D

Optical Reading of Typeset Music

by

Alan Ruttenberg

B.A., Brandeis University (1984)

Submitted to the Media Arts and Sciences Section,
Department of Architecture and Planning
in partial fulfillment of the requirements for the degree of

Master of Science in Visual Studies

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1991

Signature of Author .... [illegible]
Media Arts and Sciences Section,
Department of Architecture and Planning
January 18, 1991

Certified by .... [illegible]
Marvin Minsky
Toshiba Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by .... [illegible]
Stephen A. Benton
Chairman, Departmental Committee on Graduate Students

Optical Reading of Typeset Music

by

Alan Ruttenberg

Submitted to the Media Arts and Sciences Section,
Department of Architecture and Planning
on January 18, 1991, in partial fulfillment of the
requirements for the degree of
Master of Science in Visual Studies

Abstract

This thesis describes a program which translates a digitized image of a music score into a representation more amenable to computer manipulation of the musical content. I aim for a representation in which the information manifest on a sheet of music is easily accessible, information such as: the key signature, the pitches and durations of notes, and dynamic markings associated with parts of the score. The program is designed to read music printed by a single publisher, in a single musical font. Throughout the thesis I will comment on the issues and feasibility of extending this work to other fonts and publishers. This work is done for the most part using a Connection Machine, a data parallel computer.

Thesis Supervisor: Marvin Minsky
Title: Toshiba Professor of Media Arts and Sciences

Acknowledgments

In no particular order, and undoubtably with omissions.

This work was supported in part by Media Lab Funds, for which I am grateful.

Thanks to Sarbina Birner, Jim Davis, Peter Heller, Dave Rosenthal, Jeanne Speckman, for reading and commenting on drafts of this thesis.

The Spirit of 76 formed a supportive household during this work. Thanks Jeanne, Rick, Barry, Liz, Nancy, Mira, and kitty.

Thanks, Dave Rosenthal, for your advise and support in many ways.

Thanks to Sandy Pentland for ideas and space.

The people in the Music and Cognition Group at the Media Lab formed the core of my intellectual surroundings. Thank you.

Mike Travers, my officemate, taught me much about programming.

For various reasons: Barry Vercoe, Amy Bruckman, Marc Davis, Lori Rosenthal, Linda Peterson, Andrea Akerib, Doug Lenat, Lennart Johnsson, Ken Haase, Dave Sheppard, Stephen Benton, Henry Baird, Nicholas Negroponte, Gerald Sussman, Harold Abelson, Frank Perkins.

Betty Dexter is wonderful!

Thanks to the people at Thinking Machines, Symbolics, and Apple, for building cool machines, and writing inspiring software.

Thanks to Donald Knuth for TEX and his books, which serve as a continuing inspiration.

My parents have always been there when I needed them.

Contents

List of Figures

Chapter 1

Introduction

1.1 Goal

This thesis describes a program which translates a digitized image of a music score into a representation more amenable to computer manipulation of the musical content. I aim for a representation in which the information manifest on a sheet of music is easily accessible, information such as: the key signature, the pitches and durations of notes, and dynamic markings associated with parts of the score. This kind of encoded score could be used as input to other programs written to analyze musical structure or to translate scores into performances.

The program is designed to read music printed by a single publisher, in a single musical font. As will be seen, there are many difficulties even with this limited goal. I chose **The Complete Chamber Music of Schubert**, reprinted by Dover Press as my corpus, some 350 pages of mostly string quartet music. A substantial amount of other music is available from the publisher in the same format. I have not done any tests with other scores. Throughout the thesis I will comment on the issues and feasibility of extending this work to other fonts and publishers.

I aim for perfect accuracy in the representation of the written score, since it is my intention to use these representations in scholarly study of Schubert. Where there is a choice to be made between speed and accuracy, the choice in this work is always towards accuracy.

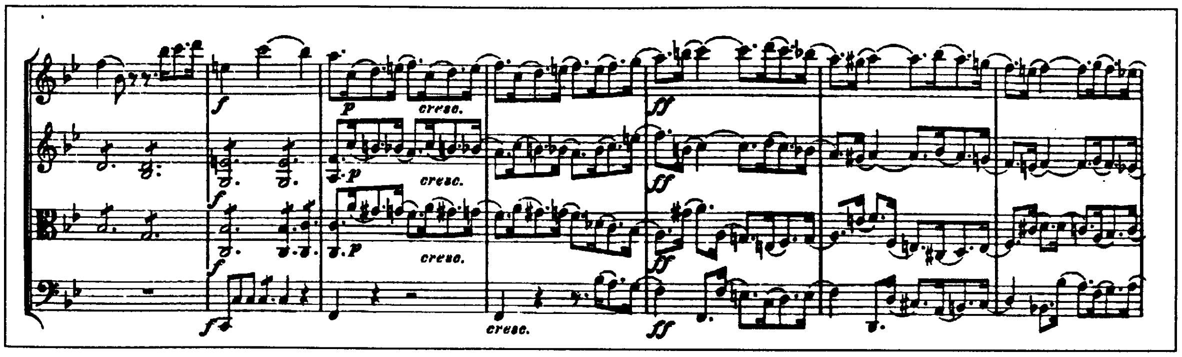


Figure 1-1: A typical system from the Schubert String Quartets published by Dover Press. This one is from page 97, the third system.

1.2 Why Optical Reading?

One can imagine transcribing the scores by hand, perhaps using some existing or special purpose program to record the information. This would be would be tedious, time consuming, and extremely prone to error. A typical page of string quartet music contains a couple of thousand marks, whose type and position are significant. Further, available representations for entering the information are limited [2], either being extremely cumbersome, or in being unable to capture the full range of expression of a printed score.

Attempts to formulate symbolic musical representations often lead to a compromise between conciseness, which demands that unimportant information be left out, and comprehensiveness - the ability of the language to express sufficient information that the score can be reconstructed. By reading the score optically, information can be kept which might justifiably be left out of some representations of the score but which might prove useful. For instance the exact positions of all the marks might be useful for studies of music typography. Thus, the question of which format to use to encode the score can be avoided by recording enough information so that the score can be visually reconstructed without loss. This is more information then one would want to enter by hand, but is easily available using the optical method, and is easy to translate to any desired format.

Another approach would be to write a program that interprets either an acoustic or gestural (e.g. midi) input and translates that to a representation of the score. The problem is that a given score has many interpretations. While the pitches often do not vary in these interpretations, the timing and dynamics do. Even supposing the performer made no mistakes, this variation in timing makes it difficult to determine a unique interpretation of the rhythmic content of the music.

I am skeptical that it is even possible in principle to return to the score from a performance. Often a phrase can be written in many notations which seem roughly equivalent. Compare four alternating eight notes and rests to four quarter notes with staccato markings; these might both legitimately be performed the same way. Examples like this suggest that there are many mappings from scores to performances and vice versa, and thus translating one unambiguously to another is impossible.

1.3 Comparison to Optical Character Recognition

It is useful to compare the score reading to text recognition. Superficial similarities between the two problems catch one's attention from the first. A comparison between the two can serve to suggest strategies for tackling the score reading problem, illuminate problems specific to score reading, and help better understand the capabilities and limitations of current character recognition technology.

To optically read text, first the lines of text are identified by searching for the long approximately horizontal space between lines. Each line is then processed in sequence. Within a line characters are handled one at a time. The reason that they can be handled individually is that they are not connected to each other, and so simple region labelling algorithms can segregate characters for individual handling[1]. The sort of errors character recognition algorithms make show that for the most part characters are individually recognized. Similarly shaped characters are often mistaken for each other, an l (letter L) for a 1 (number one) even in unlikely contexts, such as in the middle of paragraphs of narrative text. Even with the current state of the art, it is imperative to proofread the results of the text reading because errors are quite common.

In examining a page of music, one sees that the first steps towards recognition used by character recognition programs are not applicable. Isolating glyphs on a page of music is difficult because they overlap, and come in many sizes. Staff lines overlap with almost all other

[1]Ligatures are examples of connected characters, but there are only a few of them, and the ligature can be considered a single character by the recognition process.

features. Shape types range from note heads to beams to ties, each with a different characteristic scale. These features can not be isolated as easily as in the case of character recognition.

The final representation of music is also more complicated than in optical character recognition in which the relevant information is the sequence of characters and the places where the paragraphs break. Though font information would be desirable, commercial optical character readers only supply limited font information - not nearly enough to adequately parse a dictionary definition, for instance. In terms of data structures, this information is typically represented as a ordered sequence of characters with font and size properties attached.

In some forms of media it is an oversimplification to treat text as a stream of propertied characters is much too simplified. On the front page of a newspaper the relative size and position of text carries information which supplements the meaning of the words, information about importance, continuation, and relationship to surrounding information. I am not familiar with research which does text recognition at this level. Certainly what comes under the label of optical character recognition is not this, which might be more appropriately called visual language recognition (after the Media Lab's Visual Language Workshop).

The visual language of music is much more complicated than the common representation of textual information. A beam groups the set of notes *attached* to it by stems. A key signature is recognized as a *cluster* of accidentals not *beside* note heads. An accidental modifies the note to its *right*, a dot the note to its *left*. A tie modifies the performance of all the notes *between* the two notes it *points* to.

The point of view taken in character recognition is inadequate for reading music. In order to attack the problem of music recognition we need to think differently about the sort of recognition primitives to employ and about the architecture of a recognizer which can handle the diversity of visual relations which a musical score exhibits.

1.4 Considerations in Deciding what a Feature Is

Shape can be dealt with at a number of levels of detail. In talking about a character glyph, for instance, one shape-like description might be in terms of corners, edges, lines, and curves. Another might consider the shape of a particular character in a certain font to be a primitive shape. Another, such as Donald Knuth's Metafont descriptions, could deal with descriptions of the shared and unique parts of characters, and how characters are composed from them.

The choice of what to consider a feature is a design decision which involves tradeoffs between the difficulty of the programming task, the processing time, and the reliability of the methods which can be employed to locate that feature. The more complicated the primitive features are, the harder it is to write a specific program to find them. Implicit in complication is variety. There are more types of complicated shapes than there are simple types of shapes. So the more complicated that shape is, the more processing is involved because more methods have to be applied in order to find all the different shapes.

With simpler shape primitives there are also problems. The simpler a shape becomes, the more ubiquitious it is. If the shape primitive were a small curved edge, such as would be part of a note head, that same primitive would be found in many places; as part of a clef, as part of words on the page, and as part of poorly printed beams. Because the feature is found more often, it distinguishes less well between the different types of objects found. So while there is a the potential for a speedup in the feature finding process by virtue of looking for fewer features, there is the additional complexity of using those features effectively.

1.5 *Feature* Defined

Features are objects such as note heads, staff lines, beams, treble clefs, dynamic markings, and the like. They are the shapes which this program finds as a whole, rather than reconstructing out of smaller parts. A beamed group is an example of an object which is assembled from these primitives, a composite object consisting of a beam feature and several note head features. Features are partitioned into several classes: Fixed-sized, beams, ties and slurs, stems and lines. Each feature class uses a different locating algorithm.

The level of feature type which I chose is not intrinsically primitive. Just as composite structures are built from these features, these features can themselves be considered composed of smaller visual components. Some marks which might be considered single features, such as an eighth notes, are found in two pieces: the note head and the flag. But a note head is found as a single piece, rather than as the coincidence of several curved boundaries.

The principle advantages of these design choices are coverage and accuracy. All features

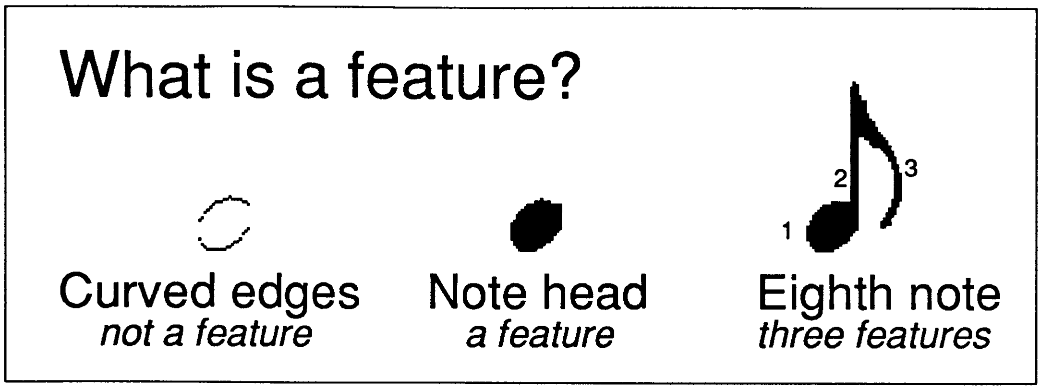


Figure 1-2: The first image, representing the curved edges of a note head could have been considered a feature, but wasn't in this work. The last is a composite feature, made up of the stem, the flag, and the note head.

except grace notes (which are rare in this music) are found by one or the other of the methods [2]. All the feature finding methods are adequately accurate. Some are virtually error free, whereas others have low enough error rates that the later processing stages can easily detect and correct the errors.

1.6 Use of Multiple Methods for Locating Features

Features come in many forms. Rather than try to create a feature model which could find all kinds of features I wrote a variety of different methods, each able to find a certain class of features. The advantage of a single feature model is uniformity of representation and processing, and perhaps elegence, but as usual there is a tradeoff. Since features come is a large range of shapes, any single model would have to be quite general. Such a model is difficult to design, and it would be difficult to design algorithms which could locate any instance of such a model.

Instead, five distinct feature finding algorithms are used, each designed to find a different class of features. These five classes are illustrated in Figure 1-3.

Fixed sized features are are always the same shape and size (within a single publication). Examples are note heads and accidentals. These are the closest to the features of traditional optical character recognition. Since they are always the same shape and size they can be found

[2]Grace notes would be found using the current scheme if the scanning resolution were greater.

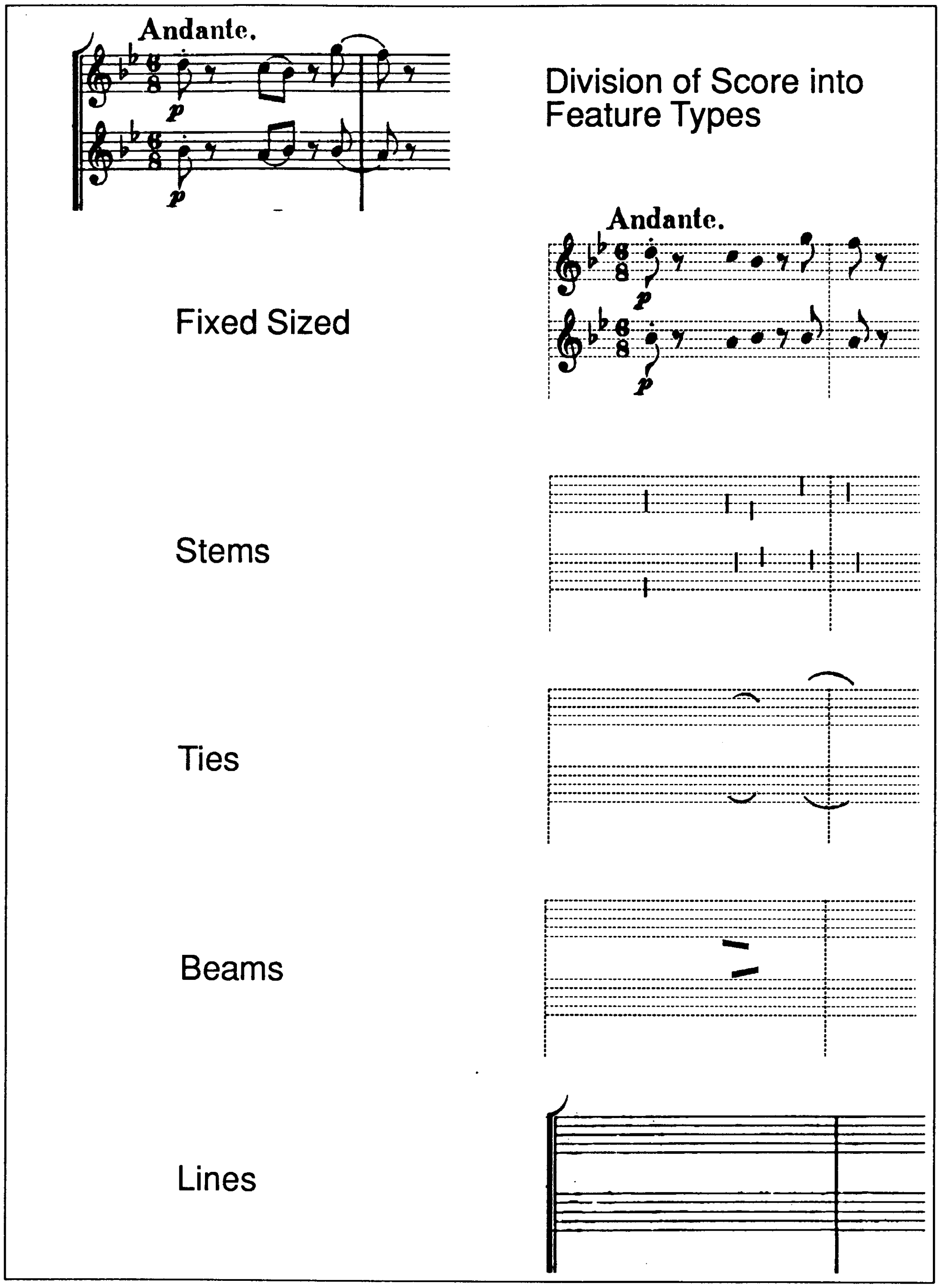


Figure 1-3: Feature are divided into five classes. The greyed staff lines are for reference

by matching the image to a template of the feature in question.

A second class of feature are beams. The length and number of ligatures of a beam varies, as does the angle of the beam, but the thickness of a single ligature hardly varies at all. Beams are found by a method which first isolates them by exploiting the fact that beams are relatively thicker than other features. Once the beams are isolated their shapes can be individually characterized.

To locate staff and measure lines I use a filter to remove other than these "almost" horizontal and vertical lines. The filter is not perfect, and leaves marks which might mistakenly be considered staff or measure lines. A second stage of processing divides the image into pieces to find line segments. These segments are then reconstructed into the desired lines.

Because stems are only a few pixels wide, there is much noise in the marks which represent them. But since stems are usually redundant information, a simple, error prone method can be used to find them. The particular method I use actually works best when the stem information is most needed.

Ties and slurs comprise the last feature class. They have various slopes and length, though have a similar line thickness. The method for finding them focuses on the characteristic of ties and slurs being "thing which look like lines", and uses a variety of filters to isolate and recontruct them.

1.7 Handling of Inaccuracy in Feature Location

The various methods for locating features are not 100 per cent accurate. There are two ways to compensate for this. Either more time could be spent making the feature locaters more accurate or auxillary methods could be used to compensate for their inaccuracies. I chose the latter method because I don't believe that any simple feature recognition algorithm can be perfect, and because I was interested in the general problem of dealing with systems with imperfect information.

In this work, inaccuracies are dealt with in three ways: a) Given that errors in feature location will happen, write algorithms which err toward finding extra features rather than missing some, b) Decide when features contain redundant information since accuracy is less important in these cases, and c) Use constraints on the relationships between objects to rule out impossible combinations.

In locating fixed sized features, the acceptability threshold is a parameter which can be varied to more or less easily accept a feature as matching the template. The template matching process for locating fixed sized features uses a threshold which functions as an acceptibility parameter. When the parameter is lower a pattern more easily matches the template, when higher it is more difficult. These threshold were individually tuned so as to more easily accept features, and thus generate spurious locations, rather than missing a valid feature. It is easier to rule out something as impossible or unlikely than to postulate its existence when it is not in fact present.

Finding stems was difficult because of quantization noise and proximity to other features. But stem information is redundant except in the case where a note lies between two staffs, in which case the stem determines which staff the note belongs to. The method for finding stems works poorly in the presence of other features. But the cases where stem information is useful are precisely when there are not other features nearby. The information derived from the stem locater is used on demand, when other information is insufficient to determine note position. By using the information on demand, the mass of incorrect stem locations do not confuse the score reader.

Once features are found, they are assembled into groups in order to reconstruct the elements of the score. Information about which features are expected to be near others is encoded in the methods which link features together into groups which make sense. Features which are not part of sensible groupings are ruled out, and additional attention is paid to groups which don't make sense, or ambiguous grouping relationships.

1.8 Assembling Composite Musical Objects

The raw positions of the features must be processed further before they can be interpreted as music. Time and pitch need to be computed, the time and key signatures need to be assembled from their components, dynamics and other expressive marks have to be attached to the notes and phrases that they modify. Mistaken features must be discovered and removed, and the score checked for consistency. Where inconsistency still exists the program needs to notify the user. Each of these tasks has in common that their execution depends on the ability to express and manipulate information about the score which is symbolic, rather than exclusively numerical.

I set up an object oriented framework in which the movement between numerical and symbolic representations is relatively uniform. Symbolic relationships are represented as objects which have pointers to each of the features which participate in the relationship. In order to flexibly handle the possibility of mistaken features, relationships may be noted as being ambiguous or missing and thus be accessible as such for further processing. [3] These same objects have methods attached to them which know, given one of the participants, how to find the other potentially related objects and verify whether the relationship in fact holds. Methods exist which can notify other objects that a relationship unambiguously holds and therefore extend to larger scale information which is initially computed locally.

Features are also represented as objects, containing information about their position, geometry, and their relationships to other features. Methods[4] on feature classes can be written as necessary in order to enable them to supply information necessary to the computations involved in finding and checking relationships. For example, each fixed sized feature can include in their representation information about their position, bounds, as well as a number of named points on their shape which are necessary for verifying certain types of relationships. An accidental has, for instance, a point called a *pitch-center*, the position of which can be queried by the assembler/relator. This position is necessary to establish the relationship between a note head and its accidental, namely that the *pitch-centers* of the head and accidental lie at roughly the same vertical height.

The initial links between objects are based on the overlap of rectangles which are computed as a function of the features in question. For instance a note head computes a rectangle the same size as its own bounds but displaced to its left as an area to search for an accidental. Features are stored in such a manner as to make efficient the operation of querying their overlap with arbitrary rectangles. Since there are roughly two thousand features per page the efficiency of this operation is of some concern.

Because of the uniform framework it is relatively easy to add new relationships and further

[3]Relations may be ambiguous because of the presense of spuriously found features, and because not all interpretations can be made without greater knowledge of music. For instance, two flags might be near enough a note head to be both, at first glance, associated with that note.

[4]I use the term method here in the object oriented sense of a function which is specialized according the type of object it operates on.

methods for handling errors in the feature location. Once all relations are established and verified a subset of the network of relations becomes, with minimal translation, a complete and adequate representation of the page of music.

1.9 Connection to Society of Mind

At the beginning of this work I was uncertain how to relate structural information about music scores to the image processing driven representations of shape and position. Initial steps in making this connection were confusing and complicated and too intertwined with the mechanism used to find the features. I consider it a success of this work that I was able to make a reasonably modular boundary between the geometric and the predicate/relational levels of the recognition process.

In Society of Mind, it is argued that the mind is composed of a large number of agencies, each with a simple function, with limited and specific means of communicating with other agencies. But many questions remain about whether this is true. Where are the boundaries placed? What information needs to be shared short of communicating *everything*, in order to sustain the functionality of a brain?

Perhaps this work gives a glimmer of what part of such a system might look like. All the feature finding algorithms work on fairly local sections of the image, offering one opportunity for parallelism. A second agency could implement the mechanism for computing overlap quickly. It need only be concerned with communicating information about simple spatial relations. Finally, by being able to move to a symbolic description (still retaining the old when necessary), yet another sort of mental activity can be engaged. The manner of each relation type encapsulating information for finding, verifying, and relaying information about its activities in a controlled way suggests one architecture that a society might have.

1.10 Influence of Parallel Programming

I was fortunate enough to have the use of a Connection Machine in doing this research. A Connection Machine is a data parallel computer with up to 64k processors. Our connection machine was equipped with 16k processors, each with 8k bytes of memory. Peripherals included a 5 Gbyte fast access disk and a 1280 by 1024 by 24 bit framebuffer.

Because the machine is very fast, it was possible to implement some of the feature locaters in what would be an inefficient way on a serial machine. This is most true for the fixed-sized feature location method, where the method of convolution was chosen because of its conceptual ease and high accuracy over other algorithms which have the potential of using less computation. The ability to trade off development time against speed made it possible for me to write enough feature locaters to challenge the assembly process and ensure that its architecture could truly deal with the complexity of a real score.

From a computational complexity point of view, a machine with N processors can run N times faster than a machine with a single processor. This is because the single processor could sequentially execute each of the computations that the larger set of processors execute. But this simple result belies the difficulty and challenge of using a parallel machine. The theoretical speedup of a factor of the number of processors is not easily achieved because it requires recasting of serial algorithms so that the computation can be shared by many processors. Part of the work done in this thesis was exploring algorithms and designing representations which could exploit the machine.

The beam locating code illustrates an example of this process. My method for locating beams employs the equivalent of a labelling of connected regions. The solution, adapted from [cite lim], involves a constant time step (with a processor per pixel) which forms a circular linked list of the the boundary points of the regions. All the information about the regions can be derived from the boundaries of the regions, and since there are typically many fewer boundary than points which cover a plane [5] fewer processors can be used to do the remaining computation on the boundary points. This smaller set of processors communicates label tokens until each point has had the chance to receive a token from any of the other processors, a process which taken a number of steps logarithmic in the size of the list. At this point each point is labelled with a token uniqe to the boundary it is part of. Compare this to the typical raster scanning methods for accomplishing the same thing serially[6], and you get some idea of the excitement of developing algorithms for a parallel machine.

[5]A fractal is an exception
[6]for instance in [5]

In this work, parallelization occured over pixels or features. Algorithms were then best expressed from the point of view of a single feature or pixel. The point of view can be seen most clearly in a cliche which I used several times. To locate a feature I use a two step process: First taking into account properties of the neighborhood of the feature, I would write a filter which disconnects it from or transforms its surroundings. By writing the filter as a function of the surroundings of a pixel it can execute quickly - all surroundings get processed in parallel. Then I would focus identification on the resultant connected fragments. It is reasonable to consider all the fragments which result in the application of the filter by allocating a processor for the task of examining each one.

In this view feature locating can be viewed as an iterative process: Instead of finding a method which locates the feature in question in a single step, repeatedly focus attention on potential features, having them eliminate themselves when some inspection of the local surroundings makes their existence seem unlikely.

1.11 Choice of Serial vs Parallel Algorithms

A symbolics lisp machine served as a front end to the Connection Machine, and could be used to do serial computation. Given the additional resource, choices needed to be made about which algorithms to implement on the serial front end as opposed to the Connection Machine.

All the algorithms written for this thesis were parallel, except for a portion of the staff finding algorithms, a small deglitching of the beam algorithm and all the score assembly work. My default was to start off with a parallel algorithm. I escaped to a serial algorithm when either the number of objects being dealt with decreased to tens, or when writing a parallel program became difficult.

The former reason accounts for the serial algorithms in the beam and staff line case. In both these cases the computation starts with a processor per pixel, and reduces the image to a small set of complete features. The complexity of writing the parallel program was the factor in the assembly process. In this case there are different sorts of interactions between different sorts of objects, and a data parallel solution did not immediately present itself. The assembling takes a small portion of the total computation time, so it wasn't worth expending more effort to develop a parallel solution, though such an implementation might be interesting in its own right.

1.12 Extending Reader Coverage

The scope of the reader was narrowly restricted to the string quartets of a single publisher in order to make the problem more manageable. However algorithms were designed with the idea of extendability in mind. In this section I will comment the possibility of modifying the reader to be able to read other scores. The reader should note that no experiments were performed to verify the accuracy of these ideas.

How can we asses the difficulty of adapting this program? A program can be viewed as a large function, with the input parameters being whatever global variables and databases are set up before running it. The code can be viewed as a parameter as well, but one which is changed with greater difficulty. The difficulty of changing the parameters so that the program works for a different task is a measure of how difficult it is to perform that task.

I'll characterize the program as three distinct modules. A template driven regular feature finder makes the assumption that instances of the same feature are very similar. It takes as input the set of templates and some variables which define thresholds of acceptance of match.

There are a series of hand coded methods for finding different features which don't fall in the first category. These routines deal with features which are fixed in some ways and vary others. They assume the basic topology of the object and recover the particular parameters of its shape. They take as input some variables related to the scale of the image which define the range of variation permissable. Methods for locating stems, beams, ties, and staff and measure lines fall into this category.

The third module deals with assembling the features into larger groups. This module has no parameters, since the methods for assembling are expressed in code, and are written in terms of relative sizes of features. The code is object oriented with methods for each type of graphical relationships and methods for handling each typical error mode.

I considered extending the reader to handle three successively more general tasks: (1) reading music scores of other publishers and styles, (2) reading arbitrary typeset scores without foreknowledge of the publisher and (3) reading handwritten scores.

Very little work needs to be done to adapt the program to a different publisher of typeset music. The template need to be changed to reflect the different typography standards, and some scale parameters need to be changed. Small conventions, for instance in the placement of dynamics might present the need to rewrite certain of the assembly modules.

To adapt the program to read arbitrary typeset music work of a different sort is needed. The strategy I have considered is to write a program which determines the necessary scale parameters automatically, perhaps by search through the parameter space that defines the operation of the staff line finder. This would give a strong indication of the basic sizes of features, and could be used as a narrower guide for search of other parameters. Since the system has the ability to check for consistency, there is potential for generate and test strategy to determine the parameters once they have been narrowly restricted.

As for the possibility of extending this reader to read handwritten scores, this is fairly remote. Almost all of the feature finding routines would need to be abandoned except, perhaps for the staff locater. Still, the assembly module could still be used with other feature finders substituted, so there is the possibility of some reuse of the ideas presented here.

1.13 Speculation: Other Applications of the Technology

I believe that some of the ideas which were developed in this thesis could be successfully adapted to other recognition tasks. Some initial work has been done is evaluating the possibility of using these methods to read a dictionary, a form of text where font and position information bears information, not only the raw text.

The ideas presented under the name of assembly might be relevant to the task of interpreting diagrams such as blueprints, line drawings, and sketched information input to a computer via a pen and tablet.

I think that in general the questions which are posed here concerning the recognition of visual language, merit further thought, and that the particular solutions used here might shed light on the general question of the nature of people's ability to perceive visually.

1.14 How to Read this Thesis

Chapter 2 contains an overview of the connection machine concepts which I use in this thesis. This is necessarily brief, and I urge the reader to consult some of the literature provided by thinking machines. The rest of the chapters are divided into two sections, a prose description of the algorithm and problems, and the code which implements it. The code is provided with few comments, but serves as a definitive definition of the implementation. Most readers will wish to only browse the occasional section.

Chapter 2

Connection Machine Concepts

This chapter introduces some concepts which are necessary in order to understand the implementation I describe in my thesis.

2.1 Programming Language

All code for this thesis was written in Symbolics Common Lisp[12], and ***lisp**[13] both extensions of commonlisp[11]. I assume that the reader is familiar with commonlisp. In addition, I used the object oriented programming system supplied on the lisp machine, called flavors. I wrote an extension of flavors to handle parallel variables which is used in the beam finding code.

2.2 Image

The *scanned images* I refer to are 1 bit images obtained by digitizing a sheet of music using an Apple scanner at 300 dots per inch (*dpi*). They are best imagined as a large array of numbers, in this case 2544 across by 3300 down for the $8\frac{1}{2}$ by 11 inch sheets of music. Imagine a fine grid which is layed over the original image. In each square of the grid, if the average density (the higher the density, the darker the image) of the square is more than a certain threshold value, the the corresponding array point is set to one, if it is less a zero. The amount of memory occupied by such an image is $\frac{2544 \cdot 3300 \text{ bits}}{8 \text{ bits per byte}} = 1,049,400$ bytes, or roughly one megabyte.

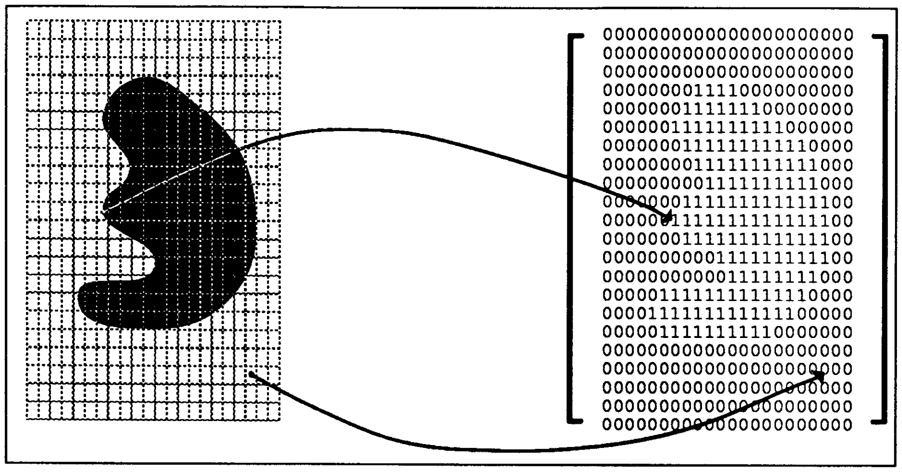


Figure 2-1: Digitization of an image: An imaginary grid is placed over the image. Each square of the grid gets translated into a 0 or a 1 depending on whether it is more white or black.

2.3 SIMD

The Connection Machine is a parallel single instruction multiple data (*SIMD*), computer. The CM I that used could be configured to use either 16k or 8k processors, each of which has 64k bits of memory.

I'll describe the operation of the Connection Machine by starting with analogy with a standard serial machine and adding features until a model of the CM sufficent for understanding of this work is built up.

Start first by imagining a unstructured collection of processors. Each processor is to be imagined as having variables and structures. Each processor has the same set of variables, but each can have different values for those variables.

Instead of each processor having a program memory, however, there is only one program memory(see Figure 2-2). On a serial machine one would write (setq a (+ b c)) to mean that the variable a should be set to the sum of b and c. For the connection machine one writes (*set a (+!! b c)) to mean that in each processor, the variable a should be set to the sum of b and c. +!! is the parallel equivalent of +, in the sense that it implements the operation of addition in each processor. Many commonlisp function have a parallel equivalent in this sense. Variables for which there is a copy in each processor are referred to as *pvars* (abbreviated from *parallel variables*).

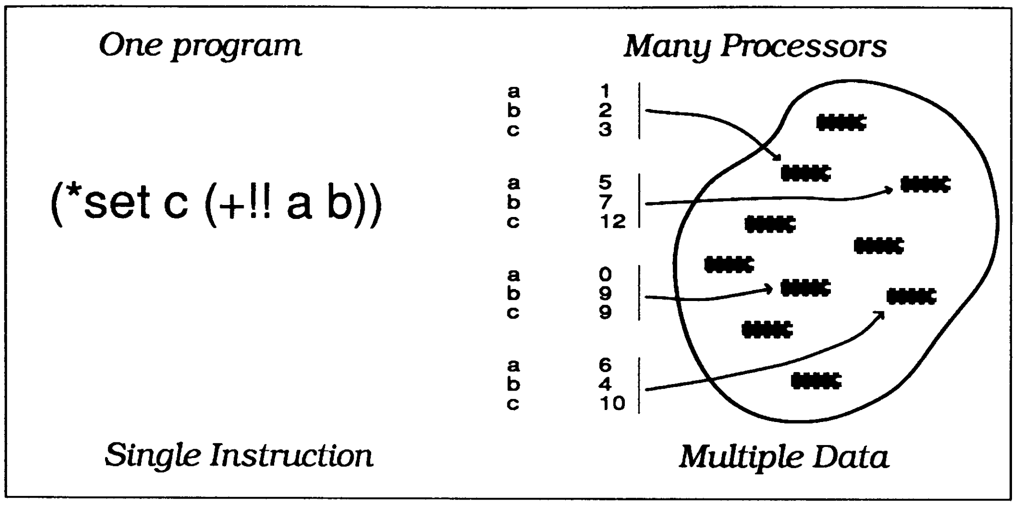


Figure 2-2: The connection machine has one copy of the program (= single instruction) and many processors, each of which has different values assigned to its variables (= multiple data). I use an icon of an integrated circuit to represent a processor.

2.4 Subselection

A subset of processors can be left out in doing some operations; this is referred to as *subselection*, the set of processors which does execute the instruction is called the the *selected set*. If select? is a boolean pvar, then the expression

```
(*when select? (*set a (+!! b c)))
```

causes a to be set to the sum of b and c only when the variable select? is nonnull. See Figure 2-3.

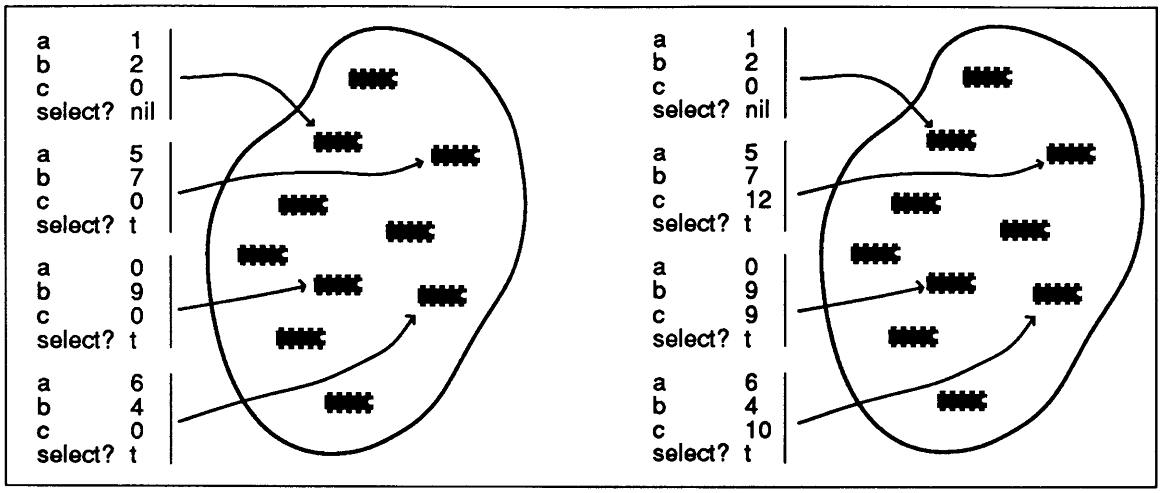


Figure 2-3: Visualization of Selected set. (*when select? (*set c (+!! a b))) gets executed only in processors where select? is t.

2.5 Global Operations

Operations which accumulate some value as a function of all the processors are called *global* operations. The expression (*sum a) would return a single value which is the sum of the variable a in each processor.

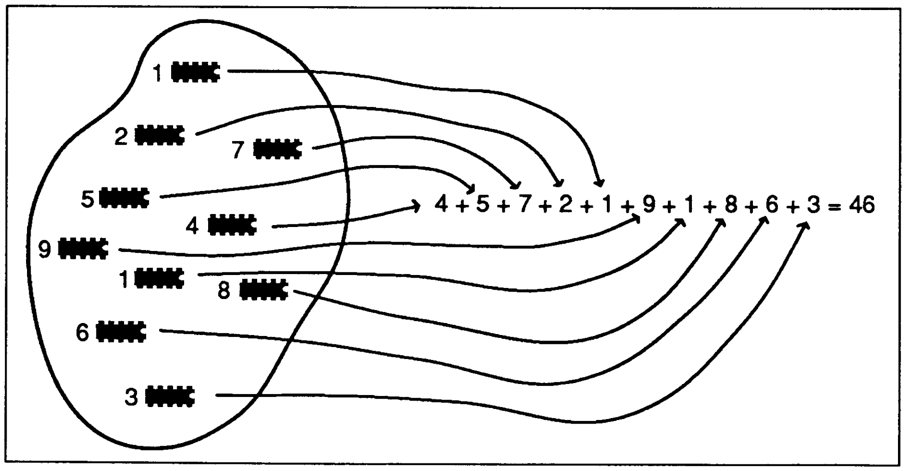


Figure 2-4: Visualization of a global operation. *sum adds up the value of some variable in all processors.

2.6 Virtual Processors Sets

The Connection Machine software supports the ability to present the illusion that the pool of processors has more than the physical number of processors that the machine has, and the ability to have multiple differently sized pools of processors. A pool of processors is called a *vpset*(for **Virtual Processor Set**). The form

```
(setq vpset1 (create-vp-set (list (expt 2 20))))
```

creates such a pool of processors with approximately one million processors. At a given moment all operations are done on a single vpset, and forms are supplied that switch the current vpset. The form

```
(*with-vp-set vpset1 (*set a (+!! b c))
```

would switch to the earlier created vpset and within it, add b to c. A variable of a given name always belongs to a particular vpset.

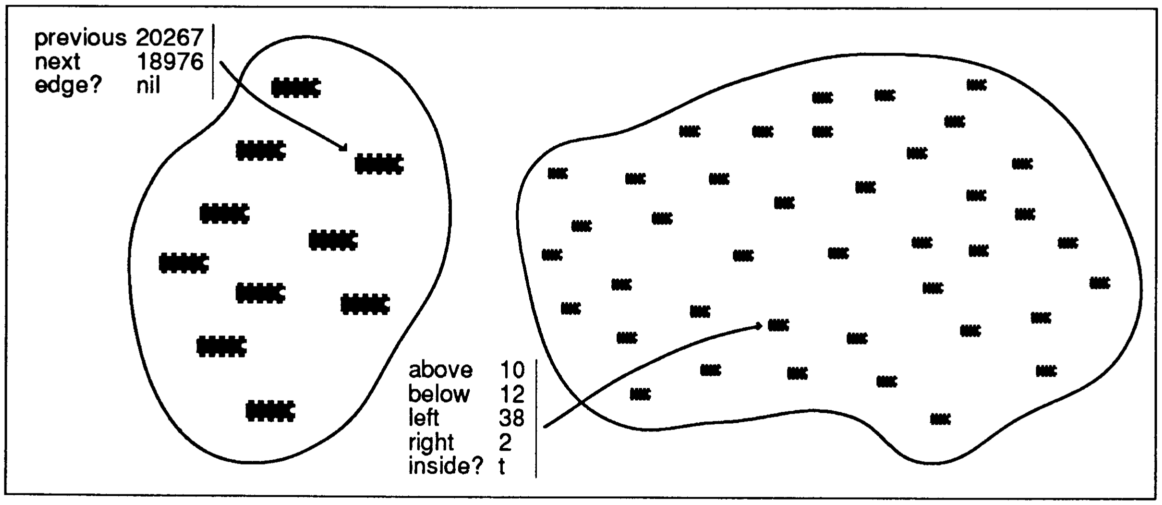


Figure 2-5: Visualization of Vp Sets. A vpset is a set of processors each of which may contain some variables. A given variable only resides in the processors of a single vpset.

2.7 Sending

Each processor has a unique numerical identifier, which can be used to address that processor. These identifiers number from 0 to one less than the number of processors in the vpset. The form (self-address!!) returns this unique integer in each processor.

Processors can *communicate* with each other by *sending* information between themselves. In order to send information, the destination processor must be specified. All the sending processors must be in a single vpset and all the receiving processors must be in a single vpset, possibly the same vpset(see Figure 2-6. The syntax for this operation is slightly confusing, but useful to understand. The form

(*with-vp-set this-vpset (*pset :overwrite a processor-address b :vpset other-vpset))

causes each processor in this-vpset to send the value of variable b to the processor in other-vpset with selfaddress processor-address (a variable in this-vpset). The variable c in that processor gets set to the sent value.

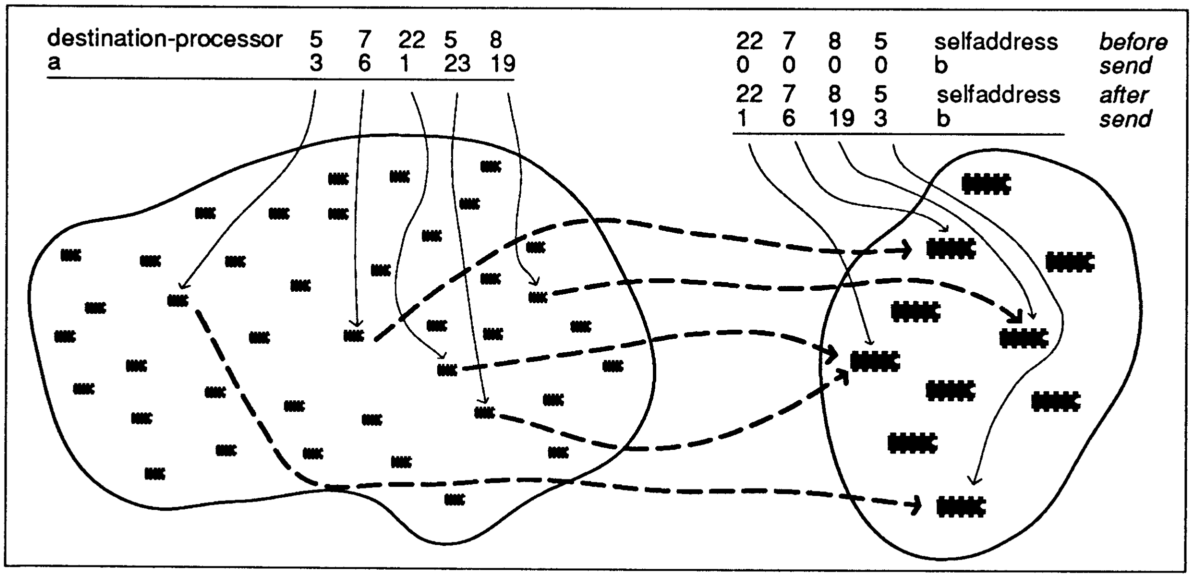


Figure 2-6: How to visualize a send operation: A variable in processors from one vpset is copied into another variable in another vpset. Note that processor 22 gets two messages, one of which is ignored.

Because processors doing the sending can independently specify a destination address, it is possible for more than one processor to send information to a single processor. The Connection Machine software allows one to specify what happens in this case, which is called *combination.* In the above *pset form, the :overwrite symbol specifies that if more than one message is sent to the same processor, earlier ones get overwritten by later ones. Other possibilities are that numbers get added, or that bitstrings get combined using logical operations.

2.8 Geometry

Thus far a vpset has been visualized as an unstructured set of processors. Quite commonly one also defines a *geometry* on this set. A geomtry introduces a virtual geometric structure on the processors by defining which processors are *close* to which other processors. Access to *close* processors is more efficient and the CM software supplies extra primitives to do these communications. by **dimension**.

The Connection Machine software supplies a limited number of built in geometries. One set of them are *grid* geometries, optimized so that communication between nearby points on the lattice is fast. A vpset created thusly

```
(setq two-d-grid (create-vpset (list 1024 1024)))
```

would create a two dimensional grid with that property. In order to take advantage of the fast *nearest neighbor* communication, instead of using ***pset**, one would use a form such as (*set a (news!! a -1 2)), which would have each variable a be set to the value of a which is one to the left and 2 below it in the grid. The coordinate functions are computed using the function self-address-grid!!, which takes a single argument which is the dimension number of the coordinate desired.

2.9 Remapping

Remapping is the process of taking variables in a set of processors in one vpset and moving to processors in another vpset.

A common application for remapping is *compacting.* Sometimes many processors are needed to calculate a result which is contained in some subset of those processors. If there is more work to be done on the remaining processors, it is often advantageous to send the values in the subset of processors into a new vpset which has fewer processors. The idea is illustrated in Figure 2-7.

Compacting occurs in beam calculation, for instance, in which a processor per pixel is used

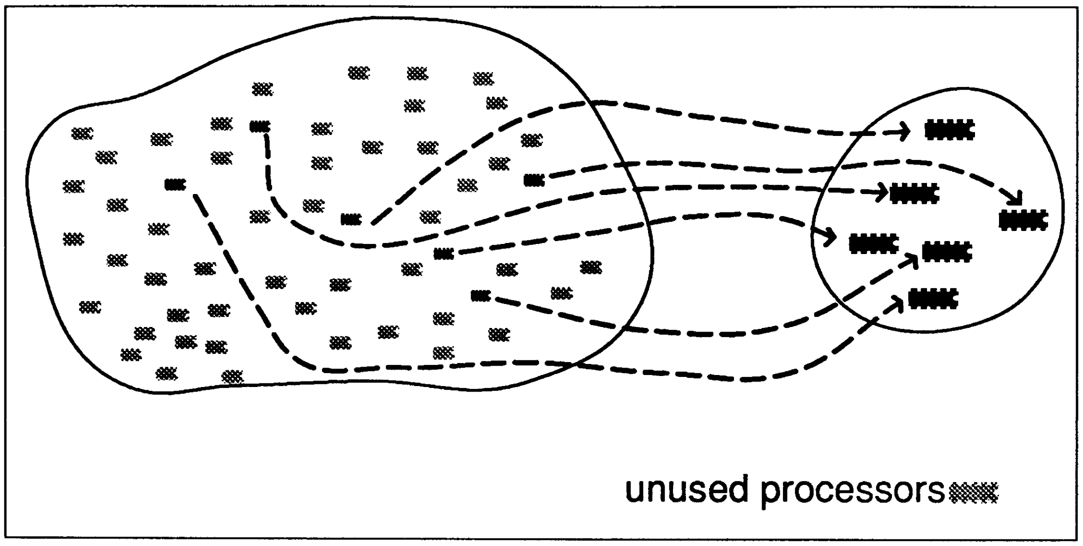


Figure 2-7: When the result of some computation reduces to values in only a few processor in the vpset, and if more processing remains to be done on these results, it often pays to create a smaller vpset in which the remaining computation can be done. This way, fewer processors remain idle.

to calculate boundary points of the beams, but then a subsequent calculation is done solely on the the beam boundaries. Other times, some aspect of a different geometry is desired in some calculation. For instance, consider a set of points in a plane. For some purpose one might want to think of these as a list of points sorted by their distance from the origin.

2.10 Movables

It is often necessary to compute a value in a single processor which is a function of values in other processors. Suppose that we wish to smooth an image which is represented on the CM by assigning a processor to each pixel. One way to do this is to replace each point value by the sum of itself and its neighboring points(Figure 2-8).

There is a choice of frame of reference here which makes the problem easy to imagine. Focus attention on a processor $\mathcal{P}$ which wants to do this calculation replacing its old value by the value of the sum of itself and its nine neighbors. From the point of view of $\mathcal{P}$ we would write:

```
(*set P (+!! P (above!! P) (below!! P) (right!! P) (left!! P)
      (above!! (left!! P)) (above!! (right!! P))
```

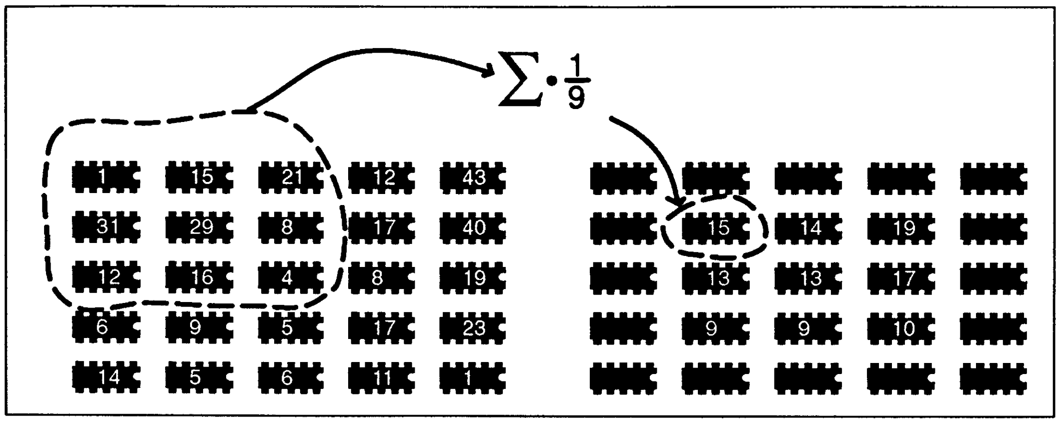


Figure 2-8: An example of a computation which involves surrounding processors. Smoothing an image by replacing each point by the average of the points around it

(below!! (left!! $\mathcal{P}$)) (below!! (right!! $\mathcal{P}$))))

The functions left!!, right!!, and so on are function which get the value of a variable passed to it which is one processor away in the direction of the name of the function. Since $\mathcal{P}$ is an arbitrary processor, the computation is done for each processor.

Usually, I implement this computation in a slightly different way. The idea is to first make a copy of variable being computed on. In the case illustrated above, imagine making a second copy of the image(as in Figure 2-9, in the variable $\mathcal{P}'$(I call this the *movable*). Then in each step, $\mathcal{P}'$ is rearranged so that if $\mathcal{P}$ and $\mathcal{P}'$ are superimposed, with an offset, so that different values of the image are at the same address (see Figure 2-9). The total set of values which visit a given processor are exactly those you need to do the smoothing.

The advantage of this method is that less communication needs to be done. The connection machine communicates in either the x or y direction in each step. Using the first strategy 12 communications are needed, using the latter uses only 8.

2.11 Slices

The size of the images is such that the CM I used did not have sufficient memory to work on the whole image at once. Instead the image was processed in pieces which I refer to as slices. A given slice is rectangular part of the whole image, and the slices are layed out over the image

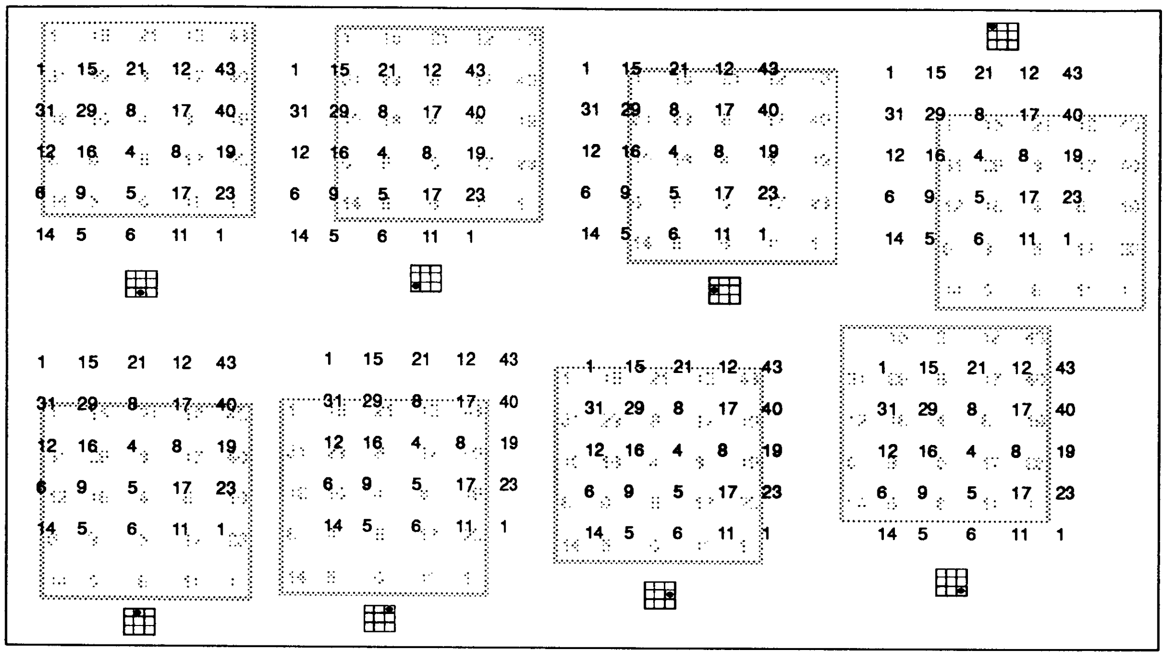


Figure 2-9: Example of a movable copy of a value in a set of processors being moved over an initial copy, in the process of computing the average at the center. The movement of the copy is such that once done, every processor has had the value of each neighboring processor visit it.

so as to have small amounts of overlap. The overlap is there to avoid the problem of features being divided up between two slices. The overlap is chosen to be bigger than the largest feature. Then, if a feature is positioned on some slice so as to be interrupted by the edge of the slice, some other slice is guaranteed to fully contained the feature.

This strategy worked adequately for most of the reader, with the exception of the beam finder section. In that case beams could be large enough so that no amount of overlap would ensure that there was no beam cut off. The solution to that problem is discussed in the chapter on beam finding.

The macro loop-over-slices, is used in the code wherever it is necessary to iterate over the complete image by processing each of the slices. The macro arranges for a variable to be set to the current slice, and to have variables bound which describe the width and height of the slice, the coordinate of the top left corner, with respect to the complete image, and other information useful in dealing with a slice.

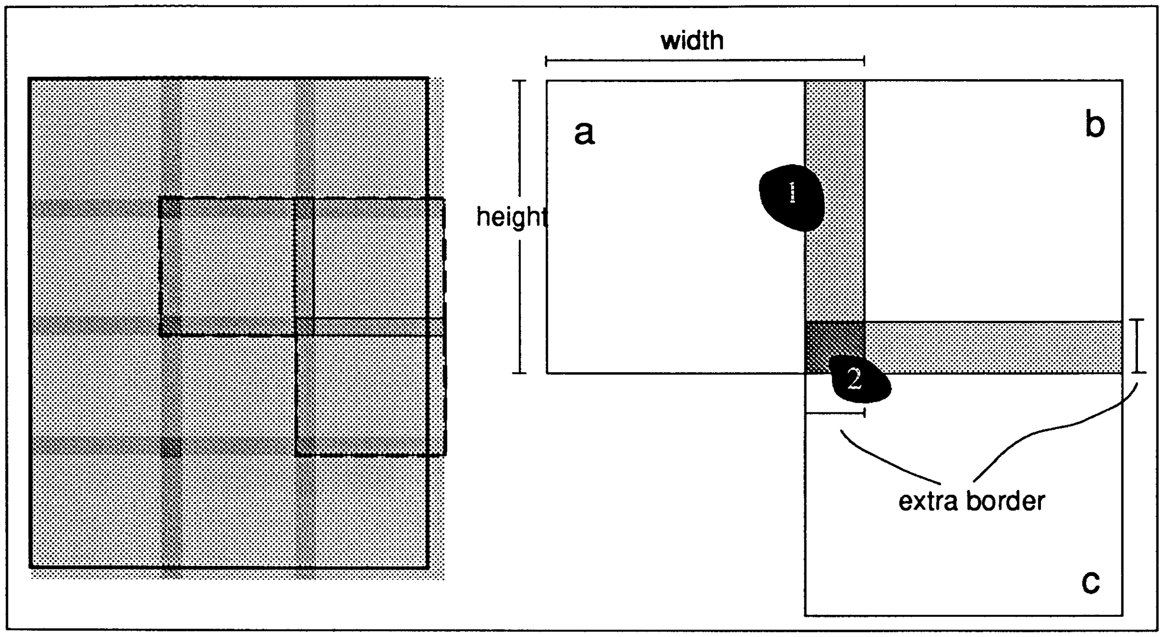


Figure 2-10: A page is processed in pieces which I call slices. Slices overlap so that the largest feature is sure to be wholly contained on at least one of them. Blob **1** is contained in slice **a** but not **b**. Blob **2** is completely contained in slice **c**.

2.12 Pointer Doubling

Sometimes the nature of a calculation is such that the processors in a vpset are logically divided into subgroups, and a calculation needs to be made which is function of all the elements of the group. In the region labelling code, for instance, I construct a linked list of all the boundary points and need to label all the points of a given boundary by a unique number.

Using a movable copy for this calculation would take take n steps for a group of n processors. But for the case where the computation is binary and associative, the same calculation can be done in a logarithmic number of steps. Each processor communicates with its adjacent processor to share its value. Then each processor communicates with a processor two away, then four away, and so on, for $\log_2 n$ steps. Figure 2-11 illustrates this process for a group of eight processors, where the operation is maximization of the alphabetic position of the value.

2.13 Scans

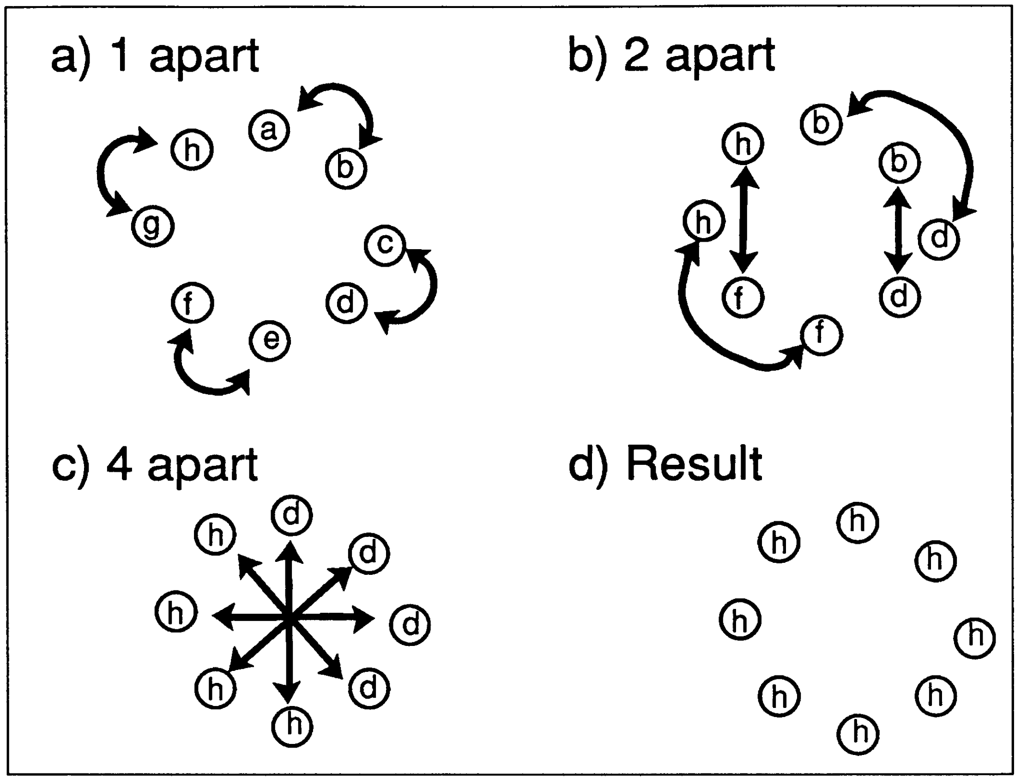
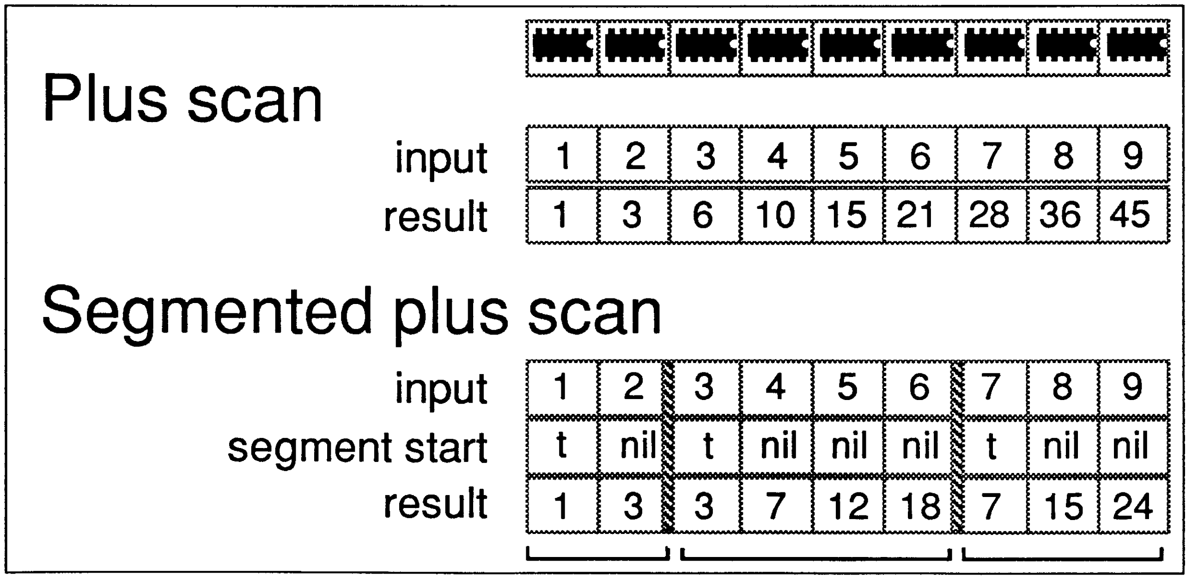


Figure 2-11: Using pointer doubling to maximize. In each step all processors share values and compute the maximum. Processors first communicate with their neighbors, then with processors successively greater powers of two apart.



Plus scan

input	1	2	3	4	5	6	7	8	9
result	1	3	6	10	15	21	28	36	45

Segmented plus scan

input	1	2	3	4	5	6	7	8	9
segment start	t	nil	t	nil	nil	nil	t	nil	nil
result	1	3	3	7	12	18	7	15	24

Figure 2-12: A plus scan operation accumulates in each processor the sum of the value in all previous processors. A segmented scan restarts the summing process wherever the segment start flag is true.

Consider a one dimensional line of processors, each with a value. Take some binary associative operation, like addition. In each processor compute the operation applied to all the processors to the left of it. A scan operation accomplishes this in time proportional to the logarithm of the number of processors, where a serial machine would do this in time proportional to the number of processors.

The connection machine software supplies two versions of this operation. One operates on all the processors in a vpset, the other on segments of the processors, where the beginning of each segment is defined by a boolean in the processor.

Scan operations are used in similar ways to pointer doubling. A scan can be in any situation where a lattice gemotry is used, in which case it is more efficient than using pointer doubling. In the code for locating beams, sets of beam boundary points are placed in segments so that calculations using segmented scan operations are possible.

Chapter 3

Locating Fixed Sized Features

3.1 Introduction

Many of the marks on a page of a printed score are always the same size and orientation within a given score. I'll refer to these marks as *fixed-sized* features. Examples of fixed-sized features are note heads, accidentals, clefs, and the numbers in time signatures. Beams, ties, crescendos, and staff lines are not. A typical page of string quartet music has one to three thousand fixed-sized features on a page, with a vocabulary of thirty to eighty symbols.

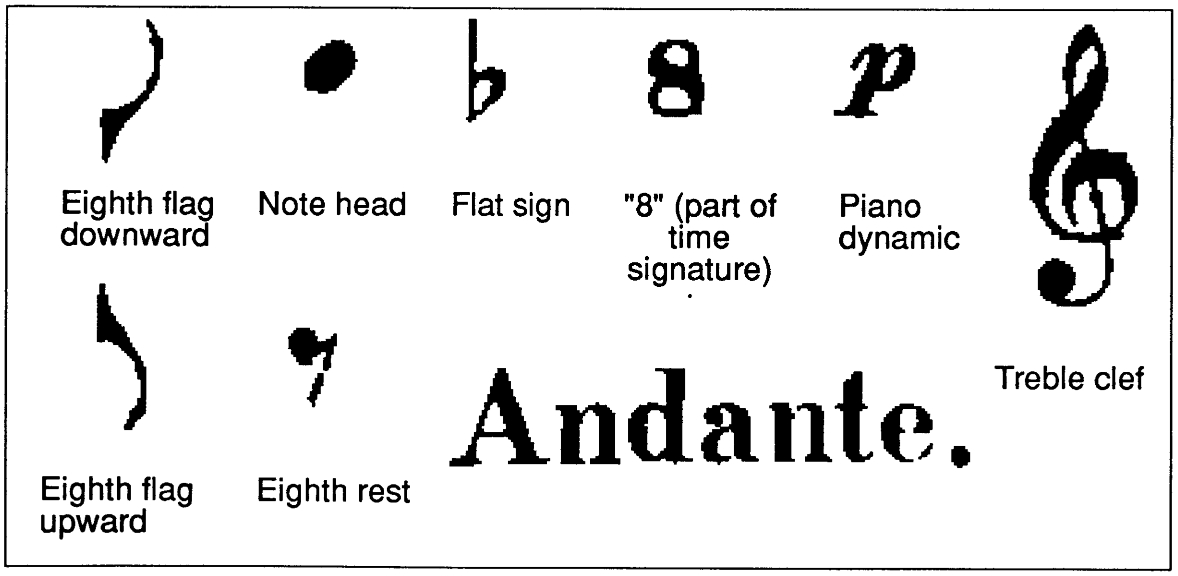


Figure 3-1: Some examples of fixed sized features.

The fact that these features are always the same size and shape makes finding them a simpler problem than if these properties varied. I locate these features by template matching. In this approach one compares an image of the feature with the scanned image. At positions where the image matches this template a feature has been identified. The connection machine proves very useful in this task. By having the template matched at many positions at once, each processor doing a match, the speed of execution is practical [1]. Still, this computation is the most time-intensive of all the feature finding, taking about two to three minutes per page on a CM2 with sixteen thousand processors.

Several things complicate finding these features on a sheet of music. Some features overlap others. The shape formed by the overlap of two features is different from the shape of either of them, making it difficult to identify either. Staff lines are a particularly large source of overlap. Staff lines not only overlap features, but connect many features together, a problem best illustrated by comparing the music recognition task to the task of optically reading text.

Programs which read text usually make the assumption that characters are disconnected blobs. This allows them to take a two step approach to their identification. First the programs identify individual blobs, then they can separately determine which character each blob represents. This method cannot be applied to music scores. Staff lines extend across the whole page, making a single connected region out of many features. Furthermore, it is not only staff lines which have this property. Often symbols are printed close enough to each other to have some small amount of overlap; note-heads in a chord touch each other and stems run from note-head to beam.

It is difficult to separate features from one another because staff lines are not printed very uniformly, and because it is hard to differentiate between noise and overlapping features. The problem is exacerbated by staff lines not being the only features which overlap others. Fully disconnecting objects requires designing a separate method for each type of overlap or extended connectivity situation. While this might be possible for staff lines, other cases seem more difficult. The approach in this work is to find features despite the overlap, and without prior

[1]To be precise, the computational complexity of locating the fixed-sized features is proportional to the combined area of all the templates, assuming a processor per pixel, and disregarding machine size effects on the speed of communication. Thus it is advantageous to reduce the number of features, and to limit their size, when possible.

disconnection.

A second problem is that features only *appear* to be exactly the same. When examined closely they are slightly different. The nonuniformity has two causes. Printing quality is not consistent. The exact shape of the impression depends on the thickness of the ink, the quality of the paper and other factors which change slightly from page to page. The scanning process also introduces some quantization noise because its finite resolution. An identically shaped object will have different pixel representations depending on its position relative to the grid because of the need to decide whether a pixel which is partially covered should be interpreted as black or white.

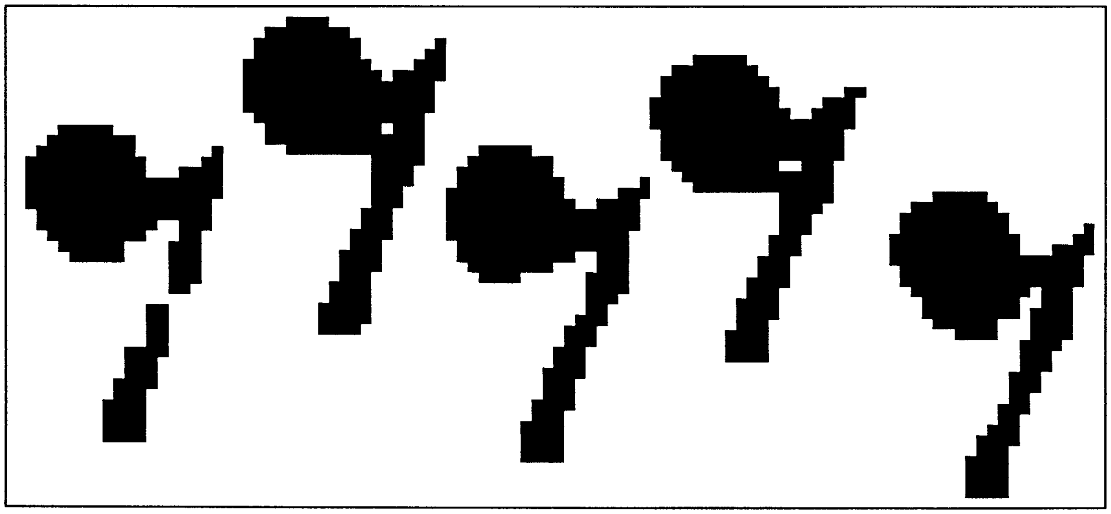


Figure 3-2: Because of quantization noise and printing irregularities, the same feature doesn't always have the same pixel representation.

In order to best avoid quantization noise, I scanned the image at 300 dots per inch, the highest resolution available to me. Since quantization noise is always of the same magnitude, roughly a pixel, the smaller a feature is, the more the noise affects the quality of the image. On features which are only a few pixels in size this noise makes it difficult to reliably identify the feature in the crowded environment of a page of music.

This noise is particularly evident in scanned images of grace notes, which are smaller than usual notes. I decided to ignore grace notes because of this problem. For the Schubert scores this is not too much of a problem, since grace notes are relatively rare, and can easily be entered by hand after the rest of the score has been optically read. Grace notes are the only featurs which are deliberately ignored.

3.2 Template Elements are Three-Valued

Template elements may be one of three values: *zero*[2], meaning look for white, *one*, meaning look for black, or neither, meaning ignore this point. How well a template matches the image is the sum of how many places a white is found where a white was looked for, plus how places a black is found where a black is desired. The ignored points do not contribute to the match.

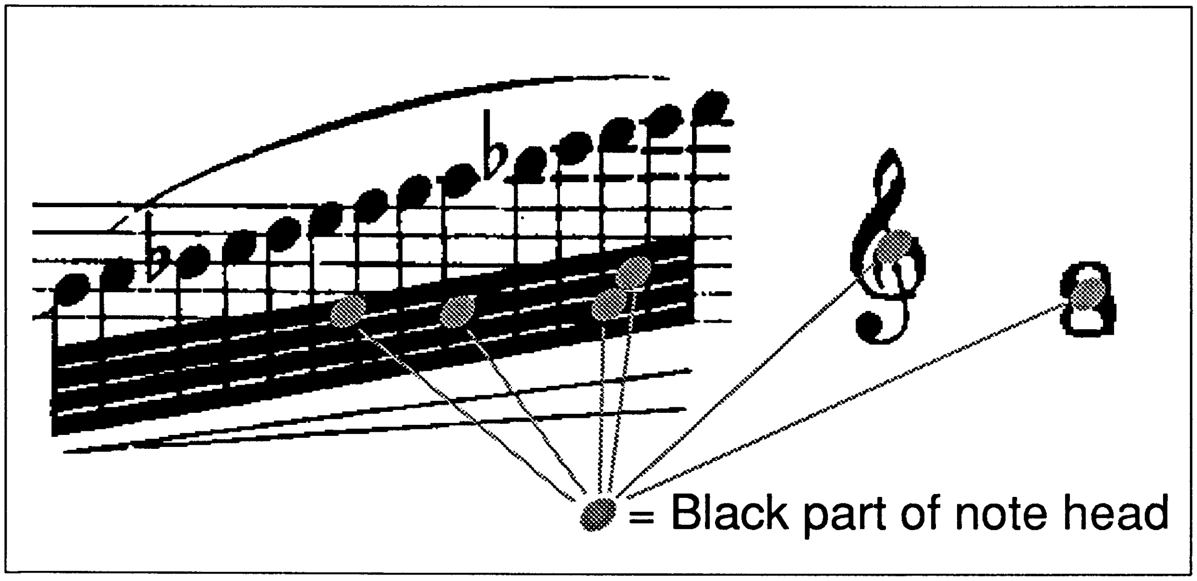


Figure 3-3: A strictly black template would match many places where the feature is not really located.

There is a reason to include each of the possible template values. The *ones* are the most obvious. One's attention is drawn to black, and it is the black part which one calls the shape of an object. When a hole (or white space) appears inside a black object, we consider the hole to be part of the containing object. If we didn't consider white space to be a distinct shape, but simply as a lack of black, a solid note head could still be distinguished from a whole note, since the whole note has less black as part of it. When a purely black template is used, features

[2]In the images black is represented by one and white by zero.

are found in all sorts of incorrect places. Figure 3-3 illustrates some places where a note head might be mistakenly found.

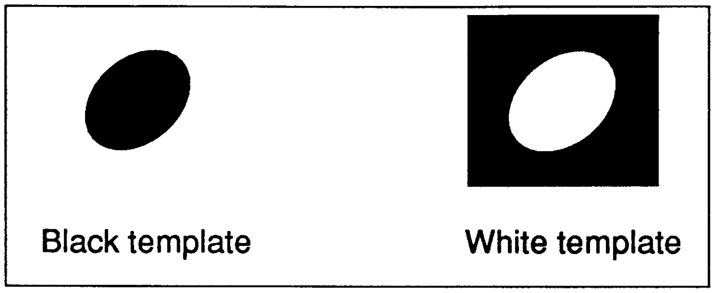


Figure 3-4: Since a black template alone won't do, a template which looks for both white and black might do better.

White space doesn't only need to be enclosed to be important. For instance the white space around flags is distinctly shaped, and useful in finding them. In these cases the white space is as distinctive as the black, and so we have a justification for looking at white space. A second approximation to a template is a combination of the black template and a white template which looks like a square with the black template cut out, illustrated in Figure 3-4.

The ignored portions of the template help solve two problems, the problem of overlap, and the problem of nonuniformity of features. Consider a note head with and without a staff line going through it. When a staff line goes through a note head, the part of the note head which is black stays black. But white surroundings and the white area in the head change. The staff line either goes through the center of the note head, or doesn't go through it. By using a template with *don't-care* values in places where a staff line might overlap, the same template can find a feature whether or not it is covered by a staff line.

Features are often near each other, with the black of a nearby mark covering an area nearby some other, for instance note heads which form part of a chord. When an occlusion occurs over part of the white space of an object, it is more difficult to use white space information, since any part of the white space might be covered by the nearby black. If features could be in any spatial relationship whatsoever, it might be impossible to use white space information, since you could never count on it being there. This is not the case in music scores.

The positions of marks on the music score have very specific relationships to each other. Staff lines either go through the center of note heads, or appear at the edges. Rests always

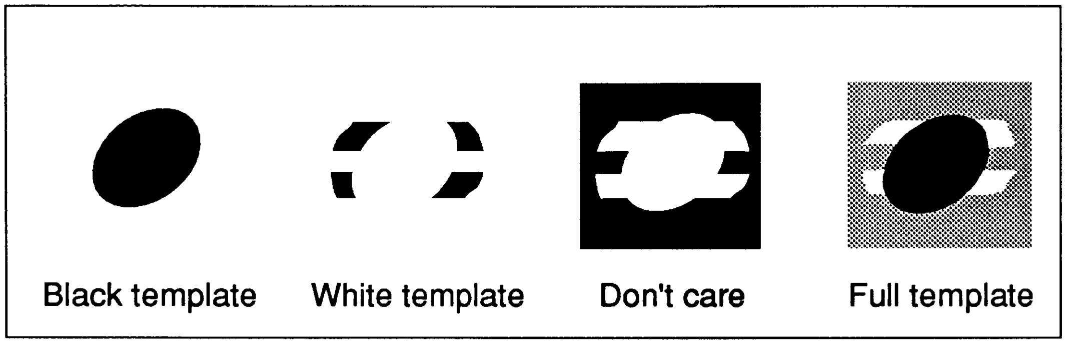


Figure 3-5: A closer approximation to what the templates should be. The first three images show each part of the template, the last a composite, with the ignored section in grey.

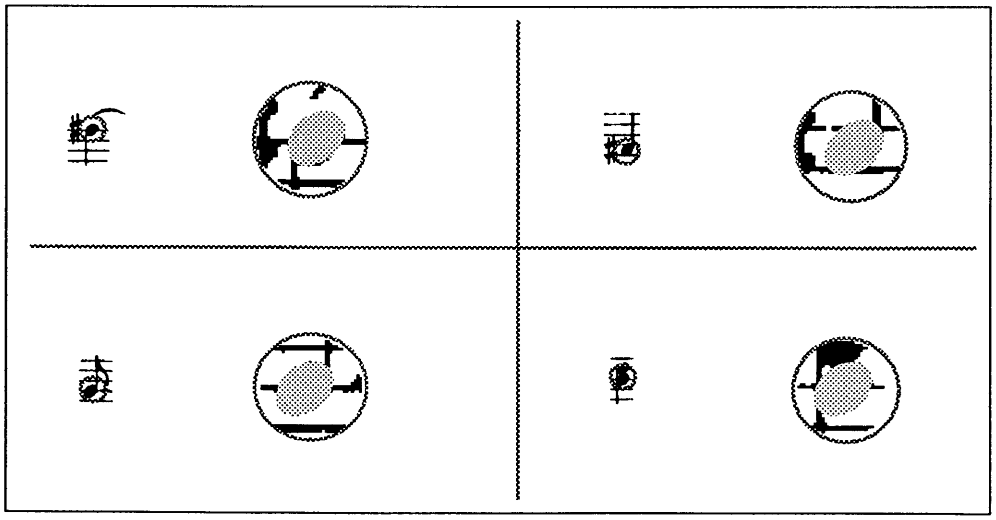


Figure 3-6: Examples of the neighborhood of a note head, showing that the assumption that there is white surrounding a note head is inadequate.

appear in the same position relative to the staff lines. Note heads appear near each other in chords, but only in certain positions relative to each other. By ignoring the portions of white space where there might be an occluding object, we make the rest of the white space usable in helping to distinguish between features.

It was not obvious how much white space to include in the template. In some cases the choice was limited since other marks were likely to be nearby, forcing the use of *ignores.* My first (wrong) intuition was to use quite a bit of white space - several times the black area - surrounding each object as part of the template. The reasoning was that the white surroundings would form a large unique template. But doing so overemphasized the importance of the white template. Because how well the template matches is the sum of the white matches and the black matches, the relative importance of the black matches was lessened by the many potential white matches. I found that about a fifty/fifty mixture of white and black was optimal. Many times this ratio was not achieved, as the ignored template values limited the total amount of white space.

3.3 Using Portions of Large Features

In some cases a feature is rather large, as in the case of the word *crescendo.* When this is the case, a smaller portion of the whole is used as a template, for two reasons. Since the time to do the template matching is proportional to the area of the template, the code runs quicker with a smaller template. Second, the extra border needed at the edge of slices [3] needs to be as large as the largest feature in order that a feature not be missed. By keeping the features smaller, less space is wasted.

When choosing a smaller portion of a feature one must take care not to choose a part which could be mistaken for some other feature, or a part which is duplicated in the feature itself, or false feature positions will be reported. When the feature for the alto clef was chosen to be upper half of the clef, the template sometimes matched twice, once to the lower half and once to the upper half. I replaced it with slightly more than half of the clef, and the problem disappeared.

[3]slices are defined and discussed in the chapter on connection machine concepts

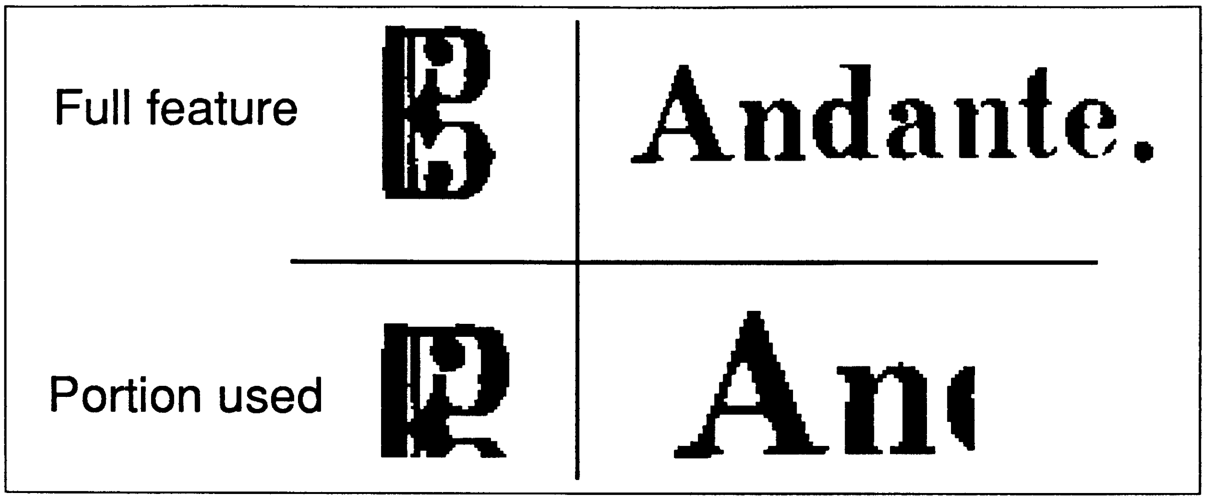


Figure 3-7: On large features, only a unique subportion is used.

3.4 Template Scores and Thresholds

If the scanned image were perfect, then it would be simple to decide at which points a feature occurred. They would simply be the points where the surroundings of that point exactly matched the template, pixel for pixel. Since the image is never perfect, the criterion for matching has to be something other than an exact match.

The template *score* is the number template elements which matched the scanned image, excluding *ignores*. Ignored points do not affect the score at all. Instead of asking that the score be perfect, a *threshold* is used to decide if a point matches a feature. Places with a score greater than the threshold are accepted as matching.

There are two issues associated with use of a threshold. First, threshold values need to be chosen. If the threshold is too low, then many places where there is no feature are matched. If the threshold is too high then some features are missed. Second, if the threshold is less than perfect, then slightly shifting the template yields a score which also might exceed the threshold.

After some experimentation with choice of threshold, I found that a threshold which was eighty five percent of the total possible score of a template was a reasonable choice. The fact that this threshold percentage was adequate for most of the features was a surprise to me, and undoubtably had something to do with how I adjusted the templates. I think that there is something interesting in this fact, but have not investigated it in detail.

When this threshold is applied to the result of the matching, a small cluster of points around where the feature was located scored high enough to pass the threshold. This is because most shapes, when only shifted slightly in position, look a lot like themselves. In order that the program didn't think that there were many note heads in a small area, each of these clusters had to be reduced to a single point representative of the position of the feature.

I experimented with several different methods for doing this. One method is to average of the positions of all the points in the cluster, weighted by score. Another is to choose the point in the cluster with maximum score. The major problem with these two methods is that they are both functions defined on a connected region, and computing a connected region is time consuming. Instead of these I chose a slightly less accurate, but local method, in which a point was considered a representative position if it was a local maxima. While this method occasionally allowed for superfluous points, these are easily removed at a later processing stage.

3.5 Creating the Templates

I created the feature templates by hand. I inspected displayed images of the score using a bitmap editor and selected portions which defined typical features. Then I edited the templates to remove noise and to specify *don't care* pixels. I used the resultant template to find features and see where it failed. After analysis of failures I was adjusted the feature until it worked well enough.

I was not able to adjust the template and threshold so that all and only features of the desired shape were found. Either some would be missed, or others found where there were in fact not any. When I could, I adjusted the feature until it tended to find all features of the desired type, and as few extras as possible. I did this since it seemed likely that I could rule out a feature which was in a place it shouldn't be more easily than hypothesizing it to be in a place where it might be.

Though features are indivisible objects from the point of view of the finding algorithm, they have parts, and these parts are necessary in later attempts to assemble the score. For a note head, the center point the head is important, since it is the position of the center on the staff that determines the pitch of a note. On a clef there is typically a spot whose placement determines what transposition to use. In such cases templates were augmented by giving symbolic names

```
(make-instance
  'feature-model
  :documentation "from 97"
  :threshold NIL
  :points '((1 :pitch-center "*"))
  :pretty-name "half note"
  :string-description
  '("================********========"
    "==============***********======="
    "============**************======"
    "----------*********-----***-----"
    "-----...-********-------***-----"
    "-----..********---------***-----"
    "-----,********----------***-----"
    "-----********----------****-----"
    ".....*******..........*****....."
    "=====*****===========*****======"
    "====*****=======1===******======"
    ".....****==========******======="
    "-----***----------*******-------"
    "-----***---------*******--------"
    "-----***-------********---------"
    "......****...*********.........."
    "======**************============"
    "=======***********=============="
    "================================")
)
```

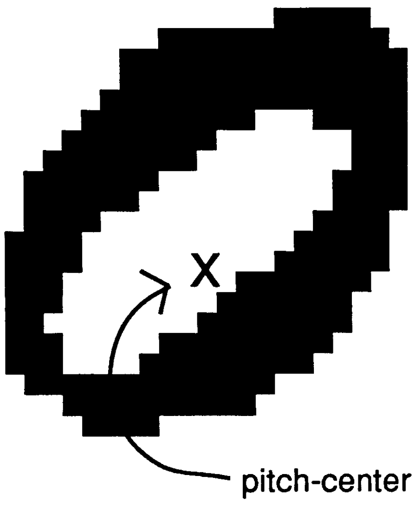


Figure 3-8: The lisp representation of a feature.

to whatever points would be useful in later computation. Figure 3-8 shows the representation of a half note.

3.6 Future Work

Though these methods work well in finding features, despite overlap and connection between them, they seems unsatisfying from a cognitive point of view. Were this same algorithm implemented in the brain its time or space requirements would be proportional to the number of features we know. The older we became, the longer it would take to recognize objects. Further, this algorithm is sensitive to the size of a feature, yet we seem able to recognize features despite their size. It would be worthwhile to do some work in making a more cognitively plausible implementation.

On a more mundane level some automation of the template acquisition would be beneficial, and also might make it possible to more easily use this work for scores printed by other publishers. One idea is to start with a rough shape of the object and refine it with search, first through size space, and later through shape space. This might be a time consuming process but only needs to be once for each publisher, after which the templates can be used for a large corpus of music.

3.7 The Code

3.7.1 Top level code for finding features

This is the top level code for finding all of the features on a given page. It is complicated by the fact that the page is divided into a number of slices. After each of the slices is processed the coordinates of the found features need to be adjusted to be in the coordinate system of the full page.

```
(defun check-if-scatterbrained (cm-image image)
  (unless (cm:attached)
    (format t "~%Forgot to attach - attaching and *cold-booting")
    (cm:attach) (*cold-boot) (setq cm-image (image-to-cm image cm-image))
    (setq image *last-image-to-cm-image*))
  (and moviep (unless *8-bit-display* (initialize-display))))

(defun coerce-to-features-list (features)
  (when (null features)
    (setq features (accept '(sequence feature-model) :prompt "Find which features?")))
  (when (or (stringp features) (keywordp features))
    (setq features (find-feature features)))
  (when (atom features)
    (setq features (list features)))
  features)

(defun local-positions-to-recognized (feature column-start row-start per-column
                                      per-row height width local-positions)
  (and local-positions
      (loop for column being the array-elements of (first local-positions)
            for row being the array-elements of (second local-positions)
            for real-column = (+ column column-start)
            for real-row = (+ row row-start)
            when  (and (< row per-row) (< column per-column)
                       (< real-row height) (< real-column width))
              collect (make-instance 'basic-recognized-feature
                                     :reduced-x real-column
                                     :reduced-y real-row
                                     :my-feature feature))))
```

```
(defun find-features-on-sheet (&optional (features *all-features*)
                                &key   (cm-image *last-image-to-cm*)
                                slices
                                &aux (results (make-hash-table)) total-to-do (progress 0)
                                (vp-set (pvar-vp-set cm-image))
                                (image (pvar-getf cm-image :image)))
  (check-if-scatterbrained cm-image image)
  (setq features (coerce-to-features-list features))
  (setq total-to-do (* (length features)
                       (if slices
                          (length slices)
                          (apply '* (*array-dimensions cm-image)))))
  (*with-all-vp-set vp-set
    (tv:noting-progress-alterable-note ("")
      (let ((valid-x-limit (- (car *current-cm-configuration*) (pvar-getf cm-image :extra-border)))
            (valid-y-limit (- (cadr *current-cm-configuration*) (pvar-getf cm-image :extra-border))))
      (loop-over-slices (slice cm-image)
        ;; get rid of some noise for now
        (when (slice-member slices)
          (for-visual-types (image-to-fb slice))
          (loop for f in features with partial do
            (tv:format-progress-note
              nil "~A ~A by ~A, top left (~a,~a)" (pretty-name f) (- row-end row-start)
              (- column-end column-start ) row-start column-start)
            (tv:note-progress (incf progress) total-to-do)
            (setq partial (local-positions-to-recognized f column-start row-start valid-x-limit
                                                          valid-y-limit image-height image-width
                           (find-basic-model slice f)))
            (setf (gethash f results) (nconc partial (gethash f results))))))
        (make-instance 'page-of-features :result-table results :image image)))))
```

3.7.2 Top level Convolution of a Feature with a Slice

```
(defvar *convolve-peak-maximum-radius* 2
  "approximate radius in pixels of a convolution peak")

(defun find-basic-model (slice model)
  "Find all occurances of feature on slice"
  (let ((conv-score-size (integer-length (send model :total-bits))))
    (*let ((score (!! 0))
           (matches nil!!))
      (*type (score conv-score-size) (matches t))
      (convolve-feature slice model matches score)
      (when (*or matches)
        (find-maxima matches score *convolve-peak-maximum-radius*)
        (extract-singles matches)))))
```

3.7.3 Convolution - Scanning Method

The first method method described is implemented in the function **convolve-feature-slower**. Slice is a portion of the input image, model a feature model object, matches is a boolean which, upon exit from the routine will be **t** wherever the convolution has scored above threshold. Score is a variable to be used to accumulate the convolution.

The feature model contains a number of pieces of information about the feature being looked for. **model-zeros** and **model-ones** are two bit arrays which are positive where the feature model expects a zero or a one respectively. The model also supplies **width** and **height**.

The loop is best understood from the point of view a single processor at a pixel $\mathcal{P}$. The pixels which contribute to a match at itself, lie downward and to the right. Imagine a sheet onto which the values of all the pixels have been copied (the variable **movable-slice**). In order to have all those pixels pass by $\mathcal{P}$ one first pulls the sheet to the left, as far as necessary to complete on row, then slides it upward a row, and pulls it in the opposite direction until the end of that row, and so on. In the code **moving-left** keeps track of which direction the sheet is currently going.

At each step of loop, one gets the value of **looking-for** from the feature model, compares it to the pixel which currently lies on top of $\mathcal{P}$, and increments **score** accordingly. The functions **left!!** and **right!!** move the sheet. Finally after all the accumulating is done, the model is asked for a threshold, which is a number of matches necessary to have the pixel considered likely to have the feature nearby.

Of course nothing changes if, instead of imagining a single processor $\mathcal{P}$, there was a processor at every pixel, which is case on the connection machine. So the convolution gets done in time proportional to the area of the feature model.

```
(defun convolve-feature-slower (slice model matches score)
  (*locally-type (:fast ((slice matches) t) (score :field))
    (*let ((movable-slice slice))
      (*type (movable-slice t))
      (*set score (!! 0))
      (*set matches nil!!)
        (loop with width = (send model :width) and height = (send model :height)
              for row from 0 repeat height
              with model-zeros = (send model :zeros)
              with model-ones = (send model :ones)
              with moving-left = t
              with maxcol = width
              do
          (loop for column from 0 repeat width
                for column-to-do = (if moving-left column (- maxcol column 1))
                for looking-for = (cond ((plusp (aref model-ones row column-to-do)) 'ones)
                                        ((plusp (aref model-zeros row column-to-do)) 'zeros)
                                        (t nil))
                do
            (case looking-for
              (ones (*set matches movable-slice))
              (zeros (*set matches (not!! movable-slice))))
            (when looking-for
              (*when matches (*set score (1+!! score))))
            (if moving-left
              (*set movable-slice (right!! movable-slice))
              (*set movable-slice (left!! movable-slice))))
          (*set movable-slice (below!! movable-slice))
          (if moving-left
            (*set movable-slice (left!! movable-slice))
            (*set movable-slice (right!! movable-slice)))
          (setq moving-left (not moving-left))))
      (for-visual-types (rainbow-scale-to-fb score (send model :threshold)))
      (*set matches (>!! score (!!i (send model :threshold)))))))
```

3.7.4 Convolution - Packing Method (faster)

The faster method for convolving incorporates a number of optimizations. In order to minimize communication time, the bits are first compacted upwards so that a feature is represented in a line of 30 or so bit fields, instead of an area of 1 bit fields. The communications speeds up because there is a constant overhead for each communications cycle which is greater than the time spent communicating a single bit. Using this method we make length+height communications, instead of length $\times$ height. The compacted bit patterns are precomputed on the front end and are stored in the feature model.

A second optimization is to write the inner part of the loop in *paris*, a lower level language for the connection machine.

A third optimization is to periodically check if the number of template elements can't possibly be enough to cross the threshold in any of the processors. When this happens we can cease computing. This is most useful when looking for sparse features such as clefs and dynamics.

```
(defun convolve-feature (slice model matches score &aux logwidth logtotalbits)
  "This time pack horizontal bits into a single field, and then iterate over height"
  (multiple-value-bind (width look-mask care-mask total-bits)
      (feature-pack-parameters model)
    (setq logwidth (integer-length width))
    (setq logtotalbits (integer-length total-bits))
    (*locally-type (:fast (slice 1) (matches t) (score logtotalbits))
      (*let ((packed-field (!! 0))
             (packed-field-sum (!! 0))
             (temp (!! 0)))
        (*type ((packed-field temp) width) (packed-field-sum logwidth))
        (with-locations (packed-field packed-field-sum score temp)
          (*set packed-field slice)
          (*set score (!! 0))
          (dotimes (i (1- width))
            (cm:get-from-news-always-1l
              (+ packed-field-loc i 1) (+ packed-field-loc i) 0  :upward 1))
          (loop for look being the array-elements of look-mask
                for care being the array-elements of care-mask
                for bits-so-far = 0 then (+ bits-so-far (logcount care))
                with cantmiss  = (- total-bits (scl:send model :threshold))
                for iter from 1
                do
            (and (zerop (mod iter 4)) ;; every 4 seems not to impose too large a penalty
                 (when (> (-  bits-so-far (cm:global-u-max-1l score-loc logtotalbits)) cantmiss)
                   (go si:end-loop)))
            (cm:logeqv-constant-3-1l temp-loc packed-field-loc look width)
            (cm:logand-constant-2-1l temp-loc care width)
            (cm:u-logcount-2-2l packed-field-sum-loc temp-loc logwidth width)
            (cm:get-from-news-always-1l packed-field-loc packed-field-loc 1 :upward width)
            (cm:u-add-3-3l
              score-loc score-loc packed-field-sum-loc logtotalbits logtotalbits logwidth)))
        (*set matches (>!! score (!! (the fixnum (scl:send model :threshold)))))
        (for-visual-types (rainbow-scale-to-fb score (scl:send model :total-bits))))))
```

3.7.5 Finding the Convolution Peaks

A point is considered to be part of the peak if it is a local maxima. This method was chosen to be fast, and involves communication over a small number of processors. **spiral-deltas** computes a series of incremental moves by 1 in either the x or y direction which taken in sequence can move a copy of the data nearby a processor over the center one. **extract-singles** gets a representation of the points where the feature was found. Initially the points are scattered on a $2d$ surface. The routine sends them so that they are adjacent in a $1d$ geometry and then uses a starlisp function to copy them to an array on the front end.

```
(defvar *convolve-max-blobs-per-slice* 500
  "Arbitrary maximum number of features found
   on a single page. If more than this there is probably an error")

(defvar *convolve-max-blobs-length* (integer-length *convolve-max-blobs-per-slice*))
```

```
(defcached spiral-deltas ((radius))
  "spiral deltas returns a list of lists (dx dy) which when used with news instructions
   have each processor in a box of width 2r + 1 be brought to the central processor"
  (flet ((duplicate-element (element times)
           (loop for i below times collecting element)))
    (let ((s (loop for r from 1 to radius
                   with directions = (vector '(1 0) '(0 1) '(-1 0) '(0 -1))
                   collect (aref directions 3)
                   nconc (duplicate-element (aref directions 0) (1- (* r 2)))
                   nconc (duplicate-element (aref directions 1) (* r 2))
                   nconc (duplicate-element (aref directions 2) (* r 2))
                   nconc (duplicate-element (aref directions 3) (* r 2)))))
      s)))

(defun find-maxima (places score &optional (radius 2) &aux (conv-score-size (pvar-length score)))
  (*locally-type (:fast (places t) (score conv-score-size))
    (let* ((diff-size (*when places (integer-length (1+ (- (*max score) (*min score)))))))
      (*let ((diff (if!! places (1+!! (-!! score (!! (*min score)))) (!! 0)))
             neighbor)
        (*type ((neighbor diff) diff-size))
        ;; isolate possible peaks
        (for-visual-types (image-to-fb places))
        (*all
          (*set neighbor diff)
          (loop for (dx dy) in (spiral-deltas radius)
                do (*set neighbor (news!! neighbor dx dy))
                   (*set places (and!! places (>=!! diff neighbor))))
          ;; a typical error mode is that two adjacent pixels are selected. Clean this up here
          (*when (or!! (above!! places) (left!! places)) (*set places nil!!))
          (for-visual-types (image-to-fb (not!! places)))
          )))))

(defun extract-singles (places)
  "pull out the x and the y coordinated of each selected processor, and return
   as a list of two vectors, one of x coordinates and one of y"
  ;; now only one processor per blob is selected
  (*locally-type ((places t))
    (*when places
      (*let ((enum (!! 0))
             coord sent-coord)
        (*type (enum *convolve-max-blobs-length*) ((coord sent-coord) :cube))
        (let ((howmanyblobs (*sum (!! 1))))
          (cond ((> howmanyblobs *convolve-max-blobs-per-slice*) (break))
                ((plusp howmanyblobs)
                 (*set enum (enumerate!!))
                 (let ((x-vector (make-array howmanyblobs))
                       (y-vector (make-array howmanyblobs)))
                   (*with-x-and-y (x y)
                     (*set coord x)
                     (*pset :no-collisions coord sent-coord enum)
                     (pvar-to-array sent-coord x-vector :cube-address-end howmanyblobs)
                     (*set coord y)
                     (*pset :no-collisions coord sent-coord enum)
                     (pvar-to-array sent-coord y-vector :cube-address-end howmanyblobs)
                     (list x-vector y-vector)
                     ))))))))))
```

3.7.6 Feature Model Flavor Definition

The feature model is an object which contains all information pertinant to a feature, including dimensions, geometry annotations and cached values for bit strings needed in the convolution routines.

```
(defparameter *feature-white-character* #\-)

(defparameter *feature-black-character* #\*)

(defparameter *feature-content-characters*
              (list *feature-black-character* *feature-white-character*))

(defvar *all-features* nil)

(defflavor feature-model
      (threshold documentation pretty-name
                 ones zeros looking-for look-mask care-mask (total-bits 0)
                 ;; coordinate are wrt to topleft corner x right y down.
                 points width height black-top black-left black-bottom black-right
                 (full-feature nil) full-feature-offset)
        ()
  (:initable-instance-variables threshold documentation
                                pretty-name points full-feature full-feature-offset)
  (:init-keywords :string-description)
  (:required-init-keywords :string-description)
  :settable-instance-variables
  :gettable-instance-variables)
```

3.7.7 Geometry Information for Feature Models

These routines supply information about the shape, size and annotated point in a feature.

```
(defmethod (point-position-maybe feature-model) (point-label)
  (declare (values x y))
  (let ((found (assoc point-label points)))
    (values (cddr found) (cadr found))))

(defmethod (some-point-position feature-model) (&rest point-labels)
  (loop for l in point-labels
        for found = (assoc l points)
        when found do (return (values (cddr found) (cadr found)))))

(defmethod (point-position feature-model) (point-label)
  (declare (values x y))
  (case point-label
    (:left (values black-left (/ (+ black-top black-bottom) 2.0)))
    (:right (values black-right (/ (+ black-top black-bottom) 2.0)))
    (:top (values (/ (+ black-left black-right) 2.0) black-top))
    (:bottom (values (/ (+ black-left black-right) 2.0) black-bottom))
    (:center (values (/ (+ black-left black-right) 2.0) (/ (+ black-top black-bottom) 2.0)))
    (otherwise
      (let ((found (assoc point-label points)))
        (assert found (points point-label) "couldn't find label ~A in ~A" point-label pretty-name)
        (values (cddr found) (cadr found))))))
```

```
(defmethod (black-bounds feature-model) ()
  (declare (values top left bottom right))
  (values black-top black-left black-bottom black-right))

(defmethod (feature-pack-parameters feature-model) ()
  (values width look-mask care-mask total-bits))
```

3.7.8 Interface for Feature Models

```
(defun find-feature (name &aux (name (string name)))
  (find-if (λ (f) (string-equal (pretty-name f) name :end1 (length name))) *all-features*))

(defun show-feature(it &optional (stream *standard-output*) location)
  (if (stringp it) (setq it (find-feature it)))
  (with-room-for-graphics (stream (send it :height))
    (present it 'feature-model :stream stream :location location)
    it))

(defmethod (draw feature-model) (x y &optional (alu :draw))
  (draw-image full-feature (+ x (car full-feature-offset)) (+ y (second full-feature-offset))
                          :opaque nil :alu alu))

(defmethod (pretty-name feature-model) () pretty-name)

(defmethod (sys:print-self feature-model) (stream &rest ignore)
  (if (member :draw-1-bit-raster (send stream :which-operations))
    (draw-image ones 0  0 :stream stream
                    :transform `(1 0 0 -1 0 ,height))
    (format stream "#<Feature model for ~a>" pretty-name)))

(defmethod (:fasd-form feature-model) ()
  `(find-feature ',pretty-name))
```

3.7.9 Construction of a Feature Model

The magic 85 percent threshold is supplied here, if one is not supplied in the feature definition.

```
(defmethod (make-instance feature-model :after) (&key string-description &allow-other-keys)
  (check-inits self string-description)
  (multiple-value-bind (first-row last-row first-column last-column)
      (content-bounds self string-description)
    (setq height (1+ (- last-row first-row)))
    (setq width (1+ (- last-column first-column)))
    (grok-points self string-description first-row first-column)
    (if full-feature
      (compute-black-bounds self full-feature
                              (- first-column (car full-feature-offset))
                              (- first-row (second full-feature-offset)))
      (compute-black-bounds self string-description first-column first-row))
    (create-arrays self)
    (fill-in-arrays self string-description first-row first-column last-column)
    (handle-full-feature self)
    (unless (and threshold (> threshold 1))
      (setq threshold (round (* (or threshold .85) total-bits))))
    (update-feature-list self)))
```

3.7.10 Parsing the marked points in a feature

The annotated points are represented by an association list pairing character to name and replacement character. When character appears in the representation, the coordinate is noted, and the replacement character is put into the representation (so as not to leave a blank).

```
(defmethod (grok-points feature-model) (string-description first-column first-row &aux result)
  (flet ((to-char (thing)
          (cond ((numberp thing) (coerce (format nil "~A" thing) 'character))
                ((null thing) nil)
                ((symbolp thing) (coerce (string thing) 'character))
                ((stringp thing) (coerce thing 'character))
                ((characterp thing) thing)
                (t (error "couldn't figure out what character ~S should be" thing)))))
    (setq points
         (loop for (original token replace) in points
               collecting (list (to-char original) token (to-char replace))))
    (flet ((get-points-from-strings (string-description offset-x offset-y &aux what)
            (loop for string in string-description
                  for line from 0
                  for position = (position-if
                                   (λ(c) (setq what (car (member c points :test 'char= :key 'car))))
                                   string)
                  when position
                    do (when (third what)
                         (setf (aref string position) (third what)))
                    and collect (cons (second what) (cons (- line offset-y) (- position offset-x))) into result
                  finally (return result))))
      (when full-feature
        (setq result (get-points-from-strings full-feature
                                              (- first-column (first full-feature-offset))
                                              (- first-row (second full-feature-offset)))))
      (setq points (append result (get-points-from-strings string-description first-column first-row))))))
```

3.7.11 Computing the Bounds of Feature

```
(defmethod (content-bounds feature-model) (string-description)
  (let* ((first-row (position-if (λ(s) (has-some-content self s)) string-description))
         (last-row (position-if (λ(s) (has-some-content self s)) string-description :from-end t))
         (first-column (apply 'min (mapcar (λ(s) (content-start self s)) string-description)))
         (last-column (apply 'max (mapcar (λ(s) (content-end self s)) string-description))))
    (values first-row last-row first-column last-column)))

(defmethod (compute-black-bounds feature-model)
           (string-description left top &aux (c (list *feature-black-character*)))
  (setq black-top
        (- (position-if (λ(s) (has-some-content self s c)) string-description) top))
  (setq black-bottom
        (- (position-if (λ(s) (has-some-content self s c)) string-description :from-end t) top))
  (setq black-left
        (- (apply 'min (mapcar (λ(s) (content-start self s c)) string-description)) left))
  (setq black-right
        (- (apply 'max (mapcar (λ(s) (content-end self s c)) string-description)) left)))

(defmethod (has-some-content feature-model)
           (string &optional (content-characters *feature-content-characters*))
  (find-if (λ(c) (member c content-characters :test 'char=)) string))
```

```
(defmethod (content-start feature-model)
           (string &optional (content-characters *feature-content-characters*))
  (or (position-if (λ(c) (member c content-characters :test 'char=)) string) most-positive-fixnum))

(defmethod (content-end feature-model)
           (string &optional (content-characters *feature-content-characters*))
  (or (position-if (λ(c) (member c content-characters :test 'char=)) string :from-end t)
      most-negative-fixnum))
```

3.7.12 Creating the Black and White Arrays

This is where the compacted bit strings are created for the fast version of the convolution.

```
(defmethod (create-arrays feature-model) ()
  (let ((width32 (* 32 (ceiling width 32))))
    (setq ones (make-array (list height width32) :element-type 'bit :initial-element 0))
    (setq zeros (make-array (list height width32) :element-type 'bit :initial-element 0))
    (setq looking-for (make-array (list height width32) :initial-element nil )))
  (setq look-mask (make-array height))
  (setq care-mask (make-array height)))

(defmethod (fill-in-arrays feature-model) (string-description first-row first-column last-column)
  (loop for string in (nthcdr first-row string-description)
        for i below height
        for care = 0
        for look = 0
        do
    (loop for c being the array-elements of string from first-column to last-column
          for j from 0
          do
      (cond ((char= c *feature-black-character*)
             (setf (aref ones i j) 1)
             (setf (aref looking-for i j) 1)
             (incf total-bits)
             (setq care (logior care (ash 1 j)))
             (setq look (logior look (ash 1 j))))
            ((char= c *feature-white-character*) (setf (aref zeros i j) 1)
             (setf (aref looking-for i j) 0)
             (incf total-bits)
             (setq care (logior care (ash 1 j))))))
    (setf (aref care-mask i) care)
    (setf (aref look-mask i) look)))
```

3.7.13 Handle the Case Where a Full Feature is Specified

When a portion of a feature is used, the full bitmap of the feature can also be specified. This is useful for two reasons: 1) For debugging purposes the full feature can be plotted, and 2) Because some of the annotated points may be outside the bounds of the portion.

```
(defmethod (handle-full-feature feature-model) ()
  (if full-feature
    (setq full-feature (make-full-feature self full-feature))
    (progn
      (setq full-feature ones)
      (setq full-feature-offset '(0 0)))))
```

```
(defmethod (make-full-feature feature-model) (string-description)
  (let* ((width (* 32 (ceiling (length (car string-description)) 32)))
         (height (length string-description))
         (full-feature (make-array (list height width) :element-type 'bit :initial-element 0 )))
    (loop for string in string-description
          for i from 0
          do
      (loop for c being the array-elements of string
            for j from 0
            do
        (when (char= c *feature-black-character*) (setf (aref full-feature i j) 1))))
    full-feature))
```

3.7.14 Bookeeping and Error Checking

```
(defmethod (check-inits feature-model) (string-description)
  (assert (apply #'= (mapcar 'length string-description)) ()
          "the strings in the string description should allbe the same length")
  (check-type threshold (or null (number 0 *))) )

(defmethod (update-feature-list feature-model) ()
  (setq *all-features* (delete-if (λ(f) (string= (pretty-name f) pretty-name)) *all-features*))
  (push self *all-features*)
  (setq *all-features* (sort *all-features* #'string<
                             :key (λ(f) (pretty-name f)))))
```

3.7.15 Definition of a Recognized Feature

A recognized feature is an instance of a feature found on the score. As such it keeps a pointer to the feature model, and information about where the feature is found. We redefine methods for getting geometry information which translate the positions to the coordinate system of the page.

```
(defflavor basic-recognized-feature ((my-feature nil) (reduced-x nil) (reduced-y nil))
           (util-mixin has-relations has-surroundings sys:property-list-mixin)
  :initable-instance-variables
  :settable-instance-variables
  :gettable-instance-variables)

(defmethod (sys:print-self basic-recognized-feature) (stream &rest ignore)
  (format stream "#<~A at (~A,~A)>" (pretty-name my-feature) (floor reduced-x) (floor reduced-y)))

(defmethod (draw basic-recognized-feature) (&rest args)
  (apply 'draw my-feature reduced-x reduced-y args))

(defmethod (point-position basic-recognized-feature) (point)
  (multiple-value-bind (x y)
      (point-position my-feature point)
    (values (+ reduced-x x) (+ reduced-y y))))

(defmethod (some-point-position basic-recognized-feature) (&rest points)
  (multiple-value-bind (x y)
      (apply 'some-point-position my-feature points)
    (and x y
         (values (+ reduced-x x) (+ reduced-y y)))))
```

```
(defmethod (point-position-maybe basic-recognized-feature) (point)
  (multiple-value-bind (x y)
      (point-position-maybe my-feature point)
    (and x y
         (values (+ reduced-x x) (+ reduced-y y)))))

(defmethod (top basic-recognized-feature) ()
  (+ (send my-feature :black-top) reduced-y))

(defmethod (bottom basic-recognized-feature) ()
  (+ (send my-feature :black-bottom) reduced-y))
```

3.7.16 Definition of a Full Page of Features

This object contains a list of all the features of a given type which occured on a single page of the score. The **draw** method here and elsewhere are used for debugging.

```
(defflavor page-of-features (result-table image elements)
          (util-mixin)
  :initable-instance-variables
  :settable-instance-variables
  :gettable-instance-variables)

(defmethod (make-instance page-of-features) (&rest ignore)
  (setq elements 0)
  (maphash (λ(ignore d) (incf elements (length d))) result-table))

(defmethod (sys:print-self page-of-features) (stream &rest ignore)
  (format stream "#<Feature Network with ~r feature~p, ~A instance~p found>"
          (hash-table-count result-table) (hash-table-count result-table)
          (if (zerop elements) "no" elements) elements))

(defmethod (draw page-of-features) (&key (reduce-by 6))
  (with-room-for-image
    (image :reduce-by reduce-by
           :label (format nil "~V⊂features from ~A~⊃" '(:eurex :italic :huge)  (send image :name))
           :label-object self)
    (maphash
      (λ(ignore a)
        (mapc (λ (n) (dw:with-output-as-presentation (:object n :type 'sys:expression)
                        (draw n))) a))
      result-table)
    nil))
```

Chapter 4

Locating Beams

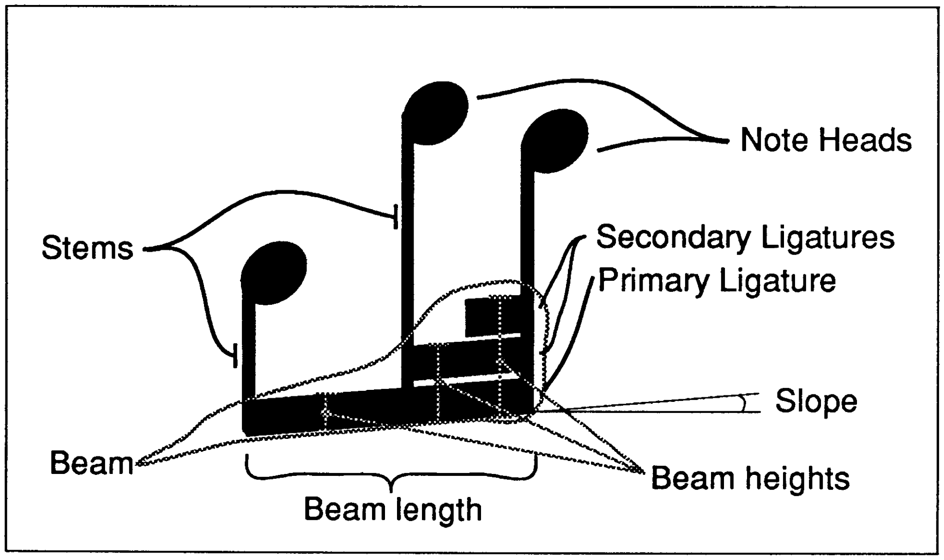


Figure 4-1: Parts of a beamed set of notes.

4.1 Introduction

Beams connect the stems of a set of notes signifying that they are to be played as s single group. In most cases a beam defines the duration of the notes which it attaches. I'll call the whole object a *beam*. Each of the separate lines of which it is made will be called a *ligature*[1].

[1]This differs slightly with [8], who calls the complete object a beam, but also calls the individual lines primary and secondary beams, which doesn't adequately make the distinction between whole and part.

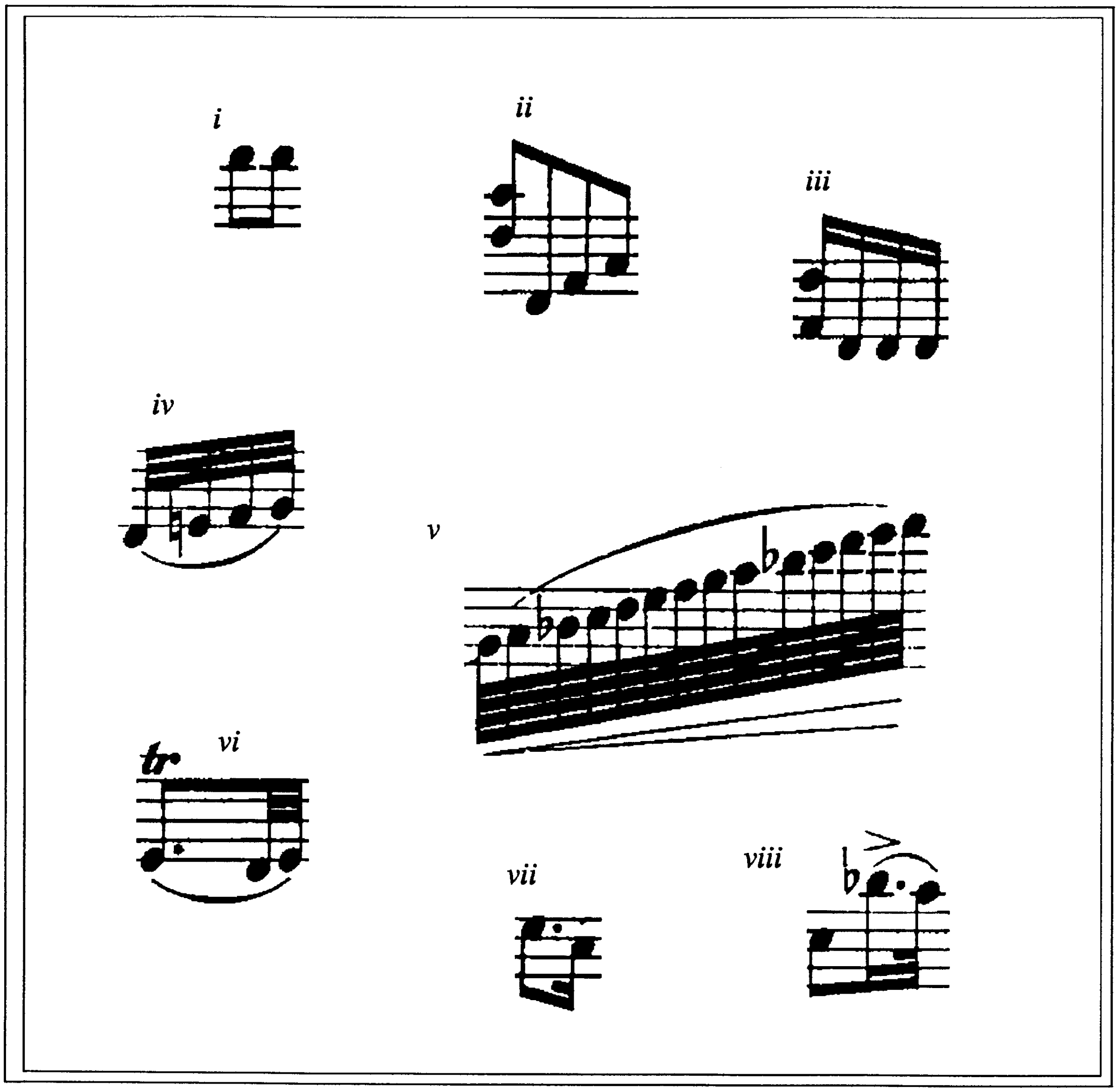


Figure 4-2: A sampling of beam shapes.

By length of a beam, I mean the distance between its leftmost and rightmost pixel, and by height, I mean the height at some particular point along the length of the beam (since beams may have different numbers of ligatures at different points). At least one of the top and bottom edges of the beam is flat. At times I'll talk about the slope of the beam, and I'll mean the slope of the flat edge.

What is interesting about recognizing beams is that while their shapes do exhibit a regularity, that regularity is not obviously parametrizable. The task of recognizing them has two fundamental parts to it. First a representation of a beam needs to be designed - we need a precise way of describing and talking about it, and second, a method needs to be designed to recover this representation from the bit images.

Visually, beam shapes vary in their length, in their slope, and in the number of ligatures at various positions along the length of the beam. In terms of their surroundings they overlap staff lines but not measure lines, they have stems connecting them to note heads, and group together a number of note heads which typically lie at varying distances either above or below the beam, and occasionally on both sides.

In recovering beams from the score, we should keep in mind the tasks that need to be accomplished. These are threefold: Each beam must be associated with the set of note heads which it groups. Each note head needs to be assigned a duration which depends on the number of ligatures above or below it. Finally we need to check whether the beam has been correctly located, and resolve cases which are uncertain.

4.2 Representation

The representation which I chose is an intermediate representation, a compromise between a form which is reasonably well suited for creation from some transformation of the image, and a form which would anticipate all uses of the beam information. The hope is that it represents the visual information completely enough, but is in a form which is easy enough to manipulate so that representations can be easily built on top of it.

I considered several choices of representation before settling on one. For instance a representation as a series of boxes or parallelograms might have worked. Whenever I thought of the representation in terms of boxes, though, it seemed that this was too general a representation.

All the boxes of the beam would have an edge which would be aligned with the others (along the top edge), but this fact would have to be grafted on to the box representation as an extra constraint, and would have to be explicitly checked for in any found beam. Also, any box representation would be ambiguous - there are several ways that a series of boxes could cover a beam, and any method for finding a beam would have to deal with this ambiguity.

My representation relies on beams having at least one flat top or bottom edge. The representation of the beam will root itself with respect to this line. The height of the beam at a given point can then be measured with respect to that line. This representation makes the assumption that one of the edges *is* in fact flat. While I believe that this is true of beams in general, and certainly is true of beams in the Schubert scores, I am not sure that this is true of all beams in common music notation. If beams are more complicated in other score then the algorithm will need to be rewritten.

There are two natural units for the representing the heights of the ligatures. One is height in pixels and the other is the number of ligatures. The latter is more interesting information, but is potentially harder to calculate. This is because one is not a priori sure that that the height of a single ligature is stable enough in the printed scores to rely on it, and because the slope of a beam is variable, the height of a ligature might vary as the slope changes.

Why might the height change? It depends on how the beams are produced. If we imagine a rectangular piece of metal which is tilted to produce the varying slope, then the height would increase as the absolute value of the slope increased. If on the other hand the production of a beam were considered a skewing of parallelogram, initially a rectangle, then the height would not change.

Fortunately it seems that in the Schubert scores we needn't worry about such intricacies. The beam height tends to be stable across pages, and the maximal slope of a beam is small enough that height changes due to angle shifts are not noticable, if present. So changing from units of pixels to ligatures is a matter of rounding to the nearest multiple of the standard beam height.

I said that the representation records the height along the length of the beam, and once again we have the issue of what unit of measure to use. In this case, however, no obvious unit presents itself. Beams for even the same type of tuplet can have radically varying lengths, since the material is typeset in reference to all the other staffs in a system. If there are few notes in other staffs then the beam will be short, if there are many the beam will be long.

The most desirable representation would be one which assigned a beam height to each head modified by it. But in order to do that we need to have interpreted much of the score by the time we interpret beams, and that hasn't been done yet. So instead the pixel scale was chosen, with the idea that at a later stage of interpretation it could be transformed to something else.

It doesn't make sense to record the beam height at each pixel along the vertical length of the beam, since it changes so slowly on a pixel scale. So, in order to conserve space, I chose to run-length encode the beam height along its length.

In summary, the representation of a beam will start with a line representing either the top or bottom edge. We'll record the slope and horizontal limits of this line, and an indication of whether it is the top or bottom of the beam. The heights of the beam are stored as a list of the height of the beam (in ligatures) and its horizontal extent (in pixels).

4.3 Finding the Beams

Given a more precise description of what is being looked for, we are ready to start discussing how to find the beams. The strategy is to disconnect the beam from other parts of the score, transform the beam into a single connected region, find the edges of that shape, and then do calculations with the edge points in order to compute the necessary parameters.

A fundamental idea used over and over again in this work is the idea of connectivity. Often a feature-finding algorithm starts by disconnecting the feature in question from other features. Then each connected region is labelled, and finally computations on these labelled regions work on distinguishing the features we are looking for from those that we don't care about.

Why is labelling important? Because while people can look at a blob of ink and identify it as a single object quite readily, that information is not explicit in a pixel representation. The process of labelling gives each connected object a unique identification, and gives each pixel, by access to that label, a method of determining which object it is part of. Once a unique label is given, we can collect all the points in a region, sort the points by region, and do other operations which facilitate computation on the feature.

To disconnect the beams from the rest of the score I exploit a fact about the relative sizes

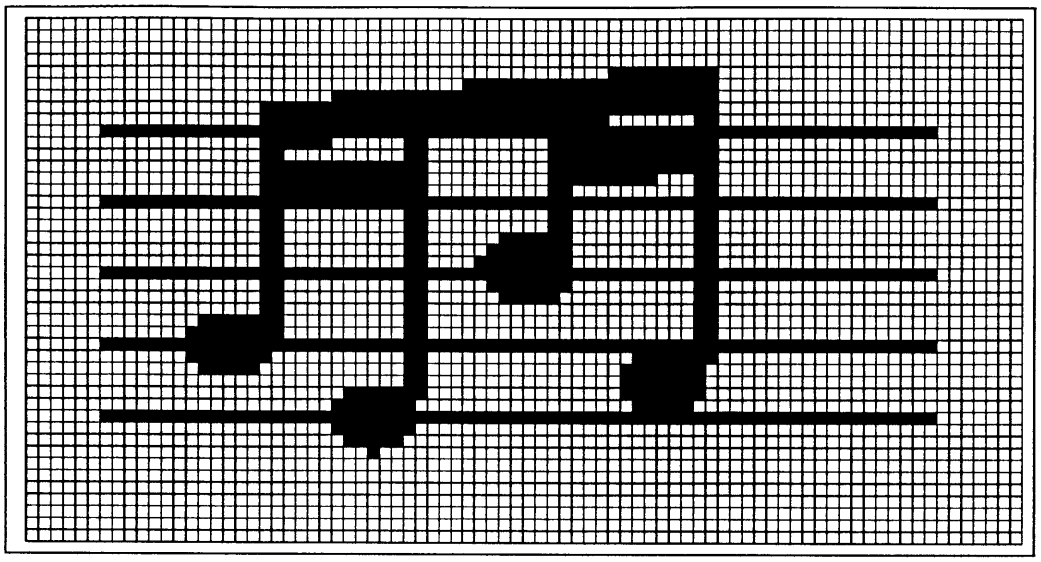


Figure 4-3: A closeup of a digitized beam. In order to illustrate the tranformations more clearly, this beam is not from the score, but instead a more coarsely digitized example.

of beam parts and their surrounding. Looked at with a squinted eye, beams look like large ink blobs, with other thin ink streaks running in and out of them. The strategy for disconnecting the beams is to erase the thin streaks, without disturbing the beam shape.

There are a couple of standard image processing methods for doing this. One idea for getting rid of the thin lines would be to conceptually erode the image at the edges, just to the point where the thin lines disappear (Figure 4-4. The problem with this is that in an erosion, the beam changes shape, since its own edges erode as well.

Or we could consider the frequency space representation of the image, and realize that the thin lines are the high frequency component of that image. By using a spatial filter, we might hope to filter out all the thin lines, but not the beam shapes. But this method suffers from the same problem that the erosion method has; low pass filtering will remove important detail from the actual beam shapes as well.

Instead I use a filtering technique which envisions each pixel in the context of the consecutive attached vertical or horizontal run of which it is part of, and take the measure of its thickness as being the number of pixels in that consecutive run. So a pixel which is at the end of a run has the same measure as a pixel which is in the middle of a run, shown in Figure 4-5. By removing pixels which have a small measure, thin sections are removed without at all changing

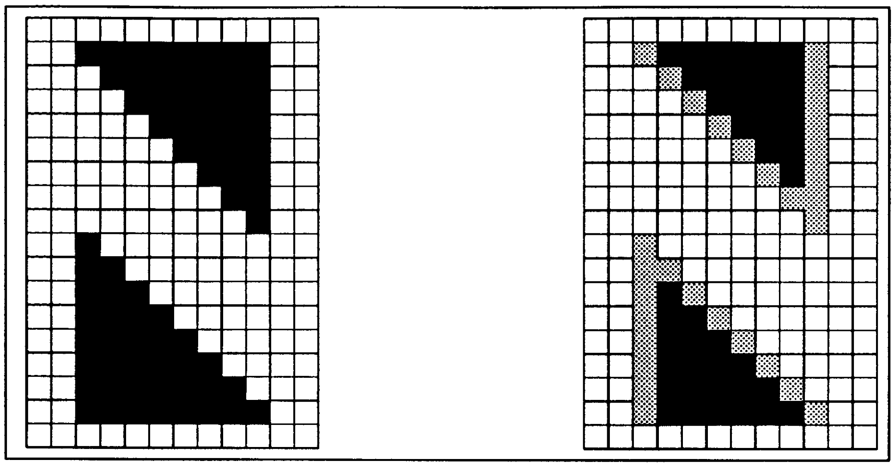


Figure 4-4: An unsuitable filtering technique: In erosion each pixel which is adjacent to a white pixel also becomes white.

thicker sections.

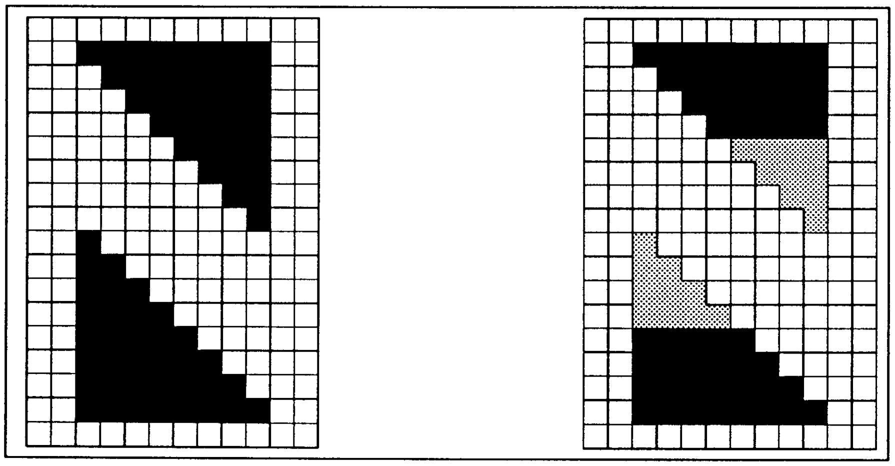


Figure 4-5: The chosen filtering method, neighbor counting: Any pixel which has less than three horizontal neighbors in a row turns white.

The disconnection of beams from their surroundings is a matter executing this filter twice, first with a vertical measure(Figure 4-6), and then with a horizontal one(Figure 4-7). As a side effect this filter also removes small objects of all sorts from the image. After this process, the only marks visible on the page are the beams, remnants of solid note heads, and remnants of treble clefs.

The remnants do not cause a problem because they are easily distinguished from beams.

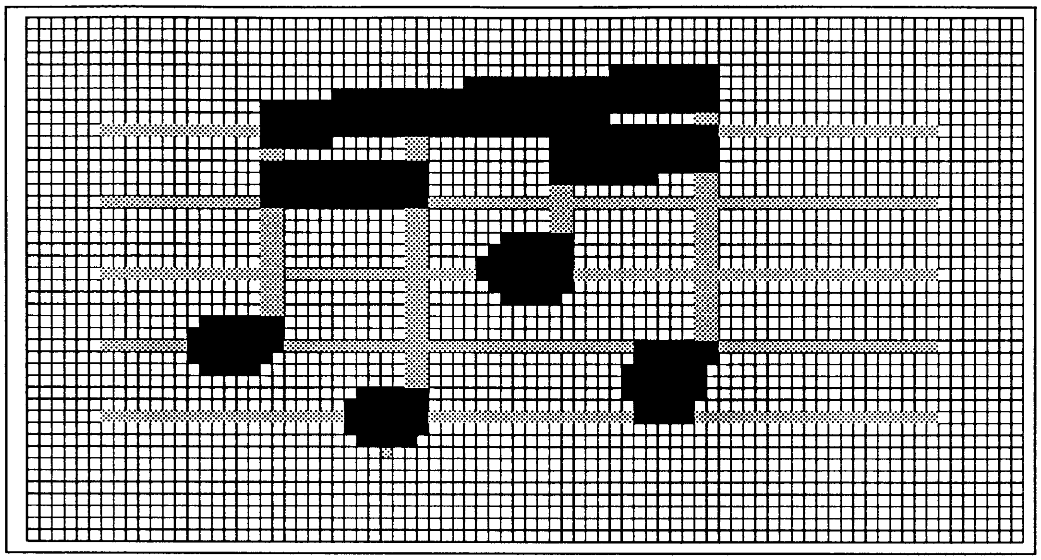


Figure 4-6: Beam after the vertical neighbor counting is done.

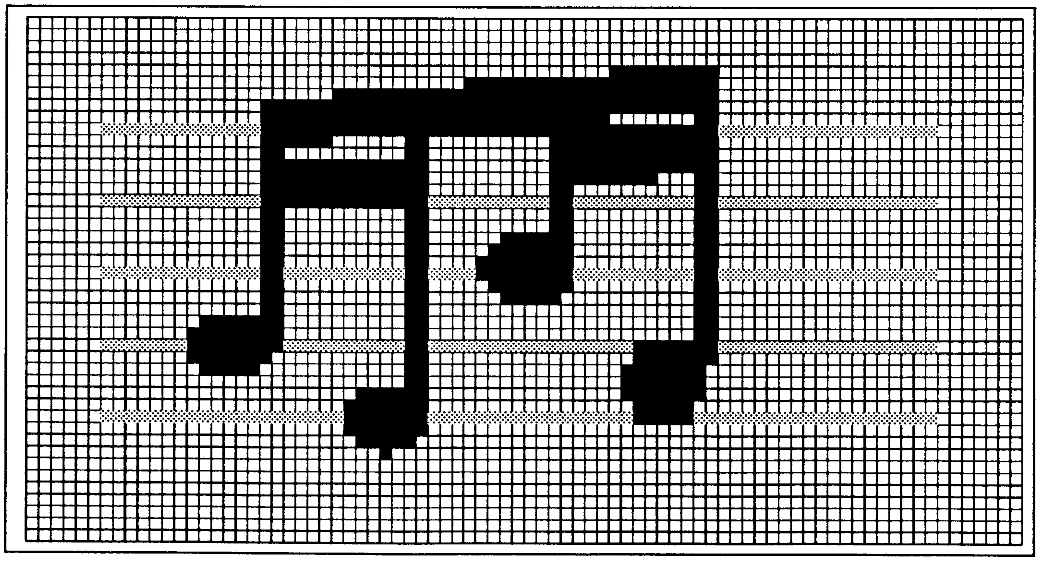


Figure 4-7: Beam after the horizontal neighbor counting is done. At this point different parts of the tuplet are disconnected.

Clefs are always in a position where beams are not possible, and there are not notes above or below a clef. Notes are easily removed because note heads are always shorter than beams, since a beam covers at least two notes.

As printed, the individual ligatures are often visible in the beam. Since we want each beam object to be a single connected region, we need to do something to reconnect these segments. In this case, we can tolerate a small distortion of the beam. In order to reconnect the segments, each pixels blackens its white neighbors, a process called bleeding (see Figure 4-8). In this way, after a few iterations of this process the thin white space between ligatures is filled in, with the slight cost of broadening the beam slightly (Figure 4-9).

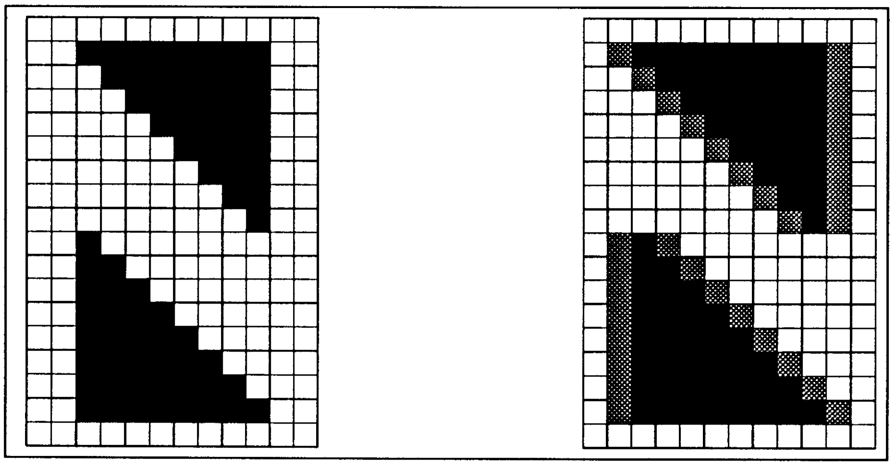


Figure 4-8: In bleeding each pixel which is adjacent to a black pixel also becomes black.

What remains is to calculate the edges(see Figure 4-10). The reason edges are calculated at all is that the parameters which we want are easily expressable as functions on the edge points, and any function on edges will be faster than functions on the whole area. Think of the edge points ordered by their x coordinate. Then for each x coordinate, we have at least two edge points - the top and the bottom. The set of points with maximal y for each x are the points on the top edge. The points with minimal y for each x are the bottom edge points, and height in pixels is the difference between these two, for each x.

Figure 4-11 illustrates the complete sequence. From the top are illustrated: a) The original score, b) the score after horizontal neighbor counting, c) the score after vertical neighbor counting, d) the score after the bleeding is done, and e) the edges of beams (before remnants

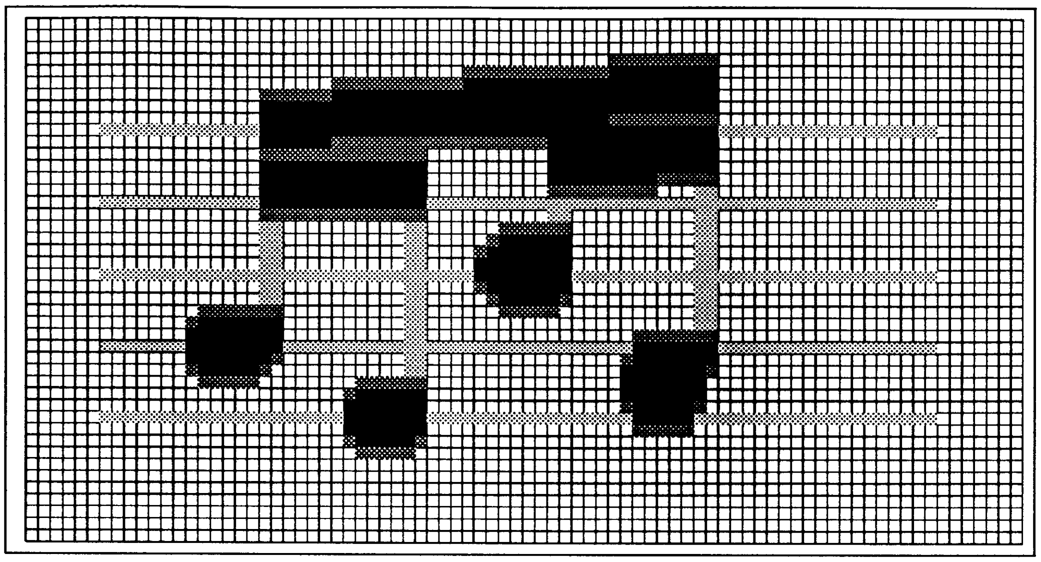


Figure 4-9: Beam after the bleed is applied. Note heads are still visible, but the beam region has been simplified.

are removed).

The details of the computation of the edges will be explained in later sections along with the code. The algorithm I use is adapted from a parallel region labelling based on that described originated by Willie Lim of Thinking Machines. Roughly, a linked list of the edge points is computed, then the points are identified as being part of the same edge by a marker spreading algorithm. The beauty of the algorithm is that it is very parallel. Creating the linked list (on a parallel machine), takes a constant number of steps, with the number of processors on the order of the number of points contained in the regions. The marker propagation step takes order of log the length of the longest border, with as many processors as there are edge points.

Last of all the flat edge of the beam must be determined. To calculate this, the boundary points are sorted by their x coordinate. For each x coordinate, the maximum and minimum y coordinate is calculated. Then a line is fit to each of these sets. The set with the line that has the better goodness of fit (in χ^2 sense) is considered the flat edge.

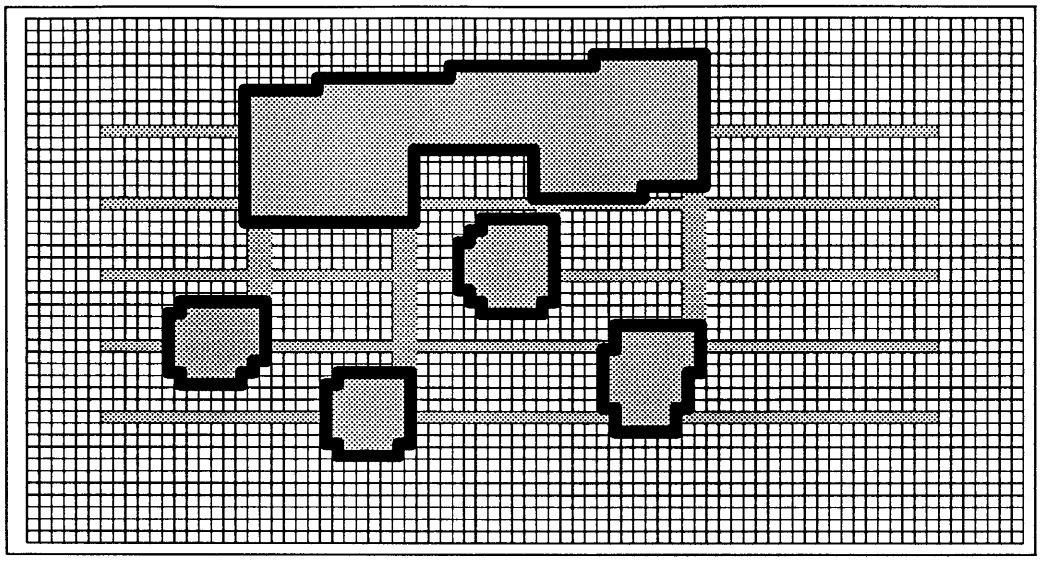


Figure 4-10: Edges of remaining areas.

4.4 Future Work

The main weakness of this method of finding beams is its dependence on several variables which express the scale of the image - numbers which measure how thin something can be without being a beam and the thickness of a single ligature. I think that a method could be designed which computed some scale statistics on a new image in order to automatically calibrate these sizes.

Other than this dependence, I think that this approach should be transportable to other music scripts. Of course the challenge to undertake would be recognition of beams in handwritten scores. In handwritten scores the width of beams is somewhat more variable, and the beams themselves are often curved somewhat in the haste of writing. Still, I think that some of the ideas here might be a help, particularly the methods used to disconnect beams from their surroundings.

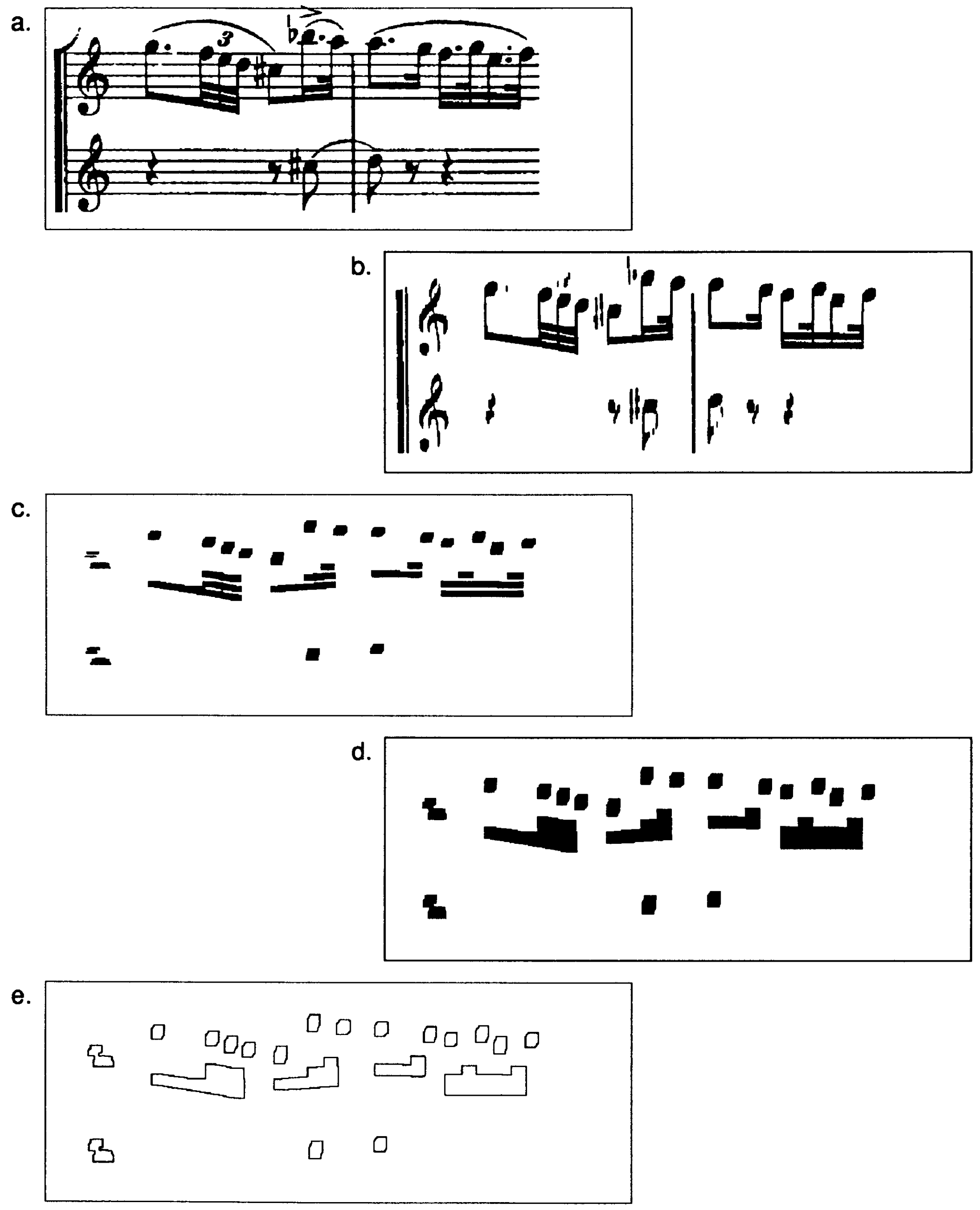


Figure 4-11: This sequence show how the successive beam filters transform the original digitized image. The image is from the top left corner of page 216 of the Schubert scores.

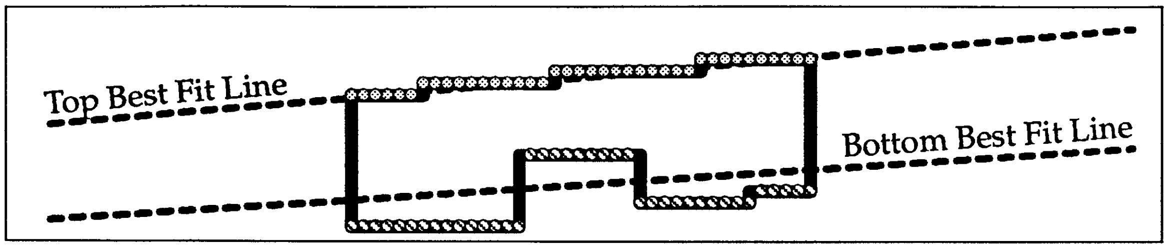


Figure 4-12: Best lines through the points constituting the top and bottom edges of the beam.

4.5 The Code

4.5.1 Top level

After writing the code for chains, I got frustrated with ***defstruct**, so I wrote an analog of **defflavor** for ***lisp**, called ***defflavor**. ***flavors** act like regular **flavors**, using ***defmethod** to defined methods. One extension to the flavors analogy is ***leti**, which allocates a ***instance** with dynamic extent.

```
(defun find-beams (&optional (cm-image *last-image-to-cm*) &aux (boundary-points 0))
  (*warm-boot)
  (unless cm-image (dv-image-to-cm 2))
  (*with-all-vp-set-of cm-image
    (*let ((filtered-image cm-image))
      (loop-over-slices (s filtered-image real-s)
        (beam-filter-slice real-s)
        (incf boundary-points (*when real-s (count-boundary-points real-s))))
      (*leti ((seams boundaries-seams-for-slices
                     :cm-image filtered-image
                     :total-bdry-points boundary-points))
        (*leti ((cb beams-compressed-boundary
                    :picture-vp-set *current-vp-set*
                    :total-bdry-points boundary-points :seams seams))
          (loop-over-slices (s filtered-image)
            (*leti ((bp beam-boundaries :seams seams))
              (new-slice seams column-start row-start column-end row-end)
              (compute-boundaries bp s)
              (compress-boundary bp cb)
              (for-visual-types (show-boundaries cb (send cb :next)))
              ))
          (*with-all-vp-set (send cb :vp-set)
            (spread-label cb)
            (*leti ((scan-bdry beams-scan-boundary :picture-vp-set *osr-image-set* :seams seams
                               :vpset (send cb :vp-set)
                               :total-bdry-points (send cb :total-bdry-points)))
              (initialize-from-compressed-boundary scan-bdry cb)
              (blank-out-regions-with-width-less-than scan-bdry *beam-width-minimum*)
              (sort-by-x scan-bdry)
              (for-visual-types (show-boundaries scan-bdry))
              (let ((the-beams (extract-beams scan-bdry)))
                (map nil 'clean-up the-beams)
                (make-instance 'page-of-beams :image (pvar-getf cm-image :image) :beams the-beams))
              )))))))
```

4.5.2 Shape Parameters

```
(defvar *beam-basic-height* 15 "height in pixels of one beam width")


(defvar *beam-width-minimum* 25)
```

4.6 Filtering Code

4.6.1 Computing Neighbor Counts

```
(defmacro with-neighbor-counts ((slice vrange hrange) &body body)
  `(let ((h-length (1+ (integer-length ,hrange)))
         (v-length (1+ (integer-length ,vrange))))
     (funcall-with-neighbor-counts
       ,slice ,vrange ,hrange
       (λ (above-count below-count left-count right-count)
         (*locally-type (:fast ((above-count below-count) v-length) ((left-count right-count) h-length))
           (ignore above-count below-count left-count right-count)
           ,@body)))))

(defun funcall-with-neighbor-counts (slice vrange hrange function
                                     &aux (h-length (1+ (integer-length hrange)))
                                     (v-length (1+ (integer-length vrange))))
  (*locally-type (:fast (slice t))
    (*let ((from-above slice)
           (from-below slice)
           (from-left slice)
           (from-right slice)
           (still-connected-above? slice)
           (still-connected-below? slice)
           (still-connected-left? slice)
           (still-connected-right? slice)
           (above-count (!! 0))
           (below-count (!! 0))
           (left-count (!! 0))
           (right-count (!! 0)))
      (*type ((save-slice from-above from-below still-connected-above? still-connected-below?) t)
             ((above-count below-count) v-length)
             ((from-left from-right still-connected-left? still-connected-right?) t)
             ((left-count right-count) h-length))
      (loop for i below (1+ vrange) do
        (*set from-above (above!! from-above)
              still-connected-above? (and!! still-connected-above? from-above)
              from-below (below!! from-below)
              still-connected-below? (and!! still-connected-below? from-below))
        (*when still-connected-above? (*set above-count (1+!! above-count)))
        (*when still-connected-below? (*set below-count (1+!! below-count))))
      (loop for i below (1+ hrange) do
        (*set from-left (left!! from-left)
              still-connected-left? (and!! still-connected-left? from-left)
              from-right (right!! from-right)
              still-connected-right? (and!! still-connected-right? from-right))
        (*when still-connected-left? (*set left-count (1+!! left-count)))
        (*when still-connected-right? (*set right-count (1+!! right-count))))
      (funcall function above-count below-count left-count right-count))))
```

4.6.2 Bleeding

```
(defun bleed-vertical  (slice &optional (radius 1))
  (*locally (*type :fast (slice t))
    (*let ((movable slice))
      (loop for r below radius
            do
        (*set slice (or!! (news-border!! slice nil!! 0 1) slice))
        (*set slice (or!! (news-border!! slice nil!! 0 -1) slice)))
      (and moviep (image-to-fb slice)))))

(defun bleed-horizontal  (slice &optional (radius 1))
  (*locally (*type :fast (slice t))
    (*let ((movable slice))
      (loop for r below radius
            do
        (*set slice (or!! (news-border!! slice nil!! 1 0) slice))
        (*set slice (or!! (news-border!! slice nil!! -1 0) slice)))
      (and moviep (image-to-fb slice)))))
```

4.6.3 Filter for Beams

Aim for multiple tiered beams to be one region

```
(defun filter-out-other-than-beams (slice &key (min-height 7) (min-width 13) (bleed 4))
  (*locally-type (:fast (slice t))
    (for-visual-types (image-to-fb slice))
    (with-neighbor-counts (slice min-height 0)
      (*set slice (and!! slice (>!! (+!! (!! 1) above-count below-count) (!!i min-height)))))
    (for-visual-types (image-to-fb slice))
    (with-neighbor-counts (slice 0 min-width)
      (*set slice (and!! slice (>!! (+!! (!! 1) left-count right-count) (!!i min-width)))))
    (for-visual-types (image-to-fb slice))
    (bleed-vertical slice bleed)
    (for-visual-types (image-to-fb slice))))
```

4.6.4 Handle Edges

We don't want spurious connected regions on the wrapparound, so clobber a couple of pixels around the edges.

```
(defun interrupt-regions-at-edges (slice &optional (edge-width 2))
  (*locally-type (:fast (slice t))
    (*all
      (*with-x-and-y (x y)
      (*when (or!! (<!! x (!!i edge-width)) (<!! y (!!i edge-width))
                   (>!! x (!!i (- (car (vp-set-dimensions *current-vp-set*)) edge-width)))
                   (>!! y (!!i (- (second (vp-set-dimensions *current-vp-set*)) edge-width))))
        (*set slice nil!!))))))
```

4.6.5 Prepare a Single Slice

```
(defun beam-filter-slice (s)
  (for-visual-types (image-to-fb s))
  (filter-out-other-than-beams s)
  (interrupt-regions-at-edges s)
  (for-visual-types (image-to-fb s)))
```

4.7 Boundaries Code

4.7.1 Boundaries Data Structure

Adapted from

```
(*defflavor boundaries (processor used labels (vp-set *current-vp-set*)
                                   (pointer-length (1- (integer-length (apply '* (vp-set-dimensions vp-set))))) total-
        ()
   (:pvars (processor (@ 4 pointer-length)) (used (@ 4 t)) (labels (@ 4  pointer-length)))
   (:other-declarations (destination pointer-length) (slice t) :fast)
   (:initable-instance-variables vp-set)
   :gettable-instance-variables
   :settable-instance-variables)

(defgeneric show-boundaries (boundary &optional display-variable)
  (:documentation "Put a representation of the boundary on the Framebuffer.
If display-variable is specified, the image is colored with that value"))

(*defmethod (show-boundaries boundaries) (&optional display-variable &aux!! slice)
  (*all
    (*set slice nil!!)
    (dotimes (i 4)
      (*set slice (or!! (aref!! used (!!i i)) slice)))
    (if display-variable
      (rainbow-scale-to-fb (if!! slice display-variable (!! 0)))
      (image-to-fb slice))))

(*defmethod (compute-boundaries boundaries) (slice)
  (compute-boundary-points self slice)
  (compute-processor-labels self)
  (compute-boundary-neighbors self slice)
  (for-visual-types (show-boundaries self)))
```

4.7.2 Define Eight Connectivity

```
(defvar *boundary-neighbors*
      (apply 'vector '((0 . -1) (-1 . -1) (-1 . 0) (-1 . 1) (0 . 1) (1 . 1) (1 . 0) (1 . -1))))

(defun boundary-dx (direction)
  (car (aref *boundary-neighbors* direction)))

(defun boundary-dy (direction)
  (cdr (aref *boundary-neighbors* direction)))
```

4.7.3 Connect Boundary Points

```
(defun count-boundary-points (slice &aux (count 0))
  (macrolet ((&2 (n) `(mod (+ 2 ,n) 8))
             (&1 (n) `(mod (+ 1 ,n) 8)))
    (*locally-type ((slice t))
      (loop for neighbor in '(0 2 4 6) do
        (*if-not-same-color slice neighbor
          (incf count (*sum (!! 1)))
          (*if-not-same-color slice (&1 neighbor)
            (*if-same-color slice (&2 neighbor)
              (incf count (*sum (!! 1)))))))
      count)))

(*defmethod (compute-boundary-points boundaries) (slice)
  ;(*all (*fill used nil!!))
  (*all (*set (cast!! used 4) (!! 0)))
  (loop for neighbor in '(0 2 4 6) do
    (*if-not-same-color slice neighbor
      (*setf (aref!! used (!!i (/ neighbor 2))) t!!)
      (*if-not-same-color slice (&1 neighbor)
        (*if-same-color slice (&2 neighbor)
          (*setf (aref!! used (!!i (/ (&2 neighbor) 2))) t!!))))))

(*defmethod (compute-boundary-neighbors boundaries) (slice)
  (loop for neighbor in '(0 2 4 6) do
    (*if-not-same-color slice neighbor
      (*if-same-color slice (&2 neighbor)
        (set-pointer self neighbor (&2 neighbor) neighbor)
        (set-pointer self neighbor 'self (&2 neighbor)))
      (*if-not-same-color slice (&1 neighbor)
        (*if-same-color slice (&2 neighbor)
          (set-pointer self (&2 neighbor) (&2 neighbor) neighbor))))))
```

4.7.4 Set Pointer used in bf compute-boundary-neighbors

```
(*defmethod (set-pointer boundaries) (direction-pointer processor-pointed-to direction-in-pointed-to-processor)
  (let* ((direction-pointer (/ direction-pointer 2))
         (direction-in-pointed-to-processor (/ direction-in-pointed-to-processor 2)))
    (if (eq processor-pointed-to 'self)
      (*setf (aref!! processor (!!i direction-pointer))
             (aref!! labels (!!i direction-in-pointed-to-processor)))
      (*setf (aref!! processor (!!i direction-pointer))
             (news!! (aref!! labels (!!i direction-in-pointed-to-processor))
                     (boundary-dx processor-pointed-to)
                     (boundary-dy processor-pointed-to))))))
```

4.7.5 Comparing Colors

The algorithm is designed for used in an image with more than just black and white, hence the plural colors. For our purposes there are only two colors, and in practive we are interested only in the black boundaries.

```
(defun if-same-color-funcall (direction if-continuation else-continuation pvar &optional (same? t))
  (*locally-type (:fast (pvar :field))
    (*let ((them (=!! (news!! pvar (boundary-dx direction) (boundary-dy direction)) pvar)))
      (*type (them t))
      (unless same? (*set them (not!! them)))
      (*if them (funcall if-continuation) (funcall else-continuation)))))
```

```
(defmacro *if-not-same-color (pvar direction if &body else)
  (declare (zwei:indentation 2 2))
  `(if-same-color-funcall ,direction (λ() ,if) (λ() ,@else) ,pvar nil))

(defmacro *if-same-color (pvar direction if &body else)
  (declare (zwei:indentation 2 2))
  `(if-same-color-funcall ,direction (λ() ,if) (λ() ,@else) ,pvar))
```

4.7.6 Accessors for Nearby Pixels

```
(defmacro-in-flavor (&2 boundaries) (n)
  `(mod (+ 2 ,n) 8))

(defmacro-in-flavor (&1 boundaries) (n)
  `(mod (+ 1 ,n) 8))
```

4.7.7 Label each Pointer Uniquely

This needs to be done as precursor to compression. The algorithm creates four pointers in each processor. Since we are going to compress just the active links, each needs to be assigned a consecutive number. The ***lisp** function **enumerate!!** assigns each active processor with a unique number between 0 and the total number of selected processors.

```
(*defmethod (compute-processor-labels boundaries) ()
  ;; compute a new processor label for every used pointer
  (*fill labels (!! 0))
  (loop for neighbor below 4
        with offset = 0 do
    (*when (aref!! used (!!i neighbor))
      (*setf (aref!! labels (!!i neighbor)) (+!! (enumerate!!) (!!i offset)))
      (setq offset (+ offset (*sum (!! 1)))))
        finally (return (setq total-bdry-points offset))))
```

4.8 Compressed Boundaries Code

4.8.1 Definition of Compressed Boundary

A compressed boundary is a set of only active boundary pointers.

```
(*defflavor compressed-boundary (id (valid nil!!) next x y picture-vp-set
                                   total-bdry-points
                                   (vp-set (vpset-to-fit total-bdry-points))
                                   (pointer-length (ceiling (log total-bdry-points 2)))
                                   (xlength (ceiling (log (first (vp-set-dimensions picture-vp-set)) 2)))
                                   (ylength (ceiling (log (second (vp-set-dimensions picture-vp-set)) 2))))
         ()
  (:pvars (id pointer-length) (next pointer-length) (x xlength) (y ylength) (valid t))
  (:other-declarations (pointer pointer-length) :fast)
  (:initable-instance-variables picture-vp-set vp-set total-bdry-points)
  :gettable-instance-variables
  :settable-instance-variables)
```

```
(*defmethod (show-boundaries compressed-boundary) (&optional display-variable)
  (*let-in-vp-set picture-vp-set ((bdry (!! 0)))
    (*type (bdry %id))
    (*when valid
      (if display-variable
        (*nowarn (*pset :no-collisions display-variable bdry
                        (cube-from-vp-grid-address!! picture-vp-set x y) ))
        (*pset :no-collisions id bdry (cube-from-vp-grid-address!! picture-vp-set x y) )))
    (*with-all-vp-set picture-vp-set
      (rainbow-scale-to-fb  (mod!! bdry (!! 255))))))
```

4.8.2 Make a Compressed Boundary from an Ordinary Boundary

```
(*defmethod (allocate-compressed-boundaries boundaries)
            (&aux smaller-set cb)
  (setq smaller-set (vpset-to-fit total-bdry-points))
  (setq cb (*make-compressed-boundary :picture-vp-set *current-vp-set* :vp-set smaller-set
                                       :total-bdry-points total-bdry-points))
  (*with-all-vp-set smaller-set (allocate-pvars-on-heap cb))
  (make-visible (compressed-boundary cb next x y valid)
     (dotimes (neighbor 4)
       (*when (aref!! used (!!i neighbor))
         (let ((destination (alias!! (aref!! labels (!!i neighbor)))))
           (*pset :no-collisions (aref!! processor (!!i neighbor)) next destination :notify valid)
           (*with-x-and-y (xhere yhere)
             (*pset :no-collisions xhere x destination)
             (*pset :no-collisions yhere y destination))))))
  cb)
```

4.8.3 Pointer Doubling

Label all of the points of a single boundary with single color, by pointer doubling.

```
(*defmethod (spread-label compressed-boundary) (&aux!! pointer (lastid pointer-length))
  (*when valid
    (*set id next)
    (*set pointer next)
    (loop do
      (*set lastid id)
      (*set id  (min!! id (pref!! id pointer :collision-mode :no-collisions)))
      (*pset :no-collisions id id pointer)
      (*set pointer (pref!! pointer pointer :collision-mode :no-collisions))
      (for-visual-types (show-boundaries self))
          until (*and (=!! lastid id)))))
```

4.8.4 Scan Ordered Boundaries

Yet another ordering for boundaries, a linear ordering in which points on the same boundary are in contionuous segments. These are useful for computations on complete boundaries, which can used segmented scan operations.

```
(*defflavor scan-ordered-boundary (x y dx dy bdry-start bdry-end (valid nil!!)
                                        picture-vp-set vpset total-bdry-points
                                        (cube-length (cmi::vp-set-address-length vpset))
                                        (xlength (ceiling (log (first (vp-set-dimensions picture-vp-set)) 2)))
                                        (ylength (ceiling (log (second (vp-set-dimensions picture-vp-set)) 2))))
         ()
  (:pvars (x xlength) (y ylength) (dx -2) (dy -2) (bdry-start t) (bdry-end t) (valid t))
  (:other-declarations (ymin ylength) (ymax ylength) (commonx xlength) (commony ylength) (slope ^) (intercept ^)
  (:initable-instance-variables vpset picture-vp-set)
  :gettable-instance-variables
  :settable-instance-variables)

(*defmethod (show-boundaries scan-ordered-boundary) (&optional display-variable &aux!! (id :cube))
  (*all
    (if display-variable
      (*nowarn (*set id display-variable))
      (*set id (scan!! (if!! bdry-start (!! 1) (!! 0)) '+!! :dimension 0)))
    (*when valid
      (*let-in-vp-set picture-vp-set ((bdry (!! 0)) (slice nil!!))
        (*type (bdry %id) (slice t))
        (*with-all-vp-set-of bdry (*set bdry (!! 0)))
        (*pset :no-collisions id bdry (cube-from-vp-grid-address!! picture-vp-set x y) :notify slice)
        (*with-all-vp-set picture-vp-set
          (rainbow-scale-to-fb (mod!! bdry (!! 256)) 256))))))
```

4.8.5 Make a Compressed Boundary into a Scan Ordered Boundary

Compute a mapping between compressed-boundary and scan ordered boundaries. Once each compressed boundary is labelled, a **rank!!** computes the mapping. This is because of the fact that if two nodes have the same id they are guaranteed to be contiguous after a rank, since no larger or smaller node can intercede.

```
(*defmethod (initialize-from-compressed-boundary scan-ordered-boundary) (cb &aux!! (mapaddress :cube))
  (set-x-y-dx-dy cb x y dx dy)
  (compute-map-to-scan-ordered cb mapaddress bdry-start bdry-end)
  (*when (the boolean-pvar (send cb :valid))
    (*bitcat-let (bunch (x y dx dy))
      (*pset :no-collisions bunch bunch mapaddress)
      (*all (*extract-x x) (*extract-y y) (*extract-dx dx) (*extract-dy dy))))
  (setq total-bdry-points (*when (the boolean-pvar (send cb :valid)) (*sum (!! 1))))
  (*when (<!! (self-address-grid!! (!! 0)) (!!i total-bdry-points)) (*set valid t!!))
  (for-visual-types (show-boundaries self)))

(*defmethod (compute-map-to-scan-ordered compressed-boundary)
            (mapaddress boundary-start boundary-end &aux!! (mappedid :cube))
  (*type (mapaddress :cube) ((boundary-end boundary-start) t) :safe)
  (*set mappedid (!! 0))
  (*when valid
    (*set mapaddress (cube-from-grid-address!! (rank!! id '<=!!)))
    (*pset :no-collisions (1+!! id) mappedid mapaddress))
  (*set boundary-start nil!! boundary-end nil!!)
  (*when (<!! (self-address-grid!! (!! 0)) (!!i total-bdry-points))
    (*set boundary-start (/=!! (previous!! mappedid) mappedid))
    (*set boundary-end (/=!! (next!! mappedid) mappedid))))
```

```
(*defmethod (set-x-y-dx-dy compressed-boundary) (xnew ynew dx dy)
  (*type (xnew :field) (ynew :field) ((dx dy) -2))
  (*when valid
    (*pset :no-collisions x xnew next)
    (*pset :no-collisions y ynew next)
    (*set dx (-!! x xnew) dy (-!! y ynew) xnew x ynew y)))
```

4.8.6 Compute Lines which Best Fit the Top or Bottom of the Beam

```
(*defmethod (sort-by-x scan-ordered-boundary) (x-seg-start x-seg-end &aux!! (rank :cube)
                                                &type ((x-seg-end x-seg-start) t))
  (*when valid
    (*nowarn "rank!! doesn't compile"
             (*set rank (rank!! x '<=!!  :dimension 0 :segment-pvar bdry-start)))
    (for-visual-types (show-boundaries self rank))
    (*bitcat-let (bunch (x y dx dy))
      (*pset :no-collisions bunch bunch
             (cube-from-grid-address!!
               (+!! rank (scan!! (self-address-grid!! (!! 0)) 'copy!! :segment-pvar bdry-start :dimension 0))))
      (*all (*extract-x x) (*extract-y y) (*extract-dx dx) (*extract-dy dy)))
    (*set x-seg-start (or!! bdry-start (/=!! (previous!! x) x)))
    (*set x-seg-end (or!! bdry-end (/=!! (next!! x) x)))))

(*defmethod (compute-and-subtract-common-mode scan-ordered-boundary) (commonx commony)
  (*set commonx (scan!! (scan!! x 'min!! :dimension 0 :segment-pvar bdry-start)
                        'copy!! :dimension 0 :segment-pvar bdry-end :direction :backwards)
        x (-!! x commonx)
        commony (scan!! (scan!! y 'min!! :dimension 0 :segment-pvar bdry-start)
                        'copy!! :dimension 0 :segment-pvar bdry-end :direction :backwards)
        y (-!! y commony)))

(*defmethod (compute-lines scan-ordered-boundary) (linestart  x y slope intercept goodness
                                                  &aux!! (sx ^) (sy ^) (sxy ^)
                                                         (sxx ^) (syy ^) (n :cube))
  (*type :fast ((x y slope intercept goodness) ^) (linestart t))
  (*set sx (scan!! x '+!! :dimension 0 :segment-pvar linestart))
  (*set sy (scan!! y '+!! :dimension 0 :segment-pvar linestart))
  (*set sxy (scan!! (*!! x y) '+!! :dimension 0 :segment-pvar linestart))
  (*set sxx (scan!! (*!! x x) '+!! :dimension 0 :segment-pvar linestart))
  (*set syy (scan!! (*!! y y) '+!! :dimension 0 :segment-pvar linestart))
  (*set n (scan!! (!! 1) '+!! :dimension 0 :segment-pvar linestart))
  (*let (denom)
    (*type (denom ^))
    (*when bdry-end
      (*set denom (-!! (*!! n sxx) (*!! sx sx)))
      (*set slope (/!! (-!! (*!! n sxy) (*!! sx sy)) denom))
      (*set intercept (/!! (-!! (*!! sxx sy) (*!! sx sxy)) denom))))
  ;who knows where i found this formule
  (*set goodness (/!! (+!! (*!! n intercept intercept)
                           (*!! (!! 2) intercept slope sx)
                           (*!! slope slope sxx)
                           (-!! (*!! (!! 2) intercept sy))
                           (-!! (*!! (!! 2) slope sxy))
                           syy) n)))

(*defmethod (line-shift-coordinates scan-ordered-boundary) (commonx commony slope intercept)
  (*incf intercept (+!! commony (-!! (*!! commonx slope)))))
```

4.8.7 Remove Short (invalid) Boundaries

```
(*defmethod (blank-out-regions-with-width-less-than scan-ordered-boundary) (min-width)
  (*when valid
    (*set valid
          (scan!! (>!! (abs!! (-!! (scan!! x 'min!! :dimension 0 :segment-pvar bdry-start)
                                   (scan!! x 'max!! :dimension 0 :segment-pvar bdry-start)))
                       (!!i min-width))
                  'copy!! :dimension 0 :segment-pvar bdry-end :direction :backwards))))
```

4.9 Front End Representation Code

4.9.1 Front End Representation for a Single Beam

This flavor is called **new-beam-feature**, because it was the second attempt at the problem.

```
(defflavor new-beam-feature (left right slope intercept points-upwards heights goodness-of-fit)
        (has-relations has-surroundings util-mixin)
  (:constructor make-beam-feature (slope intercept points-upwards goodness-of-fit))
  :settable-instance-variables
  :gettable-instance-variables)

(defmethod (draw-feature new-beam-feature) (&optional (alu :draw))
  (flet ((draw-segment (&rest poly-points) (draw-polygon poly-points :filled t :alu alu)))
    (loop for (x length rawheight) in heights
          for height = (* rawheight *beam-basic-height* (if points-upwards -1 1))
          for leftside = x
          for rightside = (+ x length)
          for topleft = (to-y self leftside)
          for topright = (to-y self rightside)
          for bottomright = (+ topright height)
          for bottomleft = (+ topleft height)
          if points-upwards
            maximize (max topleft topright) into lower
            and minimize (min bottomright bottomleft) into upper
          else
            maximize (max bottomright bottomleft) into lower
            and minimize (min topleft topright) into upper
          do
      (draw-segment leftside topleft rightside topright
                    rightside (+ topright height) leftside (+ topleft height))
          finally
             (bug-workaround "Noop option to draw rectangle causes erase to printer"
                    (dw:with-output-as-presentation (:object self :type 'sys:expression)
                      (draw-rectangle (1- left) (1- upper) (1+ right) (1- lower) :filled t :alu :noop))
               (unless (typep *standard-output* 'lgp::lgp2-page-buffering-stream)
                 ;; make the sensitive region a box because polygon sensitization is messed up in Genera 7.2
                 (dw:with-output-as-presentation (:object self :type 'sys:expression)
                   (draw-rectangle (1- left) (1- upper) (1+ right) (1- lower) :filled t :alu :noop)))))))

(defmethod (pretty-name new-beam-feature) ()
  "Beam")

(defmethod (to-y new-beam-feature) (x)
  (+ (* x slope) intercept))
```

4.9.2 Clean up the Beam Feature on the Front End

Here, very short ligatures are removed and merged with their neighbors.

```
(defmethod (clean-up new-beam-feature) ()
  (get-rid-of-height-glitches self)
  (compute-left-and-right self))

(defmethod (compute-left-and-right new-beam-feature) ()
  (setq left (caar heights))
  (let ((last (car (last heights))))
    (setq right (+ (first last) (second last)))))

(defmethod (get-rid-of-height-glitches new-beam-feature) (&aux (glitch-threshold 5))
  (loop with carry = 0
        for segment in heights
        for (x length height) = segment
        if (< length glitch-threshold)
          do (incf carry length)
        else
          collect segment into deglitched and
          do (decf (car segment) carry)
             (incf (second segment) carry)
             (setq carry 0)
        finally (setq heights deglitched)))
```

4.9.3 Compute the Number of Beam Ligatures

```
(defmethod (beam-height-at-x= new-beam-feature) (x)
  (if (or (> x right) (< x left))
    0
    (loop for (left length height) in heights
          when (< (- x left) length)
            do (return-from beam-height-at-x= (* height *beam-basic-height*)))))
```

4.9.4 Beam Geometry

For use in establishing relationships.

```
(defmethod (top-and-bottom new-beam-feature) (x)
  (let ((one-limit (to-y self x))
        upper lower)
    (if points-upwards
      (setq lower one-limit upper (- one-limit (beam-height-at-x= self x)))
      (setq upper one-limit lower (+ one-limit (beam-height-at-x= self x))))
    (values upper lower)))

(defmethod (point-in-beam? new-beam-feature) (x y)
  (and (> x left) (< x right)
       (multiple-value-bind (upper lower) (top-and-bottom self x)
           (and (> y upper) (< y lower)))))
```

```
(defmethod (bounding-box new-beam-feature) ()
  (loop for (left-point width) in heights
        with upper and lower
        do (multiple-value-setq (upper lower) (top-and-bottom self left-point))
        minimize upper into top
        maximize lower into bottom
        do (multiple-value-setq (upper lower) (top-and-bottom self (+ left-point width -1)))
        minimize upper into top
        maximize lower into bottom
        finally (return (values top left bottom right))))

(defmethod (place-relative-to new-beam-feature) (thing)
  "thing should accept the :pitch-center point-position message"
  (multiple-value-bind (x y) (point-position thing :pitch-center)
    (multiple-value-bind (top bottom) (top-and-bottom self x)
      (cond ((< y top) :above)
            ((> y bottom) :below)
            (t :overlaps)))))
```

4.9.5 Object to Hold all the Beams on a Page

```
(defflavor page-of-beams (beams image)
           (util-mixin)
  :gettable-instance-variables
  :settable-instance-variables)

(defmethod (draw page-of-beams) (&key (reduce-by 3) (coloring-function (λ(ignore) :draw)))
  (with-room-for-image (image :reduce-by reduce-by
                              :label (format nil "Beams from ~A" (send image :name))
                              :label-object self)
    (map nil (λ(f) (draw-feature f  (funcall coloring-function f))) beams)))
```

4.10 Code for full page of beams

4.10.1 Beam boundaries

```
(*defflavor beam-boundaries () (boundaries-with-seams)
   :initable-instance-variables
   :gettable-instance-variables
   :settable-instance-variables)

(*defflavor beams-compressed-boundary () (compressed-boundary-with-seams))

(*defflavor beams-scan-boundary (x-seg-start x-seg-end seams) (scan-ordered-boundary)
   (:pvars (x-seg-start t) (x-seg-end t))
   :initable-instance-variables
   :settable-instance-variables
   :gettable-instance-variables)

(*defwhopper (allocate-pvars-on-heap beams-compressed-boundary) ()
  (*with-vp-set vp-set (continue-whopper)))

(*defwhopper (allocate-pvars-on-stack beams-compressed-boundary) ()
  (*with-vp-set vp-set (continue-whopper)))
```

```
(*defmethod (show-boundaries beams-scan-boundary) (&optional display-variable &aux!! (id :cube)
                                                                    &aux (reduction (reduction seams)))
  (*all
    (if display-variable
      (*nowarn (*set id display-variable))
      (*set id (scan!! (if!! bdry-start (!! 1) (!! 0)) '+!! :dimension 0)))
    (*when valid
      (*let-in-vp-set picture-vp-set ((bdry (!! 0)))
        (*type (bdry %id))
        (*with-all-vp-set-of bdry (*set bdry (!! 0)))
        (*pset :no-collisions id bdry
              (cube-from-vp-grid-address!! picture-vp-set (floor!! x (!!i reduction))
                                                          (floor!! y (!!i reduction))))
        (*with-all-vp-set picture-vp-set
          (rainbow-scale-to-fb (mod!! bdry (!! 256)) 256))))))
```

4.10.2 Ignore the redundant white border

Modify compute-boundaries, so that only the black border is bothered with

```
(*defwhopper (compute-boundaries beam-boundaries) (slice)
  (*when slice (continue-whopper slice)))
```

4.10.3 Slight modifications for a few methods

```
(defmethod (make-instance beams-scan-boundary :after) (&rest ignore)
  (multiple-value-bind (x-address-extra y-address-extra)
      (extra-address-lengths seams)
    (incf xlength x-address-extra)
    (incf ylength y-address-extra)))

(*defwhopper (sort-by-x beams-scan-boundary) ()
  (continue-whopper x-seg-start x-seg-end))
```

4.10.4 Compute the Height of Beams, in Units of Ligature

```
(*defmethod (compute-height beams-scan-boundary) (ymin ymax beams which-beam
                                                  &optional (basic-height *beam-basic-height*)
                                                  &aux (number-of-beams (length beams))
                                                  &aux!! (height 3) (height-bdry-start t) (height-bdry-en
                                                         (count :cube) &type (which-beam :cube))
  (*set height (round!! (-!! ymax ymin) (!!f basic-height)))
  (*set height (scan!! height 'copy!! :segment-pvar x-seg-end :dimension 0 :direction :backward))
  (*set height-bdry-start (or!! bdry-start (/=!! height (previous!! height))))
  (*set height-bdry-end (or!! bdry-end (/=!! height (next!! height))))
  (*let ((startx (scan!! x 'copy!! :dimension 0 :segment-pvar height-bdry-start)))
    (*type (startx xlength))
    (*set count (-!! x startx))
    (with-temp-array (height-rle number-of-beams)
      (*when height-bdry-end
        (loop for p fixnum being the *processors do
          (push (list (pref startx p) (1+ (pref count p)) (pref height p))
                (aref height-rle (pref which-beam p)))))
      (loop for b being the array-elements of beams
            for i from 0 do (send b :set-heights (sort (aref height-rle i) #'< :key #'car))))))
```

4.10.5 Extract representation from CM

```
(*defmethod (extract-beams beams-scan-boundary) (&aux!! (slope ^) (goodness ^)
                                                        (intercept ^) (points-upward t)
                                                        (startbit t)
                                                        ymin ymax commonx commony
                                                   &aux beams)
  (*when valid
    (compute-and-subtract-common-mode self commonx commony)
    (*let (xp yp)
      (*type ((xp yp) ^))
      ;; make a bit which is t in the first end of a x segment
      (*set startbit (scan!! bdry-start 'copy!! :dimension 0 :segment-pvar x-seg-start))
      (*set xp x points-upward nil!!)
      (*set ymin (scan!! y 'min!! :dimension 0 :segment-pvar x-seg-start) yp ymin)
      (*when x-seg-end        ;compute the top line
        (compute-lines self startbit xp yp slope intercept goodness)
        (line-shift-coordinates self commonx commony slope intercept))
      (*set ymax (scan!! y 'max!! :dimension 0 :segment-pvar x-seg-start) yp ymax)
      (*let (bottomslope bottomintercept bottomgoodness)
        (*type ((bottomslope bottomintercept bottomgoodness) ^))
        (*when x-seg-end  ; compute botton line
          (compute-lines self startbit xp yp bottomslope bottomintercept bottomgoodness)
          (line-shift-coordinates self commonx commony bottomslope bottomintercept))
        ; set slope (the result slope) to the slope with the better goodness of fit
        (*when (<!! bottomgoodness goodness)
          (*set slope bottomslope
                goodness bottomgoodness
                intercept bottomintercept
                points-upward t!!))))
    (*set ymin (+!! commony ymin) ymax (+!! ymax commony) x (+!! x commonx) y (+!! y commony))
    (*let ((which-beam (1-!! (scan!! (cast!! bdry-start 1) '+!! :dimension 0))))
      (*type (which-beam :cube))
;     (*when (=!! which-beam (!!i *beam-to-debug*))
; (print (loop for p being the *processors collect (cons (pref x p) (pref y p)))))
      (*when bdry-end
        (loop for p fixnum being the *processors
              with them = (make-array (1+ (*max which-beam)))
              for bf = (make-beam-feature (pref slope p) (pref intercept p)
                                          (pref points-upward p) (pref goodness p))
              do (setf (aref them (pref which-beam p)) bf)
            finally (setq beams them)))
      (compute-height self ymin ymax beams which-beam))
    beams))
```

4.11 Code to deal with slices

4.11.1 Definition of Seams

Unfortunately we don't have enough memory to do a page at a time. Instead, the boundaries for each slice are computed. Since a beam can straddle a slice boundary, there needs to be a mechanism for hooking up the boundary in one slice to the boundary in the next. To do this, an object called a *seam* is created, which holds the results of the boundary labelling algorithm on the edges of slices. When the boundary points are moved to compressed boundary vpset, the links are restored.

```
(*defflavor boundaries-seams-for-slices (cm-image
                                         (image (pvar-getf cm-image :image))
                                         total-bdry-points
                                         (picture-vp-set (pvar-vp-set cm-image))
                                         (number-vertical-seams
                                              (1- (*array-dimension cm-image 1)))
                                         (number-horizontal-seams
                                              (1- (*array-dimension cm-image 0)))
                                         (total-seams
                                            (+ number-vertical-seams number-horizontal-seams))
                                         (pointer-length
                                            (1- (integer-length
                                                   (apply '* (vp-set-dimensions picture-vp-set)))))
                                         (all-pointers-length (* 4 pointer-length))
                                         the-seams bdry-point? (bdry-points-so-far 0)
                                         (have-seams
                                             (make-array (list number-horizontal-seams
                                                               number-vertical-seams)))
                                         (extra-border (pvar-getf cm-image :extra-border))
                                         (slice-width (first (vp-set-dimensions picture-vp-set)))
                                         (slice-height (second (vp-set-dimensions picture-vp-set)))
                                         (half-border (/ extra-border 2))
                                         top left bottom right left? top? bottom? right?
                                         (image-width (send image :width))
                                         (image-height (send image :height))
                                         (seam-length (max image-width image-height))
                                         (seams-vp-set (vpset-to-fit (* 2 seam-length)))
                                         xslice yslice)
        ()
   (:initable-instance-variables image total-bdry-points)
   (:pvars (the-seams (@ total-seams all-pointers-length)) (bdry-point? (@ total-seams t)))
   (:other-declarations (labels all-pointers-length) ; labels passed from the boundary
    (used 4)
    (parallel-address :field) (perp-address :field)
    (which-seam 2)  ; layout is boundary then overlap seam in ascending cube addresses
    )
   (:gettable-instance-variables)
   (:settable-instance-variables))

(*defwhopper (allocate-pvars-on-stack boundaries-seams-for-slices) (&rest ignore)
  (*with-vp-set seams-vp-set  (continue-whopper) (*all (*fill bdry-point?  nil!!))))

(*defwhopper (allocate-pvars-on-heap boundaries-seams-for-slices) (&rest ignore)
  (*with-vp-set seams-vp-set (continue-whopper) (*all (*fill bdry-point?  nil!!))))

(*defmethod (seam-index boundaries-seams-for-slices) (dimension which)
  (+ (if (plusp dimension) number-horizontal-seams 0) which))

(*defmethod (reduction boundaries-seams-for-slices) ()
  (1+ (max number-vertical-seams number-horizontal-seams)))

(*defmethod (valid-slice-limits boundaries-seams-for-slices) ()
  (let ((xstart (if left? 0 half-border))
        (xlimit  (if right? slice-width (- slice-width half-border)))
        (ystart (if top?  0 half-border))
        (ylimit (if bottom? slice-height (- slice-height half-border ))))
    (values left top xstart ystart xlimit ylimit)))

(*defmethod (extra-address-lengths boundaries-seams-for-slices) ()
  (values (ceiling (log (*array-width cm-image) 2)) (ceiling (log (*array-height cm-image) 2))))
```

4.11.2 Save away the seams

```
(*defmethod (cache-seam boundaries-seams-for-slices) (parallel-address perp-address
                                                       labels used dimension which start
                                                       include-extreme?
                                                      &aux (length (if (zerop dimension)
                                                                       slice-width
                                                                       slice-height))
                                                       (boundary-seam (- length half-border 1))
                                                       (overlap-seam (1+ boundary-seam))
                                                       (seam-index (seam-index self dimension which))
                                                      &aux!! which-seam)
  (*when (or!! (=!! perp-address (!!i overlap-seam)) (=!! perp-address (!!i boundary-seam)))
    (*when (and!! (>=!! parallel-address (!!i (if include-extreme? 0 half-border)))
                  (<=!! parallel-address (!!i (- length half-border))))
      (*set which-seam (-!! perp-address (!!i boundary-seam)))
      (*when (plusp!! used)
        (*let ((address (cube-from-vp-grid-address!!
                          seams-vp-set
                          (+!! parallel-address (!!i start) (*!! which-seam (!!i seam-length))))))
          (*type (address :cube))
          (*pset :no-collisions labels (alias!! (aref!! the-seams (!!i seam-index)))
                 address
                 :notify (alias!! (aref!! bdry-point? (!!i seam-index)))))))))

(*defmethod (cache-seams boundaries-seams-for-slices) (labels used)
  (*with-x-and-y (x y)
    (unless bottom? (cache-seam self x y labels used 0 yslice left left?))
    (unless right? (cache-seam self y x labels used 1 xslice top top?))))
```

4.11.3 Restore the seams

```
(*defmethod (restore-seam boundaries-seams-for-slices) (labels dimension which beginning end
                                                        &aux!! which-seam
                                                        &aux (seam-index
                                                               (seam-index self dimension which)))
  (*with-line-address (line)
    (*when (<!! line (!!i (* 2 seam-length)))
      (*set which-seam (floor!! line (!!i seam-length)))
      (*set line (mod!! line (!!i seam-length)))
      (*when (and!! (>=!! line (!!i beginning)) (<=!! line (!!i end)))
        (*when (aref!! bdry-point? (!!i seam-index))
          (if (zerop dimension)
            (*pset :no-collisions (alias!! (aref!! the-seams (!!i seam-index))) labels
                   (cube-from-vp-grid-address!!
                     picture-vp-set
                     (+!! (-!! line (!!i beginning)))
                     (+!! (!!i (1- half-border)) which-seam)) )
            (*pset :no-collisions (alias!! (aref!! the-seams (!!i seam-index))) labels
                   (cube-from-vp-grid-address!!
                     picture-vp-set
                     (+!! (!!i (1- half-border)) which-seam)
                     (+!! (-!! line (!!i beginning))))))))))))

(*defmethod (restore-seams boundaries-seams-for-slices) (labels)
  (*all
    (unless left? (restore-seam self labels 1 (1- xslice) top (1- (+ top slice-height))))
    (unless top? (restore-seam self labels 0 (1- yslice) left (1- (+ left slice-width))))))
```

4.11.4 Details

```
(*defmethod (new-slice boundaries-seams-for-slices) (xstart ystart xend yend)
  (setq xslice (/ xstart (- slice-width extra-border)))
  (setq yslice (/ ystart (- slice-width extra-border)))
  (setq left xstart right xend top ystart bottom yend)
  (setq left? (zerop xslice))
  (setq top? (zerop yslice))
  (setq right? (>= xend image-width))
  (setq bottom? (>= yend image-height)))
```

```
(*defmethod (subselect-to-protect-seam-labels boundaries-seams-for-slices) (continuation
                                                                             &aux!! (restored-seams t))
  (*all
    (*set restored-seams nil!!)
    (*with-x-and-y (x y)
      (unless top?
        (*set restored-seams
              (or!!
               (=!! y (!!i (1- half-border)))
               (=!! y (!!i half-border)))))
      (unless left?
        (*set restored-seams
              (or!! restored-seams
                    (=!! x (!!i (1- half-border)))
                    (=!! x (!!i half-border))))))
  (*when (not!! restored-seams)
    (funcall continuation)))
```

4.11.5 Boundaries Which also have Seams

```
(*defflavor boundaries-with-seams (seams offset)
          (boundaries)
   :initable-instance-variables
   :gettable-instance-variables
   :settable-instance-variables)
```

```
(*defmethod (compute-boundaries boundaries-with-seams :before) (&rest ignore)
  (let ((*used* self))
    (restore-seams seams labels)))
```

```
(*defmethod (compute-boundaries boundaries-with-seams :after) (&rest ignore)
  (let ((*used* self))
  (cache-seams seams labels used)))
```

4.11.6 More Details

```
(defmethod (more-boundary-points boundaries-seams-for-slices) (howmany)
  (incf bdry-points-so-far howmany))
```

```
(*defwhopper (compute-processor-labels boundaries-with-seams) ()
  (subselect-to-protect-seam-labels seams (λ() (continue-whopper))))
```

```
(*defmethod (compute-processor-labels boundaries-with-seams :after) (&rest ignore
                                                                     &aux
                                                                     (offset
                                                                      (send seams
                                                                        :bdry-points-so-far)))
  (dotimes (i 4)
    (*setf (aref!! labels (!!i i)) (+!! (aref!! labels (!!i i)) (!!i offset))))
  (more-boundary-points seams total-bdry-points))
```

4.11.7 Compressed Boundaries with Seams

```
(*defflavor compressed-boundary-with-seams (seams) (compressed-boundary))

(defmethod (make-instance compressed-boundary-with-seams :after) (&rest ignore)
  (multiple-value-bind (x-address-extra y-address-extra)
      (extra-address-lengths seams)
    (incf xlength x-address-extra)
    (incf ylength y-address-extra)))

(*defmethod (show-boundaries compressed-boundary-with-seams) (&optional display-variable
                                                              &aux (reduction (reduction seams)))
  (*with-all-vp-set vp-set
    (*let-in-vp-set picture-vp-set ((bdry (!! 0)))
      (*type (bdry %id))
      (*all
        (if display-variable
          (*nowarn
            (*pset :no-collisions display-variable bdry
                  (cube-from-vp-grid-address!! picture-vp-set (floor!! x  (!!i reduction))
                                               (floor!! y (!!i reduction)))))
          (*pset :no-collisions id bdry
                (cube-from-vp-grid-address!! picture-vp-set (floor!! x  (!!i reduction))
                                             (floor!! y (!!i reduction)))))
        (*with-all-vp-set picture-vp-set
          (rainbow-scale-to-fb  (mod!! bdry (!! 255))))))))
```

4.11.8 Compress the Boundary taking Seams into Account

Compress boundary is more complicated since we compress from a number of slices, instead of only one.

```
(*defmethod (compress-boundary boundaries-with-seams) (compressed-boundary
                                                       &aux (vps *current-vp-set*))
  (make-visible (compressed-boundary-with-seams compressed-boundary next x y valid vp-set)
    (*with-all-vp-set vp-set
      (*let ((recipients nil!!))
        (*type (recipients t))
        (multiple-value-bind  (xoffset yoffset xstart ystart xlimit ylimit) (valid-slice-limits seams)
          (*with-all-vp-set vps
            (*with-x-and-y (xhere yhere)
              (*when  (and!! (<=!! (!!i xstart) xhere (!!i xlimit)) (<=!! (!!i ystart) yhere (!!i  ylimit)))
                (dotimes (neighbor 4)
                  (*when (aref!! used (!!i neighbor))
                    (let ((destination (alias!! (aref!! labels (!!i neighbor)))))
                      (*pset :no-collisions (aref!! processor (!!i neighbor)) next destination
                             :notify recipients)
                      (*pset :no-collisions (+!! xhere (!!i xoffset)) x destination)
                      (*pset :no-collisions (+!! yhere (!!i yoffset)) y destination))))))))
        (*set valid (or!! valid recipients))))))
```

Chapter 5

Locating Staff and Measure Lines

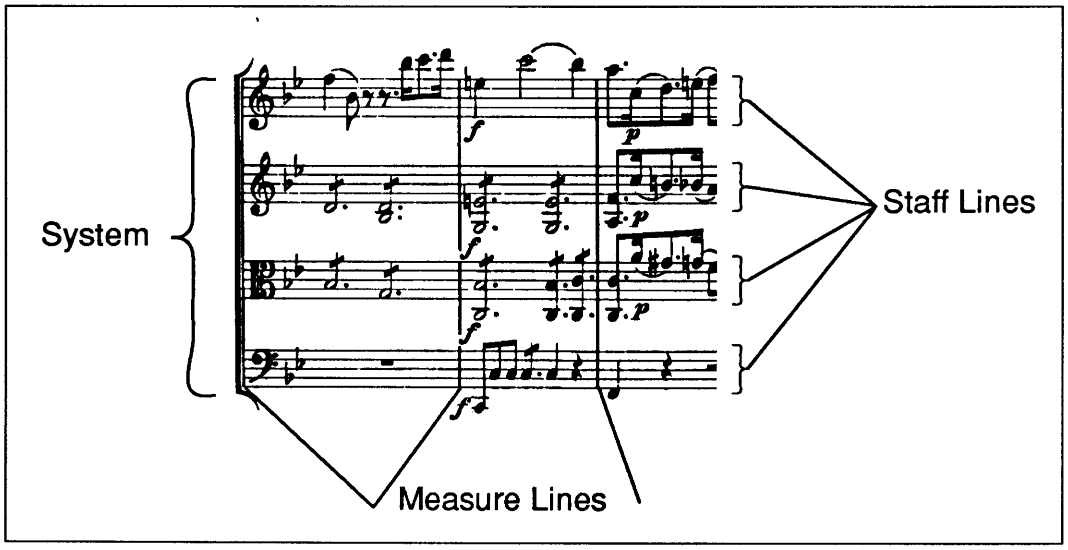


Figure 5-1: Staff and measure lines.

5.1 Introduction

Staff and measure lines impose a grid on a page of music forming a coordinate system by which time and pitch are measured. Of the various features found on a sheet of music, they are the most obviously structured. Each staff line is composed of exactly five horizontal lines which stretch across almost the whole page. Each measure line is a thin vertical line which extends across a group of staffs called a system. There are as many staffs in a system as there are instruments in the orchestration. In the Schubert string music which I worked on that meant between three and five staffs per system.

The lines which compose the staff and measure boundaries are intended to be exactly horizontal and vertical. This observation inspired my first attempts to identify them on the score. Finding staff lines and finding measure lines are very similar. In the rest of this discussion I'll mainly focus on finding staff lines, adding a few remarks at the end on the differences between finding these and measure lines.

5.2 Finding Staff Lines

Professor Sandy Pentland, of the Media Lab's Vision group, pointed out that if one considered the horizontal sum of pixels across each row of the image then most rows would not add up to much, but that lines which were part of a staff line would have a sum which was comparable to the width of the image. The horizontal sums could be viewed as a function of the vertical position on the page. Identifying the staff lines would then be a matter of finding the locations of the peaks of this function, illustrated in Figure 5-2.

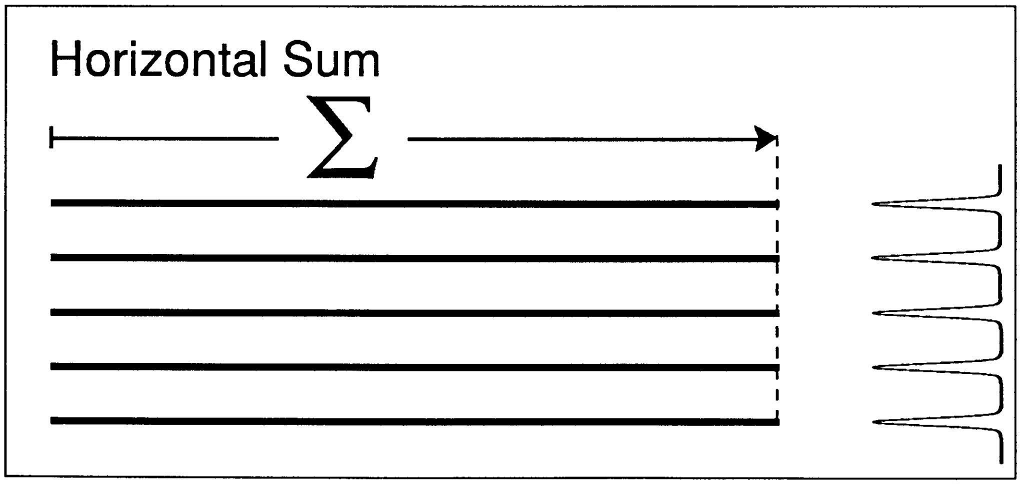


Figure 5-2: The basic way of finding staff lines is to sum pixels horizontally. To the right is a plot of what the sum looks like as a function of vertical position.

Unfortunately lines are not always horizontal. Figure 5-3 shows some typical cases. In the first place, if the image is not perfectly registered when scanned then the lines have a slope.

Think of what happens to the peaks as a line is tilted. When the line is flat at 0° the peak exists at a single point. But at 45° there is absolutely no peak. So any tilting of the page results in a rapid diminution of the peaks. The images were scanned at 300 dots per inch, and since the staff lines are only a few pixels wide, the necessary level of precision in scanning was difficult to maintain.

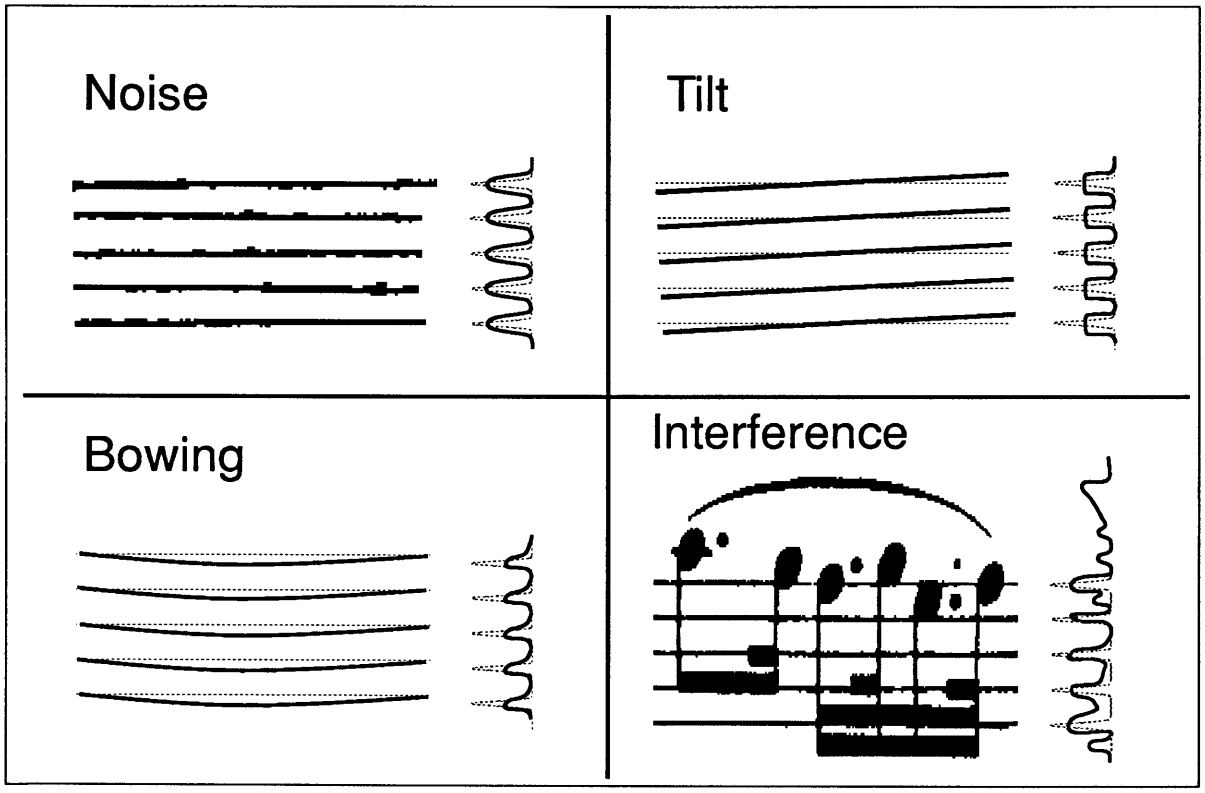


Figure 5-3: Four ways in which digitized printed staff lines differ from the ideal. To the right are the horizontal sums. The ideal sums are in grey.

To make this even worse, imperfections in printing introduced a hardly perceptible bowing of the line which amounted to tens of pixels. This imperfection appeared on the horizontal sum function in the form of peaks which were significantly smaller, and were hence more difficult to positively identify. Moreover, this imperfection in the printing process could not be corrected by any amount of care in placement of the sheet while scanning.

Another problem is that where there is a long tie, beam, or dynamic, or where many beams are at the same height on the page, the horizontal sum of pixels for that row is comparable to the sum produced by a staff line. It is difficult to distinguish between peaks which are due to staff lines and peaks which are due to beams and ties.

5.3 Filter, Divide and Conquer

The work leading to overcoming these problems was done by Bill Jarrold a student who worked with me on this phase of the project. This approach was three pronged: filter the image to reduce extraneous features, divide the page into vertical segments so that within each segment the lines were more consistently flat, and make a final pass over the found lines applying heuristics to remove unlikely lines. The next few paragraphs discuss these methods in more detail.

In order to reduce the influence of ties, beams and other features on the horizontal sums, the image is first filtered to remove image portions which are not part of somewhat longer horizontal runs(see Figure 5-6). The filter acts like a low pass horizontal filter, but is not implemented in frequency space. Instead each point looks in its horizontal neighborhood, counting how many neighbors it has within a certain radius. Points with less than some threshold were ignored in future processing. The subsequent processing is not extremely sensitive to the exact value of the radius, so not too much care was necessary in its choice. Larger radius values result in stronger filters, but take longer to compute. When the radius is too low too much of the image other than staff lines remain. When the radius gets too large, the bowing of the line results in the staff lines being filter out. Between these two extremes there is roughly an order of magnitude to choose from. For the staff lines I used a radius of about one third of an inch, for measures about a tenth of an inch.

The second strategy was to divide the page into a series of vertical segments. The idea is that although the staff lines are bowed, they are less so over smaller lengths. Segment lengths were empirically chosen to be the largest which gave sufficient accuracy so that the algorithm didn't fail in some fifty pages of testing. The segments were about an inch wide for staff lines, and about one fifth of an inch for measure lines.

The larger the segments were, the less likely that some long non staff line feature such as a tie would give a significant peak, but also the more the chance was that bowing of the line would contribute to degradation of the ability to find the peak. As with the filter, there was, despite these constraints, a large range of segment size which gave adequate results. I chose segment sizes which led to a power of two number of segments in order to simplify allocation of processors.

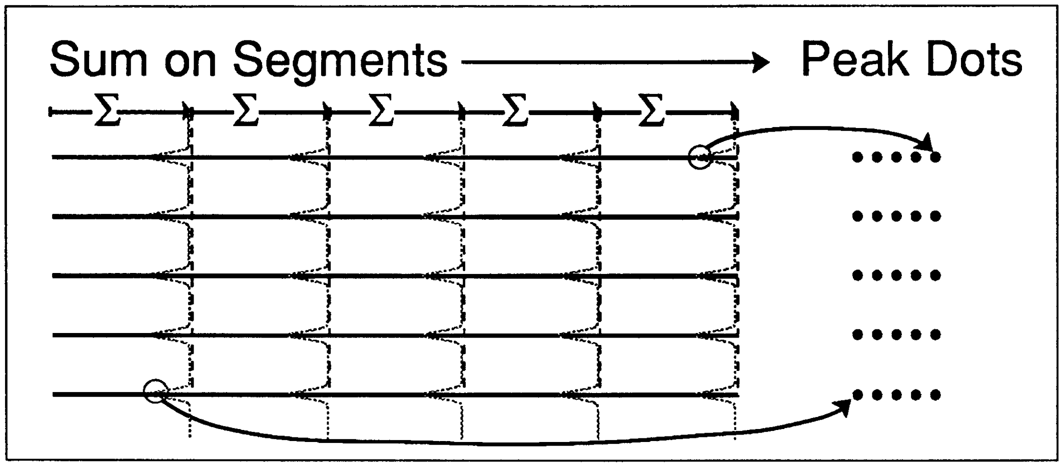


Figure 5-4: The basic way of finding staff lines is to sum pixels horizontally. To the right is a plot of what the sum looks like as a function of vertical position.

This step uses one processor per pixel, with the sums being accomplished by scan operations. With these operations the time to sum is proportional to the log of the width of a page. At the end of each segment, peaks are found. As with the original algorithm, it is possible that a string of beams or a tie would cause a spurious peak. At this stage of processing such fictitious staff line segments are tolerated. These spurious line segments will have a chance to be removed when the line segments are put back together. If they still persist, they are removed using heuristics to suggest which lines are incorrect.

5.4 Reconstructing Lines From Line Segments

What is taken apart, must also be put together. In dividing the page into a number of vertical segments the algorithm finds portions of staff lines, rather than complete staff lines. To reconstruct the full line, I construct a list of segments which might connect to each other.

After the horizontal sum is completed there are relatively few processors which have interesting information in them, specifically those processors at the end of a segment in which a peak was found. Imagine, then, removing all the processors which are allocated to points not at the end of a segment. In this new topology peaks which are near other peaks represent line segments which end near the same vertical position. I call this the *dot representation* (see Figure 5-5).

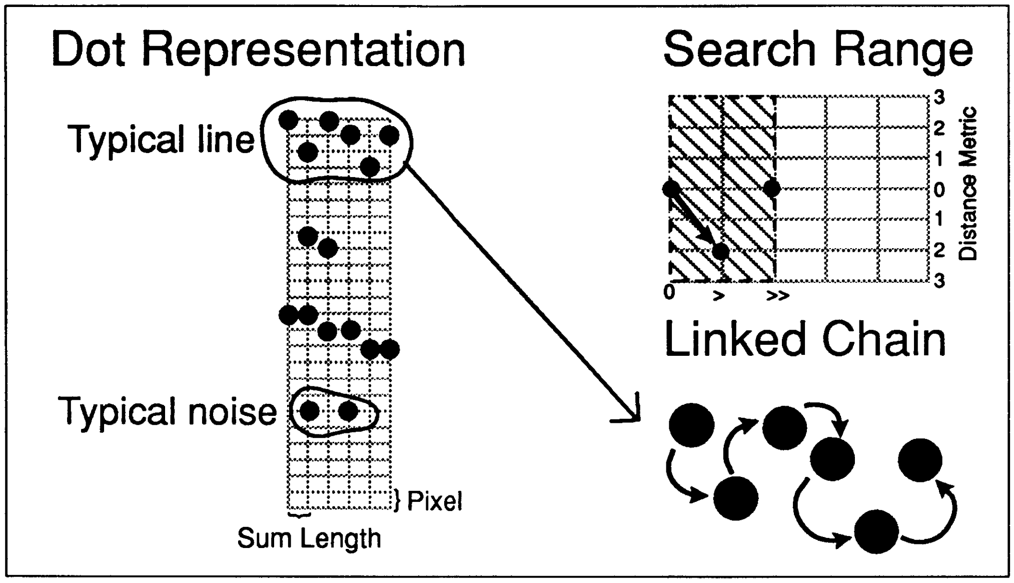


Figure 5-5: The dot representation is transformed to a chain by having each point search for another which is nearby.

Each dot searches for another dot in a small neighborhood to its right. If a dot is found then a link is constructed. Once all the links have been made[1] a pointer doubling algorithm is used to uniquely label each chain of dots, i.e. line segments[2]. Finally, a representation of the chain is extracted from the connection machine. In this extraction, chains with a single member are excluded, since they are obviously too short.

5.5 Checking and Correcting Errors

Despite the removal of the smallest line segments, occasionally a longer chain is mistaken for a staff line. In order to remove these the program repeatedly checks the set of potential staff lines seeing if they make sense. A set of staff lines makes sense if there are a multiple of five of

[1]All processors search in parallel, so the time is proportional to the search area.
[2]This takes time logarithmic in the number of segments.

them, and if each appears to be part of a group. A line is judged to be part of a group if its vertical distance to one of its neighbors is almost the same as the mode distance between lines.

If the set of staff lines is judged not to make sense, then a candidate for removal is computed. A line is most likely to be mistaken if it is short, less than one third the average line length[3]. If no such line is found, then a line which is relatively shorter than both the lines surrounding it is searched for. These heuristics have never failed.

5.6 Grouping Staffs into Systems

Systems are groups of several staffs, one staff for each of the instruments in the orchestration. The program does its best to determine how many staffs are to be grouped into a system using the heuristic that the spacing between staffs within a system is usually less than the spacing between systems. This heuristic only sometimes works, so when the program can't rely on this it asks me how many staffs there are in a system. Unfortunately this

still happens regularly. Since the number of staffs per system is the same over a complete piece it is no chore to tell the program once and for all what the answer is, and so I decided to put no more work in making this part of the program more accurate.

5.7 Finding Measure Lines

Very little changes in finding measure lines. The choice of several of the length scales changes as does the direction of application of the algorithm. Filtering tries to preserve long vertical things. The filter radius is reduced, as measure lines are shorter than staff lines. The segment size is likewise reduced so that the chances of a line only partly overlying a segment are lessened.

The measure locater uses a couple of heuristics, one of which is applied in this phase of computation and one later. Measure lines span a single system. Vertical lines which are found to lie substantially outside a system are removed from consideration as measure lines.

To test whether measures make sense the number of notes in the measure can be added up and should equal the amount in the time signature. Later this constraint will be used to help rule out incorrect measure lines.

[3]The algorithm is not very sensitive to the exact value.

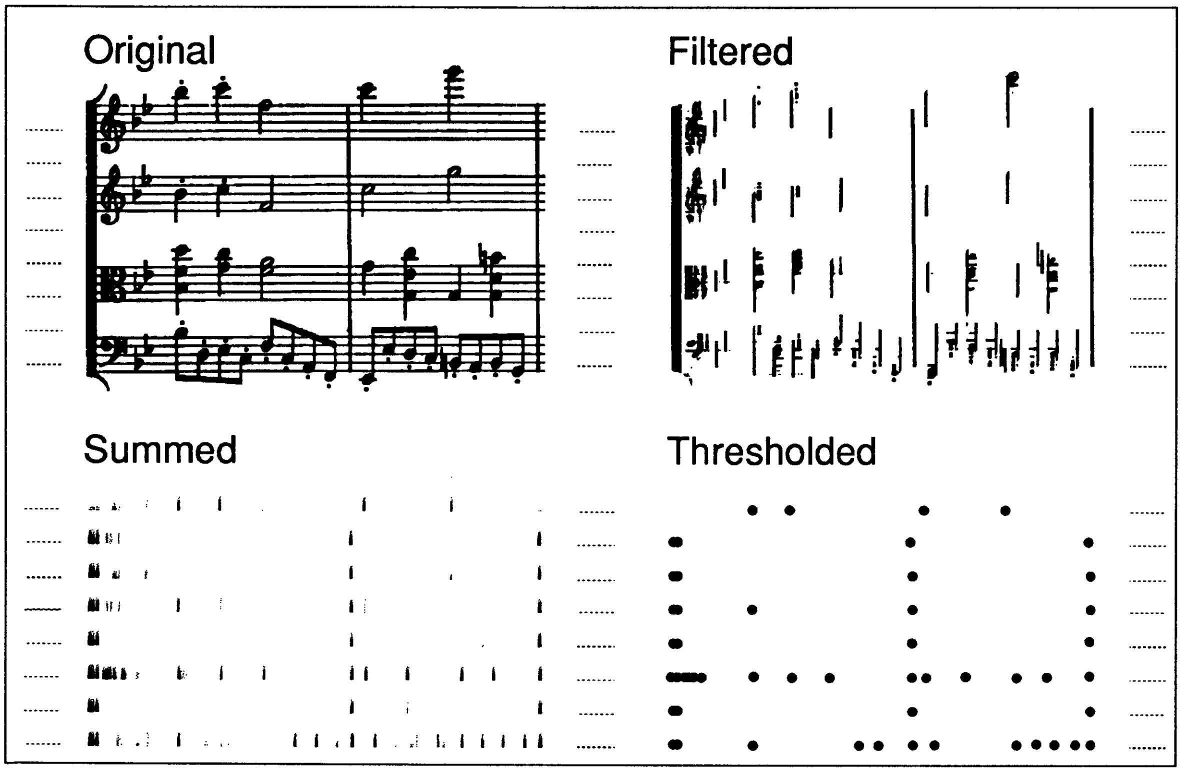


Figure 5-6: This sequence shows the process of finding measure lines. The image is filtered to remove other than thin vertical things. Then it is divided into horizontal slices, each of which is summed downward. The image shows the accumulating sum where the darker it is, the larger the sum. Finally these sums are thresholded yielding a series of points representing line segments.

5.8 Robustness

There are several choices of parameters in the staff finding algorithm: the number of sections in a division, the length over which filters are to operate, the threshold for considering peaks, the length ratios for choosing most likely wrong staff lines. In all cases the parameters have the property that changing them slightly has little effect on the performance of the algorithm. Consider this in the light of the attempts with the first(naive) algorithm. Experimenting with that method showed it was *extremely* sensitive to parameter changes. It is this insensitivity to parameter shifts which I believe gives robustness to the modified version of the algorithm.

5.9 Future Work

Several parameters are hand tuned. It would be better to have the program derive these parameters. Also, people can read a fairly tilted sheet of music and my program can't. This discrepancy should be fixed.

The tilt problem can be handled in a general way, I think. First the orientation of the page needs to be determined. This can be done in a variety of ways, using a Radon, Hough, or Fourier transform. Then the image can be rotated so that the lines are relatively horizontal. At this point the current algorithm can be used.

5.10 The Code

5.10.1 Parameters

```
(defvar *long-emphasis* 100 "was originally 70, but 5% would fail. at 100 none fail")

(defvar *vertical-slice-width* 256)

(defvar *measure-emphasis* 32)

(defvar *measure-squelch* .5)

(defvar *slack-staff-line-spacing-percentage* .30)

(defvar *double-measure-line-threshold* (* (send (find-feature "solid") :width) 2)
  "closer than this and they are considered a double bar line")
```

5.10.2 Top level construction of Bars on a Page

```
(defun grid-page ()
  (let* ((measure-lines (find-measure-lines))
         (staff-lines (find-staff-lines))
         (pb (make-instance 'page-of-bars :staff-lines staff-lines :measure-lines measure-lines)))
    (verify-and-fix-if-necessary staff-lines)
    (split-staffs staff-lines)
    (make-systems staff-lines)
    (segment pb)
    pb))

(defmethod (segment page-of-bars) (&aux delete-these)
  (clear-measure-lines staff-lines)
  (map-measure-lines measure-lines
                     (λ (which top bottom position)
                        (unless
                          (some (λ (system)
                                  (grab-potential-measure-line system top bottom position))
                                (send staff-lines :systems))
                          (push which delete-these))))
  (delete-lines measure-lines delete-these)
  (mapc 'ignore-double-staff-lines (send staff-lines :systems)))
```

5.10.3 Top Level for Finding Staff Lines

```
(defun find-staff-lines (&key (cm-image *last-image-to-cm*)
                              (cutoff .8) (range 8)
                              (width *vertical-slice-width*) &aux (sum-size (integer-length width)))
  (*locally (*type :fast)
    (*with-all-vp-set-of cm-image
      (let ((smaller-set (slice-set cm-image 0 width)))
        (*with-all-vp-set smaller-set
          (*let ((line-sums (!! 0)))
            (*type (line-sums sum-size))
            (loop-over-segmented-slices (slice width 0 cm-image)
              (fill-in-noise slice)
              (emphasize-long-things slice *left-right* :emphasis-factor *long-emphasis*)
              (*let ((hsum (scan!! (cast!! slice 1) '+!! :dimension *left-right* :segment-pvar segment-start)))
                (*type (hsum sum-size))
                (for-visual-types (grey-scale-to-fb hsum))
                (*when segment-end
                  (map-to-smaller hsum line-sums))))
            (*with-all-vp-set-of line-sums
              (*let ((centers nil!!))
                (*type (centers t))
                (find-line-midpoints line-sums cutoff centers)
                (let ((staff-lines
                        (chain-compute-segments (construct-chains centers 1 range) width 'page-staff-lines)))
                 (send staff-lines :set-image (pvar-getf cm-image :image))
                 staff-lines)))))))))
```

5.10.4 Top level for Finding Measure Lines

```
(defun find-measure-lines
       (&key (cm-image *last-image-to-cm*)
        (cutoff .8) (range 8)
        (width *measure-slice-width*) &aux (sum-size (integer-length width)))
  (*locally (*type :fast)
    (*with-all-vp-set-of cm-image
      (let ((smaller-set (slice-set cm-image *up-down* width)))
        (*with-all-vp-set smaller-set
          (*let ((line-sums (!! 0)))
            (*type (line-sums sum-size))
            (loop-over-segmented-slices (slice width *up-down* cm-image)
              (fill-in-noise slice)
              (emphasize-long-things slice *up-down* :emphasis-factor
                                     *measure-emphasis* :squelch-factor *measure-squelch*)
              (*let ((hsum (scan!! (cast!! slice 1) '+!!
                                   :dimension *up-down* :segment-pvar segment-start)))
                (*type (hsum sum-size))
                (for-visual-types (grey-scale-to-fb hsum))
                (*when segment-end
                  (map-to-smaller hsum line-sums))))
            (*with-all-vp-set-of line-sums
              (*let ((centers nil!!))
                (*type (centers t))
                (find-line-midpoints line-sums cutoff centers)
                (let ((measure-lines
                        (chain-compute-segments
                         (construct-chains centers 1 range) width 'page-measure-lines)))
                  (send measure-lines :set-image (pvar-getf cm-image :image))
                  measure-lines)))))))))
```

5.10.5 Filtering

```
(defun emphasize-long-things (slice direction &key
                                   (emphasis-factor *long-emphasis*) (squelch-factor .70)
                                   &aux (sum-size (1+ (integer-length emphasis-factor))))
  (*locally (*type :fast (slice 1))
    (*let ((sum (!! 0))
           (previous slice)
           (next slice))
      (*type ((next previous) 1) (sum sum-size))
      (dotimes (k emphasis-factor)
        (if (zerop direction)
          (*set previous (left!! previous) next (right!! next))
          (*set previous (above!! previous) next (below!! next)))
        (*set sum (+!! sum previous next)))
      (*locally (*type (slice t))
        (*set slice (and!! slice (>!! sum (*!! (reduce-and-spread!! sum 'max!! (other-direction direction))
                                               (!!f squelch-factor))))))
      (for-visual-types (image-to-fb slice)))))
```

5.10.6 Definition of the Chain Structure

A problem with ***defstruct** is illustrated here, and worked around. A chain is defined in part by the size of the vpset which it is in. However the definition is vpset independant. So I use some special variables and a form which dynamically binds them.

```
(defvar .chain-row-length.)
```

```
(defvar .chain-column-length.)

(defvar .chain-pointer-length.)

(defmacro with-chains-in-context-of (vpset &body body)
  '(let ((.chain-row-length. (integer-length (second (vp-set-dimensions ,vpset))))
         (.chain-column-length. (integer-length (first (vp-set-dimensions ,vpset))))
         (.chain-pointer-length. (integer-length (apply '* (vp-set-dimensions ,vpset)))))
     (declare (special .chain-pointer-length. .chain-column-length. .chain-row-length.))
     ,@body))

(*defstruct chain
  (row 0 :type (unsigned-byte .chain-row-length.) :cm-uninitialized-p t)
  (column 0 :type (unsigned-byte .chain-column-length.) :cm-uninitialized-p t)
  (left 0 :type (unsigned-byte .chain-pointer-length.) :cm-uninitialized-p t)
  (right 0 :type (unsigned-byte .chain-pointer-length.) :cm-uninitialized-p t)
  (left? nil :type boolean)
  (right? nil :type boolean))
```

5.10.7 Secondary Accessors for Chains

The slots of the chain are used a second time, with different meaning. I define the accessors here.

```
(defmacro chain-index!! (chain)
  '(chain-left!! ,chain))

(defmacro chain-staff-line?!! (chain)
  '(chain-left?!! ,chain))

(defmacro chain-closest!! (chain)
  '(chain-right!! ,chain))

(defmacro chain-closest?!! (chain)
  '(chain-right?!! ,chain))
```

5.10.8 Sending Chains

This is shorthand for sending to one of the processors pointed to by one of the pointers the chain structure.

```
(defmacro *chain-psetf (place value)
  '(progn (ignore chain-set) (*pset :no-collisions ,value (alias!! (,place chain)) chain-address :vp-set chain-set)))

(defun cubify-chain-pointers (chain)
  (*locally-type (:fast (chain $chain))
    (*with-all-vp-set-of chain
      (*when (chain-left?!! chain)
        (*setf (chain-left!! chain) (cube-from-grid-address!! (chain-left!! chain))))
      (*when (chain-right?!! chain)
        (*setf (chain-right!! chain) (cube-from-grid-address!! (chain-right!! chain)))))))
```

5.10.9 Finding the Closest Line Segment

```
(defun find-closest-to-staff-line (chain index column row staff-line
                                              horizontal-range hamount vertical-range)
  (*locally-type (:fast (index :field) (column :field) (row :field) (chain $chain) (staff-line t))
    (*let ((attached-to-something? nil!!)
           distance
           (closest (most-positive-fixnum!! row))
           attached-to)
      (*type (attached-to-something? t) ((distance closest) %row) (attached-to %index))
      (loop for count below horizontal-range do
        (*all (*set chain (news!! chain hamount 0)))
        (*when (not!! attached-to-something?)
          (dolist (vamount '(-1 1))
            (loop for count below vertical-range
                  do
              (*set distance (abs!! (-!! (chain-row!! chain) row)))
              (*when (and!! staff-line (chain-staff-line?!! chain) (<!! distance closest))
                (*setf closest distance)
                (*setf attached-to-something? t!!)
                (*setf attached-to (chain-index!! chain)))
              (*all (*set chain (news!! chain 0 vamount)))
                  finally (*all (*set chain (news!! chain 0 (* -1 vertical-range vamount)))))))
            finally (*all (*set chain (news!! chain (* -1 horizontal-range hamount) 0))))
      (*setf (chain-closest!! chain) attached-to (chain-closest?!! chain) attached-to-something?)
      )))
```

5.10.10 Create Pointers to the Left and Right

```
(defun set-left-right-pointers (staff-line chain chain-original hamount chain-address chain-set)
  (*locally-type (:fast (staff-line t) ((chain-original chain) $chain) (chain-address :field))
    (*when staff-line
      (*when (chain-closest?!! chain-original)
        (when (plusp hamount)
          (*chain-psetf chain-right!! (chain-closest!! chain-original))
          (*chain-psetf chain-right?!! (chain-closest?!! chain-original)))
        (when (minusp hamount)
          (*chain-psetf chain-left!! (chain-closest!! chain-original))
          (*chain-psetf chain-left?!! (chain-closest?!! chain-original)))))))
```

5.10.11 Ensure that Chains are Linear

Because the links between elements of the chain are determined by search, there is no guarantee that two chain elements don't point to an identical element. This routine removes such cases.

```
(defun detach-endpoints-at-vertices (chain)
  (compiler-let ((*use-always-instructions* nil))
    (*all
      (*locally-type (:fast (chain $chain))
        (*let ((index (self-address!!))
               (right? (chain-right?!! chain))
               (left? (chain-left?!! chain)))
          (*type (index :cube) ((right? left?) t))
          (*when (and!! right?
                        (pref!! left? (chain-right!! chain))
                        (not!! (=!! (pref!! (chain-left!! chain) (chain-right!! chain)) index)))
            (*setf (chain-right?!! chain) nil!!))
          (*when (and!! left?
                        (pref!! right? (chain-left!! chain))
                        (not!! (=!! (pref!! (chain-right!! chain) (chain-left!! chain)) index)))
            (*setf (chain-left?!! chain) nil!!)))))))
```

5.10.12 Create a Chain

```
(defun chain-compute-segments (chain width flavor)
  ;; first disconnect segments which are not contiguous.
  (with-chains-in-context-of (slice-set *last-image-to-cm* 0 width)
    (*with-all-vp-set-of chain
      (detach-endpoints-at-vertices chain)
      ;; copy-spread
      (*locally-type (:fast (chain $chain))
        (let ((iterations (ceiling (log (*max (chain-column!! chain)) 2))))
          (*let ((pointer (chain-right!! chain))
                 (valid-pointer (chain-right?!! chain))
                 (value (self-address!!)))
            (*type (pointer .chain-pointer-length.) (valid-pointer t) (value :cube))
            (dotimes (i iterations)
              (*when valid-pointer
                (*pset :no-collisions value value pointer)
                (*set valid-pointer (pref!! valid-pointer pointer))
                (*set pointer (pref!! pointer pointer))))
            (extract-chain-extents chain value width flavor)))))))
```

5.10.13 Extract a List of Chain Extents and Positions

```
(defun extract-chain-extents (chain id width flavor)
  (*all
    (*locally-type (:fast (chain $chain) (id :field))
      (*let ((left (and!! (chain-right?!! chain) (not!! (chain-left?!! chain))))
             (right (and!! (chain-left?!! chain) (not!! (chain-right?!! chain))))
             (any (or!! (chain-left?!! chain) (chain-right?!! chain)))
             (recipients nil!!)
             (chain chain)
             (segment-pvar))
        (*type ((left right any recipients segment-pvar) t))
        (let ((left-points (*when left (*sum (!! 1))))
              (right-points (*when right (*sum (!! 1))))
              (total-points (*when any (*sum (!! 1)))))
          (assert (= left-points right-points) ()
                  "Different number of endpoints on the left ~A, on the right ~A" left-points right-points)
          (*when any
            (*let ((rank (rank!! id '<=!!)))
              (*type (rank :cube))
              (*pset :no-collisions chain chain rank :notify recipients)
              (*pset :no-collisions id id rank)))
          (*when recipients
              (*set segment-pvar (/=!! (next!! id) id))
              (sort!! chain '<=!! :key 'chain-column!! :segment-pvar segment-pvar))
          (make-instance
            flavor
             :raw-y (pvar-to-array (chain-row!! chain) nil :cube-address-end total-points)
             :raw-x (pvar-to-array (*!! (chain-column!! chain) (!!i width))
                       nil :cube-address-end total-points)
             :raw-id (pvar-to-array id nil :cube-address-end total-points)
             :slice-width width
          ))))))
```

5.10.14 Reduce each line segment to a point which is at the centroid

```
(defun find-line-midpoints (line-sums cut-off centers)
  (*with-all-vp-set-of line-sums
    (*let ((sigma-sum-x (!! 0))
           (multiplicand (!! 0))
           (sigma-multiplicand (!! 0)))
      (*type :fast ((sigma-sum-x multiplicand sigma-multiplicand) ~) (line-sums %line-sums) (centers t))
      (*with-x-and-y (which-slice vertical-position)
        (*set multiplicand  (*!! vertical-position line-sums))
        (*let* ((above-threshold? (>=!! line-sums (*!! (!!f cut-off)
                                                      (reduce-and-spread!! line-sums 'max!! *up-down*))))
                (centroid-start (and!! above-threshold? (not!! (above!! above-threshold?))))
                (centroid-end (and!! above-threshold? (not!! (below!! above-threshold?))))
                (centroid (!! 0)))
          (*type ((above-threshold? centroid-start centroid-end) t) (centroid %vertical-position))
          (*set sigma-multiplicand (scan!! multiplicand  '+!! :segment-pvar centroid-start :dimension *up-down*))
          (*set sigma-sum-x (scan!! line-sums '+!! :segment-pvar centroid-start :dimension *up-down*))
          (*set centers nil!!)
          (*when centroid-end
            (*set centroid (round!! sigma-multiplicand sigma-sum-x))
            (*pset :overwrite t!! centers (cube-from-grid-address!! which-slice centroid)))
          )))))
```

5.10.15 Front end Representation of lines

```
(defflavor parallel-lines (raw-x raw-y raw-id points (image nil)
                           (lines nil) (segments nil) (slice-width nil) (deleted nil)
                           (segment-pos nil) (median-width nil) (maximum-number-of-lines-to-remove 30))
         (util-mixin)
  :initable-instance-variables
  :settable-instance-variables
  :gettable-instance-variables)

(defmethod (make-instance parallel-lines :after) (&rest ignore)
  (setq points (length raw-y))
  (assert (= (length raw-y) (length raw-x) (length raw-id)) ()
          "Something's wrong. These should all be the same length - ~A,~A,~A"
          raw-y raw-x raw-id)
  (construct-individual-segments self)
  (compute-median-width self))

(defmethod (sys:print-self parallel-lines) (stream ignore ignore)
  (format stream "#<~A lines for page ~A, ~r segments~A>"
             (moniker self)
             (string-downcase (send image :name)) lines
             (if deleted (format nil ", ~r ~A removed"
                                 (length deleted)
                                 (if (> (length deleted) 1) "were" "was")) "")))

(defmethod (moniker parallel-lines) ()
  "parallel")

(defmethod (:fasd-form-of parallel-lines) (variable)
  (case variable
    (image '(or (color:find-image ,(send image :name)) (to-image ,(send image :name))))
    (otherwise nil)))
```

5.10.16 Reducing Several Line Segments to a Single Line

```
(defmethod (construct-individual-segments parallel-lines) ()
  (setq lines (count-segments self))
  (setq segments (make-array lines :fill-pointer t))
  (loop with last and count = -1
        for el being the array-elements of raw-id
        for i from 0
        do
    (when (not (eql last el))
      (incf count))
    (setq last el)
    (push (cons (aref raw-x i) (aref raw-y i)) (aref segments count)))
  (loop for i below (length segments)
        do (setf (aref segments i) (mapcan (λ(a) (list (car a) (cdr a))) (sort (aref segments i) '< :key 'car))))
  (sort-segments self)
  (compute-segment-average-position self))

(defmethod (compute-segment-average-position parallel-lines) ()
  (setq segment-pos (make-array (length segments) :fill-pointer t))
  (loop for s being the array-elements of segments
        for count from 0
        do
    (setf (aref segment-pos count)
          (/ (loop for (x y) on (aref segments count) by #'cddr summing y)
             (/ (length (aref segments count)) 2.0)))))

(defmethod (sort-segments parallel-lines) ()
  (setq segments (sort segments '< :key 'second)))

(defmethod (compute-left-and-right parallel-lines) (left right
                                                    &optional (scale slice-width)
                                                    &aux (segments segments))
  (let ((left left) (right right))
    (declare (sys:array-register left right segments))
    (loop for s being the array-elements of segments
          for count from 0
          do
      (setf (aref left count) (floor (car s) scale))
      (setf (aref right count) (floor (loop for (x y) on s by #'cddr finally (return x)) scale)))))
```

5.10.17 Compute the Median Distance Between Lines

```
(defmethod (compute-median-width parallel-lines) ()
  (stack-let ((left (make-array (length segments)))
              (right (make-array (length segments)))
              (width (make-array (length segments))))
    (declare (sys:array-register left right))
    (compute-left-and-right self left right)
    (loop for l being the array-elements of left
          for r being the array-elements of right
          for count from 0
          do
      (setf (aref width count) (abs (- l r))))
    (sort width '<)
    (setq median-width (aref width (floor (length width) 2)))))
```

5.10.18 Generic Algorithm for Removing Lines which don't Make Sense

```
(defmethod (verify-and-fix-if-necessary parallel-lines) ()
  (compute-mode self)
  (loop until (could-lines-make-sense? self)
        for count below maximum-number-of-lines-to-remove
        do
    (remove-worst-line self)
    finally
      (when (= count maximum-number-of-lines-to-remove)
        (like-new self)
        (draw self)
        (error "Couldn't verify and fix ~A" (send image :name)))))

(defmethod (ask-user-for-line parallel-lines) ()
  (draw self)
  (let ((res (accept 'number :prompt
             (format nil "Choose a superfluous ~A line by pointing at it"
                     (moniker self))
             :default 0)))
    res))
```

5.10.19 Draw Lines for Debugging Purposes

```
(defmethod (draw parallel-lines) (&key (reduce-by 6))
  (flet ((draw-them ()
          (loop for s being the array-elements of segments for count from 0 do
            (dw:with-output-as-presentation (:object count :type 'sys:expression)
              (loop for (x y) on s by 'cddr
                    with lastx and lasty
                    with not-yet = t do
                (unless  not-yet (draw-a-segment self lastx lasty x  y ))
                (setq lastx x lasty y)
                (setq not-yet nil))))))
    (with-room-for-image
      (image :reduce-by reduce-by
             :label (format nil "~V⊂~A lines from ~A~⊃"
                      '(:eurex :italic :huge) (moniker self) (send image :name))
             :label-object self)
      (draw-them))))

(defmethod (draw-a-segment parallel-lines) (x0 y0 x1 y1 &rest args)
  (if color:color-screen
     (apply 'draw-line x0 y0 x1 y1 :thickness 2 :scale-thickness nil args)
     (apply 'draw-line x0 y0 x1 y1 args)) )
```

5.10.20 Methods to Delete One or More Lines

The lines are stored in an array, so this requires moving the lines after the one to be deleted so that they fill the space. Keep track of the deleted lines so that they can be restored, for debugging.

```
(defmethod (delete-line parallel-lines) (which)
  (flet ((doit (arr)
           (loop for i below lines
                 with count = 0 do
             (setf (aref arr count) (aref arr i))
                 when (/= i which)
                   do (incf count))
           (decf (fill-pointer arr))))
    (doit segments)
    (doit segment-pos)
    (decf lines)
    (push which deleted)))

(defmethod (delete-lines parallel-lines) (them)
  (flet ((doit (arr &aux (length (car (array-dimensions arr))))
           (stack-let ((new (make-array length :element-type (array-element-type arr)
                                              :fill-pointer (fill-pointer arr))))
             (copy-array-portion arr 0 (1- (fill-pointer arr)) new 0 (1- (fill-pointer arr)))
             (setf (fill-pointer arr) 0)
             (loop for el being the array-elements of new
                   for count from 0
                   unless (member count them :test #'=)
                     do (vector-push-extend el arr)))))
    (doit segments)
    (doit segment-pos)
    (decf lines)
    (setq deleted (append deleted them))))
```

5.10.21 Miscellaneous Functions

like-new is a routine which restores the state of the lines to what they were before any lines were deleted. This was useful for debugging.

```
(defmethod (page parallel-lines) ()
  (send (send self :image) :name))

(defmethod (count-segments parallel-lines) ()
  (loop with last and count = 0
        for el being the array-elements of raw-id
        do
    (when (not (eql last el)) (incf count))
    (setq last el)
    finally (return-from count-segments count)))

(defmethod (like-new parallel-lines) ()
  (construct-individual-segments self)
  (setq deleted nil)
  (compute-median-width self))
```

5.10.22 Definition of a Full Page of Staff Lines

The **:fasd-form** method here and elsewhere instructs the lisp machine how to save a representation of the object to a file.

```
(defflavor page-staff-lines ((mode nil) (staffs nil) (systems nil)
                              (maximum-number-of-lines-to-remove 30))
          (parallel-lines)
  :settable-instance-variables
  :gettable-instance-variables
  :initable-instance-variables)

(defmethod (moniker page-staff-lines) ()
  "staff")

(defwhopper (:fasd-form page-staff-lines) (&rest args)
  `(let ((it ,(lexpr-continue-whopper args) ))
    (flet ((set-staff-lines (thing) (send thing :set-staff-lines it)))
      (mapcar #'set-staff-lines (send it :systems))
      (mapcar #'set-staff-lines (send it :staffs)))
    (loop for s in (send it :systems) do (reset-staves s))
    it))
```

5.10.23 Compute the Mode Distance between Lines

```
(defmethod (compute-mode page-staff-lines) ()
  (let ((max-spacing
         (loop for i below (1- (length segments))
               for row = (second (aref segments i))
               for next-row = (second (aref segments (1+ i)))
               maximize (abs (- row next-row)))))
    (stack-let ((bins (make-array (+ max-spacing 10) :initial-element 0 :element-type 'fixnum)))
      (loop for i below (1- (length segments))
            for row = (aref segment-pos i)
            for next-row = (aref segment-pos (1+ i))
            do
        (incf (aref bins (abs (round (- next-row row))))))
      (loop for s being the array-elements of bins
            for count from 0
            with max = 0 and where
            when (> s max) do (setq where count max s)
            finally (return (setq mode where))))))
```

5.10.24 Decide if the Set of Staff Lines Could Make Sense

A set of staff lines makes sense when there are a multiple of five of them, and each has a neighbor whose distance is approximately the mode distance between lines.

```
(defmethod (could-lines-make-sense? page-staff-lines) ()
  (and (zerop (mod (length segments) 5))
       (loop with count = 0
             with tolerance = (* *slack-staff-line-spacing-percentage* mode)
             with could = t and last
             while (< count (length segment-pos))
             do
         (setq last (aref segment-pos count))
         (loop for i below 4 do
           (setq could (and could (< (abs (- mode (- (aref segment-pos (incf count)) last))) tolerance)))
           (setq last (aref segment-pos count)))
         (incf count)
            finally (return could))))
```

```
(defmethod (remove-worst-line page-staff-lines) ()
  (let ((line (or (surprisingly-short-line self)
                  (line-shorter-than-neighbors self)
                  (ask-user-for-line self))))
    (delete-line self line)))
```

5.10.25 Ways of Choosing a Bad Line to Remove

```
(defmethod (line-shorter-than-neighbors page-staff-lines) ()
  (stack-let ((left (make-array (length segments)))
              (right (make-array (length segments))))
    (declare (sys:array-register left right))
    (compute-left-and-right self left right)
    (loop with segments = (length segment-pos)
          for i below segments
          with biggest-diff = 2 and where = nil
          for above = most-positive-fixnum then middle
          for middle = (- (aref right 0) (aref  left 0)) then below
          for below = (if (= (1+ i) segments) most-positive-fixnum (- (aref right (1+ i)) (aref left (1+ i))))
          for diff = (if (< above below) (- above middle) (- below middle))
          when (> diff biggest-diff)
            do (setq biggest-diff diff where i)
          finally (return where))))

(defmethod (surprisingly-short-line page-staff-lines) (&aux (segments segments))
  (stack-let ((left (make-array (length segments)))
              (right (make-array (length segments))))
    (declare (sys:array-register left right))
    (compute-left-and-right self left right)
    (loop with segments = (length segment-pos) and where = nil
          for i below segments
          for width = (- (aref right i) (aref left i))
          with shortest = most-positive-fixnum
          when (< width shortest) do (setq shortest width where i)
          finally (return (if (< shortest (/ mode 3)) where nil)))))
```

5.10.26 Group Each Five Staff Lines into a Staff

```
(defmethod (split-staffs page-staff-lines) ()
  (setq staffs
        (loop for i below lines by 5 for count from 0
              collecting (make-instance
                           'one-staff
                           :the-lines (make-array 5 :displaced-to segments :displaced-index-offset i)
                           :y-position (make-array 5 :displaced-to segment-pos :displaced-index-offset i)
                           :from-top count
                           :staff-lines self
                           :mode-spacing mode))))
```

5.10.27 Try to Figure Out How Many Staffs in a System

```
(defmethod (interstaffs page-staff-lines) ()
  (loop for (s . rest) on staffs
        when rest collecting (- (top (first rest)) (bottom s))))

(defmethod (interstaff-larger-than-neighbors page-staff-lines) ()
  (loop for (last this next) on (interstaffs self)
        when (and this next) collecting (and (> this last) (> this next))))
```

```
(defmethod (determine-n-tet page-staff-lines) ()
  (let ((possibilities
         (loop for possible from 2 to 6
               with interstaff-bigger? = (interstaff-larger-than-neighbors self)
               when (possibly-n-tet self interstaff-bigger? possible) collect possible)))
    (if (= (length possibilities) 1)
      (car possibilities)
      (ask-user-to-choose-arrangement self possibilities))))

(defmethod (possibly-n-tet page-staff-lines) (interstaff-bigger? n)
  (loop for i from (- n 2) by n below (length interstaff-bigger?)
        always (nth i interstaff-bigger?)))

(defmethod (ask-user-to-choose-arrangement page-staff-lines) (possibilities)
  (draw self)
  (let ((names (vector :solo :duet :trio :quartet :quintet :sextet)))
    (1+
      (position
        (dw:accept-values
          '(((member ,@(loop for p in possibilities collecting (aref names (1- p))))
             :prompt "I can't figure out what sort of arrangment this is. What is it?")))
        names))))
```

5.10.28 Construct the System Objects

```
(defmethod (make-systems page-staff-lines) (&optional (n-tet (determine-n-tet self)))
  (let ((staffs (copy-list staffs)))
    (setq systems
          (loop until (null staffs) for count from 0
                collecting (make-instance 'one-system
                                          :the-staves (loop repeat n-tet collect (pop staffs))
                                          :from-top count
                                          :staff-lines self
                                          )))))
```

5.10.29 Definition of a Page of Measure Lines.

Mostly, I fool the staff lines into being vertical. For instance in the draw method I just modifies the usual method by switching the x and y arguments.

```
(defflavor page-measure-lines ()
           (parallel-lines)
  :settable-instance-variables
  :gettable-instance-variables
  :initable-instance-variables)

(defmethod (moniker page-measure-lines) ()
  "measure")

(defmethod (map-measure-lines page-measure-lines) (function)
  (declare (values (which top bottom position)))
  (stack-let ((left (make-array (length segments)))
              (right (make-array (length segments))))
    (declare (sys:array-register left right))
    (compute-left-and-right self left right 1)
    (loop for l being the array-elements of left
          for r being the array-elements of right
          for count from 0
          for pos being the array-elements of segment-pos
          do
      (funcall function count l r pos))))
```

```
(defmethod (sort-segments page-measure-lines) (&rest ignore) nil)

(defwhopper (draw page-measure-lines) (&rest args)
  (unwind-protect
      (progn (rotatef raw-x raw-y)
             (lexpr-continue-whopper args))
    (rotatef raw-x raw-y)))

(defmethod (draw-a-segment page-measure-lines) (x0 y0 x1 y1 &rest args)
  (apply 'draw-line y0 x0 y1 x1 args))
```

5.10.30 Definition of a Staff Line

```
(defflavor one-staff (the-lines from-top staff-lines mode-spacing
                          y-position roughly-one-line line-functions )
        (has-relations util-mixin)
  :settable-instance-variables
  :gettable-instance-variables
  :initable-instance-variables)

(defmethod (make-instance one-staff) (&rest ignore)
  (one-line self))

(defmethod (sys:print-self one-staff) (stream print-depth slashify-p)
  (ignore print-depth slashify-p)
  (format stream "#<~:r staff on page ~a>" (1+ from-top) (send (send staff-lines :image) :name)))

(defmethod (:fasd-form-of one-staff) (symbol)
  (eq symbol 'staff-lines))
```

5.10.31 Simple Functions of Staffs Lines

```
(defmethod (one-line one-staff) ()
  (setq roughly-one-line (/ (- (bottom self) (top self)) 5)))

(defmethod (relative-position one-staff) (position &aux (x (car position))) ; position is cons x y
  (cond ((< x (- (top self) roughly-one-line)) :above)
        ((> x (+ (bottom self) roughly-one-line)) :below)
        (t :inside)))

(defmethod (page one-staff) ()
  (send (send (send self :staff-lines) :image) :name))

(defmethod (top one-staff) ()
  (aref y-position 0))

(defmethod (bottom one-staff) ()
  (aref y-position 4))

(defmethod (draw-clipped-self one-staff) (left right)
  (map nil (λ(y) (draw-a-segment staff-lines left y right y)) y-position))
```

5.10.32 Compute Functions for Estimation of Line from Position.

I take the x and y positions of each of the segment and do linear regression, fitting line number against the x and y. Output are the coefficients which are used to convert back and forth between line number and position.

```
(defmethod (compute-line-functions one-staff) ()
  (let ((points (/ (length (aref the-lines 0)) 2))
        (coeffs (make-array 3)))
    (stack-let ((sums (make-array '(3 3) :initial-element 0s0))
                (rhs (make-array 3 :initial-element 0s0))
                (lu (make-array '(3 3)))
                (ps (make-array 3)))
      (macrolet ((incf-coefficient (i j v)
                   '(setf (aref sums ,i ,j) (incf (aref sums ,j ,i) ,v))))
        (setq line-functions
              (loop for segment below (1- points)
                    for (startx starty endx endy) on (aref the-lines 0) by 'cddr
                    do
                (loop for line being the array-elements of the-lines
                      for line-number from 2 by -1
                      do
                  (loop for (x y) on (nthcdr (* segment 2) line) by 'cddr for count below 2
                        do
                    (incf-coefficient 0 0 (* x x))
                    (incf-coefficient 0 1 (* x y))
                    (incf-coefficient 1 1 (* y y))
                    (incf-coefficient 0 2 (* x))
                    (incf-coefficient 1 2 (* y))
                    (incf-coefficient 2 2 1)
                    (incf (aref rhs 0) (* line-number x))
                    (incf (aref rhs 1) (* line-number y))
                    (incf (aref rhs 2) line-number)))
                (math:decompose sums lu ps)
                (math:solve lu ps rhs coeffs)
                    collect (list startx endx coeffs)))))))
```

5.10.33 Conversion between Position and Line Number

```
(defmethod (pitch-position one-staff) (x y)
  (when (not (variable-boundp line-functions)) ;; compute it on demand
    (compute-line-functions self))
  (loop for (start end coeffs) in line-functions
        until (<= start x end)
        finally (return (round (+ (* (aref coeffs 0) x) (* (aref coeffs 1) y) (aref coeffs 2)) .5))))

(defmethod (normalized-y-coordinate one-staff) (x y)
  (when (not (variable-boundp line-functions)) ;; compute it on demand
    (compute-line-functions self))
  (loop for (start end coeffs) in line-functions
        until (<= start x end)
        finally
          (let ((line (+ (* (aref coeffs 0) x) (* (aref coeffs 1) y) (aref coeffs 2))))
            (return (values (+ (aref y-position 2) (/ line (aref coeffs 1))) line)))))
```

5.10.34 Definition of a System

```
(defflavor one-system (the-staves from-top staff-lines half-staff (measure-lines nil) roughly-one-line)
           (has-relations util-mixin)
  :settable-instance-variables :gettable-instance-variables :initable-instance-variables)

(defmethod (make-instance one-system :after) (&rest ignore)
  (when (neq staff-lines t) ;;
     (setq half-staff (floor (- (bottom self) (top self)) 2)))
  (one-line self))
```

```
(Defmethod (sys:print-self one-system) (stream print-depth slashify-p)
  (ignore print-depth slashify-p)
  (format stream "#<~:r system on page ~a>" (1+ from-top) (send (send staff-lines :image) :name)))

(defmethod (reset-staves one-system) ()
  (setq the-staves (loop for which in the-staves collecting (nth which (send staff-lines :staffs))))
  (setq half-staff (floor (- (bottom self) (top self)) 2))
  (one-line self))

(defmethod (:fasd-form-of one-system) (symbol)
  (case symbol
    (staff-lines t)
    (the-staves `(list ,@(loop for s in the-staves collecting (send s :from-top))))))
```

5.10.35 Draw a System

```
(defmethod (draw-self one-system) ()
  (let ((top (top self))
        (bottom (bottom self))
        (right (reduce (λ(a b) (min a (car b))) measure-lines :initial-value most-positive-fixnum))
        (left (reduce (λ(a b) (max  a (car b))) measure-lines :initial-value 0)))
    (loop for (position) in measure-lines do
      (draw-a-segment staff-lines position top position bottom))
      (mapc (λ(staf) (draw-clipped-self staf left right)) the-staves)))
```

5.10.36 Simple Functions of System

```
(defmethod (one-line one-system) ()
  (unless (numberp (car the-staves)) ;; in case we are making instance from fasd file
    (setq roughly-one-line (/ (+ (one-line (top-stave self)) (one-line (bottom-stave self))) 2))))

(defmethod (top-stave one-system) ()
  (car the-staves))

(defmethod (bottom-stave one-system) ()
  (car (last the-staves)))

(defmethod (relative-position one-system) (position &aux (x (car position))) ; position is cons x y
  (cond ((< x (- (top self) roughly-one-line)) :above)
        ((> x (+ (bottom self) roughly-one-line)) :below)
        (t :inside)))

(defmethod (page one-system) ()
  (send (send (send self :staff-lines) :image) :name))

(defmethod (top one-system) ()
  (top (car the-staves)))

(defmethod (bottom one-system) ()
  (bottom (car (last the-staves))))
```

5.10.37 Attach Measure Lines to Systems

Each system decides which measure lines are part of it. This help rule out potential measure lines, since a line is ignored if it doesn't significantly overlap the system.

```
(defmethod (grab-potential-measure-line one-system) (top bottom position)
  (let* ((inside (self-intersection self top bottom))
         (outside (- bottom top inside)))
    (unless (variable-boundp measure-lines) (setq measure-lines nil))
    (if (and (> inside outside) (> inside half-staff))
      (push (list position (cons top bottom)) measure-lines)
      nil)))

(defmethod (self-intersection one-system) (top bottom &aux (my-top (top self)) (my-bottom (bottom self)))
  (cond ((or (< bottom my-top) (> top my-bottom)) 0) ; no intersection
        ((and (>= top my-top) (<= bottom my-bottom)) (- bottom top)) ; completely contained
        ((and (<= top my-top) (>= bottom my-bottom)) (- my-bottom my-top)) ; surrounds
        ((< top my-top) (- bottom my-top)) ; bottom is between
        (t (- my-bottom top))))
```

5.10.38 Test for Double Measure Lines.

This is a standin, I never got to the stage where it mattered.

```
(defmethod (ignore-double-staff-lines one-system) ()
  (loop for (this thother) on measure-lines
        when (and thother (< (- (car this) (car thother)) *double-measure-line-threshold*))
          do ))
```

5.10.39 Definition of a Page of Bars.

```
(defflavor page-of-bars (staff-lines measure-lines)
           (util-mixin)
  :initable-instance-variables
  :settable-instance-variables
  :gettable-instance-variables)

(defmethod (page page-of-bars) ()
  (send (send (send self :staff-lines) :image) :name))

(defmethod (draw page-of-bars) (&optional (reduce-by 7.9))
  (with-room-for-image ((send staff-lines :image) :reduce-by reduce-by
                              :label (format nil "~V⊂Measures from ~A~⊃"
                                             '(:eurex :italic :huge)
                                             (send (send staff-lines :image) :name))
                              :label-object self)
    (mapc 'draw-self (send staff-lines :systems))))
```

Chapter 6

Locating Stems

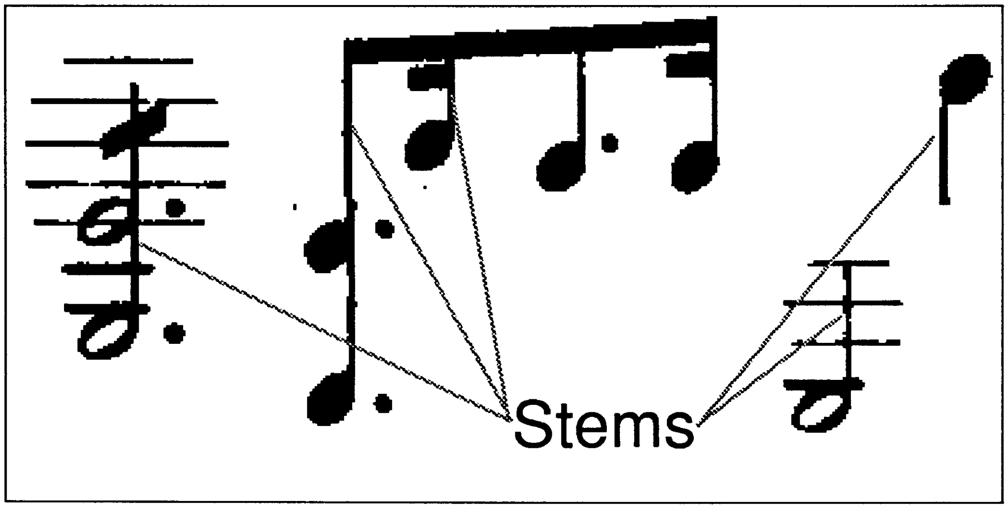


Figure 6-1: Some examples of stems.

6.1 Introduction

Stems are the small lines which lead vertically from a note head head. From the point of view of optical recognition of music, stems serve several purposes. When the note is part of a beamed group the stem leads from the note head to the beam. Since a stem or any of the flags can be attached to a solid note-head, the stem distinguishes a solid note head as being a quarter note. When several notes are played as a chord, a stem attaches them. A stem is also part of the shape of half note head. Perhaps most importantly for this work, when a note is either above or below a staff, the stem points in the direction of the staff, sometimes being the decisive factor in deciding whether a note is part of the staff above it ot the staff below.

In fact, in all the of the cases of stem use but the last, the stem represents information which is recoverable from other sources. Flags are found by the same method that note heads are found. The stem on a half-note gives no additional information. The shape of a half note, though similar to a whole note, is distinct.

The stems which attach a note to a beam are also redundant information. In fact all the notes which belong to the staff that the beam is in lie either above or below the beam, and their horizontal positions are bounded by the horizontal extent of the beam.

The only stems which are of any use at all are those stems which are attached to notes which lie between staffs. I would like to write that given that only these stems are interesting, these are the only steams that will be set about to be found. Unfortunately, the reality of the development of this module unfortunately disturbs the flow of the prose.

In fact I first made attempts to write algorithms to find all stems, and found this very difficult to do. There are several reasons for this. Stems are thin so any noise (dirt, nonuniformity) in the printing easily affects their shapes. The length of stems is not uniform. If a note is close to a beam then the stem will be short - sometimes of length smaller than the note itself. Stems are hard to disconnnect from their environment. They touch the note heads, cross several staff lines, and are perilously close to accidentals and the like.

During these experiments I noticed that all the algorithms worked the best when the stem was isolated from its surroundings, i.e. when it was attached to a note which was outside of a staff. Elsewhere, the algorithms found stems where they didn't belong and didn't find them where they did belong.

It was after this experimentation that I realized that finding stems didn't matter that much, except in certain places, and those places were exactly the places where they were easiest to find. So the strategy changed: instead of looking for all stems, use an algorithm which does well for stems which are isolated. Later, only use the stem information on demand, when the stem can supply some information that can't be derived in any other way.

To find stems I use the following series of steps, illustrated in Figure 6-2. First remove points which are part of a horizontal run wider than the average stem thickness. This leaves

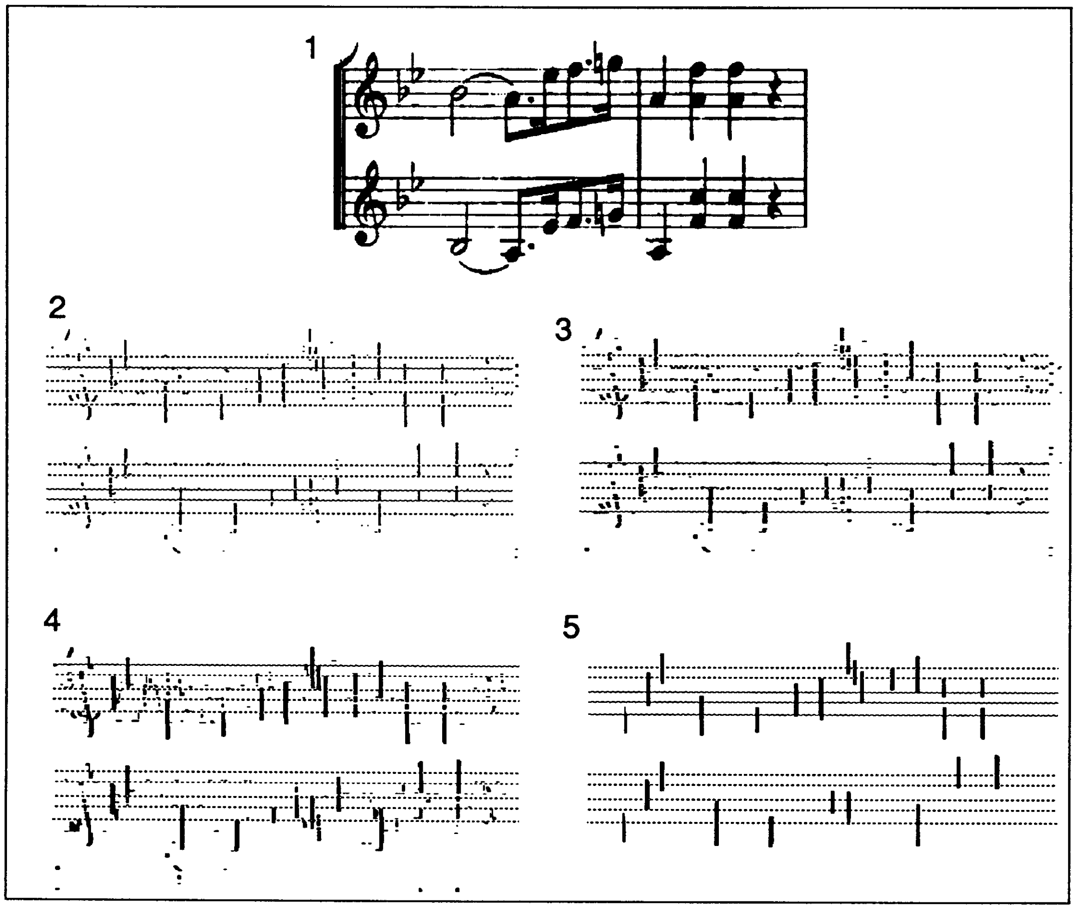


Figure 6-2: Processing stages in locating stems: 1) Original, 2) After filter, 3) after bleed 4) after removing short things 5) result.

the greater part of stems untouched, and almost everything else in the score disappears, since most things are wider than a stem. A side effect of this is that where a staff line crosses a stem, a small gap in the stem is produced, since that part of the stem is also part of the long horizontal staff line.

These gaps need to be repaired. The gap repair method has each white point looking to see if there is a black point nearby above and nearby below it. If there are then it turns itself black.

Finally the stems need to be extracted from the connection machine. In order to do this, they need to be uniquely labelled. The labelling algorithm is not at all general, it exploits the fact that the stem regions are essentially rectangular. To label a stem we first spread a unique marker horizontally, and then vertically. I call this the peanut butter region labelling algorithm, since it resembled the spreading of peanut butter on bread. The front end representation records the coordinates of the top of the stem, and its height.

6.2 Future Work

Some experiments have been done to see whether the stem information in its current form is in fact adequate to disambiguate which staff notes belong to which staffs. The results look promising, but a final assessment will only be available once more work on assembling a complete score is done.

6.3 The Code

6.3.1 Definition of a stem feature

```
(defflavor stem-feature (top left height) (basic-recognized-feature sys:property-list-mixin)
  :initable-instance-variables
  (:constructor make-stem (top left height))
  :gettable-instance-variables)

(defmethod (make-instance stem-feature) (&rest ignore)
  (setq reduced-x left reduced-y top my-feature self))

(defmethod (pretty-name stem-feature) () "a stem")

(defmethod (:my-feature stem-feature) () self)

(defmethod (draw stem-feature) (&key (alu :draw))
  (draw-line left top left (+ top height) :alu alu))
```

```
(defmethod (bottom stem-feature) ()
  (+ top height))

(defmethod (top stem-feature) ()
  top)
```

6.3.2 Filtering

Filter out stuff which isn't a good stem candidate

```
(defun find-stems (&optional (maximum-width 4) (horizontal-bleed 1)
                             (vertical-gap 10) (minimum-height 20) &aux them)
  (*locally-type (:fast)
    (loop-over-slices (slice *last-image-to-cm*)
      (with-neighbor-counts (slice 0 maximum-width)
        (*set slice (and!! slice (<!! (+!! (!! 1) left-count right-count) (!!i maximum-width)))))
      (for-visual-types (image-to-fb slice))
      (bleed-horizontal slice horizontal-bleed)
      (fill-in-vertical-gaps slice vertical-gap)
      (with-neighbor-counts (slice minimum-height 0)
        (*set slice (and!! slice (>!! (+!! (!! 1) above-count below-count) (!!i minimum-height)))))
      (for-visual-types (image-to-fb slice))
      (push (quick-label-stems slice column-start row-start)  them)))
  (make-instance 'page-of-stems :them (apply 'nconc them) :image (pvar-getf *last-image-to-cm* :image)))
```

6.3.3 Fill in Vertical Gaps

```
(defun fill-in-vertical-gaps (slice &optional (radius 1))
  (*locally-type ((slice t) :fast)
    (*let ((hasupneighbor slice)
           (hasdownneighbor slice))
      (*type ((hasdownneighbor hasupneighbor) t))
      (loop for r below radius
            do
        (*set hasupneighbor (or!! hasupneighbor (news-border!! hasupneighbor nil!! 0 1)))
        (*set hasdownneighbor (or!! hasdownneighbor (news-border!! hasdownneighbor nil!! 0 -1)))
            finally (*set slice (or!! slice (and!! hasupneighbor hasdownneighbor))))
      (and moviep (image-to-fb slice)))))
```

6.3.4 Label the Stem Regions

This is a quick and dirty region labelling. Since stems are pretty much rectangles, I can get away with token spread horizontally and then vertically.

```
(defun quick-label-stems (slice column-start row-start &aux smaller-set smaller-size)
  (*let ((label (!! 0)) top bottom left right)
    (*type :fast (label :cube) ((top bottom slice left right) t))
    ;calculate boundaries
    (*set top (and!! slice (xor!! (above!! slice) slice)))
    (*set bottom (and!! slice (xor!! (below!! slice) slice)))
    (*set left (and!! slice (xor!! (left!! slice) slice)))
    (*set right (and!! slice (xor!! (right!! slice) slice)))
    (*when slice
      ;; label
      (*when (and!! top left) (*set label (enumerate!!)))
      (setq smaller-size (1+ (*max label)))
      (setq smaller-set (vpset-to-fit smaller-size))
      ;; smoosh label up/down, left/right, up/down
      (*set label (scan!! label 'max!! :dimension 1 :segment-pvar top))
      (*set label (scan!! label 'copy!! :dimension 1 :segment-pvar bottom :direction :backwards))
      (*set label (scan!! label 'max!! :dimension 0 :segment-pvar left ))
      (*set label (scan!! label 'copy!! :dimension 0 :segment-pvar right :direction :backwards))
      (*set label (scan!! label 'max!! :dimension 1 :segment-pvar top))
      (*set label (scan!! label 'copy!! :dimension 1 :segment-pvar bottom :direction :backwards))
      (for-visual-types (*all (rainbow-scale-to-fb (if!! (zerop!! label) label (1+!! (mod!! label (!!i 254))))))))
    ;extract
    (*with-x-and-y (xpos ypos)
      (*let-in-vp-set smaller-set
           (x (y (!! 0)) (height (!! 0)) (valid nil!!))
        (*type (x %xpos) ((y height) %ypos) (valid t))
        (*when (and!! top left)
          (*pset :min xpos x label :notify valid)
          (*pset :min ypos y label))
        (*when (and!! bottom right)
          (*pset :max ypos height label))
        (*with-all-vp-set smaller-set
          (*set height (-!! height y))
          (extract-stems x y height valid smaller-size column-start row-start)
          )))))
```

6.3.5 Extract a Representation of the Stems

```
(defun extract-stems (x y height valid howmany column-start row-start)
  (*locally-type ((x :field) (y :field) (height :field) (valid t))
    (stack-let ((xa (make-array howmany))
                (ya (make-array howmany))
                (heighta (make-array howmany))
                (valida (make-array howmany :element-type 'boolean)))
      (pvar-to-array x xa :cube-address-end howmany)
      (pvar-to-array y ya :cube-address-end howmany)
      (pvar-to-array height heighta :cube-address-end howmany)
      (pvar-to-array valid valida :cube-address-end howmany)
      (loop for top being the array-elements of ya
            for left being the array-elements of xa
            for height being the array-elements of heighta
            for valid being the array-elements of valida
            when valid collecting (make-stem (+ top row-start) (+ left column-start) height)))))

(defmethod (probably-not-a-stem stem-feature) ()
  (> height #.(* 10 (send (find-feature :solid) :height))))
```

6.3.6 Definition of a Page of Stems.

```
(defflavor page-of-stems (them image positions) (util-mixin)
  :initable-instance-variables
  :gettable-instance-variables)

(defmethod (draw page-of-stems) (&key (reduce-by 4) (coloring-function (λ(ignore) :draw)))
  (with-room-for-image
    (image :reduce-by reduce-by
           :label (pretty-name (car them))
           :label-object self)
    (map nil (λ(f) (dw:with-output-as-presentation (:object f :type 'sys:expression)
                     (draw f :alu (funcall coloring-function f))))
         them)))
```

Chapter 7

Locating Ties and Slurs

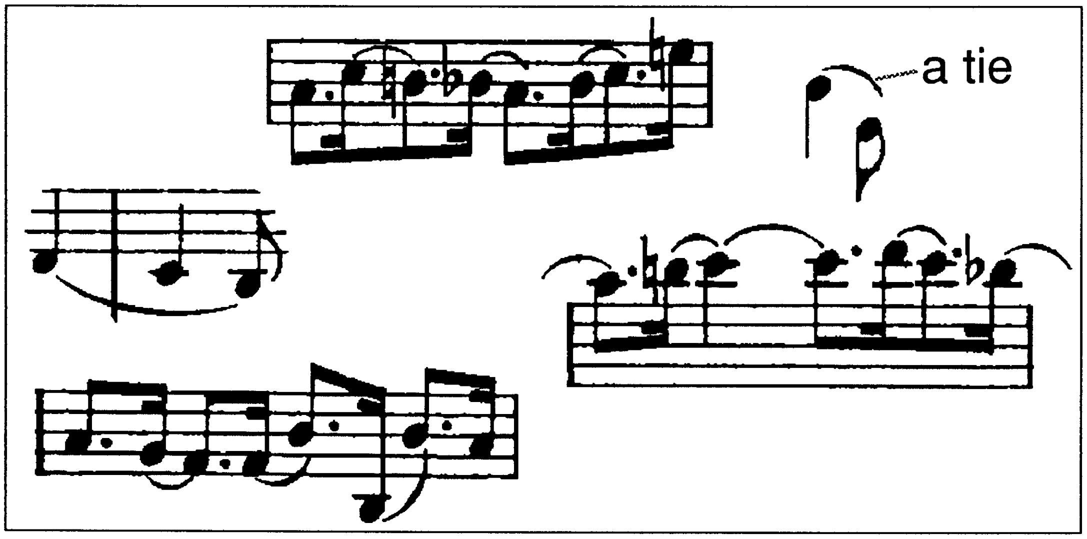


Figure 7-1: Some examples of ties and the contexts in which they appear.

7.1 Introduction

Ties and slurs are curved lines which connect one note to another. Ties connect notes of the same pitch signifying that the notes are to be played as one longer note. Slurs connect notes with different pitches and indicate that the notes should be played with each leading smoothly into the next. In the rest of this section I'll use the word ties to mean both, since they are graphically equivalent.

Ties are of variable length, and of variable curvature. Their start and end points need not be at the same height. Occasionally, a tie is broken across two systems, when a group of notes which it refers to cannot fit on a single system The thickness of a tie varies across its length, being thicker in the middle, and tapering to a point at the ends. Ties begin and end at specific notes, and modify the interpretation of all the notes between these. In searching for a representation to be used in the later interpretation of the score this extent is the only semantic information which is carried by the tie, and so the only information which is necessary to represent eventually are these endpoints.

Finding ties is hard because their shape varies a lot, and they intersect many things, at least staff lines and measure lines, and often stems, heads and accidentals. Because of the size variability the template matching approach won't work in general, though it might be useful for finding the class of short slurs which may be more fixed in their shape. An approach of trying to distinguish ties from other objects, rather than directly identifying them might have worked if ties didn't intersect all sorts of other things.

I have not yet completed the work on finding ties, but have done what I hope is a promising start. After looking at ties for while, I decided that the most salient feature of ties was that they were long "line-like" things. The question then became, how do you find line-like things on a scanned image. Here is the strategy I used.

I tried to develop a filter which removes the parts of the image which are not ties. In this way subsequent passes can more easily focus on candidate for tieness. Consider a neighborhood around each point on the score. The black points in this neighborhood are fit to line. When the goodness of fit is better than some threshold the pixel is considered to be part of some line-like thing.

When I first tried this, I found was almost every point had a poor goodness of fit. The problem was twofold. First, even line-like things in the image had a thickness of more than one pixel. Because of this I was fitting a line to a rectangle yielding a poor fit. Second, using every point in a neighborhood was a bad idea. Consider a white point which is sandwiched by two parallel black lines. Any neighborhood of the white point contains some black, and yet the point, being white, is most certainly not part of any line. Consider a point on one of the black lines. Most neighborhoods of the line the point is on also contain the other line. Even though the the point is certainly part of a line, the presence of the points on the other line will disturb the fit.

Each of these problems has a separate solution. To attack the problem of thick lines, I used a *thinning* or *skeletonizing* algorithm invented by Zhang and Suen [3]. This image transform erodes the black areas in an image in such a way as to preserve the topological properties of regions, while leaving no more than a couple of pixels in a row black. With a processor per pixel, this algorithm takes time proportional to lesser of the width or the height of the area a pixel is part of. The transform substantially distorts other parts of the image. Note-heads, for example, are usually turned into crosses. The algorithm works well, however, in the case of ties where it leaves a line a pixel wide where ties used to be.

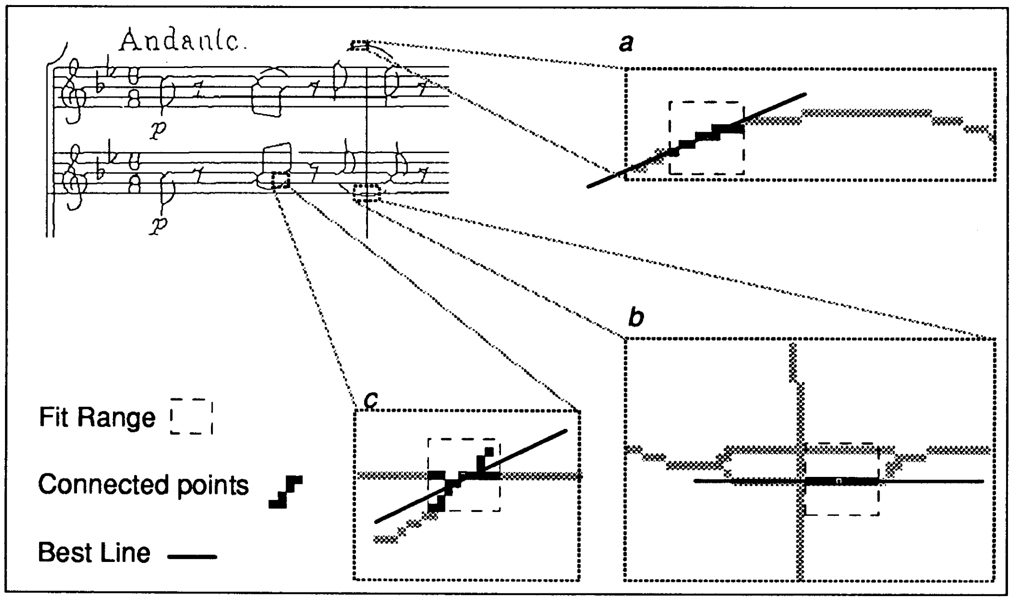


Figure 7-2: To filter for line like objects, a line is fit in the connected neighborhood of each point. When the line doesn't fit well, as in **c** the point is not considered part of a line. In **b** notice that the top grey line is not part of the fit, since it is not connected

To address the problem of undesired fit points in the neighborhood, I decided to use only a locally connected area surrounding each point in the image(as in Figure 7-2). The parallel implementation is interesting. Each processor goes through a step where it communicates with its neighbors to assemble a bitmap of the image in its neighborhood. The line is fit only to processors which are connected to the central point.

Actually, I use an approximation to connectivity which I call *local spiral connectivity*, since testing for true connectivity would take much more time. It is local in that points are not considered connected if there is no connecting path which goes completely through the neighborhood. It is spiral because the algorithm for deciding connectivity proceeds by looping through a spiral starting at the center of the bitmap. At each step it encounters a new point which needs to be classified as being connected to the center point. A new point is considered to be connected if one of its neighbors has already been noted as being connected.

In practice, this version of connectivity serves my purpose well. Fits to the points chosen this way have a goodness of fit which distinguishes well between points which are not line-like and points which are. To tighten the filter, a further test is made to see whether the line which is fit passes within a small distance of the target point. If it is too far away, the point is not considered part of as line.

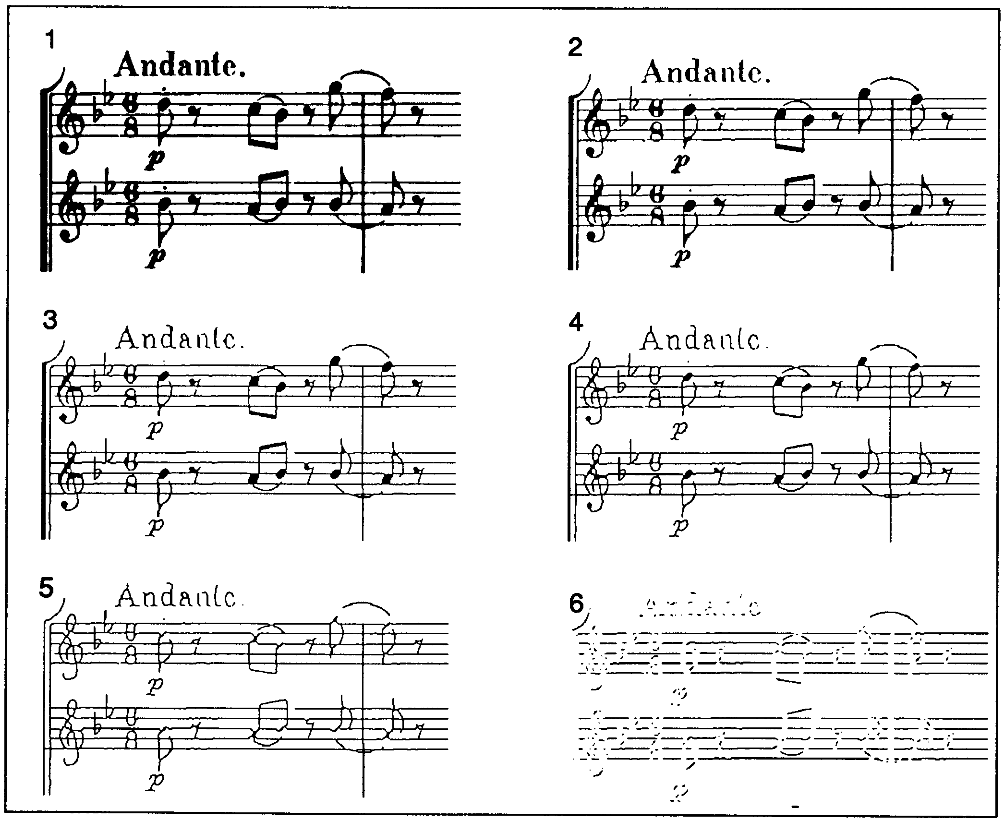


Figure 7-3: Images 1 through 5 show the progress of the skeletonizing algorithm. Image 6 show the final skeletonized image after points which are not parts of lines are removed.

Running the filter has the effect of disconnecting line segments, since vertices don't fit a line well. It also has the serendipitous effect of removing some other line-like things which we are not interested in - measure lines. Measure lines are vertical and the standard least squares fit can not fit nearly vertical lines. This is because it computes its parameters in terms of slope and intercepts, and a vertical line has an infinite slope.

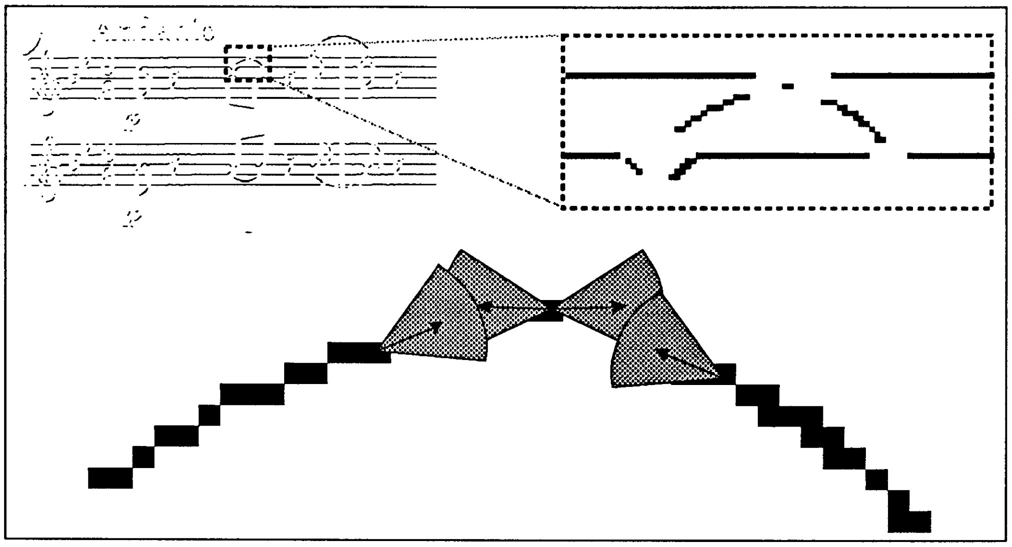


Figure 7-4: Line segments need to be reattached in order to reconstruct ties. In order to do this, one searches in a cone which the tangent bisects, for other segments. When the search areas overlap, there is a potential attachment point.

This is the point at which my work on ties stops. The image in this state looks like a bunch of disconnected line segments(see Figure 7-3), with other smaller scattered noise. Staff lines are still prominent, since they too are line-like, but they no longer connect features together. We already know, from the staff line work where they are, so they can easily be eliminated. Longer ties are interrupted at vertices. What needs to be done is to put these interrupted segments back together, and then determine which represent ties. Line segments can be reconnected by searching in a neighborhood around their endpoints(Figure 7-4). The slope of a tangent of a line continuation should be close to the tangent line at the endpoint being searched from. This search can be done in parallel. Where there are more than a single potential continuation, both should be accepted as potential lines.

Thes lines can then be checked for other constraints. The strongest constraints are that a tie starts and ends near notes(except at the end of a system line), and that the slope of the tangents to the line at each end are of opposite sign. The tie slopes upward at one end and downward at the other, or vice versa.

7.2 The Code

7.2.1 Finding the Ties

The code here is for debugging and exploration. Explore lines lets me experiment with changing various parameters and viewing the results on the framebuffer.

```
(defun test-thin-lines (&optional (radius 4) (acceptable 1) (slice '(0 0)))
  (*nowarn
    (*warm-boot)
    (*with-vp-set *osr-image-set*
    (*all
      (*let ((places (plusp!! (aref!! *last-image-to-cm* (!!i (car slice)) (!!i (second slice)))))
             (thinned nil!!))
        (*type ((places thinned) t))
        (image-to-fb places)
        (thin-image places thinned)
        (image-to-fb thinned)
        (*let ((line-places nil!!)
               (slope (!! 0))
               (goodness (!! 0))
               (intercept (!! 0)))
          (*type ((places line-places) t) ((slope goodness intercept) ^))
          (line-through-connected-region thinned radius line-places slope
                                         intercept goodness acceptable)
          (explore-line places line-places slope goodness)))))))

(defun explore-line (places line-places slope goodness)
  (format t "~%(I)mage G(oodness) S(lope) L(ine places) or E(xit)
            #=slope threshold B(ad lines out) C(olored lines)")
  (loop for instruction =  (scl:send *query-io* :tyi)
        with numbers = '(#\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9)
        with threshold = 2
        do
    (*let ()
      (*type ((slope goodness) ^) ((places line-places) t))
      (when (member instruction numbers)
        (setq threshold (position instruction numbers)))
      (case instruction
        (#\e (return-from explore-line nil))
        (#\i (image-to-fb places))
        (#\g (grey-scale-to-fb goodness (* threshold 10)))
        (#\l (image-to-fb line-places))
        (#\s (grey-scale-to-fb (abs!! slope) threshold))
        (#\b (image-to-fb (<!! goodness (!! 1.0))))
        (#\c (*let* ((them (<!! goodness (!! 1.0)))
                     (color (color-blobs them)))
               (normalize-colors line-places color color)
               (rainbow-scale-to-fb color)))))))
```

7.2.2 Thinning

This is an implementation of the thinning algorithm of Zhang and Suen [1984], as described in Digital Image Processing, by Gonzalez and Wintz, pages 398 to 402. The transform takes an image and *thins* it by eroding the edges of blobs, such that connectivity, number of holes, and edges of lines are preserved.

```
(defun thin-image (thick thinned
                   &aux (neighbor-offsets
                         '((0 -1) (1 -1) (1 0) (1 1) (0 1) (-1 1) (-1 0) (-1 -1))))
  (*all
    (*let (neigbors number-of-neigbors number-of-transitions old-thinned)
      (*type :fast (old-thinned t) ((number-of-transitions number-of-neigbors) 4)
             (neigbors (@ 8 t)) ((thick thinned) t))
      (macrolet ((not-all (field &rest bits)
                   '(not!! (and!! ,@(loop for b in bits collecting '(aref!! ,field (!! ,b))))))
                 (proper-size () '(and!! (>=!! number-of-neigbors (!! 2))
                                         (<=!! number-of-neigbors (!! 6))))
                 (one-transition-only () '(=!! number-of-transitions (!! 1))))
        (flet ((count-stuff ()
                 (loop for (dx dy) in neighbor-offsets and count from 0
                       do (*setf (aref!! neigbors (!!i count)) (news!! thinned dx dy)))
                 (*set number-of-neigbors (count!! t!! neigbors))
                 (*set number-of-transitions (!! 0))
                 (loop for count below 8 do
                   (*when (and!! (not!! (aref!! neigbors (!!i count)))
                                 (aref!! neigbors (!!i (logand (1+ count) 7))))
                     (*incf number-of-transitions)))))
          (*set thinned thick)
          (loop do
            (*set old-thinned thinned)
            (count-stuff)
            (*when (and!! (proper-size) (one-transition-only)
                          (not-all neigbors 0 2 4)
                          (not-all neigbors 2 4 6))
              (*set thinned nil!!))
            (count-stuff)
            (*when (and!! (proper-size) (one-transition-only)
                          (not-all neigbors 0 2 6)
                          (not-all neigbors 0 4 6))
              (*set thinned nil!!))
            (and moviep (image-to-fb thinned))
                until (*and (eq!! old-thinned thinned)))))))))
```

7.2.3 Creating Local Connected Area Maps

The nearest neighbor array is calculated outside the main loop, avoiding the repeated costs of checking for boundary points. The ***with-local-grid** sets up an bitmap in each processor containing a neighborhood of the specified radius.

```
(defun neighbor-offsets (connectivity)
  (ecase connectivity
    (:eight '((0 . -1) (1 . -1) (1 .  0) (1 . 1) (0 . 1) (-1 . 1) (-1 . 0) (-1 . -1)))
    (:four '((0 . -1) (1 .  0) (0 . 1) (-1 . 0)))))

(defcached nearest-neigbors-array ((width height connectivity))
  (let ((array (make-array (list width height))))
    (loop for col below width do
      (loop for row below height do
        (loop for (neighbor-offset-col . neighbor-offset-row) in (neighbor-offsets connectivity)
              for neighbor-column = (+ col neighbor-offset-col)
              for neighbor-row = (+ row neighbor-offset-row)
              when (and (not (minusp neighbor-column)) (not (minusp neighbor-row))
                        (< neighbor-column width) (< neighbor-row height))
                do (push (cons neighbor-column neighbor-row) (aref array col row)))))
    array))

(defmacro *with-local-grid ((grid radius &optional (type 'boolean) (initially 'nil!!)) &body body)
  `(let ((array-dimension (1+ (* 2 ,radius)))
         (center-row ,radius)
         (center-column ,radius))
    (ignore center-row center-column)
    (*let (,grid)
      (declare (type (pvar (array ,type (array-dimension array-dimension))) ,grid))
      (dotimes (i (1+ (* 2 ,radius)))
        (dotimes (j (1+ (* 2 ,radius)))
          (*setf (aref!! ,grid (!!i i) (!!i j)) ,initially)))
      ,@body)))

(defun compute-locally-spiral-connected-regions (places radius connected?-grid
                                                 &aux (grid-dimension
                                                        (*array-dimension connected?-grid 0)))
  (*locally (*type (places t))
    (check-type connected?-grid (pvar (array boolean)))
    (*all
      (*let ((neighboring-connected? nil!!)
             (moved-places places))
        (*type :safe ((neighboring-connected? moved-places) t)
               (connected?-grid (@ grid-dimension grid-dimension t)))
        (*setf (aref!! connected?-grid (!!i radius) (!!i radius)) places)
        (loop for (delta-row delta-column) in  (spiral-deltas radius)
              with current-row = radius
              with current-column = radius
              with diameter = (1+ (* 2 radius))
              with nearest-neigbors = (nearest-neigbors-array diameter diameter :eight)
              do
          (setq  current-column (+ current-column delta-column))
          (setq current-row (+ current-row delta-row))
          (*set moved-places (news!! moved-places delta-column delta-row))
          (*all (*set neighboring-connected? nil!!))
          (*when moved-places
            (loop for (neighbor-column . neighbor-row) in (aref nearest-neigbors current-column current-row)
                  do
              (*set neighboring-connected?
                    (or!! neighboring-connected?
                          (aref!! connected?-grid (!!i neighbor-column) (!!i neighbor-row)))))
            (*setf (aref!! connected?-grid (!!i current-column) (!!i current-row)) neighboring-connected?))
          )))))
```

7.2.4 Line fitting

Calculate a line through the locally connected region, and throw away points which aren't line-like.

```
(defun line-through-connected-region (places radius line-places slope intercept goodness
                                      &optional (acceptable-distance 1.0)
                                      &aux (diameter (1+ (* 2 radius)))
                                      (area (* diameter diameter))
                                      (area-length (integer-length area)))
  (setq acceptable-distance (float acceptable-distance))
  (*all
    (*locally (*type :safe ((slope intercept) ^) ((places line-places) t))
      (*let ((sx (!! 0)) (sy (!! 0)) (sxx (!! 0)) (sxy (!! 0)) (syy (!! 0))
             (n (!! 0)))
        (*type ((sx sy sxx sxy syy goodness) ^) (n area-length))
        (*when places
          (*with-local-grid (connected? radius)
            (compute-locally-spiral-connected-regions places radius connected?)
            (loop for column-index below diameter
                  for column = (float (- column-index radius))
                  do
              (loop for row-index below diameter
                    for row = (float (- row-index radius))
                    do
                (*when (aref!! connected? (!!i column-index) (!!i row-index))
                  (accumulate-sums sx sy sxx syy sxy n column row))
                  ))))
        (*let ((denom (!! 0)))
          (*type (denom ^))
          (*when (>!! n (!!i 1))
            (slope-and-intercept sx sy sxx syy sxy n denom slope intercept))
          (*set line-places (and!! (>!! n (!!i 1)) (<!! (abs!! intercept) (!!f acceptable-distance)))))
        (*if line-places
           (average-chi-squared goodness sx sy sxx syy sxy n slope intercept)
           (*when (not!! line-places) (*set goodness (!!f (*all (*when line-places (*max goodness))))))))
      (*when (not!! line-places)
        (*set slope (!!f *nothing*)))
      )))

(defun accumulate-sums (sx sy sxx syy sxy n xcoord ycoord)
  (*locally (*type ((sx sy sxx syy sxy) ^) (n %n))
    (*set sx (+!! sx (!!f xcoord)))
    (*set sy (+!! sy (!!f ycoord)))
    (*set sxx (+!! sxx (!!f (* xcoord xcoord))))
    (*set sxy (+!! sxy (!!f (* xcoord ycoord))))
    (*set n (1+!! n))
    (*set syy (+!! syy (!!f (* ycoord ycoord))))))
```

```
(defun slope-and-intercept (sx sy sxx syy sxy n denom slope intercept)
  (ignore syy)
  (*locally (*type :fast ((sx sy sxx syy sxy denom slope intercept) ^) (n %n))
    (*set denom (*!! n sxx))
    (*decf denom (*!! sx sx))
    (*if (plusp!! denom)
        (progn
          (*set slope (*!! n sxy))
          (*decf slope (*!! sx sy))
          (*set slope (/!! slope denom))
          (*set intercept (*!! sxx sy))
          (*decf intercept (*!! sx sxy))
          (*set intercept (/!! intercept denom)))
        (progn
          (*set intercept (!! 0))
          (*set slope (most-positive-float!! slope))))))

(defun average-chi-squared (goodness sx sy sxx syy sxy n slope intercept)
  (*locally (*type :safe ((sx sy sxx syy sxy slope intercept goodness) ^) (n %n))
    (*all (*set goodness (!! 0)))
    (*if (/=!! slope (most-positive-float!! slope))
        (progn
          (*set goodness (*!! intercept intercept n))
          (*incf goodness (*!! (!! 2) slope intercept sx))
          (*incf goodness (*!! slope slope sxx))
          (*incf goodness (*!! (!! -2) intercept sy))
          (*incf goodness (*!! (!! -2) slope sxy))
          (*incf goodness syy)
          (*set goodness (/!! goodness n)))
        (progn
          (*set goodness (!!f (*all (*max goodness)))))))))
```

Chapter 8

Assembly

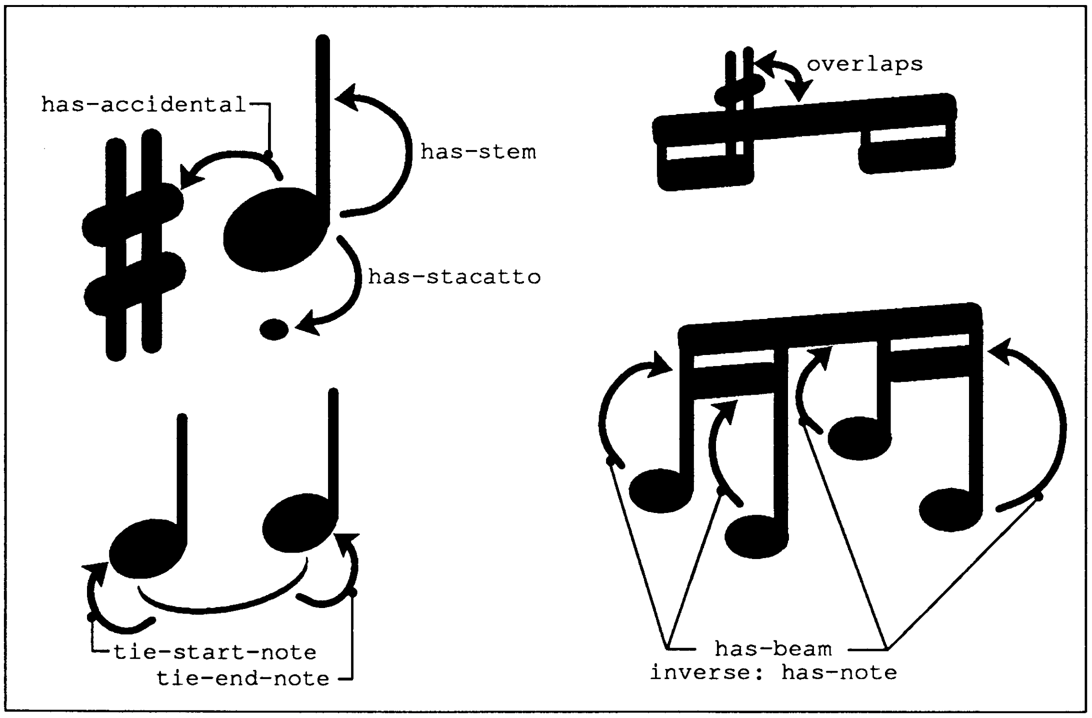


Figure 8-1: Typical relations that exist between features in a music score.

8.1 Introduction

Assembly is the process of taking the locations and types of features and translating them into a representation in which the relationships between different features are explicit. This representation should not be viewed as an ultimate "music representation" language, but as one representation in a series which enables a performer to go from looking at a score to playing it:

- The representation arrived at thus far, namely in terms of position and shape.
- A representation of the performers understanding of what the music score is communicating.
- A representation of how the performer intends to play the music.
- The motor control script which directs the performer's muscles to play the music.

This chapter's work aims at recovering something like the second representation. Some examples of things which are explicit at this level are: The pitch, duration type, and beat number of a note, the grouping of the note (whether it is part of a beamed group or is simply flagged), the starting point, and sometimes the extent of score to which a dynamic marking refers, the key signature, the time signature, what part of a score the repeat sign signifies should be repeated. Examples of things which are not explicit at this level: How loud to play each note, exactly how long to play the note, where to breath (when using a wind instrument) how to bow (for stringed instruments), how a fermata should be played.

Consider what information needs to be used in order to make explicit the aforementioned relationships. To compute what the pitch of a note is we need the clef, the key signature, whether an accidental modifies the note-head, and need to know which line or space of the staff line the note lies on. To know the duration of a note, we need the shape of the head, whether or not there is a flag (information which is determined separately this far), or whether the note is part of a beam, and whether there is a dot beside it. For each dynamic we need to know whether the dynamic operates on music below it as a trill does, or above it as a forte does, and whether it has a known extent, as a tie does and the word crescendo doesn't. How does one take the positions of features and turn in to the symbolic links which describe these relationships?

The problem is further complicated because not all features are found with 100 per cent reliability, a byproduct of the ambiguity of some shapes, and the difficulty of writing perfect feature finders. These errors are obvious to a person skilled in reading music. A sharp is found on top of a beam, a note is found at the intersection of a staff and measure line. How does one recognize and rule out these "obvious" mistakes?

In working with the various score structures, I categorize them by specifying the *objects* which form their parts, and the *relationships* between the parts and the whole. For example, a beamed group of notes has as components note heads and beam. The relationship between them is that the height of the beam above the note helps define the duration of the note. The that beam is above the notes (or below), defines a positional constraint which can be used to check for correctness.

Some visual relationships merit the status of being recognized, even though they do not form part of some musical structure. For instance, the sharp feature locator commonly makes the mistake of finding a sharp on top of where a beam is located. The consequence of this visual relationship is that either the sharp, the beam, or both have been mistakenly found.

I designed two data structures to help facilitate the recognition and representation of relationships such as these. The first I call *surroundings*, a structure which enables efficient queries by objects about others in some prescribed area. The second is the *relation* data structure. A relation is an object which records the participants in, and type of, functional or visual relationship.

8.2 Surroundings

A note has an accidental if one of five accidental shapes is not to far to the left of it, and the center line of the accidental is aligned with the center of the head, as in Figure 8-2. To determine whether or not this accidental exists, the note head needs to check its surroundings for a list of features which satisfy the type and positional constraints.

A simple strategy for satisfying this query would be to have a list of all objects, and upon a query, loop through this list, testing the type and position of each to see if it satisfies the request. But there are thousands of features on a single page of music, each of which may make many queries in the process of creating relationships. Were this implementation strategy chosen the result would be a very slow process.

Instead surroundings are implemented as a set of quad trees, one for each type of object. A quad tree is a data structure which allows the efficient computation of which of a set of rectangles intersects a given rectangle. Surroundings serve the role of responding to queries asking for specified kinds of object which occur in a given area.

8.3 Relations

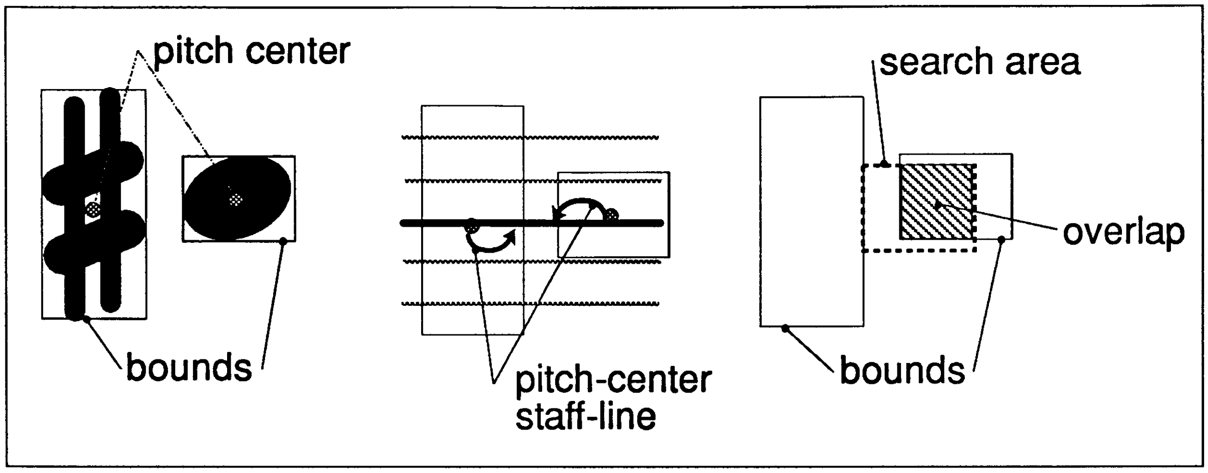


Figure 8-2: There are two constraints on the relationship between an accidental and the note it modifies. The *pitch-centers* need lie on the same line or space, and the the note-head need to be nearby and to the right.

A *relation* is an object(in the sense of object oriented programming) in which the a relationship between two features is stored. Relations are designed to handle a number of specific issues involving the reconstruction of the score from its parts. The relation stores information about the participants in the relationship, methods for checking whether the relationship exists given one of the participants. A relation can be noted as being ambiguous, for example in the case where a spurious note head is found, and a single accidental is near two note heads. Relations have procedures attached to them which take effect when the relationship is created. Relations can have inverses defined, which are automatically created. Whenever we note that a beam modifies a note head, the note is notified that it is part of the beamed group.

8.4 Disambiguation and Propagation of Relations

A specific method may be used to attempt to disambiguate an ambiguous relation. The method may ask for other information in the surroundings, or for information about other relations which have been established with the idea of figuring out which of a set of relatees may be considered invalid.

This can not automatically be done when a relationship is initially determined to ambiguous because at that time the information necessary to disambiguate the relationship may not be available. Take, for example, the the problem of deciding which notes belong to a given measure.

There are two problems that occur in this part of the reconstruction. First, some notes may lie between staffs, but may not have stem information. When a note is asked to establish a *my-staff* relation there may be two staffs which are sufficiently close to it to be considered. But one and only one measure may consider a note to be part of it.

A second problem is spuriously found note-heads. One way of deciding that such a note-head doesn't make sense is to add up all the notes in a measure and see if they add up to the time signature. If there are too many notes then the note may be removed from consideration. A good note to remove from consideration is one which may belong to some other measure.

The point that I want to make is that the computation of which measure a note may belong to proceeds along different paths. One path needs to be done first. The disambiguate method might find that it doesn't have enough information to decide now, but might have enough information later. Conceptually, we imagine that the disambiguate method is retried repeatedly throughout the calculation of other score properties, until the point it succeeds in disambiguating the relation.

Another way in which relations may affect their surroundings is by a *propagate* method. Suppose at some point a note is removed from consideration for some reason. Then we know that any relationships which depends on it being really part of the score should now be notified that this is no longer the case. An example is in the case of stems. If it is decided that a note has a stem, and if until that point it is uncertain which staff a note belongs to, then the the direction of the stem is used to make the decision. The propagate method for the stem relation encapsulates this knowledge.

Whenever there is some computation to be done, there is the question of what needs to be done first, what needs to be done before something else is done, and so on. I am trying to stay away from any order of evaluation dependency in this work on relations, since it makes it simpler to conceptualize. I want to be able to specify the relationships between objects, be able to write code to express these specifications, but to not have to worry as much about which relation is done first, particularly in the light of the fact that the process of disambiguating all relations may have to be iterative.

What is aimed for is a state of affairs in which demands for information from the score trigger off demands for the necessary establishment of positional relationships, the establishment of which which gives information which can satisfy the demands of the query. To this end, relations are established on demand and cached. A primitive called *force-relation* causes a relation to be computed the first time, and a provision is made for cases where this demand cannot be satisfied in that a function *cant-force-relation* is called. This method can take into account that fact that a portion of computation is not possible at a given time, and define a strategy for proceeding.

8.5 Related Work

This work was heavily inspired by David Waltz's[14] work on interpreting line drawings of polyhedra. That work focussed on computing several properties of lines in the drawings - whether or not both of the faces joined by the line were visible in the image, whether the line was part of a shadow a crack, or a polyhedra, and whether the line was at a concave or convex joinder of the faces. To do this he categorized all the various shapes of vertices of lines which can occur and the constraints which each vertex had on what types of lines could have formed it.

In a fully interpreted image, each of the lines had to have a single interpretation. The constraints are used to limit the possibilities for the pairs of interpretations of vertices on either side of the line. Impossible combinations are removed. When a vertex is removed from consideration by the constraints on one line, any interpretations of adjacent lines which used it would also need to be removed. Thus the constraints at one vertex propagate to others. At the end of this process either a single interpretation of the image is left, in which case it is accepted, or several interpretations are possible, as with an optical illusion, or none wass possible, in which case the drawing does not representation a physical object.

There are similarities and differences between his work and mine. In his case there were only three types of things in the world - lines vertices, and line labelings. In the music score there are many types of objects, with many type of relations. What is similar is that there may be ambiguity in the interpretation of some position of the score, which can be cleared up by constraints imposed on the interpretation of objects elsewhere in the score. Though there are more type of objects in the music score, the degree of ambiguity is less than in the line drawing world.

Waltz's work did not deal with the problem of noise and incorrectly found features. In personal discussion with Waltz, we talked about what happened if the constraints were not completely satisfiable, even though a labelling of the vertices existed. In that case each of the labeling of vertices needed to be changed to other plausible candidates and the constraint propagation run from the start to see if a consistent labeling could be found.

In this work the problem noise is considered just another potential labeling of an object. Thus the consequence of some constraint propagation might be that an object is considered noise, and this labeling might have consequences for the surroundings of the object.

8.6 Future Work

The work described in this chapter is not yet completed. I believe the basic mechanism which I outline here is adequate for the task, and illustrate the solution of several grouping problems using the mechanism. Still, the work should be viewed as work in progress - it represents new ground, and more than other methods described in this thesis, could very much use a second pass at the both framework and the implementation.

At the time that I developed the implementation of the relation data structures, I had little experience with knowledge representation systems. Given that I have an better understanding of the inference requirements of the score relationships, it might make more sense to implement these in an existing knowledge representation system.

8.7 The Code

8.8 Some Implemented Relations

8.8.1 Attaching Flags

```
(defflavor flag-relation () (relation))

(defflavor flagged-by () (relation))

(defmethod (inverse flag-relation) () 'flagged-by)
```

```
(defflavor flags-downward () (flag-relation))

(defflavor flags-upward () (flag-relation))

(defmethod (search-for-related flag-relation) ()
  (map-bounds-overlap-type
     relator      ; my feature has surroundings
     self      ; give self a bounds method (which depends on relator)
     (λ(ob) (establish self ob))
     #.(find-feature :solid)
     #.(find-feature :half)      ; can attach to solid or half
     )
  t)

(defmethod (tolerences flag-relation) ()
  (declare (values flag-to-head near-head x-tolerence))
  (values #.(* 4 (send (find-feature :solid) :height))
          #.(send (find-feature :solid) :height)
          #.(* .5 (send (find-feature :solid) :width))))

(defmethod (bounds flags-downward) ()
  (multiple-value-bind (x y)
      (point-position relator :flag-tip)
    (multiple-value-bind (flag-to-head above-head x-tolerence) (tolerences self)
       (values (- y flag-to-head) (- x x-tolerence) (+ y above-head) (+ x x-tolerence)))))

(defmethod (bounds flags-upward) ()
  (multiple-value-bind (x y)
      (point-position relator :flag-tip)
    (multiple-value-bind (flag-to-head below-head x-tolerence) (tolerences self)
       (values (+ y below-head) (- x x-tolerence) (+ y flag-to-head) (+ x x-tolerence)))))
```

8.8.2 Attaching Accidentals

An accidental is related to a note if it is in bounds, and if it is on the same line

```
(defflavor accidental-relation () (relation))

(defflavor modified-by-accidental () (relation))

(defmethod (inverse accidental-relation) () 'modified-by-accidental)

(defmethod (search-for-related accidental-relation) ()
  (force-relation relator 'my-staff)      ; make sure I have a staff relation
  (map-bounds-overlap-type
     relator      ; my feature has surroundings
     self      ; give self a bounds method (which depends on relator)
     (λ(ob) (relate-to-staff self ob))
     #.(find-feature :solid)
     #.(find-feature :half)
     #.(find-feature :whole)
       )
  t)
```

```
(defmethod (relate-to-staff accidental-relation) (ob)
  (let ((staffr (force-relation ob 'my-staff)))
    (multiple-value-bind (my-x my-y) (some-point-position relator :pitch-center)
      (multiple-value-bind (ob-x ob-y) (point-position ob :pitch-center)
        (map-cross-possibilities
          staffr (find-relation relator 'my-staff)
          (λ(obstaff mystaff)
            (and (eq obstaff mystaff)
                 (= (pitch-position mystaff my-x my-y) (pitch-position obstaff ob-x ob-y))
                 (establish self ob)))))))))

(defmethod (bounds accidental-relation) ()
  (multiple-value-bind (top ignore bottom right)
      (bounds relator)
    (values top right
            bottom (+ right #.(send (find-feature :solid) :width)))))
```

8.8.3 Relationship Between a Staff and its Constituents

```
(defflavor my-staff () (relation))

(defmethod (search-for-related my-staff) ()
  (multiple-value-bind (top ignore bottom ignore)
      (bounds relator)
    (catch 'done
      (flet ((check (staff)
               (let ((inside? (and (<= top (bottom staff)) (>= bottom (top staff)))))
                 (if inside?
                     (progn (exclusive self staff) (throw 'done t))
                     (establish self staff)))))
        (map-bounds-overlap-type
          relator
          relator
          #'check
          :staff-influence)
        t))))

(defmethod (propogate my-staff) ()
  (when (ambiguous? self)
    (propogate (force-relation relator 'has-stem-upwards))))

(defmethod (disambiguate my-staff) (which-stays)
  (when (and (= (length relatee) 2))
    (if (symbolp which-stays)
      (let ((pos1 (top (car relatee)))
            (pos2 (top (second relatee))))
        (let ((remove
                (when (neq pos1 pos2)
                  (case which-stays
                    (:upper (if (< pos1 pos2) (car relatee) (second relatee)))
                    (:lower (if (< pos1 pos2) (second relatee) (car relatee)))))))
          (when remove
            (unestablish self remove) t)))
      (unestablish self (find which-stays relatee :test 'neq))
      t)))
```

8.8.4 Relationship between a Stem and a Note-head

Propogating this relationship involves resolving any ambiguity about which staff a note is associated with

```
(defflavor has-stem () (relation))

(defflavor has-stem-upwards () (has-stem))

(defflavor has-stem-downwards () (has-stem))

(defflavor stem-of () (relation))

(defmethod (inverse has-stem) () 'stem-of)

(defmethod (search-for-related has-stem) ()
  (map-bounds-overlap-type
     relator      ; my feature has surroundings
     self     ; give self a bounds method (which depends on relator)
     (λ(ob) (establish-if-direction-ok self ob))
     :stem)
  t)

(defmethod (establish-if-direction-ok has-stem) (ob)
  (when (direction-ok? self ob)
     (establish self ob)))

(defmethod (direction-ok? has-stem-upwards) (stem)
  (< (bottom stem) (bottom relator)))

(defmethod (direction-ok? has-stem-downwards) (stem)
  (< (top relator) (top stem)))

(defmethod (offset has-stem) ()
  "amount to offset the box (1/2 dimensions )"
  (declare (values flag-to-head near-head x-tolerence))
  (values #.(* .5 (send (find-feature :solid) :height))
          #.(* .5 (send (find-feature :solid) :width))))

(defmethod (bounds has-stem-upwards) ()
  "a box displaced to upper right of head"
  (multiple-value-bind (top left bottom right) (bounds relator)
     (multiple-value-bind (offsety offsetx) (offset self)
          (values (- top offsety) (+ left offsetx) (- bottom offsety) (+ right offsetx)))))

(defmethod (bounds has-stem-downwards) ()
  "a box displaced to lower left of head"
  (multiple-value-bind (top left bottom right) (bounds relator)
     (multiple-value-bind (x y) (point-position relator :center)
        (let ((offsetx (- x left))
              (offsety (- y top)))
          (values (+ top offsety) (- left offsetx) (+ bottom offsety) (- right offsetx))))))

(defmethod (propogate has-stem) ()
  (let ((staff-relation (force-relation relator 'my-staff)))
    (when (ambiguous? staff-relation)
       (let* ((opposite (force-relation relator (opposite-direction-stem self)))
              (me-defined? (defined? self))
              (opposite-defined? (defined? opposite)))
         (cond ((and me-defined? (not opposite-defined?))
                 (disambiguate staff-relation (staff-emphasized self)))
                ((and (not me-defined?) opposite-defined?)
                 (disambiguate staff-relation (staff-emphasized opposite)))
                (t nil))))))

(defmethod (opposite-direction-stem has-stem-upwards) ()
  'has-stem-downwards)
```

```
(defmethod (opposite-direction-stem has-stem-downwards) ()
  'has-stem-upwards)

(defmethod (staff-emphasized has-stem-upwards) ()
  :lower)

(defmethod (staff-emphasized has-stem-downwards) ()
  :upper)

(defmethod (bounds stem-feature) ()
  (values top left (+ top height) (+ left 4)))

(defmethod (stem-information page-of-stems) (head &optional (htolerance 5) (vtolerance 5) &aux (direction :none))
  (assure-stem-positions self)
  (flet ((check-sense (expect stem &rest ignore)
         (when (eq (cdr stem) expect)
           (if (eq direction :none)
               (setq direction expect)
               (unless (eq direction expect) (setq direction :both))))))
    ; left side (expect down)
    (multiple-value-call
       'map-over-objects-in-range positions
       (λ(&rest args) (apply #'check-sense :down args))
       htolerance vtolerance (point-position head :left))
    ; right side
    (multiple-value-call
       'map-over-objects-in-range positions
       (λ(&rest args) (apply #'check-sense :up args))
       htolerance vtolerance (point-position head :right)))
  direction)
```

8.8.5 Measures and their Constituents

The current propogation of measure is that if you consider the box containing all unambiguous heads, then any heads still contained in that box are likewise contained, and we can disambiguate them. Possibly there are more propogations to be done. Disambiguating involves removing any extraneous measure relations

```
(defflavor measure-contains () (relation))

(defflavor in-measure () (relation))

(defmethod (inverse measure-contains) () 'in-measure)

(defmethod (ambiguous? measure-contains) ()
  (unless (undefined? self)
    (some (λ(ob) (ambiguous? (find-relation ob 'my-staff))) relatee)))

(defmethod (search-for-related measure-contains) ()
  (apply 'map-bounds-overlap-type
     relator       ; my feature has surroundings
     self     ; give self a bounds method (which depends on relator)
     (λ(ob) (establish-if-same-staff self ob))
     *heads*)
  t)
```

```
(defmethod (bounds measure-contains) ()
  (influence-bounds relator))

(defmethod (establish-if-same-staff measure-contains) (ob)
  (when (related? (force-relation ob 'my-staff) (staff relator))
    (establish self ob)))

(defmethod (propogate measure-contains) ()
  (resolve-ambiguous-head-staffs self))

(defmethod (resolve-ambiguous-head-staffs measure-contains) ()
  ;; make a box around all the unambiguous note heads
  ;; then disambiguate any ambiguous note heads which are inside that box
  (unless (undefined? self)
    (multiple-value-bind (encloses ambiguous)
        (box-enclosing-unambiguous self)
      (loop for head in ambiguous
            with top and bottom and left and right
            do (multiple-value-setq (top left bottom right) (bounds head))
            if (box-overlaps-region-p encloses left top right bottom)
              do
                (disambiguate self head)))))

(defmethod (disambiguate measure-contains) (what-stays)
  ;; what-stays is a formerly staff ambiguous head which we decied belongs with us
  ;; in order to disambiguate, we need to
  ;; 1) Tell other measures it may be part of that it now isn't part of them
  (let ((measures (find-relation what-stays 'in-measure)))
    (map-possibilities measures
        (λ(measure)
          (unless (eq measure relator)
            (unestablish (find-relation measure 'measure-contains) what-stays)))))
  ;; 2) Tell head that it has a single staff now
  (disambiguate (find-relation what-stays 'my-staff) (staff relator)))

(defmethod (box-enclosing-unambiguous measure-contains) ()
  (loop for head in relatee
        for staff = (find-relation head 'my-staff)
        with top and bottom and left and right
        if (ambiguous? staff)
          collect head into ambiguous
        else do
              (multiple-value-setq (top left bottom right) (bounds head))
            and minimize top into top! and maximize bottom into bottom!
        finally (return (values
                          (make-instance 'box :top  top! :left (from relator) :bottom bottom! :right  (to relator))
                          ambiguous))))
```

8.8.6 Overlap Relations

These are in order to compensate for features of more than one type which are found at a single place. We are going to first collect the set of selfs which overlap me. Then, Any features which are considered to be superlfluous by virtue of being overlapped, will get the overlapped-by-self relation, and be put on self-overlaps of the overlapper. This means that the computation for the set occurs the first time that the relation gets forced on one of them. So in establishing the relation, we won't look at anything which has either of these relations already on it. I really don't want to have this information around on each of the features : too much junk. Oh well. keep it for now.

```
(defflavor overlapped-by () (relation))

(defflavor overlaps () (relation))

(defflavor self-overlaps () (overlaps))

(defflavor overlapped-by-self () (overlapped-by))

(defmethod (search-for-related self-overlaps) ()
  (unless (or (find-relation-of-type relator 'overlapped-by-self))
    (let ((overlapping-set (list relator)))
      (map-bounds-overlap-type
         relator      ; my feature has surroundings
         relator      ; bounds of my relator
         (λ(ob) (and (not (find-relation ob (type-of self))) ; because there might be specializations of this relation
                         (pushnew ob overlapping-set)))
         (send relator :my-feature))
      (when (cdr overlapping-set)
         (set-up-relations self (find-dominator self overlapping-set) overlapping-set))))
  t)

(defmethod (set-up-relations self-overlaps) (dominator all)
  ;; so we come in with an empty self-overlaps and no overlapped-by
  ;;
  (let ((dom-relation (establish-relation dominator (type-of self))))
    (mapc (λ(o)
              (unless (eq dominator o)
                 (establish dom-relation o)
                 (establish (establish-relation o 'overlapped-by-self) dominator)))
           all)))
```

8.8.7 Overlap of Sharps

First pass at sharp, a common error mode

```
(defflavor self-overlaps-sharp () (self-overlaps))

(defmethod (find-dominator self-overlaps-sharp) (sharps)
  (break)
  (loop for s in sharps
        with min = most-positive-fixnum and uppermost
        for top = (top s)
        do
    (when (< top min)
       (setq uppermost s)
       (setq min top))
    finally (return uppermost)))
```

8.8.8 Current Top Level Test

```
(defvar *sharps* (list (find-feature :sharp) (find-feature :flat)
                       (find-feature :natural)))
```

```
(defvar *heads* (list (find-feature :solid) (find-feature :whole)
                      (find-feature :half)))

(defvar *upward-flags* (list (find-feature "eighth flag upward")
                             (find-feature "sixteenth flag upward")))

(defvar *downward-flags* (list (find-feature "eighth flag downward")
                               (find-feature "sixteenth flag downward")))

(defun test-relations ()
  (setq *surroundings* (make-instance 'surroundings :page 215 :book schubert))
  (trigger-relation *surroundings* 'accidental-relation *sharps*)
  (trigger-relation *surroundings* 'flags-upward *upward-flags*)
  (trigger-relation *surroundings* 'flags-downward *downward-flags*)
  (trigger-relation *surroundings* 'my-staff *heads*)
  (propogate-relation *surroundings* 'my-staff *heads*)
  (trigger-relation *surroundings* 'measure-contains (list :measure))
  (propogate-relation *surroundings* 'measure-contains (list :measure)))
```

8.9 Code Defining Behaviour of Relations

8.9.1 Definition of a Relation

```
(defflavor relation ((relatee *undefined*) relator) () :settable-instance-variables)

(defmethod (sys:print-self relation) (stream &rest ignore)
  (let ((status (cond ((undefined? self) "(undefined)") ((ambiguous? self) "(ambiguous)") (t nil))))
    (let ((*print-case* :downcase))
      (if status
        (format stream "#<~A ~A>" (type-of self) status)
        (format stream "#<~A => ~A>" (type-of self) (car relatee))))))
```

8.9.2 Default Methods for Relations

```
(defmethod (type relation) ()
   "Returns the type of relation"
  (type-of self))

(defmethod (inverse relation) ()
  "The inverse of this relation. By default it is also established, if extant. Default nil"
  nil)

(defmethod (establish-1 relation) (new-relatee)
  (if (eq relatee *undefined*)
    (setq relatee (list new-relatee))
    (pushnew new-relatee relatee)))

(defmethod (establish relation) (new-relatee &aux inverse)
  "Specifies that relatee satisfies the relation (The relation is between relator and relatee)
Relator is the object on which the relation sits. Default method just pushes it on relatees"
  (establish-1 self new-relatee)
  (when (inverse self)
    (establish-1 (setq inverse (establish-relation new-relatee (inverse self))) relator)
    (propogate self)
    (when inverse
      (propogate inverse))))

(defmethod (unestablish relation) (which)
  "Specifies that relatee doesn't satisfie the relation. Default method just removes it from relatees"
  (setq relatee (delete which relatee)))
```

```
(defmethod (undefined? relation) ()
  "Returns t if the relation is empty"
  (eq *undefined* relatee))

(defmethod (undefined? relation) ()
  "Returns t if the relation is empty"
  (eq *undefined* relatee))

(defmethod (ambiguous? relation) ()
  "Returns t is the relation is ambiguous"
 (and (consp relatee) (cdr relatee)))

(defmethod (arbitrary? relation) ()
  "When a relation is ambiguous, returns some object which satisfies the relation."
  (if (consp relatee)
    (car relatee)
    relatee))

(defmethod (disambiguate relation) (&rest info)
      "Have the relation disambiguate itself, given the information which is passed as info.
Though in most cases the diambiguation is done, it may not be for some reason
which the relation chooses. Return t if there was a disambiguation, and nil otherwise"
  (error "Don't know how to disambiguate ~A according to ~A" self info))

(defmethod (propogate relation) ()
"Have the relation do what it knows how to do in terms of using the information
which it has in order to affect other relations. This gets called after establish relation by default,
and should be called after any change to the relation. Default method does nothing"
nil)

(defmethod (disallow-1 relation) (which)
  (setq relatee (delete which relatee)))

(defmethod (disallow relation) (which)
  (disallow-1 self which)
  (when (inverse self)
     (let ((inverse-relation (find-relation which (inverse self))))
       (and inverse-relation (disallow-1 inverse-relation relator)))))

(defmethod (exclusive relation) (which)
  "Specifies that relatee satisfies the relation, and further is the sole relatee (throws away other possibilities)"
  (unless (undefined? self)
    (mapc (λ(r) (disallow self r)) relatee))
  (establish self which))

(defmethod (expected-that-relation-defined relation) ()
  (error "It was assumed that there is a least one relatee, but there is not"))

(defmethod (search-for-related relation) ()
  "Have the relation look at it's surroundings and establish relations with
objects which satisfy the relation criteria. Return t if succesfull. Default returns nil."
  nil)

(defmethod (related? relation) (what)
  "Check if objects is one of the related to objects of relation"
  (unless (undefined? self)
    (member what relatee)))

(defmethod (map-cross-possibilities relation) (other function)
  "For each pair consisiting of one possibility from relation and one from other-relation, call function on the pair"
  (unless (or (undefined? self) (undefined? other))
    (loop for r in relatee do
      (loop for r1 in (send other :relatee) do
        (funcall function r r1)))))
```

```
(defmethod (map-possibilities relation) (function)
  "map function over each of the possible relatees"
  (unless (undefined? self)
    (loop for r in relatee do
      (funcall function r))))
```

8.9.3 Pay Special Attention to Ambiguous Relations

```
(defmethod (trigger-relation-on-ambiguous surroundings) (relation-type ambiguous-relation object-types)
  (loop for type in object-types do
    (let ((area (find-area self type)))
      (map-all area
               (λ(o)
                  (when (ambiguous? (force-relation o ambiguous-relation))
                     (force-relation o relation-type)))))))

(defmethod (map-over-ambiguous-relations surroundings) (function relation-type &rest relator-types)
  (declare (values object relation))
  (loop for type in relator-types do
    (map-all (find-area self type)
             (λ(o) (let ((r (find-relation o relation-type))) (and r (ambiguous? r) (funcall function o r)))))))
```

8.9.4 Definition of Object which can have Relations

Things which may have a relation get the has-relations mixin.

```
(defflavor has-relations ((relations nil))
          () :settable-instance-variables)

(defmethod (set-relation has-relations) (type relation)
  "Sets the relation of type type to relation"
  (setq relations (delete type relations :key 'type-of))
  (push relation relations))

(defmethod (establish-relation has-relations) (type)
  (or (find-relation self type)
      (let ((relation (make-relation-type self type)))
        (set-relation self type relation)
        relation)))

(defmethod (find-relation has-relations) (type)
    "finds relation of whos type satisfies (typep relation type)"
  (find type relations :key 'type-of))

(defmethod (find-relation-of-type has-relations) (type)
  "finds relation of whos type satisfies (typep relation type)"
  (find-if (λ(ob) (typep ob type)) relations))

(defmethod (make-relation-type has-relations) (type)
  "Makes an instance of specified type of relation"
  (let ((it (make-instance type)))
    (assert (typep it 'relation) () "The flavor instantiated is not a relation")
    (send it :set-relator self)
    it))

(defmethod (force-relation has-relations) (type)
  "If the relation exists, returns it, otherwise tries its best to estabish it, eg. by search-for-related"
  (or (find-relation self type)
      (let ((new (establish-relation self type)))
        (or (search-for-related new)
            (cant-force-relation self type))
        new)))
```

```
(defmethod (cant-force-relation has-relations) (type)
  "A force-relation was tried, but no method succeeded"
  (error "Tried to force relation between ~A and ~A, but couldn't" self type))

(defmethod (kill-relation has-relations) (type)
  (let ((r (find-relation self type)))
    (when (and r (not (undefined? r)))
      (loop for relat in (send r :relatee) do
        (disallow r relat)))
    (setq relations (delete type relations :key 'type-of))))

(defmethod (kill-all-relations has-relations) ()
  (setq relations nil))
```

8.10 Implementation of Surroundings

8.10.1 Box and binning parameters

```
(defvar *osr-box-area* (make-area :name '*osr-box-area* :gc :ephemeral :n-levels 3 :capacity-ratio 2))

(defvar *area-threshold-default* .25
  "Bin here if object area less than this many times bin area. As you decrease this,
space per entry goes up, as you increase, probability of collision increases")
```

8.10.2 Definition of Boxes

```
(defflavor box (left top right bottom) ()
  :initable-instance-variables
  :writable-instance-variables
  (:ordered-instance-variables left top right bottom)
  (:default-init-plist :area *osr-box-area*)
  (:constructor make-box (left top right bottom)))

(defun boxes-overlap-p (box-1 box-2)
  (let ((left (max (box-left box-1) (box-left box-2)))
        (top (max (box-top box-1) (box-top box-2)))
        (right (min (box-right box-1) (box-right box-2)))
        (bottom (min (box-bottom box-1) (box-bottom box-2))))
    (when (and (> right left) (> bottom top))
      (values t left top right bottom))))

(defmethod (box-overlaps-region-p box) (other-left other-top other-right other-bottom)
  (and (let ((intersection-top (if other-top (max top other-top) top))
             (intersection-bottom (if other-bottom (min bottom other-bottom) bottom)))
         (> intersection-bottom intersection-top))
       (let ((intersection-left (if other-left (max left other-left) left))
             (intersection-right (if other-right (min right other-right) right)))
         (> intersection-right intersection-left))))

(defmethod (bounds box) ()
  (values top left bottom right))

(defmethod (position-string box-with-area) ()
  (format nil "x:~a,~a y:~A,~A" (floor left) (floor right) (floor top) (floor bottom)))
```

8.10.3 Definition of Boxes which Keep Track of their Area

```
(defflavor box-with-area (box-area) (box)
  (:default-init-plist :area *osr-box-area*))
```

```
(defmethod (make-instance box-with-area) (&rest ignore)
  (setq box-area (* (- right left) (- bottom top))))

(defmethod (area box-with-area) () box-area)

(defmethod (intersecting-area box-with-area) (other)
  (multiple-value-bind (valid left top right bottom)
      (boxes-overlap-p self other)
     (if valid
       (* (- left right) (- top bottom))
       0)))

(defmethod (box-overlaps-region-p box-with-area) (other-left other-top other-right other-bottom)
  (macrolet ((min1 (a b) '(if (> ,a ,b) ,b ,a)) (max1 (a b) '(if (> ,a ,b) ,a ,b)))
     (and  (> (min1 bottom other-bottom) (max1 top other-top))
           (> (min1 right other-right) (max1 left other-left)))))
```

8.10.4 Definition of a Boxes with Contents

```
(defflavor object-in-box ((object nil) id)
          (box-with-area)
  :settable-instance-variables
  (:constructor ob-in-box (object top left bottom right id)))

(defmethod (sys:print-self object-in-box) (stream &rest ignore)
  (format stream "#<~A in a box ~A>"
             (or (and (instancep object) (or (pretty-name object) (send-if-handles object :name)))
                 object) (position-string self)))
```

8.10.5 Define a Quad Tree Node

```
(defflavor area-quad
        ((topleft nil) (topright nil) (bottomleft nil) (bottomright nil)      ;the subquadrants
         (objects nil)      ;objects at this level
         (dominator :none)      ;sole object for all lower levels :none if empty, :expanded if there are
         area-threshold      ;fraction of are object should take to be saved at this level
         left-right-center top-bottom-center)      ;cached bisecting lines
        (box-with-area)      ;and of course it is a box
  :settable-instance-variables
  (:constructor make-subquad (top left bottom right)))

(defmethod (make-instance area-quad :after) (&rest ignore)
  (setq area-threshold (* *area-threshold-default* box-area))
  (setq left-right-center (* .5 (+ left right)))
  (setq top-bottom-center (* .5 (+ top bottom))))

(defmethod (sys:print-self area-quad) (stream &rest ignore)
  (let ((bcount (box-count self)))
    (format stream "#<area-quad ~A (~A instance~p)>" (position-string self) bcount bcount)))
```

8.10.6 Definition of Quad Tree Root

The top quad node is special, as it contains global information about the tree.

```
(defflavor top-level-area-quad ((count 0) (bvlength 64)
                                 (bitvector nil)
                                 (displacedbv nil)
                                 (growth-factor 2))
          (area-quad)
   (:constructor make-area-quad (top left bottom right))
   (:settable-instance-variables))
```

```
(defmethod (make-instance top-level-area-quad) (&rest ignore)
  (adjust-bitvector self))

(defmethod (sys:print-self top-level-area-quad) (stream &rest ignore)
  (let ((bcount (box-count self)))
    (if (zerop bcount)
      (format stream "#<top-area-quad ~A (empty)>"
              (position-string self))
      (format stream "#<top-area-quad ~A (~A instance~p, redundency ~,2f)>"
                (position-string self) bcount bcount (float (/ bcount count))))))

(defmethod (box-count area-quad) ()
  (+ (length objects) (if (symbolp dominator) 0 1)
     (if topleft (box-count topleft) 0)
     (if bottomright (box-count bottomright) 0)
     (if bottomleft (box-count bottomleft) 0)
     (if topright (box-count topright) 0)))
```

8.10.7 Intersection Tests

```
(defmethod (intersects-bottomright? area-quad) (box)
  (if bottomright
    (boxes-overlap-p bottomright box)
    (when (box-overlaps-region-p box left-right-center top-bottom-center right bottom)
      (setq bottomright (make-subquad top-bottom-center left-right-center bottom right))
      t)))

(defmethod (intersects-topleft? area-quad) (box)
  (if topleft
    (boxes-overlap-p topleft box)
    (and (box-overlaps-region-p box left top left-right-center top-bottom-center)
         (setq topleft (make-subquad top left top-bottom-center left-right-center))
         t)))

(defmethod (intersects-bottomleft? area-quad) (box)
  (if bottomleft
    (boxes-overlap-p bottomleft box)
    (when (box-overlaps-region-p box left  top-bottom-center left-right-center bottom)
      (setq bottomleft (make-subquad top-bottom-center left bottom left-right-center))
      t)))

(defmethod (intersects-topright? area-quad) (box)
  (if topright
    (boxes-overlap-p topright box)
    (when (box-overlaps-region-p box left-right-center top right top-bottom-center)
      (setq topright (make-subquad top left-right-center top-bottom-center right))
      t)))
```

8.10.8 The Binning Algorithm

```
(defmethod (bin-in-subquads area-quad) (ob)
  (when (intersects-bottomleft? self ob)
    (bin1 bottomleft ob))
  (when (intersects-topleft? self ob)
    (bin1 topleft ob))
  (when (intersects-topright? self ob)
    (bin1 topright ob))
  (when (intersects-bottomright? self ob)
    (bin1 bottomright ob)))
```

```
(defmethod (bin1 area-quad) (ob)
  (if (< (area ob) area-threshold)
    ;; this object is smaller than reasonable size for this quad
    (case dominator
      (:none (setq dominator ob))      ; there are no sublevels keep object here for now
      (:lower (bin-in-subquads self ob))      ; there are lower levels bin this lower down
      (otherwise      ; otherwise there is a current dominator
        (bin-in-subquads self dominator)      ; push dominator down
        (setq dominator :lower)      ; mark the node as having subquads
        (bin-in-subquads self ob)))      ; and bin self
    ;; object is similar size, so bin at this level
    (add-to-objects self ob)
    ))

(defmethod (bin top-level-area-quad) (object top left bottom right)
  (let* ((ob (ob-in-box object top left bottom right (incf count))))
    (adjust-bitvector self)
    (bin1 self ob)))

(defmethod (bin-many top-level-area-quad) (things-to-bin)
  "things to bin should support the bounds message"
  (loop for ob in things-to-bin do
    (multiple-value-call 'bin self ob (bounds ob))))

(defmethod (add-to-objects area-quad) (ob)
  (unless (member ob objects :test 'box-equal)
    (setq objects (cons-in-area ob objects *osr-box-area*))))
```

8.10.9 Routines to Map over a Range of Objects

Given a rectangular area, we want to call some function on each object which intersects that range.

```
(defmethod (map-over-objects-in-rectangle top-level-area-quad) (function in-box? top left bottom right)
  (prepare-bitvector self)
  (map-over-objects-in-rectangle-internal self function in-box? top left bottom right bitvector))
```

```
(defmethod (map-over-objects-in-rectangle-internal area-quad) (function in-box? top left bottom right seenv &aux
  (declare (sys:array-register seen))
  (macrolet ((access (object)
                `(if in-box? ,object (send ,object :object))))
     (when (and (not (symbolp dominator)) (box-overlaps-region-p dominator left top right bottom))
        (let ((count (send dominator :id)))
          (unless (plusp (aref seen count))
             (funcall function (access dominator))
             (setf (aref seen count) 1))))
     (when objects      ; checking first is a win
        (loop for o in objects do
          (when (box-overlaps-region-p o left top right bottom)
             (let ((count (send o :id)))
               (unless (plusp (aref seen count))
                  (funcall function (access o))
                  (setf (aref seen count) 1))))))
     (when (eq dominator :lower)
        (when (and topleft (box-overlaps-region-p topleft left top right bottom))
           (map-over-objects-in-rectangle-internal topleft function in-box? top left bottom right seen))
        (when (and topright (box-overlaps-region-p topright left top right bottom))
           (map-over-objects-in-rectangle-internal topright function in-box? top left bottom right seen))
        (when (and bottomleft (box-overlaps-region-p bottomleft left top right bottom))
           (map-over-objects-in-rectangle-internal bottomleft function in-box? top left bottom right seen ))
        (when (and bottomright (box-overlaps-region-p bottomright left top right bottom))
           (map-over-objects-in-rectangle-internal bottomright function in-box? top left bottom right seen)))))

(defmethod (map-over-objects-in-range top-level-area-quad) (function in-box? xrange yrange center-x  center-y)
  (map-over-objects-in-rectangle self
      function in-box?
      (- center-y yrange ) (- center-x xrange) (+ center-y yrange ) (+ center-x xrange)))
```

8.10.10 Routines to Map over all Objects

```
(defmethod (map-all top-level-area-quad) (function &optional in-box?)
  (prepare-bitvector self)
  (map-all-1 self function in-box? bitvector))

(defmethod (map-all-1 area-quad) (function in-box? seenv &aux (seen seenv))
  (declare (sys:array-register seen))
  (macrolet ((access (object)
                `(if in-box? ,object (send ,object :object))))
     (when objects
        (dolist (o objects)
           (unless (plusp (aref seen (send o :id)))
              (funcall function (access o))
              (setf (aref seen (send o :id)) 1))))
     (if (symbolp dominator)
        (when (eq dominator :lower)
           (and bottomright (map-all-1 bottomright function in-box? seenv))
           (and bottomleft (map-all-1 bottomleft function in-box? seenv))
           (and topright (map-all-1 topright function in-box? seenv))
           (and topleft (map-all-1 topleft function in-box? seenv)))
        (unless (plusp (aref seen  (send dominator :id)))
           (funcall function (access dominator))
           (setf (aref seen (send dominator :id)) 1)))))
```

```
(defmethod (map-all area-quad) (function &optional in-box?)
  (flet ((access (object)
         (if in-box? object (send object :object))))
    (when objects
      (dolist (o objects)
        (funcall function (access o))))
    (if (symbolp dominator)
      (when (eq dominator :lower)
        (and bottomright (map-all bottomright function in-box?))
        (and bottomleft (map-all bottomleft function in-box?))
        (and topright (map-all topright function in-box?))
        (and topleft (map-all topleft function in-box?)))
      (funcall function (access dominator)))))
```

8.10.11 Manipulation of the Bitvector

I use a bitvector to keep track of already mapped objects, since an object may be binned in more than one place.

```
(defmethod (prepare-bitvector top-level-area-quad) ()
  "set all entries to 0 (unseen)"
  (and displacedbv
       (fill displacedbv 0)))

(defmethod (adjust-bitvector top-level-area-quad) ()
  "resize the bitvector if necessary"
  (when (or (> count bvlength) (null bitvector))
    (setq bvlength (* (ceiling (if (> count bvlength) (* bvlength growth-factor) bvlength) 32) 32))
         (if bitvector
           (adjust-array bitvector bvlength)
           (setq bitvector (make-array bvlength :adjustable t
                                               :element-type 'bit :adjustable t :area *osr-box-area*)))
         (setq displacedbv (make-array (/ bvlength 32) :element-type 'fixnum :displaced-to bitvector))))
```

8.11 Code Implementing Surroundings

8.11.1 Definition of Surroundings

```
(defvar *surroundings*)

(defflavor surroundings (page book (areas nil) staffs systems image)
         ()
  :settable-instance-variables)

(defmethod (make-instance surroundings) (&rest ignore)
  (let ((data (get-data book page)))
    (setq image (send data :image))
    (setq staffs (send (send (send data :grid) :staff-lines) :staffs))
    (setq systems (send (send (send data :grid) :staff-lines) :systems))))

(defmethod (sys:print-self surroundings) (stream &rest ignore)
  (format stream "#<surroundings of page ~A from ~A>" page (pretty-name book)))

(defmethod (find-area surroundings) (type)
  (or (second (assoc type areas :test 'member))
      (maybe-compute-area self type)))
```

```
(defmethod (kill-area surroundings) (type)
  (setq areas (delete (assoc type areas :test 'member) areas)))
```

8.11.2 Method for Registering Staff in Surroundings

```
(defmethod (place-staff-influence surroundings) (staffs area image)
  (loop for ss on staffs
        for this = (car ss)
        for next = (second ss)
        with last
        for top = (or (and last (bottom last)) 0)
        for bottom = (or (and next (top next)) (send image :height))
        with right = (send image :width)
        do
    (bin area this top 0 bottom right)
    (setq last this)))
```

8.11.3 Method for Registering Systems

```
(defmethod (place-system-influence surroundings) (systems area image)
  (loop for ss on systems
        for this = (car ss)
        for next = (second ss)
        with last
        for top = (or (and last (bottom last)) 0)
        for bottom = (or (and next (top next)) (send image :height))
        with right = (send image :width)
        do
    (bin area this top 0 bottom right)
    (setq last this)))
```

8.11.4 Method for Registering Measures

```
(defmethod (place-measure-influence surroundings) (page image &aux (image-height (send image :height)))
  (let ((vmeasure-area (make-instance 'top-level-area-quad :top 0 :left 0
                                      :right (send image :width) :bottom (send image :height)))
        (measure-area (make-instance 'top-level-area-quad :top 0 :left 0
                                     :right (send image :width) :bottom (send image :height))))
    (flet ((place-a-measure (measure area)
             (let ((above (car (above measure)))
                   (below (car (below measure))))
               (bin area measure
                    (or (and above (bottom above)) 0)
                    (from measure)
                    (or (and below (top below)) image-height)
                    (to measure)))))
      (map-measures page (λ(m) (place-a-measure m measure-area)))
      (map-vmeasures page (λ(m) (place-a-measure m vmeasure-area))))
    (push '((:measure) ,measure-area) areas)
    (push '((:vmeasure) ,vmeasure-area) areas)))
```

8.11.5 Methods for Registering Fixed-sized Features and Beams

```
(defmethod (features-of-type surroundings) (type)
  (gethash (find-feature type) (send (send (get-data book page) :features) :result-table)))

(defmethod (place-feature surroundings) (features area)
  (loop for f in features do
    (multiple-value-call 'bin area f (bounds f))))

(defmethod (place-beam-influence surroundings) (beams beam-area)
  (loop for beam being the array-elements of (send beams :beams) do
    (multiple-value-call 'bin beam-area beam (bounding-box beam)))
  (push '((:beam) ,beam-area) areas))
```

8.11.6 Dispatch Creation of Surroundings by Type

```
(defmethod (maybe-compute-area surroundings) (type)
  (let* ((image (send (get-data book page) :image))
         (area (make-instance 'top-level-area-quad :top 0 :left 0
                              :right (send image :width) :bottom (send image :height))))
    (cond ((eq type :staff-influence)
           (place-staff-influence self staffs area image)
           (push (list '(:staff-influence) area) areas))
          ((eq type :system-influence)
           (place-system-influence self systems area image)
           (push (list '(:system-influence) area) areas))
          ((eq type :stem)
           (place-feature self (send (send (get-data book page) :stems) :them) area)
           (push (list '(:stem) area) areas))
          ((typep type 'feature-model)
           (place-feature self (gethash type (send (send (get-data book page) :features) :result-table)) area)
           (push (list (list type) area) areas))
          ((eq type :measure)
           (place-measure-influence self (make-measures book page) image))
          ((eq type :vmeasure)
           (place-measure-influence self (make-measures book page) image))
          ((eq type :beam)
           (place-beam-influence self (send (get-data book page) :beams) area))
          (t (error "didn't know how to compute area for ~A" type)))
    area))
```

8.11.7 Mapping over Portions of Surroundings

```
(defmethod (map-bounds-overlap-type surroundings) (object-with-bounds function &rest types)
  (loop for type in types do
    (let ((area (find-area self type)))
      (multiple-value-call 'map-over-objects-in-rectangle area function nil (bounds object-with-bounds)))))

(defmethod (map-point-overlap-type surroundings) (x y function &rest types)
  (loop for type in types do
    (let ((area (find-area self type)))
      (multiple-value-call 'map-over-objects-overlapping-point area function nil x y))))

(defmethod (map-over-objects-of-type surroundings) (function object-types)
  (loop for type in object-types do
    (let ((area (find-area self type)))
      (map-all area function))))

(defmethod (map-over-areas surroundings) (f)
  (loop for (types area) in areas do (funcall f area)))
```

8.11.8 Definition of Objects which can have Surroundings

```
(defflavor has-surroundings () ())

(defmethod (surroundings has-surroundings) () *surroundings*)

(defmethod (map-bounds-overlap-type has-surroundings) (object-with-bounds function &rest types)
  (apply 'map-bounds-overlap-type (surroundings self) object-with-bounds function types))

(defmethod (map-point-overlap-type has-surroundings) (x y function &rest types)
  (apply 'map-point-overlap-type (surroundings self) x y function types))
```

8.11.9 Operating on Relations in Surroundings

```
(defmethod (kill-all-relations surroundings) ()
  (loop for (types area) in areas do
    (map-all area 'kill-all-relations)))

(defmethod (trigger-relation surroundings) (relation-type  object-types)
  (map-over-objects-of-type self
      (λ(o) (force-relation o relation-type)) object-types))

(defmethod (propogate-relation surroundings) (relation-type object-types)
  (map-over-objects-of-type self
     (λ(o)
       (let ((r (find-relation o relation-type)))
         (and r (propogate r))))
     object-types))

(defmethod (kill-relation surroundings) (relation-type object-types)
  (loop for type in object-types do
    (let ((area (find-area self type)))
     (map-all area (λ(o) (kill-relation o relation-type))))))

(defmethod (map-over-relations-of-type surroundings) (f type)
  (map-over-areas self
     (λ(area) (map-all area (λ(ob) (let ((r (find-relation ob type))) (and r (funcall f r)))))))
```

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge MA, 1985.

[2] Raymond Erickson. Darms, a reference manual. 1976.

[3] R.C. Gonzalez and P. Wintz. *Digital Image Processing.* Addison Wesley, Reading MA, 1987.

[4] Daniel W. Hillis. *The Connection Machine.* MIT Press, Cambridge, 1985.

[5] Berthold K. P. Horn. *Robot Vision.* MIT Press, Cambridge, MA, 1986.

[6] Marvin Minsky. *Society of Mind.* Simon & Schuster, New York, 1987.

[7] David Prerau. *Computer Pattern Recognition of Standard Engraved Music Notation.* PhD thesis, Massachusetts Institute of Technology, 1970.

[8] Gardner Read. *Music Notation.* Crescendo Publishers, Boston, 1969.

[9] Franz Schubert. *The Complete Chamberworks (Breitkopf & Hardel Edition).* Dover Press, New York, 1973.

[10] Guy L. Steele. The definition and implementation of a computer programming language based on constraints. Technical Report AI-TR-595, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1980.

[11] Guy L. Steele. *Common Lisp: The Language (2nd Edition).* Digital Press, 1990.

[12] Symbolics, Inc. *Symbolics Common Lisp— Language Concepts*, 1988.

[13] Thinking Machines. *Starlisp Reference Manual Version 5.2*, 1988.

[14] David Waltz. Waltz filtering. In S. C. Shapiro, editor, *Encylopedia of Artificial Intelligence, Vol. 2*. John Wiley & Sons, 1987.

[15] Patrick Winston. *Artifical Intelligence*. Addison Wesley, Reading, MA, 1984.