science. The mathematical basis inherit in music gives a promising indication that new sounds and new forms of music can be made possible through advance study of the combination of these two fields assisting researchers and presenting musical technology to students and music lovers.
APPENDIX A

SOURCE CODES

Listed are the source programs used to read a BMP file image of music, picking out the staff lines for reproduction. Bmptypes.h, endian.c, endian.h, readbmp.c, readbmp.h, and the makefile were taken from Dr. Dobbs Journal Web site at http://www.ddj.com to read in a BMP file. Test.c was modified and called t1.c to pick out the staff lines.

/* bmptypes.h
 * Data types used in bitmap files.
 */

#ifndef__BMPTYPES_H_INCLUDED_
#define__BMPTYPES_H_INCLUDED_

#define Data types.
#
* INT8 is an integer of at least 8 bits wide.
* INT16 is an integer of at least 16 bits wide.
* INT32 is an integer of at least 32 bits wide.
* UINT8 is an unsigned INT8
* UINT16 is an unsigned INT16
* UINT32 is an unsigned INT32
*/

typedef char INT8;
typedef short INT16;

87

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
typedef long INT32;
typedef unsigned char UINT8;
typedef unsigned short UINT16;
typedef unsigned long UINT32;

/**************************************************************************
 *
 * Constants. Each set corresponds to a field in a structure. Note that some
 * sets only have one value. Default values for all fields are the value
 * corresponding to 0.
 */

/*
 *
 * Constants used in the "type" field of BITMAPFILEHEADER and
 * BITMAPARRAYHEADER structures. Note that these are all two-character
 * mnemonics as well as integer constants.
 */
#define TYPE_ICO (0x4349) /**<IC**/
#define TYPE_BMP (0x4d42) /**<BM**/
#define TYPE_PTR (0x5450) /**<PT**/
#define TYPE_ICO_COLOR (0x4943) /**<CI**/
#define TYPE_PTR_COLOR (0x5043) /**<CP**/
#define TYPE_ARRAY (0x4142) /**<BA**/

/*
 *
 * Compression schemes. Note that BITFIELDS (from NT) uses the same
 * number as
 * HUFFMAN1D (from OS/2)
 */
#define COMPRESSION_NONE (0)
#define COMPRESSION_RLE_8 (1)
#define COMPRESSION_RLE_4 (2)
#define COMPRESSION_HUFFMAN1D (3)
#define COMPRESSION_BITFIELDS (3)
#define COMPRESSION_RLE_24 (4)
#define COMPRESSION_LAST (4)

/*
 *
 * units of resolution
 */
#define UNITS_PELS_PER_METER (0)
#define UNITS_LAST (0)

/*
 *
 * origin of coordinate space
 */

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
/*
#define ORIGIN_LOWER_LEFT (0)
#define ORIGIN_LAST (0)
/
/*
* halftoning algorithms
*/
#define HALFTONING_NONE (0)
#define HALFTONING_ERROR_DIFFUSION (1)
#define HALFTONING_PANDA (2)
#define HALFTONING_SUPER_CIRCLE (3)
#define HALFTONING_LAST (3)
/
/*
* color table encoding
*/
#define COLOR_ENCODING_RGB (0)
#define COLOR_ENCODING_LAST (0)

/***************************************************************************/
*
* Structures.
*/

/*
* BITMAPFILEHEADER defines a single bitmap image. Its analogue in the
* Windows SDK is the BITMAPFILEHEADER. Its analogues in the OS/2 Toolkit
* are
* the BITMAPFILEHEADER and BITMAPFILEHEADER2 structures.
*
* A BITMAPHEADER structure is always concatenated to the end of a
* BITMAPFILEHEADER structure.
*/
typedef struct BITMAPFILEHEADER
{
    UINT16   type;
    UINT32   size;
    INT16    xHotspot;
    INT16    yHotspot;
    UINT32   offsetToBits;
} BITMAPFILEHEADER;

/*
* BITMAPARRAYHEADER is used to establish a linked list of
* BITMAPFILEHEADER
*/
* structures for a bitmap file with multiple images in it. There is no
* equivalent structure in the Windows SDK. Its analogues in the OS/2 toolkit
* are the BITMAPARRAYFILEHEADER and BITMAPARRAYFILEHEADER2
* structures.

* A BITMAPFILEHEADER structure is always concatenated to the end of a
* BITMAPARRAYHEADER structure.
*/
typedef struct BITMAPARRAYHEADER
{
    UINT16 type;
    UINT32 size;
    UINT32 next;
    UINT16 screenWidth;
    UINT16 screenHeight;
} BITMAPARRAYHEADER;

/*
* BITMAPHEADER defines the properties of a bitmap. Its analogues in the
* Windows SDK are the BITMAPCOREINFOHEADER and
* BITMAPINFOHEADER structures.
* Its analogues in the OS/2 Toolkit are the BITMAPINFOHEADER and
* BITMAPINFOHEADER2 structures.
*
* A color table is concatenated to this structure. The number of elements in
* the color table determined by the bit-depth of the image.
*
* Note, that if the field "size" is 12 or less, then the width and height
* fields should be read as UINT16's instead of UINT32's.
*
* Also note that if the field "size" is greater than 12, then the color table
* will have an extra byte of padding between each structures (to longword
* align it)
*
* The different sizes for the width, height, and color table are the only
* differences between the "old" and "new" bitmap file formats.
*/
typedef struct BITMAPHEADER
{
    UINT32 size;
    INT32 width;
    INT32 height;
    UINT16 numBitPlanes;
    UINT16 numBitsPerPlane;
    UINT32 compressionScheme;

typedef struct RGB
{
    UINT8 blue;
    UINT8 green;
    UINT8 red;
} RGB;

#ifndef __BMTYPES_H_INCLUDED__ /*
    RGB defines a single color palette entry. Its analogues in the Windows SDK
    are the RGBTRIPLE and RGBQUAD structures. Its analogues in the OS/2
    Toolkit are the RGB and RGB2 structure.
*/
#endif /*__BMTYPES_H_INCLUDED__ */

/*
 * Formatting information for emacs in c-mode
 */

* Local Variables:
  * c-indent-level:4
  * c-continued-statement-offset:4
  * c-brace-offset:-4
  * c-brace-imaginary-offset:0
  * c-argdecl-indent:4
  * c-label-offset:-4
  * End:
  */
/* endian.c
 * These functions read and write our basic integer types from a little-endian
 * file. The endian and word-size of the host machine will not affect this
 * code. The only assumption made is that the C data type (char) is one byte
 * long. This should be a safe assumption.
 */

#include <stdio.h>
#include "bmtypes.h"
#include "endian.h"

/*
 * Read functions. All read functions take an open file pointer as the first
 * parameter and a pointer to data as the second parameter. The return value
 * will be 0 on success, and EOF on failure. If successful, the second
 * parameter will point to the data read.
 */

/*
 * The INT8 and UINT8 types are stored as a single byte on disk. The INT8
 * type is a signed integer with range (-128..127). The UINT8 type is an
 * unsigned integer with range (0..255).
 */

int readINT8little(FILE *f, INT8 *i)
{
    int rc;

    rc = fgetc(f);
    if (rc == EOF)
        return rc;

    *i = (rc & 0xff);
    return 0;
}

int readUINT8little(FILE *f, UINT8 *i)
{
    int rc;

    rc = fgetc(f);
    if (rc == EOF)
        return rc;

    *i = (rc & 0xff);
    return 0;
}
The INT16 and UINT16 types are stored as two bytes on disk. The INT16 type is a signed integer with range \((-32768..32767)\). The UINT16 type is an unsigned integer with range \((0..65535)\).

```c
int readlNT16little(FILE *f, INT16 *i)
{
    int rc;
    INT16 temp = 0;
    temp = (fgetc(f) & 0xff);
    rc = fgetc(f);
    if (rc == EOF)
        return rc;
    temp = ((rc & 0xff) << 8);
    *i = temp;
    return 0;
}

int readUINT16little(FILE *f, UINT16 *i)
{
    int rc;
    UINT16 temp = 0;
    temp = (fgetc(f) & 0xff);
    rc = fgetc(f);
    if (rc == EOF)
        return rc;
    temp = ((rc & 0xff) << 8);
    *i = temp;
    return 0;
}
```

The INT32 and UINT32 types are stored as four bytes on disk. The INT32 type is a signed integer with range \((-2147483648..2147483647)\). The UINT32 type is an unsigned integer with range \((0..4294967295)\).

```c
int readlNT32little(FILE *f, INT32 *i)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
{  
  int rc;
  INT32 temp = 0;

  temp = ((long)fgetc(f) & 0xff);
  temp |= (((long)fgetc(f) & 0xff) << 8);
  temp |= (((long)fgetc(f) & 0xff) << 16);

  rc = fgetc(f);
  if (rc == EOF)
    return rc;

  temp |= (((long)rc & 0xff) << 24);
  *i = temp;
  return 0;
}

int readUINT32little(FILE *f, UINT32 *i)
{
  int rc;
  UINT32 temp = 0;

  temp = ((long)fgetc(f) & 0xff);
  temp |= (((long)fgetc(f) & 0xff) << 8);
  temp |= (((long)fgetc(f) & 0xff) << 16);

  rc = fgetc(f);
  if (rc == EOF)
    return rc;

  temp |= (((long)rc & 0xff) << 24);
  *i = temp;
  return 0;
}

//***************************************************************************
*  
*  Write functions. All write functions take an open file pointer as the first  
*  parameter and a data as the second parameter. The return value will be 0 on 
*  success, and EOF on failure. If successful, the second parameter will have 
*  been written to the open file. 
*/

int writeINT8little(FILE *f, INT8 i)
{
  return fputc(i, f);
}
int writeUINT8little(FILE *f, UINT8 i)
{
    return fputc(i, f);
}

int writeINT16little(FILE *f, INT16 i)
{
    int rc;
    rc = fputc((i & 0xff), f);
    if (rc == EOF)
        return rc;
    return fputc(((i >> 8) & 0xff), f);
}

int writeUINT16little(FILE *f, UINT16 i)
{
    int rc;
    rc = fputc((i & 0xff), f);
    if (rc == EOF)
        return rc;
    return fputc(((i >> 8) & 0xff), f);
}

int writeINT32little(FILE *f, INT32 i)
{
    int rc;
    rc = fputc((i & 0xff), f);
    if (rc == EOF)
        return rc;
    rc = fputc(((i >> 8) & 0xff), f);
    if (rc == EOF)
        return rc;
    rc = fputc(((i >> 16) & 0xff), f);
    if (rc == EOF)
        return rc;
    return fputc(((i >> 24) & 0xff), f);
}
int writeUINT32little(FILE *f, UINT32 i)
{
    int rc;
    rc = fputc((i & 0xff), f);
    if (rc == EOF)
        return rc;
    rc = fputc(((i >> 8) & 0xff), f);
    if (rc == EOF)
        return rc;
    rc = fputc(((i >> 16) & 0xff), f);
    if (rc == EOF)
        return rc;
    return fputc(((i >> 24) & 0xff), f);
}
/* endian.h
 * This is the header for endian.c - functions to read/write our
 * INT8, INT16 and INT32 types from/to a little-endian file.
 */

#ifndef __ENDIAN_H_INCLUDED__
#define __ENDIAN_H_INCLUDED__

/*
 * Read the basic types as little-endian values. The return value will be
 * zero if successful, EOF, otherwise.
 */
int readINT8little(FILE *, INT8 *i);
int readINT16little(FILE *, INT16 *i);
int readINT32little(FILE *, INT32 *i);
int readUINT8little(FILE *, UINT8 *i);
int readUINT16little(FILE *, UINT16 *i);
int readUINT32little(FILE *, UINT32 *i);

/*
 * Write the basic types as little-endian values. The return value will be
 * zero if successful, EOF otherwise.
 */
int writeINT8little(FILE *, INT8 i);
int writeINT16little(FILE *, INT16 i);
int writeINT32little(FILE *, INT32 i);
int writeUINT8little(FILE *, UINT8 i);
int writeUINT16little(FILE *, UINT16 i);
int writeUINT32little(FILE *, UINT32 i);

#endif /* __ENDIAN_H_INCLUDED__ */
/* readbmp.c
 * This file contains mid-level functions for reading bitmap structures and
 * high-level functions that read bitmap files.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "bmptypes.h"
#include "endian.h"
#include "readbmp.h"

//****************************************************************************
* Mid-level functions.
*
* These functions read in the various structures defined in bmptypes.h. Each
* function assumes that the file pointer is positioned at the start of the
* given structure. Upon completion, each function will leave the file
* pointer positioned on the byte immediately following the structure. Return
* values will be 0 for success and non-zero for failure. In all cases, a
* return value of non-zero means that the file position has been left in an
* indeterminate state and further reading should not be attempted.
*/

/*
 * Read a BITMAPFILEHEADER structure.
 */
int readBitmapFileHeader(FILE *fp, BITMAPFILEHEADER *bfh)
{
    int rc;
    rc = readUINT16little(fp, &bfh->type);
    if (rc != 0)
        return rc;
    rc = readUINT32little(fp, &bfh->size);
    if (rc != 0)
        return rc;
    rc = readlNT16little(fp, &bfh->xHotspot);
    if (rc != 0)
        return rc;
    rc = readINT16little(fp, &bfh->yHotspot);
}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
if (rc != 0)
    return rc;
rc = readUINT32little(fp, &(bfh->offsetToBits));
return rc;

/*
 * Read a BITMAPARRAYHEADER
 */
int readBitmapArrayHeader(FILE *fp, BITMAPARRAYHEADER *bah)
{
    int rc;

    rc = readUINT16little(fp, &(bah->type));
    if (rc != 0)
        return rc;
    rc = readUINT32little(fp, &(bah->size));
    if (rc != 0)
        return rc;
    rc = readUINT32little(fp, &(bah->next));
    if (rc != 0)
        return rc;
    rc = readUINT16little(fp, &(bah->screenWidth));
    if (rc != 0)
        return rc;
    rc = readUINT16little(fp, &(bah->screenHeight));
    return rc;
}

/*
 * Read the BITMAPHEADER structure. This one requires a bit of sanity
 * checking. The length of the structure on the disk is specified in the
 * first field. We must stop reading after that many bytes, and if that value
 * is longer than sizeof(BITMAPHEADER), we must skip over any leftover bytes.
 * Finally, if size is 12, then width an height are really 16-bit values, and
 * we have to read them differently so they'll be properly stored in the
 * 32-bit fields BITMAPHEADER uses.
 */
int readBitmapHeader(FILE *fp, BITMAPHEADER *bh)
{
    int rc, oldFormat, bytesRead;
    UINT16 tempVal;

    /*
     * Clear the structure. Default values for all fields are zeros. This
will prevent garbage from being returned if the structure is truncated on disk.

```
memset(bh, 0, sizeof(BITMAPHEADER));
```

/*
 * Read the size of the structure. From here on in, we'll have to be sure we don't read more bytes than this value.
 */
rc = readUINT32little(fp, &(bh->size));
if (rc != 0)
    return rc;
bytesRead = 4;

/*
 * If the size is 12 bytes or less, than this is an "old format" structure. So the width and height fields will have to be read differently.
 */
if (bh->size <= 12)
    oldFormat = 1;
else
    oldFormat = 0;

/*
 * Width and height are read differently for old and new format files. In the old format, they're 16-bit values. In the new format, they're 32-bits long.
 */
if (oldFormat)
{
    rc = readINT16little(fp, &tempVal);
    if (rc != 0)
        return rc;
    bh->width = tempVal;
    bytesRead += 2;
}
else
{
    rc = readINT32little(fp, &(bh->width));
    if (rc != 0)
        return rc;
    bytesRead += 4;
}
if (bytesRead >= bh->size)
    return 0;
if (oldFormat)
{
    rc = readLNT16little(fp, &tempVal);
    if (rc != 0)
        return rc;
    bh->height = tempVal;
    bytesRead += 2;
}
else
{
    rc = readLNT32little(fp, &bh->height);
    if (rc != 0)
        return rc;
    bytesRead += 4;
}
if (bytesRead >= bh->size)
    return 0;
/*
 * From this point on, old and new formats are identical to each other,
 * and we can proceed as if there was no difference. For each field, we
 * read it in and increment the count of bytes read. If at any time we
 * have read the amount we got earlier (in the size field), then stop and
 * leave the rest of the fields as zeros.
 */
rc = readUINT16little(fp, &(bh->numBitPlanes));
if (rc != 0)
    return rc;
bytesRead += 2;
if (bytesRead >= bh->size)
    return 0;
rc = readUINT16little(fp, &(bh->numBitsPerPlane));
if (rc != 0)
    return rc;
bytesRead += 2;
if (bytesRead >= bh->size)
    return 0;
/*
 * Old format stop here. But we don't have to check, because in that
 * format, 12 bytes have been read and the function will have exited
 * without any extra checking.
 */
rc = readUINT32little(fp, &(bh->compressionScheme));
if (rc != 0)
    return rc;
bytesRead += 4;
if (bytesRead >= bh->size)
    return 0;

rc = readUINT32little(fp, &(bh->sizeOfImageData));
if (rc != 0)
    return rc;
bytesRead += 4;
if (bytesRead >= bh->size)
    return 0;

rc = readUINT32little(fp, &(bh->xResolution));
if (rc != 0)
    return rc;
bytesRead += 4;
if (bytesRead >= bh->size)
    return 0;

rc = readUINT32little(fp, &(bh->yResolution));
if (rc != 0)
    return rc;
bytesRead += 4;
if (bytesRead >= bh->size)
    return 0;

rc = readUINT32little(fp, &(bh->numColorsUsed));
if (rc != 0)
    return rc;
bytesRead += 4;
if (bytesRead >= bh->size)
    return 0;

rc = readUINT32little(fp, &(bh->numImportantColors));
if (rc != 0)
    return rc;
bytesRead += 4;
if (bytesRead >= bh->size)
    return 0;

rc = readUINT16little(fp, &(bh->resolutionUnits));
if (rc != 0)
    return rc;
bytesRead += 2;
if (bytesRead >= bh->size)
return 0;

rc = readUINT16little(fp, &(bh->padding));
if (rc != 0)
    return rc;
bytesRead += 2;
if (bytesRead >= bh->size)
    return 0;

rc = readUINT16little(fp, &(bh->origin));
if (rc != 0)
    return rc;
bytesRead += 2;
if (bytesRead >= bh->size)
    return 0;

rc = readUINT16little(fp, &(bh->halftoning));
if (rc != 0)
    return rc;
bytesRead += 2;
if (bytesRead >= bh->size)
    return 0;

rc = readUINT32little(fp, &(bh->halftoningParam1));
if (rc != 0)
    return rc;
bytesRead += 4;
if (bytesRead >= bh->size)
    return 0;

rc = readUINT32little(fp, &(bh->halftoningParam2));
if (rc != 0)
    return rc;
bytesRead += 4;
if (bytesRead >= bh->size)
    return 0;

rc = readUINT32little(fp, &(bh->colorEncoding));
if (rc != 0)
    return rc;
bytesRead += 4;
if (bytesRead >= bh->size)
    return 0;

rc = readUINT32little(fp, &(bh->identifier));
if (rc != 0)
return rc;
bytesRead += 4;
if (bytesRead >= bh->size)
    return 0;

/*
 * If there are more bytes in the file than this, then the file is using a
 * future format that doesn't exist yet. Skip over the bytes. Assuming
 * this future format somewhat resembles what we know now, ignoring the
 * fields will be safe. We MUST skip them, though, since the color
 * table begins on the byte after this structure, and we have to position
 * the file pointer there.
 */
return fseek(fp, (bh->size - bytesRead), SEEK_CUR);
}

/*
 * readRgb reads in a single RGB structure from the disk. The numBytes field
 * indicates how many bytes the field occupies on the disk. It assumes that
 * each component is one byte on disk and the rest is padding. This will
 * compensate for the old/new differences in color tables. (Old format
 * bitmaps use 3 bytes per entry, while new format bitmaps use 4.) Note how
 * it will never read more than the number of bytes requested.
 */
int readRgb(FILE *fp, RGB *rgb, int numBytes)
{
    int rc;

    if (numBytes == 0)
        return 0;
    rc = readUINT8little(fp, &(rgb->blue));
    if (rc != 0)
        return rc;

    if (numBytes == 1)
        return 0;
    rc = readUINT8little(fp, &(rgb->green));
    if (rc != 0)
        return rc;

    if (numBytes == 2)
        return 0;
    rc = readUINT8little(fp, &(rgb->red));
    if (rc != 0)
        return rc;
if (numBytes == 3)
    return 0;

    /* Skip over extra bytes if more than three were requested */
    return fseek(fp, (numBytes - 3), SEEK_CUR);
}]

/*
  * A color table is a block of RGB structures, all the same size. Read it by
  * calling readRgb repeatedly.
  */
int readColorTable(FILE *fp, RGB *rgb, int numEntries, int numBytesPerEntry)
{
    int i, rc;

    for (i=0; i<numEntries; i++)
    {
        rc = readRgb(fp, &(rgb[i]), numBytesPerEntry);
        if (rc != 0)
            return rc;
    }
    return 0;
}

/*
  * ReadBitsUncompressed. Reads pixel values into an array of RGB
  * values. It assumes that there is no compression. Note that there we're
  * only handling bit depths of 1, 4, 8, 16, and 24. Note that OS/2 bitmaps
  * can (in theory) have any number of bits per pixel, so you might find a
  * strange bitmap file that this can't handle, but the chances of finding such
  * a file this are nearly zero.
  *
  * Each row of pixels is always padded to a 4-byte boundary.
  */
int readBitsUncompressed(FILE *fp, RGB *image, int width, int height,
    int depth, RGB *colorTable)
{
    UINT8 temp;
    int   rc, padBytes, i;
    long row, column, pixel, value;

    switch (depth) {
    case 1:
        /*
* For 1 bit per pixel, each byte is 8 pixels. Each one is an index
* into the color table (1 or 0). Most significant byte first. All
* is padded to 32-bit boundaries as well.
*/
pixel = 0;
if (((width % 32) == 0) || ((width % 32) > 24))
  padBytes = 0;
else if ((width % 32) <= 8)
  padBytes = 3;
else if ((width % 32) <= 16)
  padBytes = 2;
else
  padBytes = 1;

for (row = height; row > 0; row--)
{
  for (column = width; column > 0; column -= 8)
  {
    rc = readINT8little(fp, &temp);
    if (rc != 0)
      return rc;
    for (i=0; i < ((column < 8) ? column : 8); i++)
    {
      /*
      * For each byte read, bit-decompose it. Note that the
      * last byte on a row could have less than 8 bits used.
      * Most significant bits come first.
      */
      value = ((temp & (1 << (7-i))) == 0) ? 0 : 1;
      image[pixel].red = colorTable[value].red;
      image[pixel].green = colorTable[value].green;
      image[pixel].blue = colorTable[value].blue;
      pixel++;
    }
  }
  if (padBytes != 0)
  {
    rc = fseek(fp, padBytes, SEEK_CUR);
    if (rc != 0)
      return rc;
  }
}
break;

case 4:
/*
* For 4 bits per pixel, each byte is two pixels. The upper half go to
* the first pixel, and the lower half to the second.
*/

pixel = 0;
if (((width % 8) == 0) || ((width % 8) > 6))
    padBytes = 0;
else if ((width % 8) <= 2)
    padBytes = 3;
else if ((width % 8) <= 4)
    padBytes = 2;
else
    padBytes = 1;

for (row = height; row > 0; row--)
{
    for (column = width; column > 0; column -= 2)
    {
        /*
         * Each byte here is two pixels. Note that the last byte on a
         * row may only contain one pixel.
         */
        rc = readUINT8little(fp, &temp);
        if (rc != 0)
            return rc;
        /*
         * First pixel is the upper 4 bits
         */
        value = temp >> 4;
        image[pixel].red = colorTable[value].red;
        image[pixel].green = colorTable[value].green;
        image[pixel].blue = colorTable[value].blue;
        pixel++;
        /*
         * Second pixel is lower 4 bits. If this is the last byte in
         * the row, and there are an odd number of pixels per row, then
         * this is not valid data.
         */
        if (column == 1)
        {
            value = temp & 0x0f;
            image[pixel].red = colorTable[value].red;
            image[pixel].green = colorTable[value].green;
            image[pixel].blue = colorTable[value].blue;
            pixel++;
        }
}
if (padBytes != 0)
{
    rc = fseek(fp, padBytes, SEEK_CUR);
    if (rc != 0)
        return rc;
}
}

break;

case 8:
/*
 * For 8 bits per pixel, each byte is one pixel.
 */

pixel = 0;
padBytes = ((width % 4) == 0) ? 0 : (4 - (width % 4));

for (row=height; row > 0; row--)
{
    for (column=width; column > 0; column--)
    {
        rc = readUINT8little(fp, &temp);
        if (rc != 0)
            return rc;
        image[pixel].red = colorTable[temp].red;
        image[pixel].green = colorTable[temp].green;
        image[pixel].blue = colorTable[temp].blue;
        pixel++;
    }
    if (padBytes != 0)
    {
        rc = fseek(fp, padBytes, SEEK_CUR);
        if (rc != 0)
            return rc;
    }
}

break;

case 16:
/*
 * For 16 bits per pixel, you must read two bytes per pixel. But
 * there's a catch. The data is big endian! This is because all pixel
 * data (for all formats, actually) is stored as a packed array,
 * stored in pixel order.
 */
pixel = 0;
padBytes = ((width % 2) == 0) ? 0 : 2;
for (row=height; row > 0; row--)
{
    for (column=width; column > 0; column--)
    {
        /*
         * Read a 16-bit integer as big endian. Do this by reading
         * two bytes and mathematically combine them. After that,
         * proceed as usual.
         */
        rc = readUINT8little(fp, &temp);
        if (rc != 0)
            return rc;
        value = ((long)temp) << 8;
        rc = readUINT8little(fp, &temp);
        if (rc != 0)
            return rc;
        value |= temp;
        image[pixel].red = colorTable[value].red;
        image[pixel].green = colorTable[value].green;
        image[pixel].blue = colorTable[value].blue;
        pixel++;
    }
    if (padBytes != 0)
    {
        rc = fseek(fp, padBytes, SEEK_CUR);
        if (rc != 0)
            return rc;
    }
}
break;

case 24:
/*
 * For 24 bits per pixel, it's an RGB structure. Note that the color
 * table is ignore for bit depths greater than 24 bits.
 */
pixel = 0;
padBytes = width % 4;

for (row=height; row > 0; row--)
{
    for (column=width; column > 0; column--)
    {
        rc = readRgb(fp, image+pixel, 3);
    }
}
110

```c
      pixel++;
    }
    if (padBytes != 0)
      {
      rc = fseek(fp, padBytes, SEEK_CUR);
      if (rc != 0)
        return rc;
      }
    }
    break;
  }

  return 0;
}

/*
 * ReadMaskBitsUncompressed. Reads a monochrome mask into an array of
 * characters. It assumes that there is no compression. This is very similar
 * (internally) to the readBitsUncompressed function. Note that if the data
 * read isn't really one-bit-deep data, you'll probably get garbage back.
 */
int readMaskBitsUncompressed(FILE *fp, char *image, int width, int height)
{
  UINT8 temp;
  int rc, padBytes, i;
  long row, column, pixel;
  char value;

  /*
  * see the one-bit-depth part of readBitsUncompressed for comments
  */
  pixel = 0;
  if (((width % 32) == 0) || ((width % 32) > 24))
    padBytes = 0;
  else if ((width % 32) <= 8)
    padBytes = 3;
  else if ((width % 32) <= 16)
    padBytes = 2;
  else
    padBytes = 1;

  for (row = height; row > 0; row--)
    {
      for (column = width; column > 0; column -= 8)
      {
```
rc = readINT8little(fp, &temp);
if (rc != 0)
    return rc;
for (i=0; i < ((column < 8) ? column : 8); i++)
{
    value = ((temp & (1 << (7-i))) == 0) ? 0 : 1;
    image[pixel] = value;
    pixel++;
}
}
}
if (padBytes != 0)
{
    rc = fseek(fp, padBytes, SEEK_CUR);
    if (rc != 0)
        return rc;
}
return 0;

/*
* reflectYRGB takes an array of RGB values and the dimensions they represent
* and flips it vertically. This will convert a bottom-left origin array to a
* top-left origin array.
*/
void reflectYRGB(RGB *image, int width, int height)
{
    int row, col;
    RGB temp;
    for (row = 0; row < (height / 2); row++)
    {
        for (col = 0; col < width; col++)
        {
            /* Swap pixels at (x,y) with (x, height-y) */
            memcpy(&temp, image+(row * width + col), sizeof(RGB));
            memcpy(image+(row * width + col),
                    image+((height - row - 1) * width + col), sizeof(RGB));
            memcpy(image+((height - row - 1) * width + col), &temp,
                    sizeof(RGB));
        }
    }
}
* reflectYchar takes an array of char values and the dimensions they
  * represent and flips it vertically. This will convert a bottom-left origin
  * array to a top-left origin array.
  */
void reflectYchar(char *image, int width, int height)
{
    int row, col;
    char temp;

    for (row = 0; row < (height / 2); row++)
    {
        for (col = 0; col < width; col++)
        {
            /* Swap values at (x,y) with (x,height-y) */
            temp = image[row * width + col];
            image[row * width + col]=image[(height - row - 1) * width + col];
            image[(height - row - 1) * width + col] = temp;
        }
    }
}

/*********************************************/
/*
* High-level functions
*
* These functions read in specific types of bitmap files. Each assumes that
* the file pointer is positioned at the appropriate place in a bitmap file.
* (At the start of a BITMAPFILEHEADER for all functions except
* readMultipleImages, which assumes the file pointer to be positioned on the
* start of a BITMAPARRAYHEADER. These functions will leave the file pointer
* on the byte after the image's color table.
*
* The coordinate spaces in the returned arrays will have an upper-left
* origin. As before, a non-zero return value indicates that something went
* wrong.
*
* Note that the BMP and mono-ICO functions will not return 1000 if the image
* is of type color-icon. This is because a color icon consists of a bitmap
* and a monochrome icon.
*
* return values:
*  0 - success
*  1000 - incorrect file type for the routine called
*  1001 - image data out of range or damaged file
*  1002 - good data, but the routine called can't handle it (yet)
*  1003 - out of memory allocating color table
113

* 1004 - out of memory allocating image
* 1005 - out of memory allocating image arrays
* 1006 - Illegal image type in a multi-image array
* other - I/O error of some kind

/*
* readSingleImageBMP will read a single BMP image and generate an array of
* RGB triples that contain the RGB values for every pixel. It will also return
* the dimensions of the image.
* /
int readSingleImageBMP(FILE *fp, RGB *argb, UINT32 *width, UINT32 *height)
{
    BITMAPFILEHEADER bfh;
    BITMAPHEADER bh;
    RGB *colorTable, *image;
    int rc, depth, inverted;
    long numColors, numPixels, endPos;

    /*
    * First, get the file header and sanity check it. The type field must be
    * TYPE_BMP or we're aborting.
    * /
    rc = readBitmapFileHeader(fp, &bfh);
    if (rc != 0)
        return rc;
    if (! (bfh.type == TYPE_BMP) &&
        (bfh.type != TYPE ICO_COLOR) &&
        (bfh.type != TYPE_PTR COLOR))
        return 1000;

    /*
    * Immediately following a file header is always the bitmap header. Read
    * it. Sanity check any values we might need. Specifically, less than
    * 32-bit depth, known compression scheme, known origin, and known color
    * encoding, and valid height/width. Note that negative height is OK,
    * that indicates an upper-left origin for a Windows bitmap.
    * /
    rc = readBitmapHeader(fp, &bh);
    if (rc != 0)
        return rc;
    depth = bh.numBitPlanes * bh.numBitsPerPixel;
    if ((depth > 32) ||
If the height is negative, then this is a Windows bitmap whose origin is the upper-left corner and not the lower-left. The inverted flag indicates a lower-left origin. Our code will be outputting an upper-left origin pixel array.

if (bh.height < 0)
{
    inverted = 0;
bh.height = -bh.height;
}
else
    inverted = 1;

Now, sanity check a few fields that are valid, but I don’t have code to deal with them yet. This includes: more than one bit plane, any compression scheme, and bit depths that are not 1, 4, 8, 16, or 24.

if ((bh.numBitPlanes > 1) ||
    ((bh.numBitsPerPlane != 1) &&
     (bh.numBitsPerPlane != 4) &&
     (bh.numBitsPerPlane != 8) &&
     (bh.numBitsPerPlane != 16) &&
     (bh.numBitsPerPlane != 24)) ||
    (bh.compressionScheme != COMPRESSION_NONE))
return 1002;

Allocate and read the color table. The file pointer has been positioned in the right place by the readBitmapHeader function. Note that images with 24-bits or more color depth have no color table. They are already RGB. When reading the color table, be sure to check for old/new format bitmaps.

if (depth < 24)
{
    numColors = 1 << depth;
colorTable = (RGB *)calloc(numColors, sizeof(RGB));
if (colorTable == NULL)
    return 1003;
if (bh.size <= 12)
    rc = readColorTable(fp, colorTable, numColors, 3);
else
    rc = readColorTable(fp, colorTable, numColors, 4);
if (rc != 0)
{
    free(colorTable);
    return rc;
}

/*
 * We're at the end of the color table. Preserve this position. We'll
 * need to leave the file pointer there before returning from this
 * function.
 */
endPos = ftell(fp);

/*
 * Allocate the array of pixels and fill it in.
 */
numPixels = bh.width * bh.height;
image = (RGB *)calloc(numPixels, sizeof(RGB));
if (image == NULL)
{
    free (colorTable);
    return 1004;
}

/*
 * Seek to the bits
 */
rc = fseek(fp, bh.offsetToBits, SEEK_SET);
if (rc != 0)
{
    free (colorTable);
    free (image);
    return rc;
}

/*
 * Read the bits. If code for decompressing bits should be written,
 * insert the call here.
 */
switch (bh.compressionScheme) {
    case COMPRESSION_NONE:
        rc = readBitsUncompressed(fp, image, bh.width, bh.height, depth,
                                  colorTable);
        break;
}

if (rc != 0) {
    free(image);
    return rc;
}

/*
 * If the origin is lower-left, flip the image upside down
 */
if (inverted)
    reflectYRGB(image, bh.width, bh.height);

/*
 * Return the output values. Set the file pointer to the byte after the
 * color table.
 */
*argb = image;
*width = bh.width;
*height = bh.height;
fseek(fp, endPos, SEEK_SET);

/*
 * Clean up and return. Note that we don't return the color table. This
 * is because we're returning an array of RGB values for the image - such
 * a table would be redundant.
 */
if (colorTable != NULL)
    free(colorTable);

return 0;
}

/*
 * Read in a monochrome OS/2 icon/pointer. Return two arrays of bytes
 * (interpreted as booleans) for the XOR and AND masks.
 */
int readSingleImageICOPTR(FILE *fp, char **xorMask, char **andMask,
                          UINT32 *width, UINT32 *height)
BITMAPFILEHEADER bfh;
BITMAPHEADER bh;
char *mask1, *mask2;
int rc;
long numPixels, endPos;

/*
 * Read and sanity check the header. Monochrom OS/2 icons are TYPE ICO.
 * Monochrome pointers are TYPE_PTR. Color ICO and PTR is also allowed,
 * because monochrome masks are part of those images.
 */
rc = readBitmapFileHeader(fp, &bfh);
if (rc != 0)
    return rc;
if ((bfh.type != TYPE ICO) &&
    (bfh.type != TYPE_PTR) &&
    (bfh.type != TYPE ICO_COLOR) &&
    (bfh.type != TYPE_PTR_COLOR))
    return 1000;

/*
 * Now read the bitmap data and sanity check it. Since this is a
 * monochrome icon, bit depth must be 1. Additionally, a known
 * compression scheme, known origin, known color encoding, and valid
 * height/width. Height can't be less than 0, as it can with color
 * images, since this code is only used on for OS/2-type images.
 */
rc = readBitmapHeader(fp, &bh);
if (rc != 0)
    return rc;
if ((bh.numBitPlanes != 1) ||
    (bh.numBitsPerPlane != 1) ||
    (bh.compressionScheme > COMPRESSION_LAST) ||
    (bh.origin > ORIGIN_LAST) ||
    (bh.colorEncoding > COLOR_ENCODING_LAST) ||
    (bh.width < 1) ||
    (bh.height < 1))
    return 1001;

/*
 * Sanity check some valid fields that I can't deal with yet.
 */
if (bh.compressionScheme != COMPRESSION_NONE)
    return 1002;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
/*
 * Skip over the color table, since this is a monochrome mask. Note that
 * the size is already known - 2 entries - which is 6 or 8 bytes.
 * this isn't, and we don't.
 */
if (bh.size <= 12)
    rc = fseek(fp, 6, SEEK_CUR);
else
    rc = fseek(fp, 8, SEEK_CUR);
if (rc != 0)
{
    return rc;
}

/*
 * Save this file position. we'll have to seek back to it after reading
 * in the image data.
 */
endPos = ftell(fp);

/*
 * The image is actually two images. The top half is an AND mask and the
 * bottom half is an XOR mask. Allocate the images.
 */
numPixels = bh.width * bh.height / 2;
mask1 = (char *)malloc(numPixels);
if (mask1 == NULL)
    return 1004;
mask2 = (char *)malloc(numPixels);
if (mask2 == NULL)
{
    free(mask1);
    return 1004;
}

/*
 * Seek to the bit data
 */
rc = fseek(fp, bfh.offsetToBits, SEEK_SET);
if (rc != 0)
{
    free(mask1);
    free(mask2);
    return rc;
}
/*
 * Read in the bits. Note: since these are really two images, two calls
 * to readMaskBitsUncompressed are made, and the height used is 1/2 the
 * height mentioned in the header.
 */
switch (bh.compressionScheme) {
    case COMPRESSION_NONE:
        rc = readMaskBitsUncompressed(fp, mask1, bh.width, bh.height/2);
        if (rc == 0)
            rc = readMaskBitsUncompressed(fp, mask2, bh.width, bh.height/2);
        break;
}
if (rc != 0)
{
    free(mask1);
    free(mask2);
    return rc;
}

/*
 * A mask will never have an upper-left origin, since Windows will never
 * produce one in a bitmap file.
 */
reflectYchar(mask1, bh.width, bh.height / 2);
reflectYchar(mask2, bh.width, bh.height / 2);

/*
 * Return everything we've read.
 */
xorMask = mask1;
andMask = mask2;
*width = bh.width;
*height = bh.height / 2;
fseek(fp, endPos, SEEK_SET);

    return 0;
}

/*
 * Read in a color OS/2 icon, return an array of RGBs for the colors.
 * and two arrays of bytes (interpreted as booleans) for the XOR and AND
 * masks.
 */
int readSingleImageColorlCOPTR(FILE *fp, RGB **argb, char **xorMask,
char **andMask, UINT32 *width, UINT32 *height)
{
    UINT32 width1, height1, width2, height2;
    int rc;

    /*
    * Color icons consist of a monochrome icon followed by a bitmap. This
    * makes reading them easy - first do a monochrome mask read, and then do
    * a color bitmap read. We should probably add some more descriptive
    * error codes here.
    *
    * First read the mask.
    */
    rc = readSingleImageICOPtr(fp, xorMask, andMask, &width2, &height2);
    if (rc != 0)
    {
        return rc;
    }

    /*
    * Next, read the color bitmap part
    */
    rc = readSingleImageBMP(fp, argb, &width1, &height1);
    if (rc != 0)
    {
        return rc;
    }

    /*
    * Now, just sanity check the image - the dimensions for the image should
    * match the dimensions of the masks.
    */
    if (((width1 != width2) || (height1 != height2))
        return 1001;

    *width = width1;
    *height = height1;
    return 0;
}

/*
* readMultipleImage runs down the list of images in a file and returns them
* all. ImageCount is the number of images in the file. The other returned
* values are parallel arrays. If an element in argb, axorMask, or aandMask
* is NULL, then that image has no such array. (Bitmaps have no xor or and
*
masks, monochrome icons have no color arrays.
*
Note that on errors other than 1000 and 1005, the arrays will contain good
data - the images that have been read properly will be in the arrays.
Images that have not yet been read will consist of NULL pointers in the
arrays.
*/

int readMultipleImage(FILE *fp, RGB ***argbs, char ***xorMasks,
char ***andMasks, UINT32 **widths, UINT32 **heights,
int *imageCount)
{
    int rc;
    long filePos;
    BITMAPARRAYHEADER bah;
    UINT16 imageType;
    int count;

    /*
    * First count the images. Preserve the file position for later. If some
    * structure in the list isn't an array header, return 1000.
    */
    filePos = ftell(fp);
    count = 0;
    do
    {
        rc = readBitmapArrayHeader(fp, &bah);
        if (rc != 0)
            return rc;
        if (bah.type != TYPE_ARRAY)
            return 1000;
        fseek(fp, bah.next, SEEK_SET);
        count++;
    } while (bah.next != 0);
    fseek(fp, filePos, SEEK_SET);

    /*
    * Allocate the arrays. Return 1005 on any failures
    */
    *argbs = (RGB **)calloc(count, sizeof(RGB *));
    if (*argbs == NULL)
        return 1005;
    *xorMasks = (char **)calloc(count, sizeof(char *));
    if (*xorMasks == NULL)
    {
        free(*argbs);
        return 1005;
    }
• andMasks = (char **)calloc(count, sizeof(char *));
if (*andMasks == NULL)
{
    free(*argbs);
    free(*xorMasks);
    return 1005;
}
• widths = (UINT32 *)calloc(count, sizeof(UINT32));
if (*widths == NULL)
{
    free(*argbs);
    free(*xorMasks);
    free(*andMasks);
    return 1005;
}
• heights = (UINT32 *)calloc(count, sizeof(UINT32));
if (*heights == NULL)
{
    free(*argbs);
    free(*xorMasks);
    free(*andMasks);
    free(*widths);
    return 1005;
}
• imageCount = count;
/*
 * Loop through the images again. This time, read each image
 */
count = 0;
do
{
    rc = readBitmapArrayHeader(fp, &bah);
    if (rc != 0)
        return rc;
    /*
     * Get the image type. Preserve the position, since we're reading
     * into the next structure.
    */
    filePos = ftell(fp);
    rc = readUINT16Little(fp, &imageType);
    if (rc != 0)
        return rc;
    rc = fseek(fp, filePos, SEEK_SET);
/*
 * Now that we know what kind of image we're about to read, read it.
 */
switch(imageType) {
    case TYPE_BMP:
        rc = readSingleImageBMP(fp, (*argbs)+count, (*widths)+count,
                                 (*heights)+count);
        break;
    case TYPE_ICO:
    case TYPE_PTR:
        rc = readSingleImageICOptr(fp, (*xorMasks)+count,
                                 (*andMasks)+count, (*widths)+count,
                                 (*heights)+count);
        break;
    case TYPE_ICO_COLOR:
    case TYPE_PTR_COLOR:
        rc = readSingleImageColorICOptr(fp, (*argbs)+count, 
                                         (*xorMasks)+count, 
                                         (*andMasks)+count, 
                                         (*widths)+count, 
                                         (*heights)+count);
        break;
    default:
        return 1006;
    }
    if (rc != 0)
        return rc;
    
    fseek(fp, bah.next, SEEK_SET);
    count++;
} while (bah.next != 0);

return 0;
/* readbmp.h
 * This is the header for readbmp.c - functions to read the bitmap file
 * structures. See readbmp.c for details.
 */

#ifndef __READBMP_H_INCLUDED__
#define __READBMP_H_INCLUDED__

/*
 * Mid-level functions
 */
int readBitmapFileHeader(FILE *fp, BITMAPFILEHEADER *bfh);
int readBitmapArrayHeader(FILE *fp, BITMAPARRAYHEADER *bah);
int readBitmapHeader(FILE *fp, BITMAPHEADER *bh);
int readRgb(FILE *fp, RGB *rgb, int numBytes);
int readColorTable(FILE *fp, RGB *rgb, int numEntries, int numBytesPerEntry);

int readBitsUncompressed(FILE *fp, RGB *image, int width, int height,
                           int depth, RGB* colorTable);
int readMaskBitsUncompressed(FILE *fp, char *image, int width, int height);

void reflectYRGB(RGB *image, int width, int height);
void reflectYchar(char *image, int width, int height);

/*
 * High level functions.
 */
int readSingleImageBMP(FILE *fp, RGB **argb, UINT32 *width, UINT32 *height);
int readSingleImageICOPTR(FILE *fp, char **xorMask, char **andMask,
                           UINT32 *width, UINT32 *height);
int readSingleImageColorICOPTR(FILE *fp, RGB **argb, char **xorMask,
                                char **andMask, UINT32 *width, UINT32 *height);
int readMultipleImage(FILE *fp, RGB ***argbs, char ***xorMasks,
                       char ***andMasks, UINT32 ***widths, UINT32 ***heights,
                       int *imageCount);

#endif
/* t1.c
 * Test program for reading bitmap files. It accepts an input file and an
 * output file on the command line. It will read and process the input file
 * and dump an ASCII representation of the contents to the output file. The
 * dump will consist of the color image and two masks. Missing parts will be
 * indicated as such (BMP files have no masks and nonochrome ICO/PTR files
 * have no color data. In the color image, the dump will be the row and
 * ending column number that contains a certain amount of consecutive "black"
 * pixels in a row. In the masks, the dump will be represented by ":".
 * symbols representing zeros and "@" symbols representing ones.
*/

/* To compile and run on the Unix systems: Type "make", then
 * "gcc t1.c readbmp.c endian.c"
 * To run: "a.out <infile.bmp> <outfile>"
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "bmptypes.h"
#include "readbmp.h"

int main (int argc, char *argv[]) {  
    FILE *fp;
    RGB **argbs;
    char **xorMasks, **andMasks;
    UINT32 *heights, *widths, row, col;
    UINT16 fileTyple;
    long filePos;
    int numImages, i;
    int rc;
    int m;
    int c;

    if (argc < 3) {  
        printf ("usage: test <infile> <outfile>\n");
        return 1;
    }

    fp = fopen(argv[1], "rb");
    if (fp == NULL) {
        
        
}
perror("Error opening source file");
  return 2;
}

/*
 * Read the first two bytes as little-endian to determine the file type.
 * Preserve the file position.
 */
filePos = ftell(fp);
rc = readUINT16little(fp, &fileType);
if (rc != 0)
{
  perror("Error getting file type");
  return 3;
}
fflush(fp, filePos, SEEK_SET);

/*
 * Read the images.
 */
switch (fileType) {
  case TYPE_ARRAY:
    /*
     * If this is an array of images, read them. All the arrays we need
     * will be allocated by the reader function.
     */
    rc = readMultipleImage(fp, &argbs, &xorMasks, &andMasks, &heights, 
                           &widths, &numImages);
    break;
  case TYPE_BMP:
  case TYPE_ICO:
  case TYPE_ICO_COLOR:
  case TYPE_PTR:
  case TYPE_PTR_COLOR:
    /*
     * If this is a single-image file, we've a little more work. In order
     * to make the output part of this test program easy to write, we're
     * going to allocate dummy arrays that represent what
     * readMultipleImage would have allocated. We'll read the data into
     * those arrays.
     */
  argbs = (RGB **)calloc(1, sizeof(RGB *));
  if (argbs == NULL)
  {
    rc = 1005;
    break;
  }

  /*
   * ...More code...
   */
}
xorMasks = (char **)calloc(1, sizeof(char *));
if (xorMasks == NULL)
{
    free(argbs);
    rc = 1005;
    break;
}

andMasks = (char **)calloc(1, sizeof(char *));
if (andMasks == NULL)
{
    free(argbs);
    free(xorMasks);
    rc = 1005;
    break;
}

heights = (UINT32 *)calloc(1, sizeof(UINT32));
if (heights == NULL)
{
    free(argbs);
    free(xorMasks);
    free(andMasks);
    rc = 1005;
    break;
}

widths = (UINT32 *)calloc(1, sizeof(UINT32));
if (widths == NULL)
{
    free(argbs);
    free(xorMasks);
    free(andMasks);
    free(heights);
    rc = 1005;
    break;
}

numImages = 1;

/*
 * Now that we have our arrays allocated, read the image into them.
 */
switch (fileType) {
    case TYPE_BMP:
        rc = readSingleImageBMP(fp, argbs, widths, heights);
        break;
    case TYPE_ICO:
    case TYPE_PTR:
rc = readSingleImageICOPTR(fp, xorMasks, andMasks, widths, heights);
break;
case TYPE_ICO_COLOR:
case TYPE_PTR_COLOR:
    rc = readSingleImageColorICOPTR(fp, argbs, xorMasks, andMasks, widths, heights);
    break;
break;
default:
    rc = 1000;
}

/*
 * At this point, everything's been read. Display status messages based on the return values.
 */
switch (rc) {
    case 1000:
    case 1006:
        printf("File is not a valid bitmap file\n");
        break;
    case 1001:
        printf("Illegal information in an image\n");
        break;
    case 1002:
        printf("Legal information that I can't handle yet in an image\n");
        break;
    case 1003:
    case 1004:
    case 1005:
        printf("Ran out of memory\n");
        break;
    case 0:
        printf("Got good data from file, writing results\n");
        break;
default:
    printf("Error reading file rc=%d\n", rc);
    perror("Error:");
    break;
}

/*
 * If the return value wasn't 0, something went wrong.
 */
if (rc != 0)
{
    if (rc != 1000 && rc != 1005)
    {
        for (i=0; i<numImages; i++)
        {
            if (argbs[i] != NULL)
                free(argbs[i]);
            if (andMasks[i] != NULL)
                free(andMasks[i]);
            if (xorMasks[i] != NULL)
                free(xorMasks[i]);
        }
        free(argbs);
        free(andMasks);
        free(xorMasks);
        free(widths);
        free(heights);
        return rc;
    }
}
fclose(f);
fp = fopen(argv[2], "wt");
if (fp == NULL)
{
    perror ("Error opening target file");
    return 3;
}
/*
  * Dump the images.
  */
for (i=0; i<numImages; i++)
{
    /*
    * Loop through all the images that were returned and print the file
    * dimensions in a file.
    */
    fprintf (fp, "Image dimensions: (%ld,%ld)\n", widths[i], heights[i]);
    if (argbs[i] != NULL)
    {
        /*
        * If the image has colors, search for possible staff lines
        */
        ...
* storing their row and column numbers in a file.
* The hexadecimal value is used to include some shades of grey
* as black while considering others as white. The m value is
* set to a minimum amount of consecutive blackish pixels needed
* to indicate the presence of a staff line.
*/
for (row = 0; row < heights[i]; row++)
{
    m = 0;
    for (col = 0; col < widths[i]; col++)
    {
        c = col;
        if (((argbs[i][row * widths[i] + col].red) < 0x88) &&
        ((argbs[i][row * widths[i] + col].green) < 0x88) &&
        ((argbs[i][row * widths[i] + col].blue) < 0x88))
            m = m + 1;
        else {
            if (m >= 100)
                fprintf (fp, "Row %ld Col %ld\n", row, c);
            m = 0;
        }
    }
}
else
{
    /*
    * If there image has no colors, say so. (monochrome ICO and PTR
    * files)
    */
    fprintf (fp, "No color image\n");
}

if (xorMasks[i] != NULL)
{
    /*
    * If the image has an xor mask, dump it. (ICO and PTR files)
    */
    fprintf (fp, "\nXOR mask\n");
    for (row = 0; row < heights[i]; row++)
    {
        for (col = 0; col < widths[i]; col++)
        {
            fprintf (fp, "\nXOR mask\n";
            xorMasks[i][row * widths[i] + col] ? '@' : '.');
        }
    }
}
131

fprintf (fp, "\n");

}

else
{
  /*
   * If the image has no xor mask, say so. (BMP files).
   */
  fprintf (fp, "No xor mask\n");
}

if (andMasks[i] != NULL)
{
  /*
   * If the image has an and mask, dump it. (ICO and PTR files)
   */
  fprintf (fp, "\nAND mask\n");
  for (row = 0; row < heights[i]; row++)
  {
    for (col = 0; col < widths[i]; col++)
    {
      fprintf (fp, "%c",
          andMasks[i][row * widths[i] + col] ? '@' : '.');
    }
    fprintf (fp, \n");
  }
  else
  {
    /*
     * If the image has no and mask, say so. (BMP files)
     */
    fprintf (fp, "No and mask\n");
  }

  if (i != numimages-1)
  fprintf (fp, "\n-------------------------------------------\n\n");
}
fclose(fp);

/*
 * Dumping is complete. Free all the arrays and quit
 */
for (i=0; i<numImages; i++)
{ }
    if (argbs[i] != NULL)
    free(argbs[i]);
    if (andMasks[i] != NULL)
    free(andMasks[i]);
    if (xorMasks[i] != NULL)
    free(xorMasks[i]);
}
free(argbs);
free(andMasks);
free(xorMasks);
free(widths);
free(heights);

return 0; 
}
/* makefile */
all:

endian.o : endian.c endian.h bmptypes.h makefile
gcc -Wall -c endian.c

readbmp.o : readbmp.c readbmp.h endian.h bmptypes.h makefile
gcc -Wall -c readbmp.c
t1.o : t1.c readbmp.h bmptypes.h endian.h makefile
gcc -Wall -c t1.c
testBMP.o : testBMP.c readbmp.h bmptypes.h makefile
gcc -Wall -c testBMP.c
test ICO.o : test ICO.c readbmp.h bmptypes.h makefile
gcc -Wall -c test ICO.c
testCICO.o : testCICO.c readbmp.h bmptypes.h makefile
gcc -Wall -c testCICO.c
testARAY.o : testARAY.c readbmp.h bmptypes.h makefile
gcc -Wall -c testARAY.c
### OUTPUT FILE

Output file showing the number of consecutive black pixels found in a row for Figure 5 with hexadecimal value 0x88 and \( m \geq 100 \).

**Image dimensions:** (695,900)

<table>
<thead>
<tr>
<th>Row</th>
<th>Pixels (R,G,B)</th>
<th>Hex values:</th>
<th>Black pixels in a row</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>125</td>
<td></td>
<td>225</td>
<td>0</td>
</tr>
<tr>
<td>126</td>
<td></td>
<td>491</td>
<td>0</td>
</tr>
<tr>
<td>127</td>
<td></td>
<td>387</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td>108</td>
<td>0</td>
</tr>
<tr>
<td>129</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>130</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(Rows 1-124: 0 black pixels also)
Row 131 pixels (R,G,B), hex values:
0 black pixels in a row

Row 132 pixels (R,G,B), hex values:
0 black pixels in a row

Row 133 pixels (R,G,B), hex values:
229 black pixels in a row
0 black pixels in a row

Row 134 pixels (R,G,B), hex values:
510 black pixels in a row
0 black pixels in a row

Row 135 pixels (R,G,B), hex values:
356 black pixels in a row
0 black pixels in a row

Row 136 pixels (R,G,B), hex values:
0 black pixels in a row

Row 137 pixels (R,G,B), hex values:
0 black pixels in a row

Row 138 pixels (R,G,B), hex values:
0 black pixels in a row

Row 139 pixels (R,G,B), hex values:
0 black pixels in a row

Row 140 pixels (R,G,B), hex values:
107 black pixels in a row
0 black pixels in a row

Row 141 pixels (R,G,B), hex values:
365 black pixels in a row
0 black pixels in a row

Row 142 pixels (R,G,B), hex values:
506 black pixels in a row
0 black pixels in a row

Row 143 pixels (R,G,B), hex values:
207 black pixels in a row
0 black pixels in a row
Row 144 pixels (R,G,B), hex values:
0 black pixels in a row

Row 145 pixels (R,G,B), hex values:
0 black pixels in a row

Row 146 pixels (R,G,B), hex values:
0 black pixels in a row

Row 147 pixels (R,G,B), hex values:
0 black pixels in a row

Row 148 pixels (R,G,B), hex values:
0 black pixels in a row

Row 149 pixels (R,G,B), hex values:
365 black pixels in a row
0 black pixels in a row

Row 150 pixels (R,G,B), hex values:
517 black pixels in a row
0 black pixels in a row

Row 151 pixels (R,G,B), hex values:
226 black pixels in a row
0 black pixels in a row

Row 152 pixels (R,G,B), hex values:
0 black pixels in a row

Row 153 pixels (R,G,B), hex values:
0 black pixels in a row

Row 154 pixels (R,G,B), hex values:
0 black pixels in a row

Row 155 pixels (R,G,B), hex values:
0 black pixels in a row

Row 156 pixels (R,G,B), hex values:
112 black pixels in a row
0 black pixels in a row

Row 157 pixels (R,G,B), hex values:
431 black pixels in a row
0 black pixels in a row
Row 158 pixels (R,G,B), hex values:
438 black pixels in a row
0 black pixels in a row

Row 159 pixels (R,G,B), hex values:
164 black pixels in a row
0 black pixels in a row

Row 160 pixels (R,G,B), hex values:
0 black pixels in a row

Row 328 pixels (R,G,B), hex values:
272 black pixels in a row
0 black pixels in a row

Row 329 pixels (R,G,B), hex values:
530 black pixels in a row
0 black pixels in a row

Row 330 pixels (R,G,B), hex values:
326 black pixels in a row
0 black pixels in a row

Row 331 pixels (R,G,B), hex values:
0 black pixels in a row

Row 332 pixels (R,G,B), hex values:
0 black pixels in a row

Row 333 pixels (R,G,B), hex values:
0 black pixels in a row

Row 334 pixels (R,G,B), hex values:
0 black pixels in a row

Row 335 pixels (R,G,B), hex values:
0 black pixels in a row

Row 336 pixels (R,G,B), hex values:
265 black pixels in a row
0 black pixels in a row

Row 337 pixels (R,G,B), hex values:
507 black pixels in a row
0 black pixels in a row

(Rows 161-327: 0 black pixels also)
Row 338 pixels (R,G,B), hex values:
298 black pixels in a row
0 black pixels in a row

Row 339 pixels (R,G,B), hex values:
0 black pixels in a row

Row 340 pixels (R,G,B), hex values:
0 black pixels in a row

Row 341 pixels (R,G,B), hex values:
0 black pixels in a row

Row 342 pixels (R,G,B), hex values:
0 black pixels in a row

Row 343 pixels (R,G,B), hex values:
0 black pixels in a row

Row 344 pixels (R,G,B), hex values:
385 black pixels in a row
0 black pixels in a row

Row 345 pixels (R,G,B), hex values:
546 black pixels in a row
0 black pixels in a row

Row 346 pixels (R,G,B), hex values:
216 black pixels in a row
0 black pixels in a row

Row 347 pixels (R,G,B), hex values:
0 black pixels in a row

Row 348 pixels (R,G,B), hex values:
0 black pixels in a row

Row 349 pixels (R,G,B), hex values:
0 black pixels in a row

Row 350 pixels (R,G,B), hex values:
0 black pixels in a row

Row 351 pixels (R,G,B), hex values:
0 black pixels in a row

Row 352 pixels (R,G,B), hex values:
332 black pixels in a row
0 black pixels in a row

Row 353 pixels (R,G,B), hex values:
527 black pixels in a row
0 black pixels in a row

Row 354 pixels (R,G,B), hex values:
187 black pixels in a row
0 black pixels in a row

Row 355 pixels (R,G,B), hex values:
0 black pixels in a row

Row 356 pixels (R,G,B), hex values:
0 black pixels in a row

Row 357 pixels (R,G,B), hex values:
0 black pixels in a row

Row 358 pixels (R,G,B), hex values:
0 black pixels in a row

Row 359 pixels (R,G,B), hex values:
158 black pixels in a row
0 black pixels in a row

Row 360 pixels (R,G,B), hex values:
489 black pixels in a row
0 black pixels in a row

Row 361 pixels (R,G,B), hex values:
389 black pixels in a row
0 black pixels in a row

Row 362 pixels (R,G,B), hex values:
122 black pixels in a row
0 black pixels in a row

Row 363 pixels (R,G,B), hex values:
0 black pixels in a row

Row 531 pixels (R,G,B), hex values:
215 black pixels in a row
0 black pixels in a row

(Rows 364-530: 0 black pixels also)
Row 532 pixels (R,G,B), hex values:
530 black pixels in a row
0 black pixels in a row

Row 533 pixels (R,G,B), hex values:
373 black pixels in a row
0 black pixels in a row

Row 534 pixels (R,G,B), hex values:
0 black pixels in a row

Row 535 pixels (R,G,B), hex values:
0 black pixels in a row

Row 536 pixels (R,G,B), hex values:
0 black pixels in a row

Row 537 pixels (R,G,B), hex values:
0 black pixels in a row

Row 538 pixels (R,G,B), hex values:
0 black pixels in a row

Row 539 pixels (R,G,B), hex values:
164 black pixels in a row
0 black pixels in a row

Row 540 pixels (R,G,B), hex values:
539 black pixels in a row
0 black pixels in a row

Row 541 pixels (R,G,B), hex values:
342 black pixels in a row
0 black pixels in a row

Row 542 pixels (R,G,B), hex values:
0 black pixels in a row

Row 543 pixels (R,G,B), hex values:
0 black pixels in a row

Row 544 pixels (R,G,B), hex values:
0 black pixels in a row

Row 545 pixels (R,G,B), hex values:
0 black pixels in a row
Row 546 pixels (R,G,B), hex values:
0 black pixels in a row

Row 547 pixels (R,G,B), hex values:
272 black pixels in a row
0 black pixels in a row

Row 548 pixels (R,G,B), hex values:
610 black pixels in a row
0 black pixels in a row

Row 549 pixels (R,G,B), hex values:
298 black pixels in a row
0 black pixels in a row

Row 550 pixels (R,G,B), hex values:
0 black pixels in a row

Row 551 pixels (R,G,B), hex values:
0 black pixels in a row

Row 552 pixels (R,G,B), hex values:
0 black pixels in a row

Row 553 pixels (R,G,B), hex values:
0 black pixels in a row

Row 554 pixels (R,G,B), hex values:
0 black pixels in a row

Row 555 pixels (R,G,B), hex values:
252 black pixels in a row
0 black pixels in a row

Row 556 pixels (R,G,B), hex values:
612 black pixels in a row
0 black pixels in a row

Row 557 pixels (R,G,B), hex values:
203 black pixels in a row
0 black pixels in a row

Row 558 pixels (R,G,B), hex values:
0 black pixels in a row
Row 559 pixels (R,G,B), hex values:
0 black pixels in a row

Row 560 pixels (R,G,B), hex values:
0 black pixels in a row

Row 561 pixels (R,G,B), hex values:
0 black pixels in a row

Row 562 pixels (R,G,B), hex values:
0 black pixels in a row

Row 563 pixels (R,G,B), hex values:
345 black pixels in a row
0 black pixels in a row

Row 564 pixels (R,G,B), hex values:
606 black pixels in a row
0 black pixels in a row

Row 565 pixels (R,G,B), hex values:
208 black pixels in a row
0 black pixels in a row

Row 566 pixels (R,G,B), hex values:
0 black pixels in a row

Row 735 pixels (R,G,B), hex values:
322 black pixels in a row
0 black pixels in a row

Row 736 pixels (R,G,B), hex values:
612 black pixels in a row
0 black pixels in a row

Row 737 pixels (R,G,B), hex values:
263 black pixels in a row
0 black pixels in a row

Row 738 pixels (R,G,B), hex values:
0 black pixels in a row

Row 739 pixels (R,G,B), hex values:
0 black pixels in a row

(Rows 567-734: 0 black pixels also)
Row 740 pixels (R,G,B), hex values:
0 black pixels in a row

Row 741 pixels (R,G,B), hex values:
0 black pixels in a row

Row 742 pixels (R,G,B), hex values:
0 black pixels in a row

Row 743 pixels (R,G,B), hex values:
397 black pixels in a row
0 black pixels in a row

Row 744 pixels (R,G,B), hex values:
577 black pixels in a row
0 black pixels in a row

Row 745 pixels (R,G,B), hex values:
139 black pixels in a row
0 black pixels in a row

Row 746 pixels (R,G,B), hex values:
0 black pixels in a row

Row 747 pixels (R,G,B), hex values:
0 black pixels in a row

Row 748 pixels (R,G,B), hex values:
0 black pixels in a row

Row 749 pixels (R,G,B), hex values:
0 black pixels in a row

Row 750 pixels (R,G,B), hex values:
0 black pixels in a row

Row 751 pixels (R,G,B), hex values:
350 black pixels in a row
0 black pixels in a row

Row 752 pixels (R,G,B), hex values:
613 black pixels in a row
0 black pixels in a row
Row 753 pixels (R,G,B), hex values:
192 black pixels in a row
0 black pixels in a row

Row 754 pixels (R,G,B), hex values:
0 black pixels in a row

Row 755 pixels (R,G,B), hex values:
0 black pixels in a row

Row 756 pixels (R,G,B), hex values:
0 black pixels in a row

Row 757 pixels (R,G,B), hex values:
0 black pixels in a row

Row 758 pixels (R,G,B), hex values:
0 black pixels in a row

Row 759 pixels (R,G,B), hex values:
521 black pixels in a row
0 black pixels in a row

Row 760 pixels (R,G,B), hex values:
446 black pixels in a row
0 black pixels in a row

Row 761 pixels (R,G,B), hex values:
0 black pixels in a row

Row 762 pixels (R,G,B), hex values:
0 black pixels in a row

Row 763 pixels (R,G,B), hex values:
0 black pixels in a row

Row 764 pixels (R,G,B), hex values:
0 black pixels in a row

Row 765 pixels (R,G,B), hex values:
0 black pixels in a row

Row 766 pixels (R,G,B), hex values:
0 black pixels in a row
Row 767 pixels (R,G,B), hex values:
466 black pixels in a row
0 black pixels in a row

Row 768 pixels (R,G,B), hex values:
546 black pixels in a row
0 black pixels in a row

Row 769 pixels (R,G,B), hex values:
115 black pixels in a row
0 black pixels in a row

Row 770 pixels (R,G,B), hex values:
0 black pixels in a row

(Rows 771-899: 0 black pixels also)
REFERENCES


Meyer, Darren. (1997). Graphics File Formats. WPI CS Department. meyer@cs.WPI.EDU


