PIPELINING BETWEEN *JMSL* AND *Quintet.net*

Adam Siska

Budapest University of Technology and Economics
Ferenc Liszt Academy of Music, Budapest

Abstract

Although *Quintet.net* is an appropriate tool for networked, real-time performances, until now it could use only static score sources (scores previously engraved with the Score Editor of the Quintet.net CDK). On the other hand, there were professional software to produce musical data in real-time since a long time ago. JMSL is one of these professional environments, which is able to produce algorithmical music data in real-time.

The evident claim of connecting these two environments, however, raises to several problems to resolve. As there is a conceptual difference between the data models used by these two systems, an external tool must be developed to carry out the conversion between these data models. Although it would be enough a simple data conversion tool, the need for future expansion possibilities leads to the developement of a stand-alone translator object with its own, extended data model.

This data model must be planned with the aim of getting a structure which would be able to represent scores imported from many different formats. To obtain this, it'll proved that although tree structures could be used to represent many scores, they are basically not good enough to represent any kind of score. With this in mind, we'll present a new data model that has the structure of a directed graph. This is a good decision, as tree structures are subsets of directed graphs – so data importing issues can easily be solved – and directed graphs can hold musical information that trees can't hold in some cases presented below as well.

As Quintet.net doesn't have an own – automatized – engraving tool, an engraving tool is to be developed and added to the data exporting module as well. This engraving tool has to be as simple as possible, however, some parameters should be set externally to take control over the exact behaviour of the engraver. With this last step, we'll be able to send musical data obtained with algorithmical routines of JMSL to the Quintet.net score engine in real-time, thus we'll manage to enter to the world of real-time network music performances based on dynamically created music material.

1 Introduction

Quintet.net, a network performance environment created and maintained by *Georg Hajdu* is a powerful tool for networked, real-time music performances. As a real-time environment, one of its important features is that it can display scores on a screen for the players, thus the musicians are able to play their parts. [HAJDU, 2005] As one can see, an interesting question related to this engine is the source of the displayed scores. Until now, there were basically two different kind of score sources for Quintet.net: the so-called „*performance notation*", and the „*score notation*".

Performance notation is a simple way for achieving basic real-time notation. The score generated this way is in space notation, but these scores only display the currently played material.

In *score notation* one can create a score with the *Score Editor* of the *Composition Development Kit* of Quintet.net. Later this score can be loaded by the engine and displayed for the musicians. Of course, it is possible to create almost any score this way, but this is not a real-time method.

One of the goals of the Quintet.net environment was the ability of creating network music in real-time; including the ability of high-level real-time composition, supporting algorithmic methods also. Since algorithmic composition tools have a long tradition as well, it was evident that an idea have arised to create data bridge (a so-called *pipeline*) between algorithmic tools and Quintet.net (instead of developing an own module for this purpose inside the Quintet.net environment). The finally selected tool was the *Java Music Specification Language* (JMSL), a Java API for composition and performance created by *Phil Burk* and *Nick Didkovsky*. [BURK-DIDKOVSKY, 2004]

This paper discusses how such a data pipeline was built between JMSL and Quintet.net. The usage of this data bridge lets the user connect MaxScore, a JMSL-based Max/MSP notation patch developed by Nick Didkovsky with the score engine of Quintet.net, thus giving us the possibility of including real-time score creation in the world of network music performance.

2 Different Data Models

The aim of such a bridge is translating the output of JMSL to the input format of Quintet.net. As there is a big conceptual difference between the data models used by these environments, this is a quite painful process. To understand the main difficulties, we should examine these two data models separately.

2.1 Quintet.net

Quintet.net uses a low-level notation format, where the data stream contains so-called *score sprites*, that is, identifiers for different score objects (like notes, beams etc.) and positioning info for these. The system has a big canvas, and the different score objects get displayed at the position indicated by their coordinates. Although this is a very fast and compact way for displaying scores (and for transmit them by network), it leads to several problems:

- The environment doesn't need any engraver module. In some cases, this isn't a problem but a feature, but in this particular case this means that the engraving process should be done outside the existing environment, before sending any data to Quintet.net. As engraving decisions can't be separated from the special displaying engine of the environment, the engraving process can't be done by an existing engraver tool, like Lilypond or JMSL's own score display environment. This all shows us that the engraver module must be embedded in the data bridge.

- The bigger problem is that the internal format of the environment doesn't tell us anything about the musical content of the scores. Therefore, it is almost impossible to do serious manipulations with the data in the data model. This shows also the fact that it would have been almost impossible the development of a powerful algorithmic compositional API *inside* the Quintet.net environment.

2.2 JMSL

The I/O data model used by JMSL is a DOM tree, a high-level description of the musical events in a score. The output of JMSL gives us every musical–logical information about the music in the score. The different levels of the tree-nodes are `score`, `measure`, `staff`, `voice` and `note` (there are other nodes also, representing instruments etc., but these are enough to represent a score). Additional information for time signatures, clefs, beaming, slurs etc. are embedded in these nodes at corresponding levels of the tree. It can be seen that this way of score representation makes data manipulation very easy, but the structure doesn't have any explicit information about the engraving process of the respective score.

3 Extended Data Model

As presented above, the data bridge between the two systems has to realize two main tasks:

1. Convert the DOM nodes and the embedded implicit data to explicit separate score sprites, and

2. Add engraving information to the sprites (that is, calculate positions on the fixed canvas).

The simplest way to achieve this would have been the creation of a simple, sequential program that first decomposes the output of JMSL and after that calculates the exacts positions of the gained score sprites. Although this solution could have worked as well, code maintenance would have been very difficult, and it wouldn't have offered any serious upgrade possibility (for example, the possibility of importing also from other score sources). These reasons led to the decision of doing an abstract object which was independent of any score format, and had the ability of store every kind of score data. If such an object exists, the remainder of the task is to write the appropriate import/export methods (in our case, an import method for JMSL and an export one for Quintet.net).

When planning the data model used by the internal object, it was to take in consideration that it should be able to import as many different score formats as possible. Therefore the representation must be a high-level score representation. If a tree representation would be suitable for this purpose, this part of the task would have been done, as the XML output of JMSL has a tree representation of the musical events. Unfortunately, representing musical content in a tree is not very convenient. In a tree structure every node can have at most one parent node. This means that in this representation musical contexts are nested in each other, which is not necessarily the case in real musical notation. A good example is a chord that runs through several staves: the individual noteheads of such a chord belong to different staves, but all of them belong (at the same time) to the chord, and through this single chord, they belong also to the same voice. So, it can be proved that although a voice node can have a child node that is the child node of a staff node at the same time, the voice node itself is not nested necessarily in the staff context, and vice versa (of course, workarounds can be made to achieve the results *graphically*, but that's not the same as having this *logically* in the score represented with the data model).

A good choice for the data model can be a directed graph. Having this, we can keep a hierarchy within the score events, but this hierarchy doesn't have to be cascade. Of course, there is a problem with the representation of a directed graph. Unlike there exists the widely used DOM for representing data in a tree structure, there is no common way to represent a directed graph. This means that the representation must be handled by own code, which always has bigger risk than using a factory code provided by one of the big IT groups (for representing DOM in Java, there are very stable packages provided by organizations or companies like W3C, Apache, Sun, Oracle etc.). The problem arises conspicuously if one wants to export the graph, as for directed graphs the most widely used mathematical representation is a big matrix, which in our case would have extreme sizes (as the major part of the matrix would be empty), and it would be almost impossible to edit by hand (which in some cases is an important feature). But it should be considered that in fact there is no need to export the graph itself, as the initial purpose was to create a powerful score representation which is present *only in the memory*, and add methods to import and export scores in whatever format we wanted.

Let's examine now a possible representation of a score in a directed graph.

3.1   The Model

A *graph* is a collection of points and lines connecting some (possibly empty) subset of them. The points of a graph are called „*nodes"* or „*vertices"* and the lines connecting the nodes of a graph are called „*edges"* or „*arcs"*. A *directed graph* is a graph whose edges are imbued with directedness. In our representation, the score elements are represented by nodes, and the logical relations between them are the edges. Having directed arcs, we can represent the hierarchy of these elements. From now on, if we have the nodes A and B, and we have an edge e pointing from A to B, then we say that A is the *parent node* of B and B is the *child node* of A.

Apartly of the common score tags, a *timeline* was also introduced in the model. The timeline is a direct child of the `score` tag, and contains `time event` tags. These tags serve to synchronize the different elements of the score, and also this is the parent node for staves and voices (this way it's quite easy to insert *ossia* staves or additional voices in the middle of a score).

It should be noticed at this point that this data model is currently under development, so at this time there can be score elements that don't take part of the representation. The nodes included in the current development phase are presented in Table 1.

Node	Possibly parent nodes	Possibly child nodes
score		time event
time event	score	bar line, staff, voice, clef, key signature, time signature, chord
bar line	time event, staff	
staff	time event	bar line, clef, time signature, key signature, notehead
voice	time event	chord
clef	time event, staff	
key signature	time event, staff	
time signature	time event, staff	
chord	time event, voice	notehead, beam, crescendo, mark, ottava, slur, stem, tuplet

Node	Possibly parent nodes	Possibly child nodes
notehead	staff, chord	tie
beam	chord	
crescendo	chord	
mark	chord	
ottava	chord	
slur	chord	
stem	chord	flag
tuplet	chord	
tie	notehead	
flag	stem	

Table 1: The nodes included in the current development state.

As one can see, there's a big difference between the usual tree representation of a score and this one, as `staff` and `voice` are both children of `score`.

It should be pointed out that currently nodes like `instrument` or `staff group` are not part of the score structure. These items are planned to be added in a later state of development. Also `measure` nodes are missing from the model. This is because there is no need for them. Measures graphically would only inform us about bar lines, but as bar lines are explicitly included in the model, there is no need for explicit `measure` tags. Logically, measures only serve to show a fixed time amount, but this can also be calculated using the timeline (and the current time signatures).

3.2   Data I/O

Equipped with this structure, it's quite easy to implement methods that import data from a score represented as a tree, like JMSL's scores. Since a tree is also a directed graph, the JMSL DOM tree could be used easily to build a graph like the one above. Only a few modifications had to be done:

- The `voice` tags had to be separated from the `staff` ones.

- `bar line` tags had to be added to the timeline.

- `note` tags had to be converted to `chord` and `notehead` tags.

- `staff`, `voice` and `chord` tags had to be added to the timeline, while `notehead` tags had to be connected directly to `staff` tags.

- The implicit score data of JMSL had to be converted to explicit tags (tags like `clef`, `beam` etc.) and had to be connected to the respective other tags (including `time event` tags for some of these newly created ones).

The export process to Quintet.net is even more simple, as Quintet.net doesn't need to get any logic between the separate tags. Therefore, the export process is quite simple: one only has

to iterate over the elements of the score, and export them node-by-node (with every attribute of the nodes), in whatever possible order. But before this would be done, a very important step can't be left out of the process: engraving.

4   Internal Engraver Tool

As part of the data bridge, the high-level logical score info had to be expanded with low-level information about positioning. In order to achieve this, an engraver module was developed. This module operates *over* the score in the memory, represented as a directed graph, but it is totally independent from the score routines (as I/O methods are also independent from the score managing functions).

The engraver calculates the x and y coordinates of the elements (as every engraver). The graphical parameters of the score sprites are to be described in a separate parameter file, so that design changes in Quintet.net doesn't need to affect the engraver, only the parameters used for engraving. Now I would like to present some of the solutions that the engraver uses for position calculations.

4.1   Horizontal Positioning

The engraver takes in consideration every object that has to have a fixed width (these are *bar lines*, *clefs*, *key-* and *time signatures*, *chords* – including their affected *noteheads* – , *stems* and *flags*). The total length of these elements is first substracted from the full length of their respecting staves, and the remainder space is divided by the following algorithm: first, rubbers are inserted between every object. Next to this, weight values are added to these rubbers (depending on the time amount represented by these rubbers), and finally the free space will be divided between these rubbers according to the ratios of their respective weights. If the gained distance between two of these elements is less than the sum of the respective minimal distances, the two elements get their minimal required spaces (the whole procedure is analogous to the condensation process of gases). It should be pointed out that the routine that calculates weights for the rubbers can be changed externally. Currently, the weights are set to be the logarithms of the respective durations, but this can be changed anytime to any other function, without the need of changing the engraver itself. (By any other function, such extreme ones can also be imagined like sine or inverse function, or any other. Using the sine function as a weight, for example, can result in a funny score where some of the elements are very close to each other while others have big distances between them.)

The horizontal positions of the other elements (like beams etc.) are being calculated using the above gained positions.

4.2   Vertical Positioning

There are objects like *beams*, *crescendi*, *tuplets*, etc. whose vertical positions depend strongly on their parents' positions. To calculate the positions of these elements, a list is first created with the affected chords' positions, and this list is sent to a specific method. This method calculates the inclination and anchor points of the elements, using some possible parameters, but without considering the type of the object actually processed. The algorithm used by this routine can also be changed externally, without changing the engraver's own code. The method currently used first calculates a line connecting the first and last chords of the list, and then moves this line in a way that it doesn't cross any of the affected chords. This algorithm might be changed in a further state of developing to a nicer one that could calculate an optimal inclination using the Karush-Kuhn-Tucker conditions.

It should be pointed out here that *slurs* and *ties* have their own positioning calculation algorithm that generates cubic Bézier-curves. This codepart has several unsolved issues at the current state of development.

References

[BURK-DIDKOVSKY, 2004] Burk, L. B. & Didkovsky, N.: *Java Music Specification Language. An introduction and overview.* Proceedings from the International Computer Music Conference, 2004.

[HAJDU, 2005] Hajdu, G.: 'Quintet.net: An environment for composing and performing music on the Internet.' Leonardo Journal, vol. 38 no. 1, 2005.