



City Research Online

City St George's, University of London

Citation: Hall, T., Armstrong, N. & Hoadley, R. (2025). Notations and Score-making. In: The SuperCollider Book. (pp. 677-706). Massachusetts, USA: MIT Press. ISBN 9780262049702

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: <https://city-ref-test.eprints-hosting.org/id/eprint/29970/>

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to [City Research Online policy](#).

26 Notations and Score-making

Tom Hall, Newton Armstrong, Richard Hoadley

26.1 Introduction

In 1959 Hiller and Isaacson speculated on the future application of computers to music, noting that '[the use of computer] by the composer to produce a written score as well as recorded sound should be highly efficient compared with current methods of composition' (Hiller and Isaacson, 1959, p.175). Aside from a few significant exceptions (see Weibel, 2020), the predominant focus of notational software development over the following decades was on fixed 'traditional' common-practice Western notation (CPWN) (Mathews & Rosler, 1968; Roads, 1981).

Meanwhile, last century compositional and performance practice in the analog domain embraced notions both of so-called indeterminacy (see Behrman, 1965) as well as the live or dynamic score. For example, Earle Brown, reflecting many years later on his early iconic indeterminate composition, *December 1952*, explained how the composition's original conception was a motorized dynamic score in a box (Brown, 2008, p.3). Similarly, by 1969 the German composer Dieter Schnebel noted the projection of scores using slides in performance, whilst altering slide durations and appearance (Schnebel, 1969, p.291). It has only been in the 21st Century that the notion of an algorithmic, interactive, dynamic or live computer-based 'digital score' that may be used in similar performance contexts has also become well-established (see for example, Vear, 2019).

Our approach in this chapter is to consider the varieties of notational experience as –to paraphrase Iverson– tools of compositional thought (Iverson, 1980). In a sense both SuperCollider code and music notational outputs can be thought of as notations, however the focus here is on case studies which illustrate the use of SuperCollider to create both CPWN as well as examples of so-called ‘graphic notation’, for which there has been renewed interest in recent years (see, for example, Sauer, 2009).¹ In doing so, we take in both ‘fixed’ notation and scores, and live or dynamic interactive notational practices. Implicit in our approach is the hands-on interactive code-based experience afforded by the use of SuperCollider in the creation of a range of music notations. In general, we aim to highlight ways of making musical notation and scores that go beyond modes of interaction using conventional music software, or present forms of notation that could not be easily attainable using such tools, e.g. dedicated CPWN software or DAWs.

We aim to demonstrate the use of SuperCollider’s rich algorithmic paradigms within two primary contexts. The first is the use of SuperCollider alone to create different forms of musical notation outlined above; we begin with by considering approaches to using traditional music glyphs within a SuperCollider digital score ‘page’, tackling Unicode and SMuFL fonts along the way. Changing tack, we then present a case study of an interactive project to make grid-based graphic scores within SuperCollider.

The two sections that follow present projects that use SuperCollider as an interactive environment to create notation and scores which are rendered visually in third party software. Using these approaches we are able to interact with other software using approaches and code interfaces native to SuperCollider, while also providing a straightforward interface for ‘plugging

in' existing capabilities afforded through other software. A custom SuperCollider class that interacts with the INScore software is presented as a means to code audio and generative processes, allowing tightly synchronized synthesis and live scores. The **Fosc** project, on the other hand, enables complex and flexible compositional interaction primarily for the production of fixed LilyPond scores – whether algorithmically generated or 'traditionally' composed.

26.2 CPWN and graphic notation within SuperCollider

26.2.1 Introduction

A benefit to working with music notations solely within SuperCollider is the tight integration between sound and image that can be obtained using the coding environment of a single piece of software. This section is divided into three parts: in the first we consider some low-level tools for managing SuperCollider as a 'blank canvas' for score and notation creation. The second part introduces classes for working directly with CPWN within SuperCollider, and the last section presents a case study in graphic score notation.² To assist in creating responsive and structured compositional environments, the overall approach has been to use model–view–controller (MVC) and other related design patterns.³

26.2.2 Page, frame, view

For many, the blank screen now replaces the former blank page or canvas. In two-dimensional digital or printed work, the aspect ratio of the dimensions of a computer screen is a useful starting point, creating as they do the look and 'frame' of a work's presentation. The SuperCollider helper class **MITHScreenRatios** assists in translation between the 'frame' of pixel dimensions at a given ratio (e.g. 16:9) and a required ratio and available size to fit within it.

For example, the following code returns the dimensions of the largest frame with the ratio of an ANSI standard US Letter (11: 8.5) that will fit within the current screen pixel dimensions in landscape orientation: **MITHScreenRatios().ratioToDims(\usLetter)**. Below, the same ratio is used to set dimensions for a **Window** in the view class **MITHRatiosView**, then resized to a portrait-oriented *iso* ratio (**[sqrt(2): 1]**, e.g. A4) frame, adjusted to fit below the OS menu bar (on macOS and similar):

```
m = MITHScreenRatios(); m.viewDims_(m.ratioToDims(\usLetter)); MITHRatiosView(m);  
  
m.viewDims_(m.ratioToDims(\iso, landscape: true));
```

Whilst **MITHScreenRatios** assists with set-up of a digital page, **ViewCentral** and the MVC ‘façade’ design (Gamma et al., 1995) subclass **WinBlock** are tools for presenting work within a digital page centered within a full-screened window (this is achieved by nesting **VLayout** and **HLayout** in the view class). The ‘block’ models the traditional textblock used in page design for text or music, i.e. the area surrounded by the margins (see Bringhurst, 1999, Chapter 8). As suggested in Figure 26.1, using the class’ **gui** method, the block can be moved within the page to create an intentional design. We return to these classes in the next subsections.

Figure 26.1: **WinBlock** used with **MITHScreenRatios**

```
// dimensions for an iso ratio landscape Window, width 800px  
  
~dims = MITHScreenRatios().ratioToDims(\iso, 800) // default is m.screenDims  
  
// make a WinBlock, args: win, dims, margins, usrViewBool  
  
b = WinBlock.new(dims: ~dims); b.moveWin(0, 0)
```

```

// view is by default an UserView

(

b.marginCol_(Color.white);

b.view.drawFunc = { |self|

    Pen.addRect(Rect(0.5, 0.5, self.bounds.width-1, self.bounds.height-1));

    Pen.stroke;

};

b.view.refresh;

)

b.viewDims // return current block dimensions

// experiment with blockSize / ratio

g = b.gui

// shape view dimensions

d = MITHScreenRatios().ratioToDims(\r5_3, ~dims * (5/6));

b.viewDims_(*d); b.centerView;

```

```
// show area revealed in fullscreen mode

b.winCol_(Color.new255(255, 85, 0)); b.resizeWin(*(~dims * 1.2));

// timed fullScreen, also ESC key (MBP) to endFullScreen

r{ b.fullScreen; 3.wait; b.endFullScreen}.play(AppClock)

b.shrinkWin // hide area beyond main margin

g.close; b.close;
```

26.2.3 Unicode and common-practice Western notations

Having introduced simple tools for shaping and presenting the digital page, we next consider tools to assist working in SuperCollider with extended character sets such as those required for displaying CPWN music notation fonts. The easiest way to display CPWN musical symbols – ‘glyphs’ – within SuperCollider is in a **Window**, within which a specialist music font can be specified for displayed **Strings**.⁴ Since these characters are non-ASCII, it is irksome to enter them via a computer keyboard. The Unicode standard assigns a hexadecimal ‘code point’ to individual symbols and character sets.⁵ The class **MITHUnicode** assists with translation between Unicode code points and their encoding within SC as multi-byte UTF-8 characters (see also the related Quark **Strang** which shares some functionality). As with **ControlSpec**, the main way of accessing the class is via methods to other classes, e.g **String**. For the ‘musical keyboard’ Unicode character: "".asCodePoint // U+1F3B9 and "U+1F3B9".asGlyph // .

Standard Music Font Layout fonts (‘SMuFL’)⁶ standardize access to music notation glyphs via Unicode code points. An example is Steinberg’s ‘Bravura’ font.⁷ Once the font is downloaded

and installed on a computer, the code below should display a treble clef⁸ in a SuperCollider **Window**; also shown is concatenation of glyphs and ASCII text.

```
(  
  
w = Window.new.front;  
  
a = StaticText(w, Rect(100, 50, 300, 300));  
  
a.font = Font("Bravura", 72);  
  
a.string = "U+E050".asGlyph  
  
)  
  
a.string = ["U+1F1FA", "U+1F1E6"].scramble.asUnicodeString
```

Since the number of glyphs is very large, the **SMuFLtools** classes enable convenient browsing and viewing of the SMuFL font Bravura. Font data is parsed from a downloadable JSON file⁹ (a demo subset is included with the class) and font information and the glyph is rendered in a SuperCollider **Window**.

The façade class **MITHNoteViewer** uses **MITHUnicode** and **ViewCentral** to display CWMN dynamically using a SMuFL font (default is Bravura: install before using the class). There's also a default glyph mapping class **MITHNoteMidiMap**, which enables regular 12-TET notational display in a manner similar to Max software's nslider object¹⁰ (a custom mapping class could be substituted to display glyphs for, e.g. a SMuFL microtonal notation). The simplest view shows a single note, added as a MIDI integer. Adding a new note clears the existing one, as does the applying the 'clear' method. A Boolean flag is used to force display of accidentals; the default mapping uses sharp signs for accidentals, a negative MIDI integer spells the note as a flat. (This mapping can be overridden as desired by altering the **accidentalsMap** Event method variable, or

through using the **addCustom** glyphs method). Figure 26.2 code creates a view with eight columns, each of which is accessed by a slot integer, or else using the **addAll** method via an **Array**. The figure demonstrates note creation, removal and color being changed dynamically, as also shown in screenshot Figure 26.3.

Figure 26.2: **MITHNoteViewer** dynamic note array presentation with **Synth**.

```
(  
  
s.waitForBoot({  
  
  SynthDef(\percSine, {|midi = 69, mul = 0.2, out=0|  
  
    var sig, env = Env.perc(releaseTime:0.4);  
  
    sig = SinOsc.ar(midi.midicps, mul: mul)  
  
    * EnvGen.kr(env, doneAction: Done.freeSelf);  
  
    Out.ar(out, [sig, sig])  
  
  }).add;  
  
  l = MITHNoteViewer.new(8, 350);  
  
  a = [60, 62, -63, 65, 67, 69, 71, 72, -70, -68, 67, 65, -63, 62, 60];  
  
  r = Routine{  
  
    l.fullScreen;  
  
    1.wait;  
  
    a.size.do{|i|  
  
      Synth(\percSine, [\midi, a[i].abs]);  
  
      l.add(a[i], slot: i.fold(0, 7), color:Color.red);  
    }  
  }  
})
```

```

        if(p.notNull){l.glyphsCol_(Color.black, p)};

        p = i.fold(0, 7);

        (60/94).wait

    };

    l.glyphsCol_(Color.black, p);

    p = nil;

    l.endFullScreen;

    2.wait;

    l.clearAll;

    l.close

};

AppClock.play(r)

})

)

```

Figure 26.3: Screenshot of **MITHNoteViewer** as shown in Figure 26.2.

[Figure 26.3 IMAGE HERE]

26.2.4 Model, grid, score

Working directly with music glyphs purely within the SC environment soon becomes complex; possibilities for other music notations involving graphical symbols directly created in

SuperCollider are more programmatically manageable. The idea of a grid as a notational model for music goes back many centuries (Higgins, 2009), and can be thought of as a level of abstraction away from modern CWMN. A literal visual grid or graph was utilized by US composer Morton Feldman from 1950 in the score notation of his so-called graph music (Cline, 2016). A related graph approach later formed the basis of the model for the interactive GRIN (GRaphical INput) language cited in the chapter introduction (Mathews & Rosler, 1968). This section concludes with a project, **GMITH**, which uses SuperCollider to create a dynamic compositional environment for making Feldman-like notation and finished musical score.¹¹ The code design uses a hierarchy of MVC classes to draw a graphic score using the **Pen** class, and an **Array**-centric score class to represent the customizable graphical elements.

GMITH's score class makes a distinction between the type of musical event presented within a grid and the customizable form of its presentation. Feldman's scores are similar to 'piano roll' layout of musical events, in which an X-Y grid specifies units of time and pitch, respectively, except that the conception of pitch is somewhat indeterminate: precise pitch is not specified beyond high, medium or low (for any given instrument).¹² Feldman used three rows to indicate these relative registers, with integers within cells to indicate number of notes and various symbols or text to indicate playing technique (e.g., a diamond for a note played as a harmonic). Figure 26.4 shows a short example of the system, in which a performance event is represented as an **Association** of arrays for grid location and event type. Score events can also be added or removed singly via the add or remove methods (or clean slate via removeAll).

If the score is displayed, as here, with one or more 'galley' views, clicking on a cell will post its **[row, column]** location and, if relevant, event information to the post window. Other interactive mouse actions which use modifier keys include <ALT>-click to remove a score event and

<CTRL>-click to add a single cell event, or after clicking a cell location, use the method **addMouseCell** to create more complex or multi-cell events (see Figure 26.4). Lastly, to move an event, the **move** method can be accessed using a mouse by clicking on an event, then <SHIFT>-clicking to a new cell location, which will move the last clicked event to this new location. Evaluating the **undo** method will revert to the previous score state for these actions.

Score events can be sorted by row or column and posted using the **postEvents** method. In order to create a flexible compositional environment, the galley view can be expanded, for example, to display empty cells using the **gvColRange_** method, and score events can be moved by column using the **moveAll** method. Morton Feldman varied the format of the grid display itself, but typically indicated groupings of 4 icti using dotted 'bar' lines. The **gvGridPatterns** method allows customization of that part of the notation, in which arrays set the displayed pattern. Within these arrays, the integer 1 creates a solid line, 0 no line and -1 a dotted line. Solid lines (the default) is represented as [1], [1]. A variation of blank, dotted and solid lines is used in Figure 26.4 (as seen also in the inner part of Figure 26.5).

Figure 26.4: Simple **GMITH** score creation and galley view display.

```
(  
  
l = GMITH.new;  
  
// [row, column] -> [number, type, duration]  
  
l.addAll([  
  
    [1, 0],  
  
    [2, 1] -> [3],  
  
    [0, 2] -> [nil, \h],
```

```

[1, 3] -> ["Pz 3", nil, 2],

[2, 5] -> [[2, 1]],

[0, 6] -> [nil, \a]

]);

)

// |key, rows, cols, vHeightPct, win|

w = l.makeGalley(\gv1, cols: 8)

// to complete the score, click the bottom right (empty) cell, then evaluate:

l.addMouseCell([nil, \p]);

// post sorted score events l.scoreArr

l.postEvents(order: 0) // row order. order: 1 posts column order

// grid line patterns

l.gvGridPatterns(\gv1) // [ [ 1 ], [ 1 ] ]

l.gvGridPatterns_(\gv1, [1, 0, 0], [-1, 0, 0, 0]);

// display adjustments

l.gvColRange_(\gv1, 0, 63)

```

```

l.gvGridLineWidth(\gv1)

l.gvShapeLnWidth_(\gv1, 2) // adjust, default is 3

l.gvFontSizeScale_(\gv1, 0.8) // adjust

l.moveAll(12); // move score events to columns to the right

l.moveAll // return to 0 column as first event

l.gvColRange_(\gv1); // default view range is filled cell range

```

GMITH can also make a score comprising multiple screen-based ‘pages’ for printing or dynamic display, as shown in in Figure 26.5 (code for this example is included in the chapter’s code listing). As with the galley view, cell events are updated as relevant if the score is changed. This is achieved via a hierarchy of classes including **WinBlock shown earlier**, into which a custom class, **NotnFlow** is inserted. **NotnFlow** uses the **Layout** design as well as a class, **TextModel** as a model for placing **StaticText** into the page as needed. Page elements can be adjusted via the page’s main model, as shown . An effective process using these classes is to decide view dimensions and margins, then adjust system(s) size (width and height) to match the size and ratio of the **UserView** that contains the notation system notation, as determined by **pageSysSqSize_**. Future work could include further consideration of row management and additional approaches to live dynamic display in a performance context.

Figure 26.5: Example of **GMITH** score view using makePage method

[Figure 26.5 here]

26.3 INScore

26.3.1 Introduction

INScore is an “environment for the design of interactive, augmented, dynamic musical scores”¹³ in which one can dynamically create, enhance and maneuver a variety of visual media, including graphic objects, text and musical CPWN. INScore’s elasticity with these media, which are in the main created and manipulated programmatically, make it a useful notational tool.

A key motivation for using SuperCollider with INScore for musical composition is the ability to create unified interfaces enabling live generative audio processes to be synchronized in real-time with a live-generated performance score of arbitrary complexity (Hoadley, 2017). These processes are coded in SuperCollider using the custom classes, which communicate with INScore via the Open Sound Control (OSC) protocol (Wright & Freed, 1997).

We begin by demonstrating the creation of both CPWN and graphic images, with an emphasis on ‘dynamic’ live notation. The concluding sections demonstrate the use of the software using a case study of a completed composition with links to online videos.

26.3.2 INScore and SuperCollider

Whilst there are a number of well-documented ways of working with INScore,¹⁴ the approach taken here is to design composition and score data within SuperCollider, which is then sent to the INScore app for real-time display. Although INScore allows for two-way communication with another application using OSC,¹⁵ in general it is more efficient to construct and transform INScore’s object attributes within SuperCollider using the **INScoreSceneMIT** class.¹⁶ This process can then be synchronized with audio generated in SuperCollider. For creating CPWN, we focus on the capabilities of INScore’s embedded GUIDO engine,¹⁷ which we demonstrate next.

Once the INScore app is downloaded and installed,¹⁸ depending on your OS and installation location you should be able to open within SuperCollider by evaluating something like `"Applications/INScoreViewer.app".openOS`. The INScore app by default will open a Window scene called ‘scene’ to which notational data can be sent. Working natively within the app, to display a middle-C quarter-note using GUIDO music notation (‘gmn’), we could open a plain text file (extension ‘inscore’) containing `/ITL/scene/score0 set gmn "[ c ]"`; To do this at a low level within SuperCollider using OSC, we set up a **NetAddr**, and send the message as follows:

```
// assumes INScoreViewer listening for OSC on port 7000

~addr = NetAddr.new("localhost", 7000);

~addr.sendMsg("/ITL/scene/score0", \set, \gmn, "[ c ]");
```

The **INScoreSceneMIT** class simplifies interaction with INScore and can also co-ordinate broadcasting to multiple computers. The following code uses this class to display the equivalent of the example above:

```
~scene = INScoreSceneMIT.new("localhost", 7000);

~scene.note(\c);
```

INScore allows multiple windows (‘scenes’), each of which could be controlled by a **INScoreSceneMIT** class instance. Using the **addINScoreView** method allows synchronized control of a single scene across multiple computers, for example:

```
// synchronize with a remote machine

~scene.addINScoreView(NetAddr.new("192.168.178.64", 7000)) // needs correct remote IP address
```

The class **INScoreManagerMIT** manages bi-directional communication with the INScore application, can be used to query the application on remote computers to confirm, for instance,

scene window creation and deletion. This is useful when a remote screen is not visible, such as when separate computers are used for each performer, but managed by a primary computer. In cases, however, where only a local machine is being used, the **INScoreSceneMIT** class is adequate for basic scene creation and deletion. The following explicitly shows the second argument to the note method, **componentNum**, which in this case is used to create an INScore object **score0** to which the GUIDO note data string is sent, as follows:

```
j = INScoreSceneMIT.new("localhost", \joni).window

j.foreground;

j.note(pitch: "b&1/2 a", componentNum: 0);

j.close; // close window
```

Returning to our main window, the following shows a more complex example of a GUIDO string display using the **staves** method, where each voice is surrounded by square brackets:

```
// make chords manually (_ for rest), two staves

~scene= INScoreSceneMIT.new; // new class instance as needed

~scene.staves("[{a/4, b, c} {b3,e2, g&} _/8 {c#2/4., e0}], [{g/1, f}]" );
```

This is followed by the use of a small GUIDO string manipulation helper class:

```
// helper string GUIDO string manipulation class

g = ~scene.guido; // otherwise: g = SimpleGuidoMIT.new;

~scene.note(pitch: g.clef(\f) + g.meter(3, 8) +g.key(-2) + "c0/8. b-1");
```

To create a ‘dynamic’ score, one could create a function containing an algorithm to create a short unmeasured sequence of chords. This is shown in Figure 26.6, which uses the **midiNoteMap** method to translate between integers and GUIDO notation.¹⁹

Figure 26.6: Dynamic CPWN using a function to generate an algorithmic chord sequence.

```
~scene= INScoreSceneMIT.new; // new class instance as needed

(

~randomChordsFn = {

    var chordArr, chord3, noteFN, chosenDur, melodicArr;

    var guidoOutput = "", guidoChord = [], g = ~scene.guido;

    noteFN = {rrand(30, 90)};

    chordArr = [[50, 54, 57, 59], [34, 56, 78], 3.collect({noteFN.value})];

    chord3 = [chordArr[1], chordArr[1]].choose;

    melodicArr = [chordArr[0], chordArr[0]+[2, -2].choose, chord3, chordArr[2]];

    melodicArr.do({| chord|

        chosenDur = ["/16", "/4", "/2."].choose;

        chord = chord.collect({| note| g.midiNoteMap(note, "mixed") ++ chosenDur});

        chord = [g.makeChord(chord), "_" ++ chosenDur].wchoose([0.8, 0.2]);

        guidoOutput = guidoOutput + chord

    });

    guidoOutput;
```

```

};

~scene.osaActivate;

~scene.verbose_(false);

Routine.new({

    6.do{{|i|

        ~scene.note(pitch: ~randomChordsFn.value);

        ~scene.htmlFull(fontsize: "40pt", text: (i+1) + "/" 6");

        ~scene.move(-0.4, -0.6, component:"html");

        2.5.wait

    }};

    ~scene.clear; // clear existing scene

}).play;

)

```

Before we turn to other types of graphic objects, Figure 26.7 presents a further CPWN example with audio, using a **Routine** to play a dynamic score in combination with a simple **Synth**.

Figure 26.7: Dynamic score with audio.

```

~scene= INScoreSceneMIT.new // new class instance as needed

(

var acc, glyph, glyphRed, string, oldString;

s.waitForBoot({

```

```

SynthDef(\percSine, {|midi = 69, mul = 0.2, out=0|

    var sig, env = Env.perc(releaseTime:0.4);

    sig = SinOsc.ar(midi.midicps, mul: mul)

    * EnvGen.kr(env, doneAction: Done.freeSelf);

    Out.ar(out, [sig, sig])

}).add;

m = [60, 62, -63, 65, 67, 69, 71, 72, -70, -68, 67, 65, -63, 62, 60];

g = ~scene.guido;

string = g.stemsOff;

oldString = string.copy;

~scene.osaActivate;

r = Routine.new({

    1.wait;

    m.do({|i, j|

        Synth(\percSine, [\midi, i.abs]);

        acc = if(i.isNegative){"flats"}{""};

        glyph = g.midiNoteMap(i.abs, acc);

        glyphRed = g.noteFormat(color: \red) + glyph;

        // start a new string of pitches

        if(j==8){

```

```

        string = g.stemsOff + g.midiNoteMap(m[j-1].abs, acc);

        oldString = string.copy;

    };

    string = (oldString + glyphRed).postIn;

    oldString = oldString + glyph;

    ~scene.note(pitch: string);

    (60/94).wait

});

0.5.wait;

~scene.note(pitch: oldString);

1.wait;

~scene.note(pitch: g.emptyStaff);

}).play;

})

)

```

INScore allows the creation and manipulation of a wide range of non-CPWN graphic objects (based on the open standard SVG XML-based image format),²⁰ regarding which the reader is directed to INScore's online documentation. The following examples demonstrate the creation and manipulation of a simple geometric shape. First we may wish to scale, move or delete the existing GUIDO **score0** scene object:

```
// scale, move or delete the existing GUIDO score0 scene object
```

```
~scene= INScoreSceneMIT.new // new class instance as needed
```

```
~scene.note(~scene.guido.emptyStaff); // from previous example
```

```
~scene.move(y: -0.8).scale(0.5);
```

```
~scene.deleteComponent(component: "score", componentNum: 0);
```

Next we can create a polygon and change its color as follows:²¹

```
~scene.poly([0.7, 0.05, 0.1, 0.1, 0.1, 0.0]);
```

```
~scene.color(\orange, component: "poly"); // https://en.wikipedia.org/wiki/Web\_colors
```

INScore's graphic space ranges by default from -1, to 1 in the usual dimensions, with 0 being the centre of a Window. An object's position can be changed by either relative or absolute positioning; the following places the centre of the polygon at the center or either horizontal edge of the scene window:

```
~scene.move(x: [-1, 0, 1].choose, component: "poly");
```

INScore's graphic space is designed to be stretchable: changing the Window size scales the size of the objects within it, and the graphic space can be extended or reduced from the default +/-1.

This can cause complexities in scaling objects and windows on screen or via a projector in fullscreen mode. Building on the INScore *poly* object created above, Figure 26.8 shows a screenshot from an example animation that combines multiple dynamic objects, including CPWN. The code listing is included in chapter code listing. This example demonstrates a simple approach to screen ratio and window management; the example temporarily fullscreens the INScore window.

Figure 26.8: Screenshot for **INScore** scene scaling, object and dynamic CPWN animation

[Figure 26.8 HERE]

26.3.3 *Unthinking Things*

The interactive nature of using INScore makes it a powerful tool for the creation and public performance of live, dynamic scores. In this scenario, musicians may either read from a separate computer screen or from a publicly displayed screen.²² We next turn to a case study using SuperCollider and INScore in order to give a sense of an entire musical composition created using these tools.

Unthinking Things is written for sixteen-voice mixed choir, physical materials and electronics, one of a series of compositions by Hoadley involving dynamic, generative music notations.²³ The composition includes both CPWN, text and non-traditional graphic visual objects placed and moved within a ‘scene’.²⁴ Displaying the influence of Cornelius Cardew’s graphic score *The Great Learning* (Cardew: 1971), this approach allows non-metric music without requiring complex forms of CPWN. INScore allows the graceful and interconnected dynamic combination of all of these elements.

Figure 26.9 shows an example taken the ‘metals’ section of the piece, which comprises a generative audio track made from looped recordings of singing bowls and the real-time sounds of the performers’ striking resonant metal objects. The figure shows varied elongated isosceles triangles similar to those shown in Section 26.3.2, intentionally indicative of an amplitude envelope, but allowing interpretation by the performer according to the visual properties of the triangle.²⁵

Figure 26.9: Screenshot taken from the “metals” section of *Unthinking Things*.

[Figure 26.9 HERE]

Given that the performer reads the notations in real time, there are two principal options for a dynamic score: the notations move against a static cursor, or a cursor moves across a field of static objects (the latter best reflecting standard fixed notation). The ‘metals’ section in the piece uses the former approach, whereas the ‘stones’ section uses the moving cursor model (Figure 26.10). For this section, performers are asked to bring along stones which they can hit or rub together,²⁶ offering a sound palette of point-like, dry precision in contrast to the resonance of metals. As a result the algorithms of the ‘stones’ section concentrate on rhythms, contrasting pulses and flock-like clouds of particles.

Figure 26.10: A stone ‘constellation’ from *Unthinking Things*

[Figure 26.10 HERE]

INScore is used in *Unthinking Things* to create a score with representative, decorative and performance-focused features. Figure 26.11 show screenshots from sections using CWPWN elements as in traditional performance scores. The first section a), features a graphical intertwining of notations and text, the latter read live and so displayed in an eye-catching color prefixed by the performer’s number. The music notation as presented is rendered by a variety of synthesized and sampled keyboard and harp-like sounds. Figure 26.11 b) shows an approach to multi-part writing for voices.

Figure 26.11: Other notations including CPWN from *Unthinking Things*.

a) text and CPWN (‘melismata’)²⁷

b) ‘vocal rhythms’²⁸

[Figure 26.11 IMAGES HERE]

26.4 Fosc

26.4.1 Introduction

Fosc (**F**ormalised **S**core **C**ontrol/**F**or **S**uper**C**ollider) is an Application Programming Interface for generating musical notation in LilyPond²⁹ using SuperCollider. **Fosc** is a close relative of Abjad³⁰ and ports much of Abjad's Python code base to SuperCollider. The motivation for the initial development of Fosc was to fill a gap in the software tools available within existing computer-assisted composition environments. In short, Fosc was designed to feel native to SuperCollider users.

Fosc provides an object-oriented interface to LilyPond's engraving library, allowing for the generation and transformation of musical scores of arbitrary complexity using extended CPWN. LilyPond's engraving library comprises a vast number of typographical elements. While **Fosc** provides low-level access to all of these elements, composers working with algorithmic systems will in most instances want to maintain focus on high-level concepts and constructs. To this end, **Fosc** provides a single, customisable and easily extensible factory class for the high-level algorithmic specification of musical materials.

Fosc has been used extensively for compositional projects since 2018 by Armstrong. One advantage enabled by the system is that it affords a wide range of styles of working. These range from 'pure' approaches — in which the final score is generated entirely as a patchwork of algorithmically specified outputs (Armstrong, 2018) — to work (Armstrong, 2019) that emerges as a more reflective 'back-and-forth' between formal specification and informal transcription and reinterpretation.

The section that follows begins with a short overview of low-level **Fosc** usage scenarios, and concludes with examples using the high-level **FoscMusicMaker** factory class. **Fosc** and LilyPond both need to be installed to run the code examples.³¹

26.4.2 Low-level interface

Fosc's low-level interface models musical notation at the level of notes, chords, voices, staves, staff groups, and the score. Figure 26.12 presents a simple example.

Figure 26.12: Display notes on a staff.

```
(  
  
a = [FoscNote(60, 1/4), FoscNote(62, 1/8), FoscNote(63, 1/8), FoscNote(65, 1/2)];  
  
FoscStaff(a).show;  
  
)
```

Figure 26.13: Result of code in Figure 26.12.

[Figure 26.13 IMAGE HERE]

Indicators of various types can be attached to the notes and chords in a musical score. Indicators include articulation and dynamic markings, along with many other standard musical typographical elements (time signatures, clefs, fermata signs, ottava markings, etc.). Figure 26.14 shows indicators attached to a **FoscNote**.

Figure 26.14: Attach articulation and dynamics marks to a note.

```
(  
  
a = FoscNote(60, 1/4);
```

```

a.attach(FoscArticulation('>'));

a.attach(FoscDynamic('f'));

a.show;

)

```

Figure 26.15: Result of code in Figure 26.14.

[Figure 26.15 HERE]

Typographical elements that span sequences of notes and/or chords — such as slurs, ties, and dynamic hairpins — can be added using **Fosc**'s selection mechanism and spanner methods, as shown in Figure 26.16.

Figure 26.16: Attach slur and dynamic hairpin spanners to a selection of notes.

```

(

a = [FoscNote(60, 1/4), FoscNote(62, 1/8), FoscNote(63, 1/8), FoscNote(65, 1/2)];

b = FoscStaff(a);

b.selectLeaves.slur.hairpin('p < f');

b.show;

)

```

Figure 26.17: Result of code in Figure 26.16.

[Figure 26.17 HERE]

Fine-grained formatting of typographical elements can be performed by attaching LilyPond literal strings directly to score objects (Figure 26.18).

Figure 26.18: Attach a LilyPond literal string to the first note in a selection.

```
(  
  
a = [FoscNote(60, 1/4), FoscNote(62, 1/8), FoscNote(63, 1/8), FoscNote(65, 1/2)];  
  
b = FoscStaff(a);  
  
b[0].attach(FoscLilyPondLiteral("\\tweak NoteHead.style #'harmonic"));  
  
b.show;  
  
)
```

Figure 26.19: Result of code in Figure 26.18.

[Figure 26.19 HERE]

Fosc supports a range of *mutation* methods for transforming selections of music in place. These include methods for pitch transposition and duration scaling, as shown in Figure 26.20, each part of which shows the original staff and mutated copy.

Figure 26.20: The **transpose** and **scale** mutation methods.

```
// Make a copy of a staff. Transpose the contents of the copied staff up two semitones  
  
(  
  
a = [FoscNote(60, 1/4), FoscNote(62, 1/8), FoscNote(63, 1/8), FoscNote(65, 1/2)];  
  
b = FoscStaff(a);  
  
b.selectLeaves.slur.hairpin('p < f');  
  
c = b.deepCopy;  
  
mutate(c.selectLeaves).transpose(2);
```

```

FoscScore([b, c]).show;

)

// Make a copy of a staff. Scale the contents of the copied staff by a factor of 2

(

a = [FoscNote(60, 1/4), FoscNote(62, 1/8), FoscNote(63, 1/8), FoscNote(65, 1/2)];

b = FoscStaff(a);

b.selectLeaves.slur.hairpin('p < f');

c = b.deepCopy;

mutate(c.selectLeaves).scale(2);

FoscScore([b, c]).show;

)

```

Figure 26.21: Results of code in Figure 26.20.

[Figure 26.21 a) and b) HERE]

26.4.3 Making sound

Fosc score objects maintain an internal representation of note, duration, and amplitude parameters as a list of Events, accessible through an *eventList* property. Score objects can be played directly in *scsynth*, as shown below using the default settings:

```
s.waitForBoot { FoscChord([60,64,67], 1/2).play };
```

The following shows how a sequence of notes may be played back using the default settings.

Note how the musical dynamics are reflected in playback:

```
(
  a = FoscStaff((60..72).collect { |pitch| FoscNote(pitch, 1/16) });

  a.selectLeaves.hairpin('ppp < fff');

  a.play;
)
```

The *eventList* property allows for the default settings to be transformed and/or overridden.

Figure 26.22: Provide a custom SynthDef for playing a FoscStaff.

```
(
  SynthDef('ping', { |midinote=60, amp=1|

    var envgen, src;

    envgen = EnvGen.kr(Env.perc(0.01, 0.5), doneAction: 2);

    src = Pan2.ar(SinOsc.ar(midinote.midi cps, 0, amp), 0);

    OffsetOut.ar(0, src * envgen);

  }).add;

)

(
  a = FoscStaff((60..72).collect { |pitch| FoscNote(pitch, 1/16) });

  a.selectLeaves.hairpin('ppp < fff');

  Pbindf(Pseq(a.eventList), \instrument, 'ping').play;
)
```

26.4.4 High-level interface: **FoscMusicMaker**

FoscMusicMaker is a factory class for the high-level algorithmic specification of musical materials. It encapsulates the more granular detail of **Fosc**'s low-level classes, allowing composers to focus on the musical parameters that are generally of the most immediate interest when composing with standard musical notation: durations, rhythms, and pitches.

FoscMusicMaker follows a two-step configure-once / call-repeatedly pattern. At the configuration stage, preferences can be specified for beaming and the spelling of durations and rhythms. A **FoscMusicMaker** instance is called using the **value** method (the following examples use the method shortcut in the form **a.(args)**). The same instance can be called repeatedly to produce multiple musical variants from distinct sets of parameters. In Figure 26.23, an instance of **FoscMusicMaker** is called with *durations* and *pitches*. The sequence of *pitches* repeats cyclically to match the length of the sequence of *durations*.

Figure 26.23: Simple usage of **FoscMusicMaker**.

```
(  
  
a = FoscMusicMaker();  
  
b = a.(durations: [1/4, 1/8, 1/8, 1/2], pitches: #[60,62]);  
  
b.show;  
  
)
```

Figure 26.24: Result of code in Figure 26.23.

[Figure 26.24 HERE]

The basic *durations* and *itches* usage of **FoscMusicMaker** is suitable to musical situations based on simple *additive* rhythmic structures. More complex *divisive* rhythmic structures can be realized using the *divisions* argument in calls to a **FoscMusicMaker** instance, as shown in Figure 26.25. *Divisions* are rhythmic ratios that embed into *durations*.

Figure 26.25: **FoscMusicMaker** used to create *divisive* rhythmic structures.

```
// Embed divisions into durations. Negative divisions are interpreted as rests

(

a = FoscMusicMaker();

b = a.(durations: [1/2, 1/8, 3/8], divisions: #[-1,3],[3,2],[2,2,-3]], pitches: #[60,62]);

b.show;

)

// Divisions can be nested using a variant of rhythm-tree syntax

(

a = FoscMusicMaker();

b = a.(durations: [1/2, 1/8, 3/8], divisions: #[-1,3],[3,2],[2,[3,2]],-3]], pitches: #[60,62]);

b.show;

)
```

Figure 26.26: Results of code in Figure 26.25.

[Figure 26.26 a) and b) HERE]

Patterns can be ‘etched’ into musical sequences using the *mask* argument in calls to a **FoscMusicMaker** instance. As shown in Figure 26.27, when a *mask* is applied, contiguous musical events are fused in groupings that correspond to the group sizes in the *mask* array.

Figure 26.27: Melodic sequences demonstrating the use of a *mask*

```
// An unmasked periodic sequence. The sequence of divisions repeats cyclically to match the length of the sequence of
durations

(

a = FoscMusicMaker();

b = a.(durations: 1/4 ! 4, divisions: #[[1,1,1,1]], pitches: #[60,62]);

b.show;

)

// The same sequence with a [2, 1] mask applied. The mask pattern repeats cyclically. Pitches are added after the mask is
applied.

(

a = FoscMusicMaker();

b = a.(durations: 1/4 ! 4, divisions: #[[1,1,1,1]], mask: #[2,1], pitches: #[60,62]);

b.show;

)

// The same sequence with a [2, -1] mask applied. Negative mask values are interpreted as rests.

(

a = FoscMusicMaker();
```

```

b = a.(durations: 1/4 ! 4, divisions: #[[1,1,1,1]], mask: #[2,-1], pitches: #[60,62]);

b.show;

)

```

Figure 26.28: Results of code in Figure 26.27.

[Figure 26.28 IMAGES HERE]

Despite its simple interface, **FoscMusicMaker** provides a framework for complex and expressive specification of musical materials. Multiple variations on an algorithmic idea can be easily generated as standalone segments of music. The musical segments can be further decorated with indicators and spanners, allowing for a high level of specification and customization of notational detail.

26.4.5 FoscMusicMaker: An extended example

The interface provided by **FoscMusicMaker** is generic by design. It affords customized extensions — whether through subclassing or encapsulation — that are oriented towards well-defined *types* of musical material. A compositional project may comprise many such types of material, and many variants of each type. Such an approach allows for compositional projects to evolve incrementally, defining new types of material along the way, while at the same time continuously refining the specification details of extant types. As new musical segments are generated through calls to individually customized **FoscMusicMakers**, they can be inserted into an expanding musical score.

A simple class, **InconjunctsMaker**, is included as an example of how **FoscMusicMaker** can serve as a basis for customized extensions. The example in Figure 26.29 calls

InconjunctionsMaker's **value** method to recreate a texture for a quartet of violins that appears at measure 103 of Brian Ferneyhough's *Inconjunctions* (2014).

Figure 26.29: A customized extension of **FoscMusicMaker**: **InconjunctionsMaker**.

```
(  
  
~maker = InconjunctionsMaker();  
  
~selections = ~maker.(  
  
    durations: #[[4,8]],  
  
    // divisions: the rhythmic ratio/s that embed/s into durations  
  
    divisions: #[3,2,1,2].wrapExtend(24),  
  
    // groupSizes: per voice segmentations  
  
    groupSizes: #[  
  
        [5,5,5,5,4],  
  
        [6,6,6,6],  
  
        [5,6,7,6],  
  
        [3,4,5,6,6]  
  
    ],  
  
    // pitches: per voice ordered pitch sequences that repeat cyclically at each segment  
  
    pitches: #[  
  
        "gqf" bf" cqf"" d"" ef""",  

```

```

        "d' g' bqs' aqf'' bf'' bqf'",

        "aqf d' af' bqs' gqf'' af'' bqf'",

        "gqs aqf d' g' af' bqs'"

    ],

    // hairpins: per voice, repeating cyclically at each segment

    hairpins: #['fff > f'],

    // articulations: per voice, applied cyclically to the first event in each segment

    articulations: #['>'],

    // finalize: ad hoc per voice modifications

    finalize: { |sel, i|

        // override default spelling of tuplet ratio

        sel.selectComponents(FoscTuplet).do { |tuplet|

            tuplet.denominator = 4;

            tuplet.forceFraction = true;

        };

    };

};

// collect selections of music into a score and add final details

~score = FoscScore(~selections.collect { |sel| FoscStaff([FoscVoice(sel)]) });

```

```

~score.selectComponents(FoscStaff).do { |staff, i| set(staff).instrumentName = "Vln. %".format(i + 1) };

~score.leafAt(0).attach(FoscTimeSignature([4,8]));

~score.show(staffSize: 14);

)

```

Figure 26.30: Results of code in Figure 26.29.

[Figure 26.30 IMAGE HERE]

There would be little to be gained from creating a custom class if one were interested only in producing the single measure of music in Figure 26.30. The power of customized factory classes that extend **FoscMusicMaker** lies in their capacity to produce varied populations of musical material. For instance, a wide range of musical outcomes can be generated by calling **InconjunctionsMaker** with algorithmic variations on any or all of the *durations*, *divisions*, *groupSizes*, *pitches*, *hairpin*, and *articulations* parameters. One could even traverse the multi-dimensional parametric space combinatorially, filtering the output populations by user-specified preference rules.

26.5 Concluding Thoughts

As practitioners involved with different types of music notation, we believe different forms of musical notation broaden the possibilities of sonic thought and its expression. In support of this idea, we have created different interactive computer-assisted composition and notational environments using a wide variety of notations, fixed, algorithmic, ‘graphic’ and traditional and extended CPWN.

For sound practitioners working with SuperCollider, the integration of notational tools into ways of working can leverage fluency and expressivity gained through programming experience into their sonic work. Whether using SuperCollider on its own or in conjunction with other specialist notational software, we have suggested that tight integration between approaches to the sonic and the visual creates benefits for work involving notation, whether fixed and live. We are aware that each of the projects in this chapter presents certain musical and notational affordances as well as constraints. Each brings styles of interaction to the fore that we hope may suggest new and effective approaches to compositional thinking with sound and its visual prescriptions and representations.

References

Armstrong, N. 2018. 'Thread—surface,' in Armstrong, N., 2019. *The way to go out* [CD].

Another Timbre, at167.

Armstrong, N. 2019. 'A line alongside itself,' in Armstrong, N. 2019. *The way to go out* [CD].

Another Timbre, at167.

Behrman, D. 1965. 'What Indeterminate Notation Determines', *Perspectives of New Music*, Vol 3, No. 2, 58-73.

Berkeley, G. 1710. *A Treatise Concerning the Principles of Human Knowledge*. Dublin: Jeremy Pepyat. Available online: <https://www.gutenberg.org/ebooks/4723>

Bringhurst, R. 1999. *The Elements of Typographic Style*. 2nd Edn. Point Roberts, WA: Hartley & Marks.

Brown, E. 2008. 'On December 1952.' *American Music*, Vol 26, No. 1, 1–12.

Cardew, C. 1971. *The Great Learning*, London: Experimental Music Catalogue.

Cline, D. 2017. *The Graph Music of Morton Feldman*. Music since 1900. Cambridge: Cambridge University Press.

Ferneyhough, B. 2014. *Inconjunctions*, for chamber orchestra. Leipzig: Edition Peters.

Gamma, R. et al. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley

Hall, T. and A. F. Blackwell. 2014. 'Sharing digital performance notation with the audience.' In *Proceedings of the 9th Conference on Interdisciplinary Musicology – CIM14*, Staatliches Institut für Musikforschung, Berlin. Available online: <http://www.ludions.com/texts/>

Higgins, H. 2009. *The Grid Book*. Cambridge MA: MIT Press.

Hiller, L. A., and L.M. Isaacson. 1959. 'Experimental Music / Composition with an Electronic Computer'. New York: McGraw Hill.

Hoadley, R. 2017. 'Homenaje a Cervantes: For Violin, Computer And Projections'. In *Proceedings of the International Conference on Technologies for Music Notation and Representation (TENOR)*. A Coruña, Spain. Available online: https://hal.archives-ouvertes.fr/hal-03165874/file/27_Hoadley_tenor2017.pdf

Iverson, K. E. 1980. 'Notation as a tool of thought'. *Communications of the ACM*, Vol 23, Issue 8 (Aug), pp 444–465

Mathews, M. V., and L. Rosler. 1968. 'Graphical Language for the Scores of Computer-Generated Sounds.' *Perspectives of New Music* 6, no. 2: p.92–118.

- Regnierz, P. 1986. *Cornelius Cardew 1936-1981* [Film] England: Arts Council England.
- Available online: <https://player.bfi.org.uk/free/film/watch-cornelius-cardew-1936-1981-1986-online>
- Sauer, T. 2009. *Notations 21*. New York: Mark Batty.
- Schnebel, D. 1969. 'Visible Music' in *Classic Essays on Twentieth-Century Music / A Continuing Symposium*, ed Kostelanetz, R., Darby, J. Santa, M., 1996. New York: Schirmer Books.
- Veer, C. 2019. *The Digital Score / Musicianship, Creativity and Innovation*. New York: Routledge.
- Weibel, P. 2020. 'The Road To The Upic. From Graphic Notation To Graphic User Interface'. In Weibel, Brümmer, Kanach, eds., 2020. *From Xenakis's UPIC to Graphic Notation Today*. Berlin: Hatje Cantz. Available online: <http://www.interfacesnetwork.eu/post.php?pid=47-upic-book>
- Wright, M., and A. Freed. 1997. 'Open SoundControl: A New Protocol for Communicating with Sound Synthesizers'. In *Proceedings of the International Computer Music Conference (ICMC)*, pp 101-104.

Notes

¹ In covering this notational territory we acknowledge the wide pedigree of the work of others in using SuperCollider in the context of music notation as here described, including that of Nick Collins and Fredrik Olofsson (see Sauer, 2009, p.56), Rohan Drape, John Eacott, Thor Magnusson, Josh Parmenter, and Andrea Valle, among others. See also responses at <https://scsynth.org/t/do-you-make-visual-scores-music-notation-using-sc/5219>

² For an example of another type of SuperCollider live graphic notation by one of the authors, see Hall & Blackwell, 2014.

³ The relevant SuperCollider classes for this section can be downloaded from

<https://github.com/ludions/scbook-2nd-edn>

⁴ UTF-8 music font characters can be freely cut and pasted, e.g. from the Internet, however won't display in SuperCollider's IDE unless the font is installed and the editor set to that font; see the **String** help file.

⁵ See <http://www.unicode.org/versions/latest>

⁶ <https://w3c.github.io/smufi>

⁷ <https://www.smufi.org/fonts>

⁸ <https://w3c.github.io/smufi/latest/tables/clefs.html>

⁹ <https://github.com/w3c/smufi>

¹⁰ <https://docs.cycling74.com/max8/refpages/nslider>

¹¹ See the neoscore project for a related approach in the recreation of the score of Feldman's *Projection #2*. Available at <https://github.com/DigiScore/neoscore>

¹² A classic early text discussing this general approach, as well as an example of Feldman's graph scores is Behrman, 1965.

¹³ <https://inscore.grame.fr>

¹⁴ See <https://inscoredoc.grame.fr>

¹⁵ You can experiment with this using the **INScoreListenerMIT** class.

¹⁶ The relevant SuperCollider classes for this section can be downloaded from

<https://github.com/richardhoadley/inscoremit>

¹⁷ <https://guidodoc.grame.fr>. Discussion of INScore's MusicXML engine is beyond the scope of this chapter.

¹⁸ <https://github.com/grame-cncm/inscore/releases>

¹⁹ On macOS, applications that do not have focus can be slow to reflect changes. The `osaActivate` method shown in the current example will bring INScore to the foreground on macOS, which should solve any delays in updating changes via communication with SuperCollider. Note that for communication with secondary computers, INScore must first be opened for the method to work. The `osaActivate` method is ignored by other operating systems.

²⁰ See for example <https://inscoredoc.grame.fr/refs/6-setsect/#vectorial-graphics>

²¹ In INScore the native code for this is **ITL/scene/polygon0 set poly [0.7, 0.05, 0.1, 0.1, 0.1, 0.0]**;

²² Whether audience members are able to read CPWN or not, audience reaction to the public screen display has been positive; feedback has generally indicated an interest in the appearance and behavior of the varieties of notation displayed.

²³ The composition's title and much of the text used is drawn from George Berkeley's *A Treatise Concerning the Principles of Human Knowledge* (Berkeley: 1710), in which the author explores the distinction between thinking and 'unthinking' things —inanimate objects.

²⁴ Texts are obtained algorithmically from Berkeley's *Treatise* and from algorithmically generated haiku, the available vocabulary focusing on elements of Berkeley's philosophical interests.

²⁵ A rendition of this can be seen at: <https://youtu.be/64HImuOHtpc?t=734>, which links to a recording of a performance from 17th April 2021. This performance was held online due to the prevailing international health conditions. All rehearsals and performances took place from the singers' private residences with networked video, audio and data.

²⁶ A rendition of this section can be seen and heard here: <https://youtu.be/64HImuOHtpc?t=319>. In the 1986 English Arts Council documentary about Cornelius Cardew (Regnietz 1986), Eddie Prévoist provides an entertaining account of a time when, in preparation for a concert, members of the improvisation group the AMM were encouraged by Cardew to search an area within of the UK's Peak District which apparently hosted a special type of 'musical' stones. Performers might be encouraged to use similar motivations when locating the materials used for performance here.

²⁷ <https://youtu.be/64HImuOHtpc>

²⁸ <https://youtu.be/64HImuOHtpc?t=374>

²⁹ <https://lilypond.org>

³⁰ <https://abjad.github.io>

³¹ <https://github.com/n-armstrong/fosc>