ABSTRACT
The consideration of existing systems of music notation on computers has been the conclusion of the thrust for electronic music notation. But few of these note-oriented systems have managed to deal with the definition of signs on a multi-device basis. Thus these signs must be included in structured objects on which a set of primitive rules in order to help the user to deal with (eg) connectivity. In the near copy, compound objects are programmed, an user-defineable constraint propagation scheme organizes spatial dependencies. In this last step, each application program in score formulates can be build.
At the present stage of the project, an object oriented extension to the programming language PostScript™ and a set of primitives for objects commonly have been experimented.

(*) PostScript is a trade mark of Adobe System inc.

GENERAL PHILOSOPHY
The aim of our project is to provide the musician with a system that allow him to express his musical thinking in his familiar writing style without any inhibition as to style or complexity. For example, the user should be able to write such score-schemata as these mentioned in fig. 1, and of course, much more complex things.

The output of such a system are essentially devoted to human composers as well as to:
- traditional instrumental input (diffusion, notation, etc.) of the score
- symbolic or mixed music
- the trace of the composer's work (as a feed-back tool for performance)
- as command context (as an output for composition tools)

Many attempts have been made to solve the problem of traditional and contemporary music notation on computer. It is surprising to observe how although important work has been done, the results are still unsatisfactory.

In our opinion, none of the existing system fall into one or more of the following tenet exceptions:
- combination of components for a single program : some systems have the ambition to solve at the same time such different problems as : real time input and symbolic pattern recognition from a keyboard, applications of expert rules for musical notation, sound output of the score, data encoding for special protocols, etc.
- the attempt to make the layout complexe-automatic generally implies the allowed complexity is elementary or very limited (eg music score)
- generally, the user does not have the ability to introduce his own rules, he is less subjected to the object oriented formulation.
- existing systems are generally closed, preventing their extension or the use of tools of them in other context.

We think that it is ended possible to respond to the most demanding needs (eg CCONS 85) of our requires the following Above:

- provide a uniform instead of a complex and fragile application, one must feel free, like a programmer using a general purpose and powerful programming language. Furthermore, out of a toolbox facilitates the utilization of very different applications as well as experimentation of high level algorithm.
- focus on the purely graphical aspect of the system:
- no hypothesis should be made about the underlying musical structure of a score, leaving this kind of choice to the user (is a tree a tree, a graph, any other hierarchic structure ? it is an open question).
- the system must provide facilities for the user to define his own graphical objects (icons or are complex objects) and also its two tools for editing objects:
- let the system handle all the disastrous work, leaving the intelligent part to the human expertise.
- provide, at the moment, device independence (using pens, mice, pointer, pointers, bitmap screen etc) and interactive capabilities.

Application programs score format and extensions including graphical examples, even and interactions with other fields sound work, etc.) are made natural if the previous points are respected.

These steps are required:

abstract
Design and manipulation of elementary graphical objects of two kinds:
- iconic object that can only be sketched translate (notes, stems, accidentals etc.)
- algorithmic objects that must be composed (fonts, beams, \ldots).

Object-oriented programming (OOP) provides a way of managing objects and their relationships. In OOP, objects are defined as instances of classes, and methods are associated with these objects. This approach is based on the idea of encapsulation, where the internal states and behaviors of an object are hidden from the outside world, and accessible only through a well-defined interface.

In OOP, a class is a blueprint for creating objects that share characteristics and behavior. Objects are instances of classes, and they have their own unique state and behavior. Classes can inherit properties and methods from other classes, allowing for a hierarchical organization of code.

Object-oriented programming also supports the concept of polymorphism, where an object can take on multiple forms and can respond to a common interface in a variety of ways. This flexibility makes it easier to write reusable and maintainable code.

Object-oriented programming languages, such as Java, C++, and C#, are widely used in software development for their ability to handle complex systems and large-scale applications. They provide powerful tools for organizing code and managing dependencies, making it easier to develop, test, and maintain software.
AN OBJECT-ORIENTED EXTENSION OF POSTSCRIPT

PostScript offers facilities for processing and manipulating code as a data structure, as well as a powerful associative memory mechanism that allows a user to quickly find the most appropriate and useful code. The result is a more powerful and flexible PostScript interpreter.

A Class is a data structure that defines a model for objects to behave like. It consists of a list of fields defined by a name and an initial value, and a list of methods, i.e. procedures that perform a certain work specific to the class. Classes may have attributes that indicate the fields and the methods of their parents.

Methods are active (instructs of a class, defined by a vector of values for the fields, and a pointer to the class hierarchy they belong to. Methods are retrieved through messages sent to objects.

For instance:

```
Note Maid
[ % fields 300
     % class 30
     % name 1 ]
...
]

 DefClass

 defines the generic class Note.

 Note icon [... 11 1 1 DefClass]

 defines the subclass Note of Icon. Note has its own fields and
 methods and will inherit those of Icon. Thus several levels of generality
 are possible.

 note Note New def
 creates an instance of note, named 'myNote', with the fields set
 to default values.

 x y myNote Move
 myNote Draw

 successively sends the messages Move, with arguments x, y, and
 the message Draw to the newly created object. Note that all
 messages Move and Draw are defined for other classes, moving
 and drawing objects of different types will be associated with
 the same syntax, although different algorithms may be provided.

PRIMITIVES PHILOSOPHY

The purpose of level two is to implement objects. The philosophy adopted here is to remove much of the explicit object
pointers from the display. In other words, instead of having a list of objects, it is only necessary to keep
track of the object, is x-coordinate and y-coordinate must be stored in the objects, not in some message. For example:

part GetRight
3 value GetLine

correctly connects the bottom part of 'bar' to the right part of
'terminal to the third line of 'bar'.

This approach is a store general form, one in which there are fixed
operations available between them. For example a sequence like:

myNote GetRightDefaultPos system Move

that draws for the current procedures point, in order, to connect a
direction to the right of a note, would work correctly in the case of a
single note, but not in the case where we want to position the
leftmost relative to a row that is part of a column (see fig. 2).

For the dynamic objects, (beams, lines, etc.), the MoveDraw
scheme is not sufficient. It is practical for these objects to think in
terms of displacement, even if it is not an intuitive notion. For
example, since we are positioning relative to a note, then we need

to get its initial length, or to stretch a beam, because it will be
calculated relatively to a zero endpoint, then stretched linearly to match another parts, or vertically to get a certain shape, etc.

```

Fig. 3 the top side of the beam is moved to the top left side of
the upper note. Its bottom is stretched down as an y position of
the lower note.

PROBLEMS OF POSITIONING

L Icons

initialization

GetX, GetY get x, y icon's reference points. For a CounterValue
Head, it will be its x position and the place where its is formed by a
Circle. i.e. GetLeft, GetRight, GetUp, GetDown, get the
represents, in the units background, a point in the output
```

175

ICMC 86 Proceedings
The generic `Move` message looks like:

\[ x \text{ move-mask mm Mou} \]

Permanently, the `Move` message by a legitimate mask allows the user to choose which part of the note is to match the x,y coordinate on the display surface (e.g., right-side, down-jaw, counter, up-lip, etc.). For visual motions, specific messages are provided such as `Mask=Left moves the (left-side, y, rel) of an mm, etc.`. In fig. 4, the whole note is positioned by its center, while for quantizations are connected by their left and right side to the stems.

![Fig. 4 example of left, right and center positioning of note heads](image)

2. stems

General-purpose

GetLeft, GetCenter, GetRight, GetDown, GetUp have the same effects as for an icon. GetWidth yields the stem width. GetHeight, BeamPos, GetStem, BeamPos get the post where secondary beams are to be connected to the stave (see fig. 5).

Motions:

\[ xy \text{ move-mask mm Mou} \]

Same as icon, except with an additional parameter: a vertical shift towards both directions can be added in some or correct properly the stem to a notehead (see fig. 6). Always specific motion messages, MoveDownToNote and MoveUpToNote use the proper mask (with shifting) to connect a stem to a notehead.

Stretching:

\[ y \text{-value length-stretch-mask nmm Stretch} \]

The stretch-mask will specify if the outside part or the bottom part of the stem is to be stretched, if the stem is stretched up or down to a certain absolute y, or if it is stretched until it reaches a certain length. The stretching parameter is always valid in cases when the stem is stretched up or down to a notehead (see fig. 7).

The StretchUp, StretchDown, [stretchUp]Length, StretchDownLength, StretchUpToNote, StretchDownToNote messages are provided for usual purposes.

3. beams

General-purpose

GetWidth gets the beam's thickness. GetLeft, GetRight gets the beam's left and right boundary. GetSlope gets the beam's slope. GetIntersection gets the y-coordinate of the intersection point between the beam and a vertical line specified by it x-coordinate. This allows one to connect a stem inside a group to the group's beam. (Fig. 8)

Motions:

A beam can be moved only with regard to its left or right point. (MoveXf, MoveYf)

Stretching:

\[ x \text{ or } y \text{ length-stretch-mask beam-stretch} \]

A beam may be stretched in three ways:

- In order for its right or left point to reach a certain y-coordinate, with this point's x-coordinate being invariant (see fig. 9).
- In order for its right or left point to reach a certain x-coordinate, with the beam's slope being invariant (fig. 10).

ICMC36 Proceedings 175
- In order for the beam to reach a certain length, the StretchLeft and StretchRight messages realize the x-coordinate stretching.

**Function**

- **x-coordinate angle arm Rotate**
- A beam can be turned about a point (inside or outside the beam itself) specified by a x-coordinate. The y-coordinate is obviously determined by the GetInView message. This message is generally used to move a beam around the points where a score instance occurs (see Fig. 6).

4. **group**

An ordering of operation for connecting a group of, e.g., three notes with increasing note size might be as follows:

1. Create 3 Quant/Nonhead notes, 3 Stems, and one beam object.
2. Move note, note 2, and note 3 to their final position.
3. Position every voice with the following score mask:
   - (right side, down side) point of the stem to the (right side, up side) point of the note heads.
   - Sliding option must be on.
4. Set the left and right lines to their length by stretching them upward.
5. Set the beam's endpoints to the (left side, up side) point of the left stem and the up side, up side) point of the right stem.
6. Switch the middle beam with the following stretch mask:
   - (up side, up side) point of the stem.
   - Proper stretching
   - No sliding
   - up to the y-coordinate given by the GetInView query (in order to match the beam).

The following code is given as an example of what a firmware program would have to generate in P6Lecture:

- Create object:
  ```
  % beam object
  
  % note: Quant/Nonhead New def
  % note: Quant/Nonhead New def
  % note: Quant/Nonhead New def
  % note: Quant/Nonhead New def
  % note: Quant/Nonhead New def
  % note: Quant/Nonhead New def
  % note: Quant/Nonhead New def
  % note: Quant/Nonhead New def
  % note: Quant/Nonhead New def
  
  % beam: Beam New def
  
  % position noteheads on the stems considering their left side:
  
  s1 1 note GetLine note1 MoveLeft
  s2 2 note GetLine note2 MoveLeft
  s3 3 note GetLine note3 MoveLeft
  
  % connect stems to noteheads:
  
  note: GetRefRight note1 MoveToNote
  note: GetRefRight note2 MoveToNote
  note: GetRefRight note3 MoveToNote
  note: GetRefRight note4 MoveToNote
  note: GetRefRight note5 MoveToNote
  note: GetRefRight note6 MoveToNote
  note: GetRefRight note7 MoveToNote
  note: GetRefRight note8 MoveToNote
  note: GetRefRight note9 MoveToNote
  note: GetRefRight note10 MoveToNote
  note: GetRefRight note11 MoveToNote
  note: GetRefRight note12 MoveToNote
  
  % note: and stems keep their distance length (as occurs)
  % while note 1 is set to a length of two times the note
  % inquarting
  ```

177    ICMC 86 Proceedings
In order to meet the functionalities of our level 2 implementation, a C program has been written which takes a list of groups of notes as input, then generates a PostScript program that uses the connectivity facilities in order to draw connectly the groups, i.e. with correct positioning and slope of the beam. The interesting point is that a lot of local control is committed to PostScript, which means a full use of the object structure as well as it’s own control structures (loops, tests, etc.)

The Algorithm is as follows:

1. set min and max to the minimum and maximum value in the current group. If max = max, iterate on the rest of the group to find another pair of values for min and max.

2. if max > -1 * min (i.e. max and min are above middle 9 or the distance between max and the middle of the staff is greater than the one between min and the middle of the staff) then the beam is above the group.

3. set min to the true minimum and max to the true maximum. If min is equal to the first note of the group and max to the last (or the opposite) and min > max, then the beam is not horizontal.

4. if the beam is not horizontal, proceed as follows to set its slope (Fig. 13):

   - set the beam’s left and right boundaries to the first and last stem’s upper limit.
   - for each stem between the first and last, check if the stem passes beyond the beam. If it does, set the beam’s left bound to the current stem’s upper limit.
   - stretch the beam (x-type left stretching) to the x-coordinate of the first stem.
   - for each stem, stretch the stem up to reach the beam, using the GetIntersection message.

Fig. 14 shows a sample of the formater’s output.

ACKNOWLEDGMENTS

We wish to thank Clare Haggerty from Adobe for providing us with a pre-release of Adobe’s Musical font.

REFERENCES


Note: graphical examples in this paper have been realized with an Apple LaserWriter running our extended form of PostScript.