IST-Africa 2011 Conference Proceedings
Paul Cunningham and Miriam Cunningham (Eds)
IIMC International Information Management Corporation, 2011
ISBN: 978-1-905824-24-3

tactusLogic: Programming Using Physical Objects

Andrew Cyrus SMITH[1], Heinrich SPRINGHORN[2], Steven Bruce MULLIGAN[2], Ireyan WEBER[2], Jackie NORRIS[2]
[1]*CSIR Meraka Institute B43, PO Box 395, Pretoria, 0001, South Africa*
[1]*School of Computing, University of South Africa*
Tel: +27 12 841 4626, Fax: + 27 12 841 4720, Email: acsmith at csir.co.za
[2]*Department of Computer Science, University of Pretoria, Private bag X20, Pretoria, 0028, South Africa*
Tel: +27 72 368 1167, Email: zenoxhein at gmail.com

Abstract: We describe a new programming language that is based on physical elements and especially developed to test the hypothesis that a physical computer programming language is possible. This imperative language is deliberately limited to a few operators and constructs to simplify the language for the novice programmer. No computer knowledge is required once the programming system has been configured for use. Our evaluations reveal that the concept of variables, and redirecting program execution, are considered the most obscure amongst novice programmers.
Keywords: Input surface, novice programmer, tangible programming, physical interface, fiducial.

1. Introduction

Computer programming is currently an elitist activity, discriminating between those who have the funds and opportunity to become familiar with a personal computer, and those who do not. The aim of this research is to ‘push’ the computing device into the background and instead investigate softer programming paradigms. Should a programming environment exist where the traditional keyboard and mouse are no longer a prerequisite for the successful execution of the programming task, then we would have achieved our goal. But what type of interface could possibly replace the keyboard and mouse? We conjecture that using physical objects could potentially provide an alternative.

Therefore, our objective with the present study is to determine what a physical programming environment could consist of, if at all possible. Should it prove to be possible, we anticipate that it would have numerous benefits, specifically to three groups of people.

The first of these groups are those people in developing regions (such as South Africa) who do not have ready access to a personal computer (PC) due to a lack of financial means. The second group of possible beneficiaries of this work are those people who feel intimidated by the PC. It is conjectured that the intimidation might be a result of the seemingly complicated interface, consisting of a myriad of buttons (the keyboard) and onscreen icons. The third group of beneficiaries are those people who prefer to work in a group-setting as opposed to working in isolation.

The current research addresses the first need by providing a potentially affordable computing device that may simultaneously be used by multiple people. This is because a number of users can potentially share a single PC using the system described.

The second group benefits because the user interface is simplified, with only a limited set of operations exposed to the user. Potentially damaging (to the computer) instructions are not possible because of the added layer provided by the system.

The third group could find this system beneficial because it supports group collaboration due to the large physical interface objects.

In this paper we report on the design of a tangible programming environment which does not require the user to read and write, and neither does the user need to operate a computer. This paper's contribution lies in the description of the language and the design of the physical representation of the language.

The paper is structured as follows. We first discuss related work done by others, followed by the methodology used in this research. A description of our system and its implementation is then given. The usability tests and the test results follow this. Sections on proposed future work and our conclusions finish off this paper.

2. Related Work

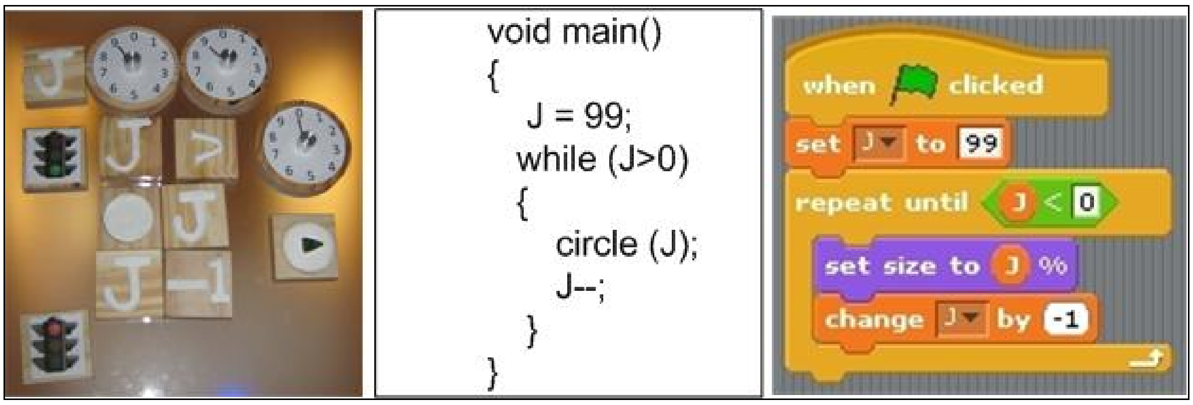
Attempts have previously been made to make computer programming attractive to certain groups. One such group is the female school-going population of developed regions. This group is targeted in these areas because computer programming seems to attract mostly the male population for reasons which won't be covered in this paper. Alice 2 [1] is an example of an experimental programming environment specially developed to test the idea of a language targeted at a specific group.

Another group targeted is the pre-letterate (not able to read or write) [2]; this group typically consists of pre-school aged children. Programming environments aimed at this group consists of icons, that are either manipulated on a computer screen using a keyboard or mouse, or in the physical world by manipulating real-world objects [3].

Scratch [4] is a programming environment that lowers the entry level significantly for older children who are comfortable with the manipulation of the mouse and can read and write (Figure 1).

Tern [5] is targeted at the class room where children that are able to read and write can assemble a program by using a selection of physical two-dimensional objects that resemble jigsaw pieces. Tern offers the potential benefit of using a single PC to control multiple instances of this physical programming environment, resulting in a cost saving.

The work reported on is this paper can be distinguished from prior work based on the sensing mechanism used to differentiate one object from another, that is, our use of optical markers at the bottom of the physical programming objects.



```
void main()
{
   J = 99;
  while (J>0)
  {
     circle (J);
     J--;
  }
}
```

```
when [flag] clicked
set J to 99
repeat until J < 0
   set size to J %
   change J by -1
```

Figure 1: Sample Scripts Using tactusLogic, Text, and Scratch

3. Methodology

The research was conducted over a period of 7 months during 2010. After the concept had been finalised, a programming language had to be devised that could exploit the attributes of physical objects. In addition, the language had to be simple so that it could be implemented with cognisance of the restrictions imposed by physical objects.

3.1 Research Methodology

A desktop study, using both academic- and public search engines, was conducted to identify related research. The study specifically targeted the use of physical objects as interfaces to a computing device, and programming languages targeting the computer novice. From this we identified a gap in the collection of approaches currently used to implement tangible programming environments. Our design, and implementation, exploits this gap.

3.2 Software Testing Methodology

The system design incorporated use-cases and narratives to capture the requirements.
Three software iterations were tested using different methods. The test plan for the first software iteration included design verification, Unit testing and refactoring of the code. Iteration 2 testing addressed integration and system testing. Iteration 3 utilised batch testing. The agile software development methodology was used for software generation.

We next describe the methodology applied in the usability tests.

3.3 Usability testing methodology

Development was followed by usability testing, using questionnaires, the latter supported by the participants' informed consent. Usability tests were conducted using a combination of a structured questionnaire with provision made for user comments.
Tests were conducted in a comfortable environment with a research team member present. Recording methods employed during the tests were: a) software to capture screen displays, and b) written notes taken by the researcher present. Having given written consent, the user had to perform seven short tasks that covered various aspects of the programming language. These tasks also served as a short introduction to computer programming.

Once the participant had completed the seven tasks, the participant completed a questionnaire related to these tasks. The average duration of a test was 45 minutes.

4. System Overview

A user of the tactusLogic programming system defines lines of computer executable code by placing various codeBlock objects onto an Input Surface (Figure 2). Three output interfaces are provided in this system. Two are in the form of computer displays, and the third are the speakers attached to the system. The primary display is used to inform and assist the user, as well as to provide the result of the executed physical program. Assistance to the user includes error checking and a help function.

Error checking is done synchronously with the construction of the physical program. As objects are placed and manipulated on the Input Surface, the underlying compiler interprets the images and provides feedback to the user through the primary display.

Assistance is available through the help function which provides instructions on the use of the system.

An optional second display is available for research purposes. On this display additional results are made available, such as Processing code.

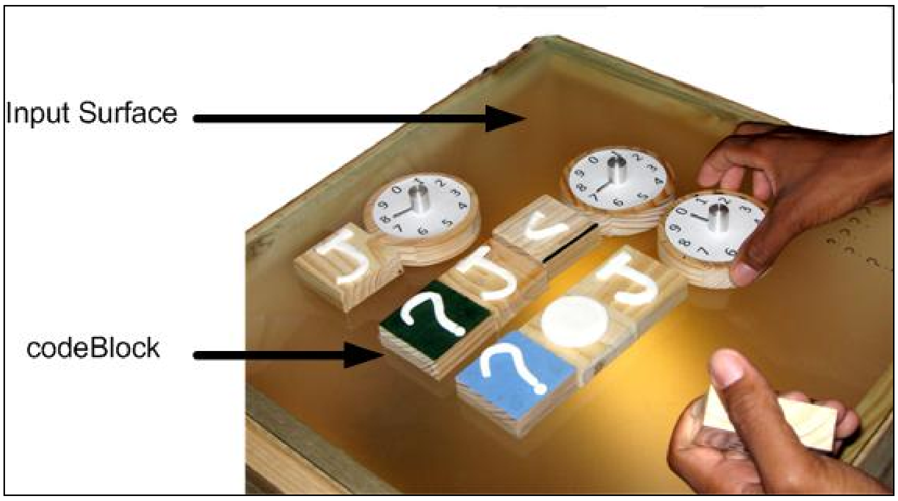


Figure 2: The tactusLogic system in use. codeBlock objects are placed on a translucent surface, called the Input Surface.

codeBlock objects can be freely placed on the surface, noting that the horizontal positioning of objects are irrelevant to how the system interprets them. Only the vertical alignment is important and is later discussed in more detail.

Objects on the Input Surface are interpreted in a top-left to bottom-right fashion.

Two very distinct marking systems are used to convey intent and representation in this system. One marking system is intended for interpretation by the user, and the other for the underlying computing system.

To the user, both colours and semiotics [6] are used to indicate the function of a block.

A different marking system is used to indicate the function of the blocks to the underlying information system. Fiducial markers and recognition software are used for this purpose.

4.1 User Actions

Three major user actions prompt a response from the system: a) the addition of an object on the Input Surface, b) the removal of an object from the Input Surface, and c) the placement of a ‘Play’ object.

When the ‘Play’ codeBlock is detected on the Input Surface, the visual data is compiled to Java code and in turn compiled to Processing code. The Processing code is a simplified version of the Java code which allows later modification by an experienced coder. Java output is required to provide a usable code source from which the Processing code can be derived.

4.2 Help Functionality

A ‘Help’ function is available in addition to the executable functions. This is implemented by monitoring a reserved corner in the bottom right of the Input Surface. Request for assistance is identified when a specific block is placed in this area. At this point an associated video clip is routed to the user’s output monitor.

4.3 - Tolerance for User Inaccuracies

tactusLogic exploits some of the advantages of using physical objects on an unstructured surface: The continuous analogue nature of the Input Surface does not place severe restrictions on the user when placing a number of codeBlock objects on the Input Surface. In contrast, textual programming environments require that the user conform to the virtual lines on the computer screen when entering statements. Rather, the user only has to ensure that a new codeBlock be placed roughly in line with the other objects already on the Input Surface. A deviation of ~30% from the mean is permissible (Figure 3). Horizontal spacing between blocks is however ignored.

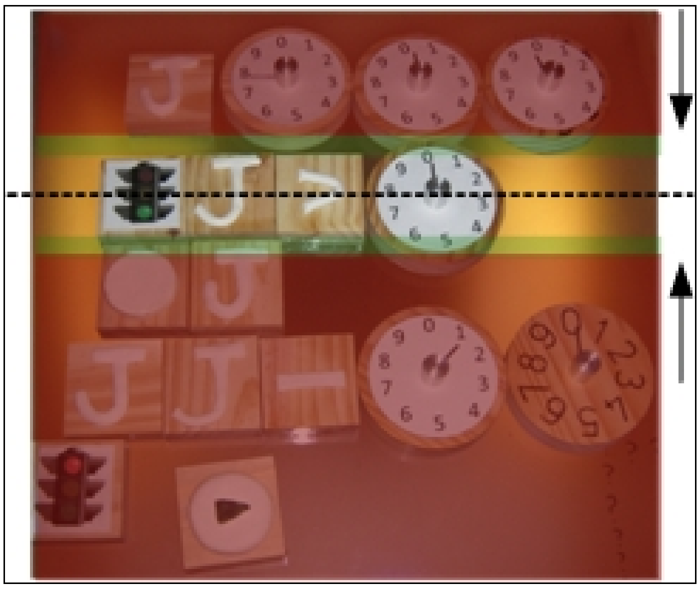


Figure 3: Alignment Requirements are Relaxed

5. The Implementation

The programming system is implemented using optical recognition technology and a custom-developed compiler. The compiler interprets information from the optical system and generates Java code. The Java code is then interpreted to produce Processing code, to be later used for research purposes which won't be discussed further in this paper.

The fiducial markers [7] are part of an optical subsystem used to uniquely identify a class of codeBlock object in the tactusLogic system. Using the optical markers, the visual subsystem can determine both the position and orientation of codeBlock objects placed on the Input Surface.

In this section be describe the system from two perspectives: a) a components point of view, and b) as a collection of subsystems. These two perspectives are covered separately in the following sections.

The system's physical, and logical (language) components are described next.

5.1 Physical Components

Input Surface
The Input Surface in the area on which the user constructs the program logic, using a combination of various codeBlock objects.

codeBlock Objects
A codeBlock is a physical object that represents an element of a programming language, specifically the variables, operators and flow controls.

A selection of codeBlock objects has been developed to introduce a novice to the basic concepts of computer programming. The selection includes a) predefined variable names, b) a method for assigning values to variables, c) conditional statements, and d) the ability to generate sounds and simplistic on-screen graphics.

Wooden blocks of 50x50x5mm for the basis of the codeBlock objects. Clay is shaped and added to the top of the wooden blocks and reflects the block's function to the user. A fiducial is glued to the bottom of the wooden block to identify it to the optical subsystem. Figure 4 is an example of a fiducial used in this system.

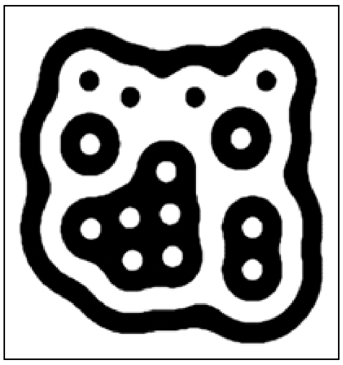


Figure 4: An Example of a Fiducial

When placing the wooden block on the Input Surface, the user orientates the block in such a way so that the fiducial is visible to the web camera from below the surface. A light illuminates the fiducial from below, assisting the optical system in differentiating the fiducial from other objects above the Input Surface.

A set of operations have been implemented as codeBlock objects (Table 1).

Table 1: Entities Represented by Means of codeBlock Objects

Start of, and end of, an execution loop.	Draw a circle of a given diameter.
Decision statements: `if then else end-if`	Play a specified audio clip.
Equality operations: ≤ ≥ = < >	Four basic mathematical operations: + − x ÷
Positive integer values in the range 0 – 999.	Increment, and decrement, unary operations.
A selection of 10 variables.	Jump to a position in the code, plus labels.
'Print' to display a variable on the screen.	'Play' to initiate code execution.

5.2 - The Programming Language

A new and simplified programming language was developed for use in tactusLogic, called the Physical Programming Language (PPL). PPL was designed to accommodate the limitations posed by the use of physical objects on a two-dimensional input surface. One of the limitations of a physical programming language is the representation of numerical values when constructing the program logic. As a means of limiting the quantity of physical objects to represent large numbers, a rotary device was designed and implemented. This device can represent 10 discreet values, ranging from 0 to 9. Placing multiple rotary devices sequentially allows multiple digit numbers to be easily represented.

PPL consists of basic operations that can be performed with the aid of variables and constants.

Software loops are represented by 'traffic lights' (Figure 5). A green light indicates the start of a loop sequence, and a red light the end of the sequence. A conditional expression is placed immediately to the right of the green light. The loop is continuously executed as long as the conditional expression remains true.

Test statements (also known as “if” statements) are implemented in the form of a question mark [?] placed on a coloured background (Figure 6). Green- and red backgrounds indicate the start and end of a test statement respectively. A default execution path (also known as a “then” statement) is indicated by a blue background, and the alternative execution path (also known as an “else” statement) is indicated by a yellow background.

On-screen output is limited to numerical values and circles. The radius of the circle can be specified and so can the pre-recorded sound sample that can be produced as a third output medium.

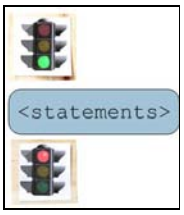


Figure 5: Loop Indicators

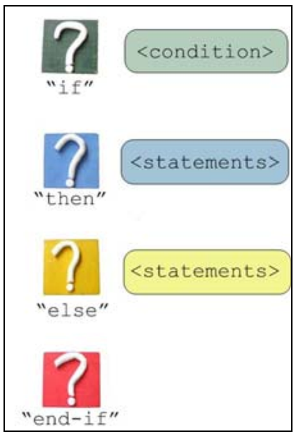


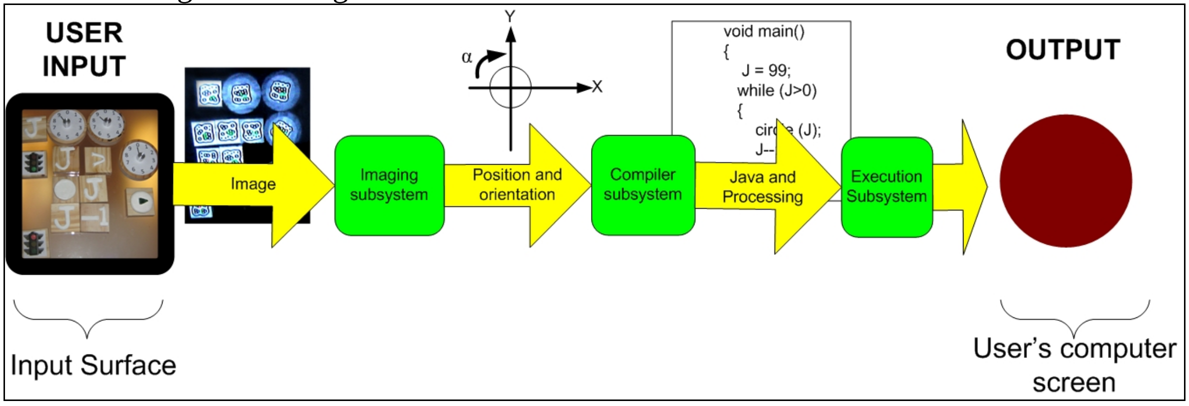
Figure 6: Conditional Execution

The system, as a collection of subsystems, is described next.

5.3 Subsystems

The system consists of three subsystems: the imaging subsystem, the compiler subsystem, and the execution subsystem.

What follows is a description of the subsystems and the data that flows between these. Figure 7 contains a code example which is used to support the description. The example contains the logic for drawing consecutive circles, each of diminishing radii. Data flows from left to right in this figure.



```
void main()
{
  J = 99;
  while (J>0)
  {
    circle (J);
    J--
```

Figure 7: System Overview Showing the Data Exchanged Between Subsystems

Imaging subsystem: The Imaging subsystem comprises of a low-cost web camera, a glass surface, a light, numerous fiducials attached to physical objects, and the reacTIVision software framework [7]. Here, data about each fiducial is extracted, including the position and orientation of each fiducial. User input is detected on the Input Surface as shown on the extreme left in Figure 7. This surface is lit from below and the web camera senses the light reflected from the fiducials. Images are sent at a constant rate to the reacTIVision software framework.

Compiler subsystem: This data is made available to the Compiler subsystem where both Java and Processing executable code is generated. A code compiler was designed and implemented to convert the observed fiducial set into computer executable code. This compiler makes use of a two-stage process. The first stage requires the conversion of the optical images to Java code. The second step in the conversion process requires the interpretation of the Java code, and the subsequent generation of Processing [8] executable code.

Execution subsystem: The Execution subsystem receives the executable code from the compiler subsystem and executes it. Visual program results are displayed on the computer screen, and audible output is played through the computer's speakers.

6. Usability Tests

The participants were all non-programmers, and for 80% this was the first exposure to programming. Usability tests were conducted during October 2010 with the assistance of 10 individuals (40% male). The age of the participants ranged between 18 to 55 years (10% 15-17yrs, 50% 18-25yrs, 10% 25-35yrs, 30% 35-55yrs) with computer experience stated by them as ranging from basic (70% of participants) to functional (30% of participants). The scale used for stating computer experience is as follows: Minimal (hardly any skill), Basic (a basic knowledge), Functional (a generally good knowledge, but no programming skills), and Advanced (can control a computer well, includes programming experience). The objectives of the tests were to observe and measure the usability of the system and also its aesthetic qualities.

7. Results

It appears that the physical alignment of the codeBlock objects do not pose problems to the Imaging subsystem. In any case, the alignment could be improved by adding physical guides for the rows, similar to rule lines on a sheet of paper. However, this addition would eliminate the some of the benefits of using a free-form area as is currently the case. Careful consideration is therefore required before implementing this option.

We were impressed on how space-efficient the rotary dial is for indicating a numerical value.

The very nature of a physical programming system limits the complexity the program logic can take on. This is however not a problem when a system such as this is targeted at the novice.

The following was evident from the usability tests: a) the Help functionality can possibly be improved by providing additional detail and through the addition of a voice prompt and subtitles; b) lightning from inside the box, used to illuminate the fiducials, need to be improved.

All participants reported that they enjoyed using the system, as could be expected when evaluating something novel.

According to the comments received, the jump function appears to be the most difficult for a novice programmer to comprehend. However, when considering the responses provided in the structured questionnaire, the use of variables and mathematical operations are reported to be the most difficult for this test group. In general, the participants commented that this system was an appropriate introduction to computer programming for the novice, and specifically the logical analysis required in constructing a computer program.

8. Future Work

From our experience with the current design, we have identified the following points that require further consideration: a) We do not yet have a good indication of which programming concepts should be added to the current system, if at all. b) Delays should be added when program loops are executed. This is so that the user can notice the loop action while it is taking place, and not only see the final result of the looping. This is important because the main beneficiaries of the system are the computer novice. c) The targeted end-users (pre-reading and writing) should be involved in the conceptualisation and testing phases of future iterations.

9. Conclusion

We have briefly explained a simple programming environment which takes its input from physical devices and produces outputs on a computer screen. Using optical markers attached to physical objects, it is possible to define a novel programming language which is simple and does not require computer literacy from the user. The system presented also supports group participation because of the large size of the objects manipulated and the large work-surface.

Our optical input mechanism is ideal for the small hands of pre-letterate [2] children, but also poses challenges for the image recognition subsystem. Future designs would benefit greatly if close attention is given to the lighting mechanism used.

As is often noted in the literature, our evaluations confirmed that inexperienced programmers have difficulty in grasping the concept of iteration. Our advice is to remove iteration objects from the system and only introduce these once the user has mastered sequential programming concepts.

Although the single usability test that was conducted at the end of the development cycle is not ideal, we hope that the results will nevertheless inform similar research projects yet to come.

Acknowledgments

We wish to acknowledge the University of Pretoria's COS301 course presenter for the support given, along with his assistants. This work was partly funded by the South African Department of Science and Technology.

References

[1] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. Storytelling Alice motivates middle school girls to learn computer programming. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1455–1464, New York, NY, USA, 2007. ACM.

[2] S. Papert. *The Children's Machine: Rethinking school in the age of the computer*, Basic Books, 1993.

[3] Andrew Cyrus Smith. Dialando: Tangible programming for the novice with Scratch, Processing and Arduino. In *Technology for Innovation and Education in Developing Countries*, 2010.

[4] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: programming for all. *Commun. ACM*, 52(11):60–67, 2009. printed 9 march 2010.

[5] Michael S. Horn and Robert J. K. Jacob. Tangible programming in the classroom with tern. In *CHI '07: CHI '07 extended abstracts on Human factors in computing systems*, pages 1965–1970, New York, NY, USA, 2007. ACM Press.

[6] Sean Hall. *This Means This, This Means That: A User's Guide to Semiotics*. Laurence King, 2007.

[7] R. Bencina and M Kaltenbrunner. The design and evolution of fiducials for the Reactivision system. In *Proceedings of the Third International Conference on Generative Systems in the Electronic Arts*, 2005.

[8] Casey Reas and Ben Fry. Processing: programming for the media arts. *AI & Society*, 20(4):526–538, September 2006.