Code-bending: a new creative coding practice

Ilias Bergstrom[1], R. Beau Lotto[2]

[1]EventLAB, Universitat de Barcelona, Campus de Mundet - Edifici Teatre, Passeig de la Vall d'Hebron 171, 08035 Barcelona, Spain. Email: iliasbergstrom@ub.edu, onar3d@gmail.com

[2]Lottolab Studio, University College London, 11-43 Bath Street, University College London, London, EC1V 9EL. Email: lotto@ucl.ac.uk

Key-words: creative coding, programming as art, new media, parameter mapping.

Abstract

Creative coding, artistic creation through the medium of program instructions, is constantly gaining traction, a steady stream of new resources emerging to support it. The question of how creative coding is carried out however still deserves increased attention: in what ways may the act of program development be rendered conducive to artistic creativity? As one possible answer to this question, we here present and discuss a new creative coding practice, that of *Code-Bending*, alongside examples and considerations regarding its application.

Introduction

With the creative coding movement, artists' have transcended the dependence on using only pre-existing software for creating computer art. In this ever-expanding body of work, the medium is programming itself, where the piece of *Software Art* is not made *using* a program; instead it *is* the program [1]. Its composing material is not paint on paper or collections of pixels, but program instructions.

From being an uncommon practice, creative coding is today increasingly gaining traction. Schools of visual art, music, design and architecture teach courses on creatively harnessing the medium. The number of programming environments designed to make creative coding approachable by practitioners without formal software engineering training has expanded, reflecting creative coding's widened user base. An additional enabling factor is the particular prevalence of the open-source ethos, the majority of programming environments being available as free open-source software. There are also a vast and ever-growing number of libraries extending the functionality of these environments, and example programs or even entire productions available to use, extend and learn from.

In traditional art, artists can choose from numerous *art practices*, by which we mean how an artist goes about doing her work: visual arts examples include sketching, oil painting, found art and live painting. Note the notions of art practice and of technique overlap somewhat: A painter may regard applying oil paint with a palette knife instead of a paintbrush as employing a new technique. A painter in his studio versus a live painter in front of an audience, may on

the other hand employ the same techniques, but engage in different practices. While programming as a medium for art is established, there is still much room for analogous discussion on how creative coding practice may be carried out. In what ways may software development, largely established as meticulous, systematic engineering practice, also be made conducive to artistic creativity? The contribution of this article is to introduce an account of such a creative coding practice, that of *Code-Bending*: inspired from circuit-bending [2], in code-bending the internally used programming interface of open-source software is re-purposed, so that instead of fulfilling its intended purpose of internal communication between outwardly inaccessible components of the software, this programming interface allows external communication from and to elements, previously neither exposed to users, nor to other software.

Background

Procedural art long predates computers, a common example being Islamic Art. In modern times, many artists have followed in the footsteps of pioneers Ben Laposky and John Whitney [3] in creating procedural art using computers, a practice which has come to be termed *Digital Art* [4], or *New Media Art,* referring to art created using *New Media* technology [5]. *Aesthetic Computing*, "the application of the theory and practice of art to the field of computing" [6], widens the scope of aesthetics in computing, emphasizing how artistic aesthetics may inform all computing practice. The account of how computing and art have informed each other is extensively covered in existing literature [7], [8]. Without attempting an exhaustive background review on the topic of the interrelation of art and computing, we will describe the practices that have most directly inspired us, framing the context into which we introduce code-bending. Note that we here focus on practices and not their supporting tools, although several tools will inevitably be mentioned for their relation to particular practices.

Creative coding

Programmers with artistic intent have by McLean and Wiggins [9] been described as frequently following a *bricolage* approach, a notion first introduced in the context of programming by Turkle and Papert [10], who in turn adopted the term from Lévi-Strauss [11]. McLean and Wiggins do not provide their own definition, instead citing Turkle and Papert:

"The bricoleur resembles the painter who stands back between brushstrokes, looks at the canvas, and only after this contemplation, decides what to do next. For planners, mistakes are missteps; for bricoleurs they are the essence of a navigation by mid-course corrections. For planners, a program is an instrument for premeditated control; bricoleurs have goals, but set out to realize them in the spirit of a collaborative venture with the machine. For planners, getting a program to work is like "saying one's piece"; for bricoleurs it is more like a conversation than a monologue."

A parallel development to the emergence of computing and its application in art has been the process of making programming languages easier to use. Some initiatives have explicitly ventured beyond lowering the learning threshold, towards encouraging a different approach to coding which, although not explicitly stated, bears much resemblance to the bricolage approach. The *Sketching* approach is first encountered in writing by Miller Puckette, originator

of the prominent visual programming languages (VPL) for creative coding, Max/MSP and Pure Data [12]. Puckette states he emphasized the sketching analogy, by presenting users with a blank canvas on to which the program is incrementally drawn up as a directed graph, through placing interconnected boxes upon it, each performing a particular function based on the data it receives through its *inlets*, before it sends the result on through its *outlets*, see Figure 1. Sketching is also prominently promoted in the Processing language and environment [13]. Processing expands upon the ideas in John Maeda's *Design by Numbers* language, created for teaching the *"idea of computation to designers and artists"* [14]. In all mentioned environments, the goal in their design is facilitating a creative approach to programming, analogous to traditional media artists sketching out their work, be it a drawing, a sculpture, or musical score.

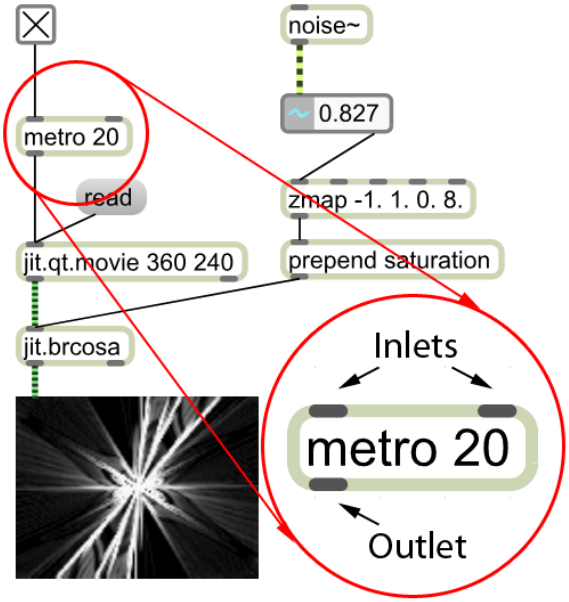

Figure 1: Max/MSP patch, showing data-processing and data-generating boxes connected to each other through their respective inlets and outlets, thus forming a visually drawn computer program (figure created by authors).

Taken to creative coding's extreme, in the practice of live-coding, artists write code as a means of performance, commonly while presenting the output in conjunction with a projection of the continuously modified program code [15], [16]. Predominantly live-coding is used in musical performance, but visuals performances are not uncommon, and there is nothing to keep the practice from being applied to any other context in which live performance and creative coding meet. Live-coding requires the use of specialized programming environments capable of interpreting the code on the fly as it is entered by the performer, without restarting or recompiling the whole program. While several environments exist with this capability (for example Max/MSP and Pure Data), some have also been created specifically with live-coding in mind (for example SuperCollider, ChucK, Impromptu and Fluxus).

Of relevance is also *hacking*. While the term may vary greatly in definition depending on context see [17], here it is used to refer to the modifying of a pre-existing piece of closed-

source software, with the intention of causing it to perform differently from what its original designers intended. This requires a very deep understanding of the working of computers and the software executed on these. Commonly it requires machine-level assembly-language programming, the only means of controlled modification of software which was not intended to be altered by end-users. Often hacking is therefore illegal, as it violates the software's end-user license agreements. There is a dearth of academic discussion on hacking as a creative art-form, regardless of the definition employed. An example where hacking as art appears in public discourse however, is the 2011 Netherlands Media Art Institute exhibition "the art of hacking" [18].

Software engineering practice

Between the two extremes of creative-coding practice on the one hand, and the development processes taught at university software engineering courses on the other, a tradeoff may often be necessary. The focus of the latter is on ensuring a predictable, rigorous, transparent structure throughout development, towards eliminating mistakes, and delivering well-functioning software, within time and on budget. Several methodologies exist, e.g. the waterfall model, spiral model, agile development, etc. [19], each detailing stages of formulating requirements, designing, implementing, testing and maintaining a software system. While the goals of these processes are desirable irrespective of artistic intent, the rigor they require easily impedes the explorative bricolage process most conducive to creativity. Consequently, even practitioners trained in these methods may not always follow them, or may apply them selectively, perhaps only after the creative content has materialized. Rapid Application Development (RAD) methods on the other hand are more akin to creative coding, their distinction sometimes even amounting only to difference in the efforts' stated intent.

The notion of design patterns is also particularly applicable to creative-coding. Initially it was conceived of within the context of architecture, and then translated to the context of software engineering and introduced to a wider readership with the seminal "Gang of Four" book [20]. The beautifully simple concept is that of cataloging succinct, abstract solutions to commonly occurring design problems, serving as a resource from which to derive solutions, through applying and combining these design patterns.

Code-Bending

The spiritual precursor to code-bending is that of *Circuit-Bending*, a term coined by Reed Ghazala, although he explicitly makes no claim on being the first to carry out the practice [21]. Circuit-bending encourages a tacit, explorative experimentation through modifying an existing electronic circuit, rather than the strict approach of traditional electronic engineering. It also requires little understanding of how the circuitry at hand works: although such knowledge is undeniably beneficial, extensive training is not an absolute barrier to entry. Reed Ghazala refers to his technique as anti-theoretical; not in the sense of rejecting theoretically informed practice, but as providing a complementary alternative to it. A popular circuit-bending practice is to modify electronic toys, for their subsequent use as musical instruments. Most famously, Mattel's Speak & Spell voice synthesis toy is often bent and repurposed as a musical instrument, generating otherworldly vocal sounds.

Analogously, in code-bending, the internally used programming interface of open-source software is re-purposed, so that instead of fulfilling its intended purpose of internal communication between outwardly inaccessible components of the software, it allows external communication from and to elements previously neither exposed to users nor to other software. We now expand description.

In programming, the term *Interface* refers to a program component's specified set of access points, over which communication with other software components is made possible. A whole program is built through defining numerous components, each with its internal functionality concealed, allowing its manipulation by other components only over this interface. While in many applications an *Application Programming Interface* (API) is deliberately formulated for third parties to use towards extending the functionality of the program, there is always a further, extensive set of interfaces, never intended to be outwardly accessible. To explain through an analogy, a home-stereo system consists of separate components (radio, amplifier, cd, etc.): each having connectors on the back, which can be likened to the components' API. A working stereo system results from connecting these together. But by lifting a component's lid, one finds that it too internally consists of sub-components, with their own additional internal interface. It is in circuit-bending practice these internal connections that are used to change the functionality of the appliance. Going back to software, in closed-source programs these internal interfaces remain concealed. But with open-source software, also these can be identified and used even if undocumented, as the code is freely available, and thus one can figuratively "lift the lid" and unrestrictedly access also these interfaces: It is these that are harnessed in code-bending practice.

Existing software can thus, in a comparatively rapid, playful manner, be repurposed, encouraging an explorative approach to implementation. Connections may be creatively re-arranged (re-mapped), either until the desired final shape is reached, or as a form of live performance, in which case the emphasis lies not on the final product, but on continuous transformation.

The term *Mapping* refers in this context to the linking of variables across two or more parameter spaces. There is much active research on mapping, primarily relating to the development of *Digital Musical Instruments* (DMI), where the control devices used by performers are separated from the sound generators producing the actual output, and defining mappings is needed for their communication [22]. This contrasts to traditional musical instruments, where physical connections between controller and sound generator are unalterable. Several mapping approaches have been discussed: one-to-one, one-to-many, many-to-one, multi-level mapping, as well as abstract mathematical or stochastic models. By *Mutable Mapping* [23], we refer to the conduct where mappings need not remain fixed during performance, but are instead gradually altered created and destroyed as a form of expression in and of itself. For this use we created the application *Mediator*, purpose-made to be a general tool for mutable mapping performance. An earlier instance of live improvisation of mappings as performance is described by Collins and Olofsson, for their klipp-av performances [24], where mappings between elements in cut-up sound and video were improvised through live-coding. Mapping concepts can apply to all new media art, including modern dance, interactive installations and live audio-visual performance. For example, parameters derived

from musical notation, pre-recorded or generated live by musicians, can in live audio-visual performance be mapped to control aspects of live procedural animation (e.g. [25]), while gestures of dancers are mapped to control their musical and/or visual accompaniment [26].

For mapping between exposed interface points in code-bent programs, the *Open Sound Control* (OSC) digital communication protocol [28] is ideal. Its advantage is that messages follow a straight-forward schema with meta-data that describe the messages' content, so the user can inspect them and decide on their interpretation. OSC provides a common, standardized message passing format between software, straightforwardly achieving interoperability between an arbitrary number of disparate sources and destinations.

Concretely, the practice of code-bending is necessarily conducted in two phases: first exposing the previously inaccessible contact points in the involved open-source programs, and subsequently executing these programs, so that while they are running, one can experiment with altering mappings between now exposed contact points, and eventual additional sources of data; either in search of an ideal mapping to subsequently finalize, or continuously, as a form of performance.

The exposing of contact points has been conceived of and carried out in three different manners, although these may not be the only approaches possible:

A. Re-adapting a creative code programming language, so that programs written in the language no longer are stand-alone applications, but instead behave as plug-ins, effectively repurposing the language as an API, allowing the flexible dynamic combining of the output from programs written in this language, within a program written to host these as plug-in modules.
B. Inserting functionality for sending and receiving OSC messages in the methods where the Graphical User Interface (GUI) communicates with the main application code. In this manner the GUI can be bypassed and the main application instead be interfaced with using OSC messages. Such functionality may also be inserted at arbitrary locations in the code, but will then require far greater care to avoid breaking program stability, se explication below.
C. Inserting functionality for sending and receiving messages to and from a visual programming language, in a manner analogous to approach B, but with messages conforming to the VPL's API instead of the OSC protocol. The application may then be hosted in a visual programming language such as Max/MSP, allowing for the newly exposed contact points to be used as inlets and outlets, patchable with other boxes on the VPL canvas, while the bent program still remains largely unmodified.

The reason contact points should always if possible be inserted where GUI code interfaces with the main application code is to avoid advanced technical obstacles caused by program multi-threading (e.g. racing conditions, deadlocks). At the layer where the GUI interfaces with the main application however, such issues should have already been protected against, since GUI and main program code virtually always reside in separate threads. Since code-bending is meant to be a playful, tacit practice, such advanced problems are better avoided than tackled head-on. Another obstacle to avoid is creating feedback loops. While desirable in analog

circuits, in software they immediately freeze execution, and should either be avoided, or better still, programmatically detected and interrupted.

Mappings to and from exposed contact points may subsequently be defined in any programming language capable of sending OSC messages in cases A and B, or through patching to and from the exposed programs' inlets and outlets, in case C. Alternatively, software for defining mappings between OSC parameter spaces may be used, such as the Mediator software. Another viable approach we have experimented with is modifying mappings through live-coding, as previously practiced by Collins and Olofsson [27].

Existing creative coding practice inspired by circuit-bending

Previous practitioners have like us taken inspiration from circuit-bending, seeking an equivalent practice in creative coding, each apparently independently formulating their own interpretation of the concept. Chris Novello [29] is first to also describe what he means with term, and present concrete work following his conception of it. In his description, code-bending depends on using software already able to *"send messages when events happen in the program"* and *"receive external messages that control the activities of the program"*. By then re-routing these messages, he incorporates this software into larger systems, in a manner akin to the notion of mapping: manipulating connections between already exposed, purposely created contact points in existing software. Alberto de Campo and Julian Rohrhuber have released an extension library to the Supercollider language that they've christened 'Bending' [30]. Paraphrasing from personal communication with Julian Rohrhuber:

"While not elsewhere documented, the Quark stands as its own theoretical documentation, in the sense that also code is a form of theory. Similarly to circuit bending, it serves as a way to go back behind parameter controls and graphical user interfaces to a far more generic and abstract mode of interaction. It allows you to modify a graph of unit generators without explicitly adding anything to the code.

It is also intended as a complement to live coding, which could be performed in turns or in parallel with it. As I prefer programs to be as abstract as possible, rather than programs as specification for user interfaces, as is often the case for musical real-time interaction, code bending is a way to directly interact on the level of the graph rather than on the level of a preconceived interface.

On a conceptual level, speaking in Bruno Latour's terms, it is a combination of white boxing (a mode where all lies open and the functionality is to be negotiated) and black boxing (a mode where all is implicit and presumed to work in the dark): we operate on the level of internal functionality and yet what we modify is specified abstractly".

The ideas we present here, while sharing commonalities with the above descriptions, also contribute to completing the conception of what code-bending practice may entail. Further from employing the notion of mapping, we detail how existing software which lacks the desired exposed contact points can be incorporated in code-bending. This is what allows for "lifting the lid" also on software, an idea central to the spirit of circuit-bending.

Examples of Code-Bending in practice

The here introduced ideas have been gradually refined through a number of years, during the development of software for live audio-visual performance, and creation of interactive installations. Following are examples from our own practice and that of others, to illustrate how code-bending ideas may be applied.

The common Processing tool is by procedural graphics artists used to quickly sketch interactive graphics algorithms. The combining of multiple such sketches into a complex program from within Processing however is challenging, discouraging explorative creative experimentation, and limiting the achievable output complexity. This issue is common across live visuals performance software: adding your own live visuals algorithms to the repertoire of any of the available applications requires significant software engineering effort and expertise. Addressing this obstacle, the program *Mother* (Figure 2) treats stand-alone Processing sketches as plug-ins, requiring only few straightforward alterations. To enable this functionality, the existing open-source Processing programming language was thus bent (approach A), repurposing it into behaving as a plug-in development API for the host application, accessible to non-expert programmers. Following this initiative, artists can now dynamically layer the output of modules, gradually adding, removing and altering these, both during experimentation, and to create a narrative during a live performance. Mother is released as open-source software, attracting several artists to employ it in their practice, and code contributions to further expand its functionality. For a more detailed discussion refer to Bergstrom & Lotto [31] and [32]. The source code is available at [33].

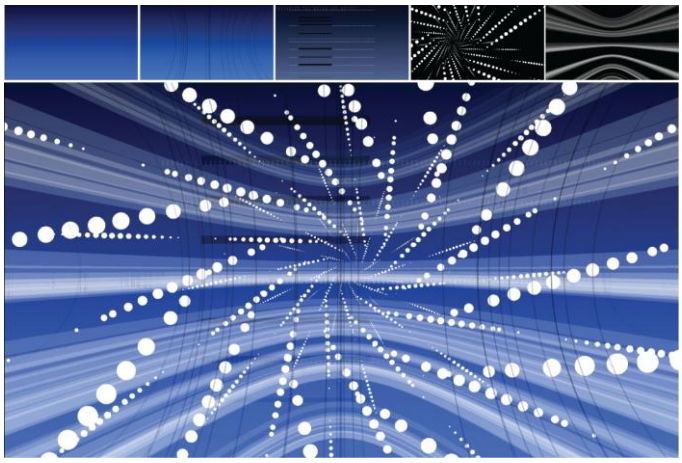

Figure 2: Illustration of how several processing sketches (each a small image at the top) are within Mother layered to produce a single complex output (all images and programs involved created by I. Bergstrom).

In the development of the Mother software, code-bending is a characteristic central to the functionality provided by the final released software. However, code-bending is useful also when exploratorily prototyping a design to be subsequently completed following different methods. We employed this approach when developing interactive installations: The “Hearing Colour” series, developed in collaboration with Sam Walker and Erwan Le Martelot, and exhibited at the London Passing Through exhibition (Figure 3) and at the Wellcome trust (2009); and the “Music from colour” project, which in several incarnations has been exhibited

at the London Science Museum's Lottolab, and used for holding "Synaesthetic Workshops" in the UK. All built on the premise of re-mapping information between sensory modalities, the output being musical and / or visual. During the design phase for all installations, existing open-source programs for interfacing with cameras and processing the visual input, and programs for real-time music synthesis were bent following approaches B & C, so they could quickly be incorporated into a system, thus facilitating experimentation towards concretizing the design of the installation at hand. The incarnation illustrated in Figure 3 comprises of two computers running software continuously deriving image information from two video feeds, re-mapping this information to musical (MIDI) instructions controlling a realistic sample-based classical orchestra. The video feeds are provided by two cameras in front of which gallery visitors can move freely, their appearance influencing the musical output. If no visitor is present the cameras see one canvas each, of which one is showing a continuously flowing generative graphics projection (left), and another onto which gallery visitors can collaboratively paint (right).

Figure 3: "Hearing Colour" installation at Passing Through exhibition, James Taylor Gallery, London, 2009. The work was created by Ilias Bergstrom (concept and video-to-music software development), Sam Walker (concept and hardware implementation), Erwan Le Martelot (Fugitive Moments generative computer graphics software), and Beau Lotto (concept).

In his practice, Chris Novello makes use of computer game software already providing contact points exposed for receiving control messages that alter the games' functionality. His work is an illustrative example of what applying code-bending ideas can manifest as, beyond the authors' own practice. For generating control data, and altering the mappings between the control data and the game patch-points, he has created a hardware controller with patch points resembling a modular analog synthesizer (Figure 4). While his initial work concentrated on computer games, his later work has expanded to also including music synthesis modules, as well as a text editor application.

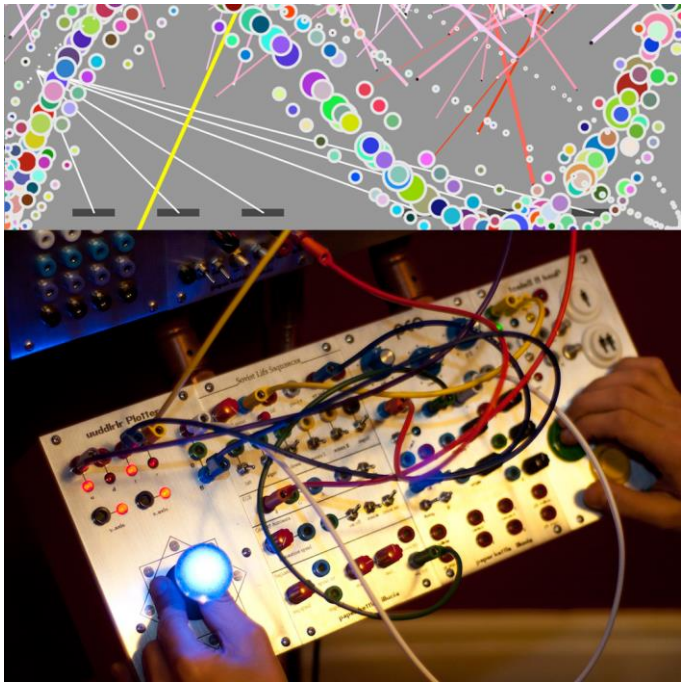

Figure 4: Chris Novello's Illucia "modular codebending instrument" (bottom), and "War Machine", one of his games created to be controlled and bent using Illucia (top). © Chris Novello.

Discussion

Many practices from the arts find their counterparts in creative coding. Sketching of code allows for a direct analogy to sketching in drawing: one starts working without having a clear goal, or any goal at all, and through exploring various ideas, a creative outcome takes shape. Improvisation, either as another means for sketching, or as live performance in front of an audience, is facilitated through live-coding as well as through mutable mapping, constructing a continuous narrative output through manipulating program instructions and interconnections. Just as some modern art movements have ceased catering to the establishment, instead seeking to subvert it [34], a counterpart in computational art is found with hacking. Extending the above narrative, code-bending facilitates a wide range of collage strategies, analogous both to collage of images / sound, and to artwork incorporating found objects. Individual artists thus have a spectrum of approaches to pick from, as best suits their idiosyncrasy and the current project at hand.

While currently the code-bending ideas described here require bending open-source software to expose patch-points for explorative mapping, a recent development has appeared which given time may complement this practice. A small number of applications are developed with patch points already exposed at varying levels of granularity, through their implementation of the OSC communication protocol, employed such that the program's user interface may be bypassed, and the application remote-controlled. Uniquely, the music composition and performance application Ableton Live, has in collaboration with Cycling74, the makers of Max/MSP, facilitated the embedding of Max programs within Ableton Live, thus exposing a large number of patch points to remote control, provided the user implements OSC or another form of communication.

However, there will always be applications which have not been implemented with such functionality in mind, as well as software including only some such functionality, where more is desired. The relevance of the code-bending approach is therefore only likely to increase as applications with already exposed patch points become more common.

We envision that as a future continuation of the efforts presented here, more creative coding practices appear and are divulged publicly so that they can be incorporated into the future practice of software artists. It is our hope that through the present discussion we will inspire software artists to create interesting new work, following interesting new approaches!

Acknowledgements

We are grateful to Patti Brennan, Elias Giannopoulos and the reviewers of this article, for their invaluable comments on earlier version of this draft. This work has been supported by an EU Presenccia grant, and Spanish INNPACTO Melomics project reference code IPT-300000-2010-010.

References

1. John Maeda, *Creative code* (London: Thames & Hudson 2004).
2. Reed Ghazala, "The Folk Music of Chance Electronics: Circuit-Bending the Modern Coconut," *Leonardo Music Journal*, pp. 97–104, 2004.
3. *Digital Art Museum (DAM),* Retrieved 11 October 2012 from <www.dam.org>.
4. Christiane Paul, *Digital art* (London: Thames & Hudson, 2008).
5. Mark Tribe, Reena Jana, and Uta Grosenick, *New media art* (New York: Taschen, 2006).
6. Paul A. Fishwick, Ed., *Aesthetic Computing* (Cambridge, MA: MIT Press 2008).
7. Steve Dixon, *Digital performance: a history of new media in theater, dance, performance art, and installation* (Cambridge, MA: MIT Press, 2007).
8. Bruce Wands, *Art of the Digital Age* (London: Thames & Hudson, 2007).
9. Alex McLean, Geraint Wiggins, "Computer Programming in the Creative Arts", *Computers and Creativity* (Berlin – Heidelberg: Springer Verlag, 2013).
10. Sherry Turkle, Seymour Papert, "Epistemological pluralism: Styles and voices within the computer culture", *Signs*, pp. 128/157, 1990.
11. Claude Lévi-Strauss, *The savage mind,* (Chicago: University of Chicago Press, 1968).
12. Miller Puckette, "Pure Data: another integrated computer music environment," *Proceedings of the Second Intercollege Computer Music Concerts*, pp. 37–41, Tachikawa, Japan, 1996.
13. Chris Reas and Ben Fry, *Processing: A Programming Handbook for Visual Designers and Artists* (Cambridge, MA: MIT Press, 2007).
14. John Maeda, *Design by numbers* (Cambridge, MA: MIT Press, 2001).
15. Nick Collins, "Live Coding of Consequence," *Leonardo*, 44, No. 3, pp. 207–211, 2011.
16. Nick Collins, Alex McLean, Julian Rohrhuber and Adrian Ward, "Live Coding in Laptop Performance," *Organised Sound*, 8, no. 03, pp. 321–330, 2003.
17. Joe Erickson, *Hacking: The art of exploitation* (San Francisco: No Starch Press, 2008).
18. *The Art of Hacking Exhibition*, Retrieved 11 October 2012 from <www.nimk.nl/eng/the-art-of-hacking-exhibition>.

19. Ian Sommerville, *Software Engineering*, 9th ed. (Boston, MA: Addison Wesley, 2010).
20. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: elements of reusable object-oriented software*, 1st ed. (Boston, MA: Addison Wesley, 1994).
21. See Ghazala [2].
22. Eduardo R. Miranda and Marcelo M. Wanderley, *New Digital Musical Instruments: Control and Interaction Beyond the Keyboard* (Middleton, WI: AR Editions, 2006).
23. Ilias Bergstrom, Anthony Steed, and Beau Lotto, "Mutable Mapping: gradual re-routing of OSC control data as a form of artistic performance," in *Proceedings of the international conference on Advances in computer entertainment technology*, pp. 290–293, Athens, Greece, 2009.
24. Nick Collins and Fredrik Olofsson, "klipp av: Live algorithmic splicing and audiovisual event capture," *Computer Music Journal*, 30, no. 2, pp. 8–18, 2006.
25. Ilias Bergstrom and Beau Lotto, "Harnessing the Enactive Knowledge of Musicians to Allow the Real-Time Performance of Correlated Music and Computer Graphics," *Leonardo*, 42, no. 1, pp. 92–93, 2009.
26. See Dixon [7].
27. See Colling and Olofsson [24].
28. Matthew Wright, "OpenSound Control: A New Protocol for Communicating with Sound Synthesizers", *Proceedings of 1997 International Computer Music Conference, ICMA*, pp. 101–104, 1997.
29. *Codebending FAQ*, Retrieved 17 August 2012 from <www.paperkettle.com/codebending/>.
30. *SourceForge.net Repository - [quarks]*, Retrieved 17 August 2012 from <http://quarks.svn.sourceforge.net/viewvc/quarks/DIRECTORY/Bending.quark?view=markup&pathrev=2132>.
31. See Bergstrom & Lotto [25]
32. Ilias Bergstrom and Beau Lotto, "Mother: Making the Performance of Real-Time Computer Graphics Accessible to Non-programmers," in *(re)Actor3: The Third International Conference on Digital Live Art Proceedings*, pp. 11–12, Liverpool, UK, 2008.
33. *Processing-Mother - A program that allows VJing with Processing.org sketches*, Retrieved 11 October 2012 from <http://code.google.com/p/processing-mother/>.
34. Ernst H. Gombrich, *The story of art* (London: Phaidon, 1995).