MusicSpace: a Constraint-Based Music Spatializer

François **Pachet**, Olivier **Delerue**
SONY CSL Paris, 6, rue Amyot 75005, Paris, FRANCE
Tel: (33) 1 44 08 05 16, Fax: (33) 1 45 87 87 50, E-mail: pachet@csl.sony.fr

Abstract

We describe a system in which users may control in real time the position of sound sources. We introduce the problem of *mixing consistency*, as the maintenance of good properties of sound source configurations, and propose a solution based on a constraint propagation mechanism. In the authoring mode, sound engineers may specify which spatialization constraints should be satisfied. In the listening mode listeners can modify the position of sources, and the constraint solver ensures the constraints are satisfied.

1. Music Spatialization

We believe that listening environments of the future can be greatly enhanced by providing users more meaningful user control. The most straightforward musical control is probably the spatialization of sound sources.
Music spatialization has long been an intensive object of study in computer music research. Most of the work so far has concentrated in building software to simulate acoustic environments for sound signals. These techniques typically exploit difference of amplitude in sound channels, delays between sound channels to account for interaural distances, and sound filtering techniques such as reverberation to recreate impressions of distance (e.g. [6]). These spatialization techniques are mostly used for building virtual reality environments, such as [3] or [7]. We propose in this work to exploit spatialization technologies in a active way, by letting users change arbitrarily the spatialization of sound sources. We show that this control induces a risk that the user creates configurations of sound sources which are not satisfactory, from the viewpoint of the sound engineer or composer. We propose a system – *MusicSpace* - in which the user may change the positions of sound sources, while ensuring that resulting spatializations are always "correct" in some precisely defined sense.

2. MusicSpace

MusicSpace is not a spatialization system *per se*, but rather an interface for issuing high level commands to a spatializer. The basic idea in MusicSpace is to represent graphically sound sources in a interface, as well as an avatar that represents the listener itself. In this window, the user may either move its avatar around, or move the instruments. The relative position of sound sources to the listener's avatar determine the overall mixing of the music, according to simple geometrical rules mapping distances to volume and panoramic controls (see Figure 1). The real time mixing of sound sources is then performed by sending appropriate commands from MusicSpace, to whatever spatialization system connected to it, such as a mixing console, a Midi Spatializer, or a more sophisticated spatialization system such as [6].

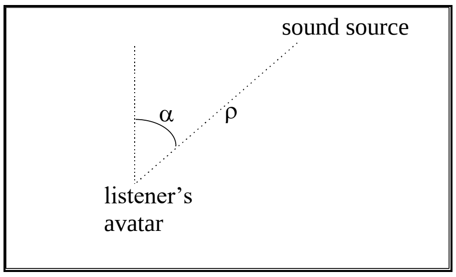

Figure 1 : Volume of sound_source$_i$ = f(distance(graphical-object$_i$, listener_avatar)). f is a function mapping distance to Midi volume (from 0 to 127). Stereo position of sound source i = g(angle(graphical_Objecti, listener_avatar)), where angle is computed relatively to the vertical segment crossing the listener's avatar, and g is a function mapping angles to Midi panoramic positions.

In this context, MusicSpace is a command generator for an arbitrary spatialization system (see Figure 2).

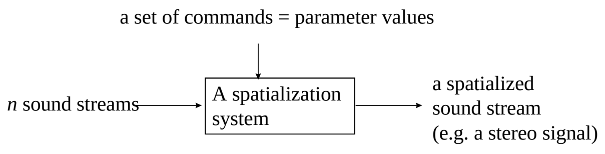
a set of commands = parameter values
n sound streams
A spatialization system
a spatialized sound stream (e.g. a stereo signal)

Figure 2 : A Spatializer module

3. Mixing Consistency

The problem with allowing users to change the configuration of sound sources, and hence, the mixing, is that they do not have the knowledge required to produce coherent, nice-sounding mixings. Without any supervision, users can end up with configurations of sound sources which are not acceptable in a number of ways. For instance, by putting related sound sources too much apart, or, conversely, by grouping somehow "incompatible" sources together. Another example is for instance that some sound sources should always be heard but should never be to loud and thus should remain in between some precisely defined boundaries.

Indeed, the knowledge of the sound engineer is difficult to acquire –it takes years to become a sound engineer -, and hence to explicit and represent. In our context, however, we claim that a purely syntactical viewpoint is enough to address the issue of supervised mixing. The basic actions of the sound engineer are atomic actions on controls and parameters such as faders and knobs on the mixing table. However, mixing also involves higher level actions that can be defined as compositions of atomic actions. For instance, sound engineers may want to ensure that the overall energy level of the recording always lies between reasonable boundaries. Conversely, several sound sources may be logically dependent. For instance, the rhythm section may consist in the bass track, the guitar track and the drum track. Another typical mixing action is to assign boundaries to instruments or groups of instruments so that they always remain within a given spatial range. The consequence of these actions is that sound sources are not configured independently of each another. Typically, when a fader is raised, another one, (or a group of other faders) will be lowered.

We propose in this paper to encode this type of knowledge on sound spatialization as *constraints*, which are interpreted in real time by an efficient constraint propagation algorithm, integrated in MusicSpace.

3.1 Constraints for Interactive Systems

Constraints can be defined as relations that should always be satisfied. Constraints are interesting because they are stated declaratively by the programmer, thereby avoiding him to program complex algorithms. Constraint propagation algorithms are particularly relevant for building reactive systems typically for layout management of graphical interfaces [5].

3.2 Constraints and Mixing Consistency

We defined a set of constraints appropriate for specifying interesting relations between sound sources. Each sound source is represented by a point, i.e. two integer variables (one for each coordinate): $p_i = \{x_i, y_i\}$ with $x_i, y_i \in [1, 1000]$ (a typical screen). An additional variable l represents the position of the listener's avatar, itself consisting of two integer variables: $l = \{x_l, y_l\}$ with $x_l, y_l \in [1, 1000]$.

Most of the constraints on mixing involve a collection of sound sources and the listener. We describe here the most useful ones.

- Constant Energy Level

This constraint states that the energy level between several sound sources should be kept constant. According to our model of sound mixing, this constraint can be stated between variables $pi, i = 1, .., n$ as follows: $\prod_{i=1}^{n} \|p_i - l\| = Cte$. Intuitively, it means that when one source is moved toward the listener, the other sources should be "pushed away", and vice-versa. The constant value on the right-hand side of the constraint is determined by the current values of p_i and l when the constraint is set. Note that this constraint is non linear, and not functional (except in the case of two sources).

- Constant Angular Offset

This constraint is the angular equivalent of the preceding one. It expresses that the spatial configuration of sound sources should be preserved, i.e. that the angle between two objects and the listener should remain constant. It can be stated between variables p_1 and p_2 as: $(p_1, \hat{l}, p_2) = Cte$. It is easily generalized to a collection of objects $p_1, \ldots, p_i \ldots, p_n$.

- Constant Distance Ratio

The constraint states that two or more objects should remain in a constant distance ratio to the listener:

$$\|p_1 - l\| = \alpha_{1,2} \|p_2 - l\|$$

- Radial Limits of Sound Sources

This constraint allows to impose radial limits in the possible regions of sound sources. These limits are defined by circles whose center is the listener's avatar (see Figure 7).

$\|p_i - l\| \geq \alpha_{\inf}$ (lower limit), $\|p_i - l\| \leq \alpha_{\sup}$ (upper limit)

- Grouping constraint

This constraint states that a set of n sound sources should remain grouped, i.e. that the distances between the objects should remain constant (independently of the listener's avatar position):

$$\forall i, j \leq n : (x_i - x_j) = Ctx_{i,j} \text{ and } (y_i - y_j) = Cty_{i,j}$$

Other typical constraints include symbolic constraints, holding on non geographical variables. For instance, an "Incompatibility constraint" imposes that only one source should be audible at a time: the closest source only is heard, the others are muted. Another complex constraint is the "Equalizing constraint", which states that the frequency ratio of the overall mixing should remain within the range of an equalizer. For instance, the global frequency spectrum of the sound should be flat.

3.3 Constraint algorithm

The examples of constraints given above show that the constraints have the following properties:

- the constraints are not linear. For instance, the constant energy level (between two or more sources) is not linear. This prohibits the use of simplex-derived algorithms.
- The constraints are not all functional. For instance, geometrical limits of sound sources are typically inequality constraints.
- The constraints in our context induce cycles. For instance, a simple configuration with two sources

linked by a constant energy level constraint and a constant angular offset constraint yields a cyclic constraint graph.

There is no general algorithm, to our knowledge, which handles non linear, non functional constraints with cycles. *Indigo* [1] is an algorithm for functional constraints with inequalities, but does not handle cycles. Conversely, cycle solvers such as *Purple* (linear constraints) and *DeepPurple* for linear inequalities, do not handle non linear constraints. The general solution as proposed in the literature consists in using hybrid algorithms such as *Detail* or *UltraViolet* as mentioned in section 3.1. However, these algorithms add a considerable level of complexity: they are difficult to implement and tune, and may have unexpected behavior [4].

Instead, we designed a simple propagation algorithm which implements only a part of our requirements, but with predictable and reactive behavior [8] . The current algorithm we use is based on a simple propagation scheme, and allows to handle functional constraints, inequality constraints. It handles cycles simply by checking conflicts. Each variable *v* is associated to the set of constraints holding on it (predicate *constraints(v)*). Each functional constraint has a set of procedures or *methods*, used to compute values of output variables from values of input variables. The algorithm is triggered by the modification of one variable, and is described below:

```
// Each variable holds a list of constraints, and each
// constraint holds the list of its variables
// The propagation depends on the type of the constraint
propagate (Constraint c, Variable v)
    if c is functional: propagateFunctional(c, v)
    if c is inequality: propagateInequality(c, v)

propagateFunctionalConstraint(Constraint c, Variable v)
    result = true
    for each variable v' in c. variables, such as v' ≠ v,
        new-value = perform-method (v', v, v.new-value)
        result = result && perturbate(v', new-value, c )
    endfor
    return result

// Inequality constraints are just checked
propagateInequalityConstraint(variable v , perturbation v-
perturbation )
    return c.isSatisfied()

// Each variable holds a value (actual current value), and a
// new-value, which represents a perturbation, either triggered
// by the user or computed
perturbate(Variable v, Value new-value, Constraint c)
    result = true
    if  v.value ≠ v.new-value // v has already been perturbated
        //perturbation is the same
        return (v.new-value = new-value)
    endif
    v.new-value= new-value
    for each constraint c' in v.constraints such as c' != c
        result = result && propagate(c', v)
    enfor
    return result
```

Figure 3. Propagation algorithm of MusicSpace

An important property of the algorithm is that new constraint classes may be added easily, just requiring the definition of a set of propagation procedures (*perform-method*).

3.4 Handles as one-way constraints

There are cases when full-fledged constraints are not appropriate for the task. For instance, when a sound source is constrained in two (or more) incompatible ways, in different contexts.

This situation typically occurs when we need to specify different usages of sound sources, independent of each others. For instance, it makes sense to group together all the sound sources under a single constraints representing the global volume or "presence" of the piece. In this case, we would like all the sound sources to be linked by a constant distance ratio constraint. However, this constraint should be enforced only when the user wants to actually modify the global volume/presence of the whole piece. Similarly, it can be interesting to group related sound sources (e.g. voice sources) together, independently of other possible constraints. If we represent all these requirements as standard, multi way constraints, these constraints will be mutually incompatible.

To solve this problem, we introduce the notion of *handle*. A handle is an extra object added in the interface, which represents a particular usage of the sound sources: for instance, grouping the rhythm section, grouping the human voices of the piece, or balancing between the rhythm section and the voicing. When the piece contains a large number of sound sources, these handles provide a way of splitting the constraint set in different, mutually incompatible subsets. Handles are represented in the interface by green balls that can possibly be given a name (see Figure 10).

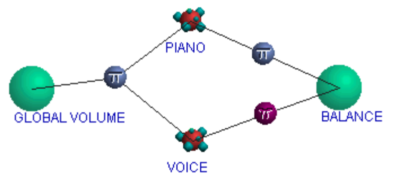

Figure 4: a problematic example

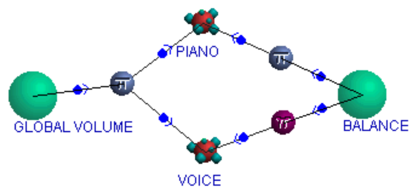

Figure 5: Solving the problem by introducing One-way constraints

To implement handles, we introduce one-way constraints in the algorithm. One way constraints can be seen as constraints which are activated conditionally.
The proposed algorithm makes it easy to integrate so-called one way constraints: one way constraints are binary constraints that come between a constraint and each of its constrained object. According to its state (object to constraint, or constraint to object) the one way constraint propagation method will simply transmit or hide the perturbation.
We introduce the function `one-way(constraint, variable)` which yields true if the constraint must propagate its change to the variable.
In the case One-way constraint to object, the only change to do in the algorithm of section 3.3 is the following:

```
propagateFunctionalConstraint(Constraint c, Variable v)
    result = true
    for each variable v' in c. variables, such as v' ≠ v,
        new-value = perform-method (v', v, v.new-value)
        if (1-way(c, v')) "do nothing"
          else result = result && perturbate(v', new-value, c )
    endfor
    return result
```

In the case of a perturbation from an object to a one-way constraint to object, the only change to do is the following:

```
perturbate(Variable v, Value new-value, Constraint c)
    result = true
    if  v.value ≠ v.new-value // v has already been perturbated
        //perturbation is the same
        return (v.new-value = new-value)
    endif
    v.new-value= new-value
    for each constraint c' in v.constraints such as c' != c
    if (1-way(c', v)) result = result && propagate(c', v)
        else  "do nothing"
    enfor
    return result
```

4. The interface

The MusicSpace interface represents both the sound sources and the listener's avatar. On Figure 6, a Jazz trio music file is loaded: the user can move around not only its avatar but also the sound sources (piano, bass and drums) and listens in real time the music, mixed according to the configuration of these sound sources.
The MusicSpace interface proposes several display modes. These modes correspond to different filters on the objects shown in the interface.

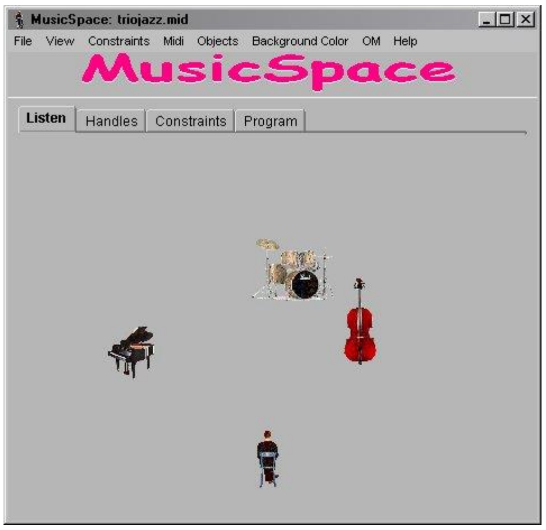
MusicSpace: triojazz.mid
File View Constraints Midi Objects Background Color OM Help
MusicSpace
Listen Handles Constraints Program

Figure 6: The MusicSpace interface in the "listening mode"

The listen mode (See Figure 6) is the simplest mode, as it shows only the essential objects for user control (i.e. the avatar and the sound sources). The user can move the sound sources or the avatar without seeing the underlying constraints.

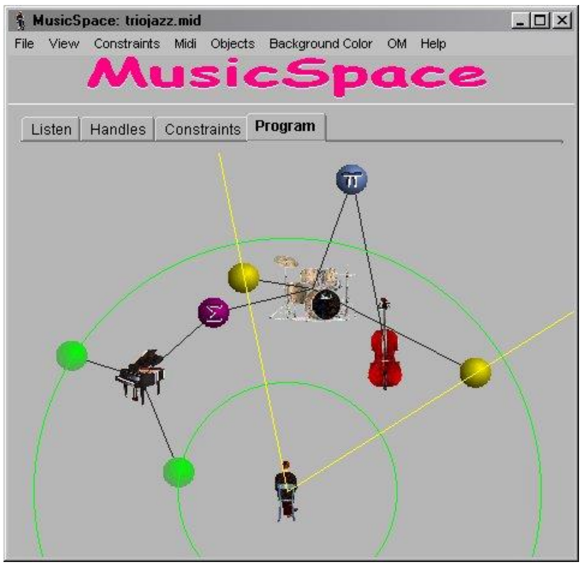
MusicSpace: triojazz.mid
File View Constraints Midi Objects Background Color OM Help
MusicSpace
Listen Handles Constraints Program

Figure 7 : The MusicSpace Interface in the "programming" mode.

In the program mode (See Figure 7), the user can visualize, remove or edit the set of constraints that operate on sound sources. In this mode, the programmer can create constraints corresponding to the specific properties on the desired configuration of sound sources.
For instance, in Figure 7, a typical set of constraints corresponding to a jazz trio has been created:

- The bass and drum sound sources are linked by a "constant distance ratio" constraint, which ensures that they remain grouped, distance wise.

- The piano is linked with the rhythm section (bass and drums linked together) by a “balance” constraint (constant energy level constraint). This ensures that the total level between the piano and the rhythm section is constant.
- The piano is limited in its movements by a two limit constraints. This ensures that the piano is always heard but never too loud.
- The drum is forced to remain in an angular area by two “angle constraints”. This ensures that the drum is always more or less in the middle of the panoramic range.

Adding a constraint in the interface is straightforward: The user first selects the sound sources to be constrained (the arguments), and then clicks on the appropriate constraint in a constraint palette (see Figure 8). This instantiates a corresponding constraint in the interface, which is represented by a small ball linked to the constrained sound sources by lines (see Figure 7).

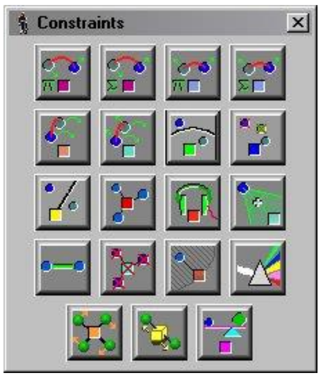

Figure 8: the constraint palette

Other intermediary modes have been defined, between Listen and Program. In particular, the mode Handles show the various handles that have been defined to ease the control of the sound sources.
For instance, when controlling a large number of sound sources (See Figure 10), it is convenient to represent the essential properties of the sound configuration and then let the user control the handles instead of controlling directly the sound sources.

Eventually, configurations of sound sources and related constraints can be saved and restored in external files, using a proprietary meta data text format.

5. MusicSpace-Audio

An audio version of MusicSpace has been prototyped. This version allow to mix directly audio files on the PC instead of using an external spatializer. This extension uses the integrated Microsoft DirectX 3D technology. Beside the better sound quality and variety brought by the audio files, the audio version of MusicSpace raises new constraints issues since the audio objects provide different and specific parameters such as for instance the sound sources orientation. This topic, as well as the problem of multiplexing audio files for multi-track audio streaming will be discussed in a forthcoming paper.

6. Applications

Our project has led to numerous applications both for end users and professionals. We review here the most promising ones.

6.1 Midi File Player

The basic MusicSpace interface (as shown in Figure 6) allows to play Midi files that conform to the General Midi specification: in this case, the file is parsed and the system generates automatically the appropriate icons (according to the midi program change number) for each instrument.

6.2 Remote Mixing Table Controller

MusicSpace includes also a number of midi objects that allow to control a sound mixer. For instance, a full implementation of commands has been created to control a Yamaha O2R mixer (See Figure 9).

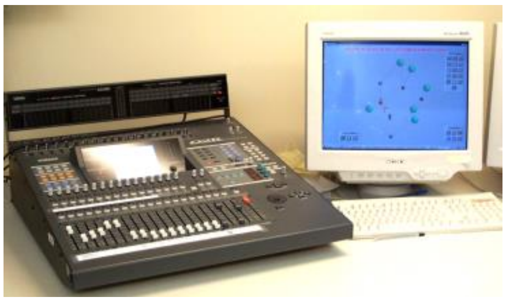

Figure 9: MusicSpace controlling a Yamaha O2R sound mixer.

Additionally, we also introduced generalized Midi objects with learning capabilities (automatic parsing of incoming midi messages) that allow to control virtually any remote controllable sound mixer.
This application of MusicSpace extends the possibilities of the sound mixer by allowing to set up constraints between any of its parameters. For instance, it is possible to control graphically the auxiliary output of a channel of the mixer to create a surround mixing with an ordinary stereo oriented mixer.

6.3 Audio Mixing Interface

Figure 10 shows the interface of the audio version of MusicSpace. Here 11 audio tracks are represented (from a French Hit from Sony Music).

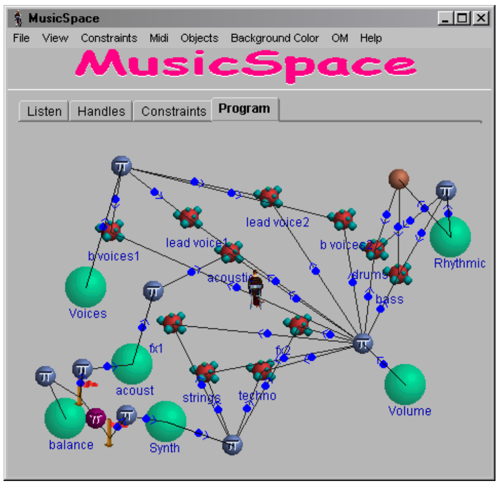

Figure 10: a full configuration with 11 audio tracks and 6 handles.

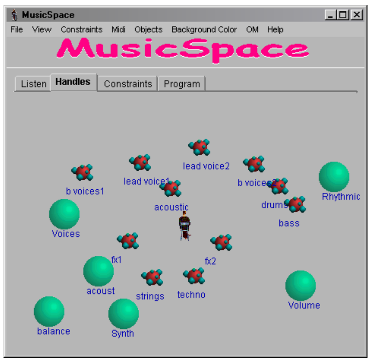

Figure 11: In the handle mode MusicSpace displays only the avatar, the sources and the handles; constraints are hidden.

In this example, we created six handles on the sound sources corresponding to the main features of the mix:

- Rhythmic: controls the rhythm section of the piece, i.e. bass and drums.
- Voices: controls all the human voices of the piece (leading voices and back voices).
- Acoust.: controls the acoustic instruments of the piece.
- Synth: controls the synthesized instruments
- Balance: is a handle on the two previous handles. This handle allows to make a balance between the amount of acoustic sound and the amount of synthesized sound.
- Volume: is a handle on all sources and allows to make all sources closer or farther without changing the proportions of distances between them.

The blue arrows shown on Figure 10 between the sources represent one-way constraints of which we make extensive use in this example to avoid incompatibility problems between the handles.

7. Conclusion

The MusicSpace system shows that it is possible to give users new degrees of freedom in sound spatialization, while preserving some semantics on the mixing of sound sources. MusicSpace provides a high level command language for moving groups of related sound sources, and may be used to control an arbitrary spatialization system. MusicSpace was connected successfully to a Midi Spatialization system for playing midi files, to a midi-controlled audio mixing console for mixing multi-track recordings, as well as to Ircam's spatialization system [6]. These applications promote the idea of dynamic mixing, where sound engineers can delegate safely a part of their responsibility in the mixing to listeners.

8. References

[1] Borning A., Anderson R., Freeman-Benson B., "Indigo: A Local Propagation Algorithm for Inequality Constraints", Proceedings of the ACM Symposium on User Interface Software and Technology, pp. 129-136, 1996.

[2] Borning A., Freeman-Benson B., "Ultraviolet: A Constraint Satisfaction Algorithm for Interactive Graphics", Constraints, Special Issue on Constraints, Graphics, and Visualization, Vol. 3 No. 1, pp. 9-32, April 1998.

[3] Eckel G., "Exploring Musical Space by Means of Virtual Architecture", Proceedings of the 8th International Symposium on Electronic Art, School of the Art Institute of Chicago, 1997.

[4] Hosobe H., Matsuoka S., Yonezawa A., "Generalized local propagation: a framework for solving constraint hierarchies", Proceedings of CP' 96, Boston, 1996.

[5] Hower W., Graf W. H., "a Bibliographical Survey of Constraint-Based Approaches to CAD, Graphics, Layout, Visualization, and related topics", Knowledge-Based Systems, Elsevier, vol. 9, n. 7, pp. 449-464, 1996.

[6] Jot J.-M., Warusfel O., "A Real-Time Spatial Sound Processor for Music and Virtual Reality Applications", Proceedings of ICMC, 1995.

[7] Lea R., Matsuda K., Myashita K., *Java for 3D and VRML worlds*, New Riders Publishing, 1996.

[8] Pachet F., Delerue O., "A Temporal Constraint-Based Music Spatializer", ACM Multimedia Conference, Bristol, 1998.