basil.js – Bridging Mouse and Code Based Design Strategies

Ludwig Zeller[1], Benedikt Groß[2], and Ted Davis[1]

[1] Academy of Art and Design Basel HGK FHNW,
Visual Communication Institute / The Basel School of Design,
Vogelsangstr. 15, 4058 Basel / Switzerland
{ludwig.zeller,theodore.davis}@fhnw.ch
[2] Benedikt Groß, Werastr. 134,
70190 Stuttgart
bg@benedikt-gross.de

Abstract. In this paper we present our JavaScript library basil.js that makes scripting and automation in Adobe InDesign accessible to designers with little previous knowledge in programming. We outline how we derived our API design from the Processing project and applied it to Adobe InDesign. We explain the benefits of combining code and mouse based design strategies within one software package and show how creative users can benefit from the possibility to extend their existing software tools. Lastly the current state of our project is reported and application examples in the form of student projects are given.

Keywords: Generative design, computational aesthetics, tool development, tool modification, educational programming language, Adobe InDesign, Processing, JavaScript.

1 Introduction and Motivation

Digital approaches for print publishing became the common practice in the 1990's. Today, Adobe InDesign is the de facto standard and both a respected and reliable tool for the creativity of layout designers. Thus, it is the primary software package that is used by students and staff in education and third-party commissions.

The Basel School of Design has a long tradition in the education of young designers and a high expertise in visual communication through editorial, book and information design. It is one of our main goals to expand the methodologies of design and to educate students in developing a set of individual, unique styles and aesthetics. One possibility to achieve this is bringing the students in contact with programming and the development of generative and/or interactive digital systems.

We see generative design as an interesting possibility for our students in order to create a unique visual language and handwriting. In particular we are interested in the creation of digital, visual tools as an individual means of expression. The complex behavior of a self-executing formal system such as a programming language can offer both control and surprise. Often the production of partly unexpected output yields an interesting source of inspiration.

A. Marcus (Ed.): DUXU 2014, Part I, LNCS 8517, pp. 686–696, 2014.

2 Tools for Designers

2.1 Generating Design through Code

The introduction of digital desktop publishing to the mass market in the 1990's brought the step from designing something by hand to using screens and mouse driven interfaces in order to do the same thing. Working with the computer obviously made the design experience less tangible, but on the other hand it also promised a vast set of new possibilities like the use of digital fonts, "copy-and-paste" and the "undo" function.

Generative design is the use of formal rules in the shape of algorithms in order to produce a design output. For instance Processing [1] is one of the most commonly used software platforms for this purpose and has become increasingly popular over the past decade. After moving from hand to mouse we have moved from creating with the mouse to creating through code. Or to be more precise the idea of procedural design is being revisited. Before the WYSIWYG paradigm had been established by the computing industry in order to make computers more accessible to a wider audience, the expression through code already had been a very common way to produce visual output. Designers and artists such as Manfred Mohr, Georg Nees and Frieder Nake have been active in the pre-GUI era of visual computing [2]. Nonetheless, their work was already circulating around the core concepts that we link with generative design nowadays.

Already these protagonists faced the situation that generative design is at the border between two different worlds. Common assumptions for this clash of roles put the idea that "artists and designers can't program" against the idea that "a programmer can't design". Therefore, educating designers in this field is an interdisciplinary endeavor. While computational design and art was already known in the 1960's as mentioned above, the idea that artists and designers would program these formal rules themselves was not necessarily commonplace back then. Frieder Nake describes in his article "Teamwork zwischen Künstler und Computer" a cooperating system consisting of an artist, a human programmer, a computer and a visual output machine such as a printer or monitor [3]. This exemplifies a major difference of today's generative design practice where it became normal that the artist and programmer are the same, broadly educated person. An explanation for this could be the availability of easy to use programming environments, our current zeitgeist that is influenced by the tight integration of computers in daily life and an according curriculum in art and design education.

But it has to be emphasized that basil.js and its integration in education is not aimed at making a programmer or even software engineer out of a designer, but to allow for visual experiments and results with greater ease and fewer requirements of their technological knowledge. Additionally we find that creating generative design within an existing WYSIWYG software package such as Adobe InDesign offers a useful bridge between generating through code and adjusting by mouse.

2.2 Modifiable Meta Tools vs. Closed Design Platforms

In general, tools are enabling and limiting at the same time. In the words of Marshall McLuhan: „First we shape our tools, thereafter they shape us.“ [3] We think and

perceive the world and its possibilities through the functional facilities we possess and know. Or to put it in the words of Ludwig Wittgenstein and his thoughts on the borders of "Welterkenntnis": "the limits of language mean the limits of my world". [4]

Many software packages have a closed, specialized nature. They offer a valuable set of pre-defined, common solutions for specific problems. They are relatively easy to learn and use, but their extension is difficult. A programming language on the other hand is a kind of meta tool that allows for questioning the set of available methods and for extending it by creating new tools.

In fig. 1 a number of common software tools and languages are put into a relation of required learning time to the amount of aesthetical quality that is pre-defined by their use. Packages such as the Adobe Creative Suite are relatively easy to learn for beginners offering a great starting point into the world of digital content creation and manipulation but are meant to be used as deployed.

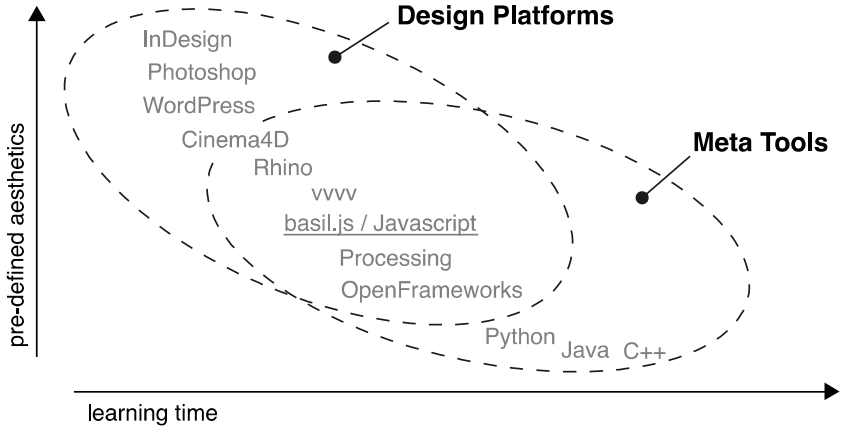

Fig. 1. Mapping common design/programming tools regarding their pre-defined aesthetics and required learning time [Adapted from 5]

On the other far end of the scale you find lower-level programming languages such as Java and C++. These require a far greater amount of time to learn, but on the other hand offer a tremendous amount of possibilities, since eventually e.g. C++ is the foundational layer of software packages such as InDesign. Unfortunately due to the extreme requirements to the user's experience in software design, bringing designers in direct touch with these languages is usually an inappropriate strategy. It would mean a too demanding involvement of them into an area that does not belong to their aesthetical interest. Nonetheless, this might be a meaningful option for some extreme scenarios.

Looking into the middle range of the graph we find software dialects such as Processing, which bundles a subset of the existing possibilities of Java into an easier to learn approach for graphics programming. Complex software packages such as Maxon Cinema 4D that feature built-in scripting facilities are usually relatively easy to learn and allow for the adaptation of the package's tools in order to realize complex and unique projects. JavaScript is regularly used as the language of choice for built-in

scripting, since its powerful interpreter engines are available to software developers to integrate into their software.

We found that both specialized and adaptable approaches have benefits and downsides. In the case of using code as a medium for visual expression the benefits are the ability to build your own tools, to foster the precision of your digital computing, to gain control over your digital production techniques and ideally to discover previously unseen aesthetics.

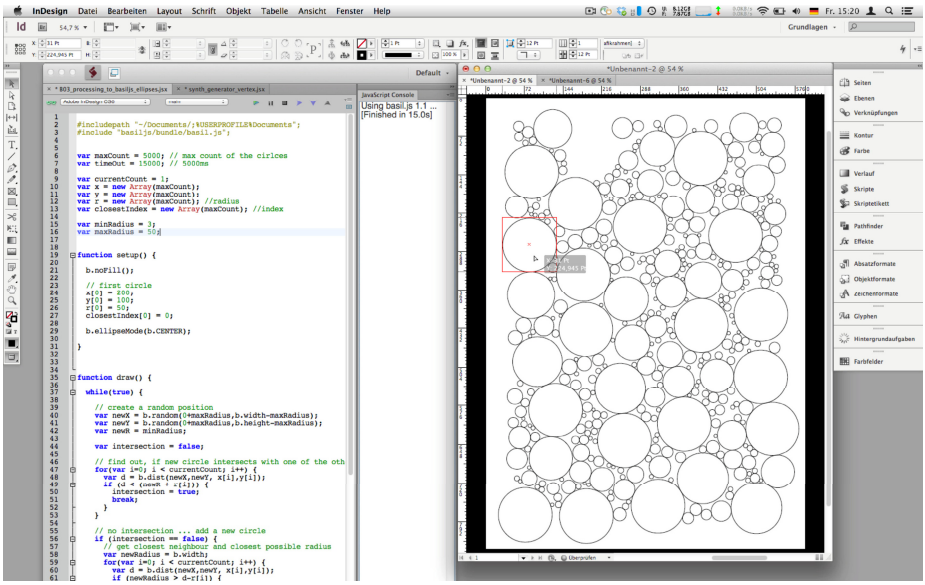

Fig. 2. basil.js code in the Adobe ExtendScript Editor (left) populated a page with circles in Adobe InDesign (right)

But we aren't advocating that code is the best way to design. Instead we want to highlight that there are disadvantages as well, such as the designer not receiving an immediate visual feedback, the manual fine-tuning of generated output is hard or even impossible, it is building upon a very different skill set than design itself and is therefore unintuitive to learn for designers. Therefore, offering a design environment that enables both the possibility for manual mouse based and algorithmic code based design tools is helpful. With Adobe InDesign containing a JavaScript programming interface it becomes a mix between a closed design platform and a modifiable meta tool.

3 Introducing basil.js

The aim of the developer team was to incorporate these scripting facilities in our teaching curriculum. We invited Benedikt Groß to offer an InDesign scripting workshop to our students in Spring 2012. We found out that the Java API of the co-existing software package "InDesign Server", which has been the starting point for the

JavaScript API, is aimed at professional software engineers. This can be derived from its very high-end pricing, the database-driven pipelines it is intended to be connected with and the formal style of coding it requires. Therefore this programming interface is unfortunately especially hard to be learnt and used by designers, who usually do not have much or any experience in programming. This seems like a contradiction since the GUI version of Adobe InDesign is obviously aimed at graphic designers.

basil.js in order to make the InDesign JavaScript API more approachable for designers and artists. This can be seen in the tradition of educational programming languages but in the form of a library or dialect instead of a full language. We aimed at creating an extension to InDesign that brings automation and scripting into layout and makes computational and generative design possible from within InDesign. Additionally, it also includes workflow improvements for data imports from various sources, indexing and complex document management.

3.1 Bridging WYSIWYG and Generative Design

As mentioned above bridging mouse and code based design paradigms was important for us in this project. Fig. 2 explains a common basil.js usage scenario: on the left you see the Adobe ExtendScript Editor, which is used for developing, running and debugging basil.js scripts. While we recommend using this default editor, other coding tools can also be used. On the right side of the screen an Adobe InDesign project is open that shows the result of the execution of the script.

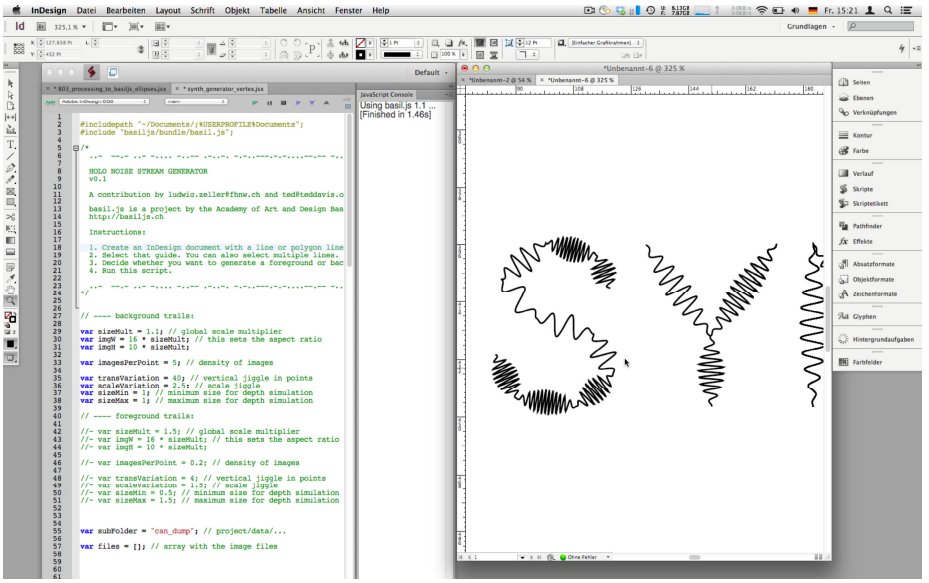

Fig. 3. Manually drawn vector lines have been used as an input for a basil.js script that transforms these according to algorithmic rules

The program filled the page with non-intersecting circles at different sizes. Different to Processing, these circles are then available as live entities within Adobe InDesign and therefore can be altered and deleted through the mouse just like manually produced graphical elements. Fig. 3 shows another example in which manually designed vector paths are being used as an input to the basil.js script on the left. In this case the original vector lines that made up simple typographic characters are being replaced with modulated sine waves that follow the exact path of the original manual input and can be configured in more detail by setting variables in the script.

3.2 Designing an Accessible API

Since the original Adobe InDesign JavaScript API was so hard to use for students we wanted to design basil.js to be significantly easier for integration into our curriculum. Processing has been an important and efficient part of our curriculum already for a number of years and therefore was suitable as a role model for starting basil.js. Since we already teach Processing for a couple of years and made good experiences with this, we took that platform as an inspiration. This brought us to the following question: What made Processing so widely adopted and is it possible to translate this to Adobe InDesign?

```
var doc = app.activeDocument;
var layer = doc.layers.add("layer with a line");
var color = doc.colors.add({model: ColorModel.process,
space: ColorSpace.CMYK, colorValue: [20,100,50,0]});
var lines = doc.graphicLines;
var newLine = lines.add( layer );
newLine.strokeWeight = 1;
newLine.strokeColor = color;
newLine.paths.item(0).entirePath = [[0,0], [300,400]];
```

Code snippet 1: Adding a line with a new color to a new layer without basil.js

```
b.layer("layer with a line");
b.stroke(20,100,50);
b.line(0,0,300,400);
```

Code snippet 2: Same task executed with basil.js

Without conducting an in-depth case study we concluded that the open-source spirit, the well integrated user community, the availability of online learning material and galleries, the integration in academic curricula world wide and its easy to grasp structure made Processing successful. We decided to transfer this “spirit of Processing” to Adobe InDesign by offering a similar programming API and online community. The initial goal to be 100% compatible to Processing codes could not be achieved, but suitable programs can be converted in a short amount of time.

We analyzed the existing functionalities of the Processing and InDesign API and selected a subset that would be especially useful in a common print design scenario. The according InDesign core functions were redesigned in its usability from the viewpoint of a visual designer and have been code-wrapped into basil.js. Processing uses very short and clear names, avoids the Java typical camel case naming and provides overloaded method signatures for getters and setters depending on the argument count. Additionally, the Processing toolkit loads most of its functionality into one very big class. From a software engineering point of view this could be seen as a clearly bad practice, but this avoids the need to understand more sophisticated concepts of organizing codes in classes, packages and software design patterns. Again, we took this as a guideline for our own development.

For many casual scripting tasks we managed to highly reduce the code complexity compared to native InDesign code and therefore improved the experience for the scripting designer. For instance in order to draw a line with a new color on a newly created layer we managed to replace the relatively complex code in snippet 1 with the shorter basil.js version in snippet 2. Please note that this comparison leaves out all the error detection and convenience that basil.js is offering on top of that basic functionality.

The most obvious difference in the two coding approaches can be seen in the demanded sequence of function calls. In the Processing style you are first setting default colors for fillings and strokes, default stroke weights, etc. before you actually apply a stroke to the canvas. This sequence stays in the metaphor of actually using a pencil: once you've taken it at hand its visual characteristics are applied to all the following strokes you are going to bring to the paper. The InDesign API on the other hand demands you to configure the appearance of each single stroke after it has been placed to the canvas. A very modular approach from a professional programmer's perspective, but completely counter-intuitive to graphic designers. This kind of gap between InDesign's API and a designer's understanding of the production of visual output can be seen throughout large parts of the API. In the above example this can also be observed in the overly explicit configuration of new colors and the actual creation of the line: First an "empty" and therefore invisible stroke object is added to the document and only after that the visual features such as colors and even the actual geometric appearance (start and end point) are defined. Without deeper knowledge we assume that the Processing team faced similar awkwardness when they wrapped the desired visual functionalities of the Java Advanced Window Toolkit (AWT) into their dialect.

We managed to keep the naming conventions of Processing for many functions such as color(), line(), rect(), etc. in order to make the transition from Processing to basil.js easy. In cases where no direct Processing equivalent was available we tried to invent new functions with a similar coding style in mind. Unfortunately, we had to bind basil.js to a global object "b" in order to avoid namespace collisions with third-party JavaScript libraries that would occur in a global scope. This is a limitation of JavaScript compared to the well-organized Java language.

4 Project Discussion

Since education is the primary goal in this project, we started to integrate basil.js in the seminars at our institute as a test-run since Winter 2012. The public release took place in February 2013. Since then a number of projects have been created, of which three three examples will be presented briefly.

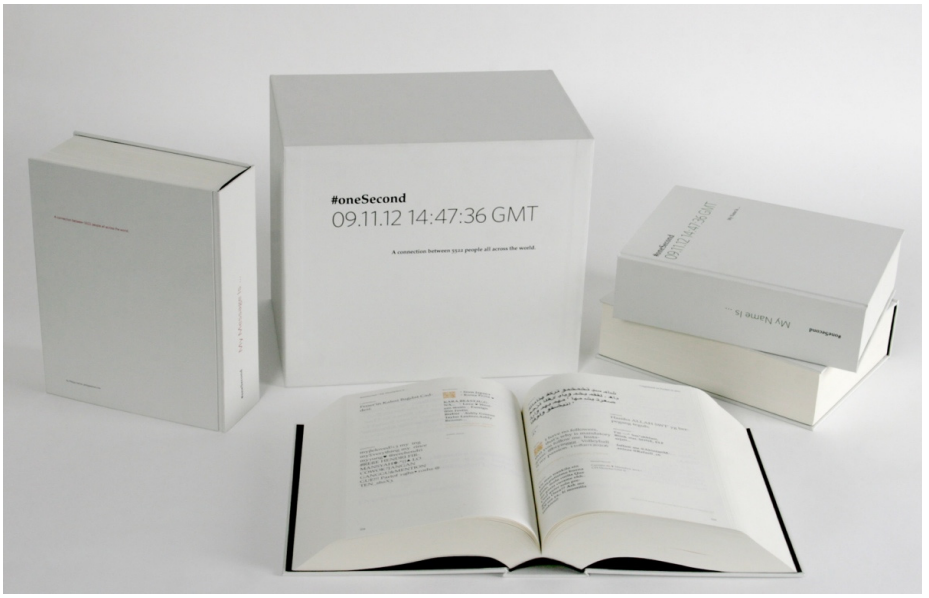

Fig. 4. #oneSecond project by Philipp Adrian. Archiving the data generated in one second on Twitter as a four volume print publication.

In fig. 4 we see Philipp Adrian's project *#oneSecond* that has been advised by Prof. Marion Fink. A one second time span on Twitter has been recorded with the help of a commercial marketing analysis service. The collected data was then transformed and decorated with additionally aggregated information about the involved users and their location, preferences, languages, etc. In total over 4000 tweets were archived in this manner, which lead to a four-volume book publication. While not directly being a generative design project *#oneSecond* gives a good example of how basil.js and InDesign can be used for customized data flows and template based print presentation.

The project *"Romeo and Juliet"* by Patrick Baumann and Inken Zierenberg shown in fig. 5 takes Shakespeare's entire play transcription and presents it in small point size on five posters. basil.js was then used to draw lines between all occurrences of the words "Romeo" and "Juliet". Together the poster series indirectly produces a visual representation of the narrative dramaturgy of the story by showing increasing and decreasing densities of line connections. In this example it was helpful to use InDesign's typesetting engine that works together seamlessly with basil.js in order to find the positions of individual words in flowing text.

Fig. 5. "Romeo and Juliet" project by Patrick Baumann and Inken Zierenberg. Visualizing the dramatic intensity through applying a simple algorithmic rule.

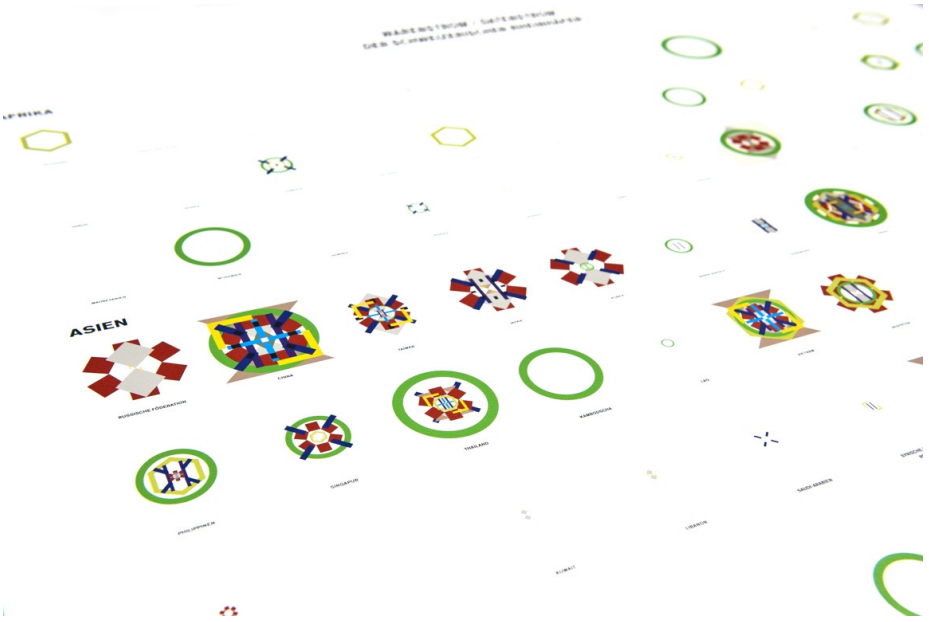

Fig. 6. "Flow of Goods / Data Stream" project by Simon Ziffermeyer. Transforming a database from the Swiss border police to an encoded visual system.

Finally, fig. 6 shows the project *"Flow of Goods / Data Stream"* by Simon Ziffermeyer. This project consists of several data visualizations of the import and export numbers at the Basel Rhine harbor in the form of posters and a book publication. Simon received these numbers separated according to the different types of goods as well as destination and origin countries as Excel tables and loaded the information with basil.js. These numbers were then being mapped to a visual system of overlaid pictograms that make up visual collages in order to give a quickly graspable representation of several numeric values at a glance. This project shows how basil.js is used in a more traditional generative design scenario and benefits from the fact that manually prepared vector shapes can be easily used and duplicated through code.

5 Conclusion

In this paper we positioned today's generative design approach as a revisiting of early concepts of computational design after the introduction of WYSIWYG platforms in the 1990ies. We described the possibilities of adaptable design tools in the creation process and discussed the integrated combination of mouse and code based design paradigms.

In this picture we introduced basil.js as a contribution to our research and education agenda at the Visual Communication Institute in Basel. It has been explained how we improved the accessibility to Adobe InDesign's scripting facilities for non-experts through partly redesigning its programming API. We exemplified the aesthetical explorations that are made possible through bringing automation and scripting into the WYSIWYG design and layout process such as processing and transforming large amounts of external data and to generate designs that would be too time demanding to achieve without code. The aesthetical and practical benefits of the tight iteration between feeding mouse-based designed assets into the generation and transformation through formal rules and vice versa have been shown. Eventually we outlined meaningful areas of specialization for our project that can be seen as an extension for existing poster and book projects in Adobe InDesign, the intense work on typography and its manual corrections as well as the usage in generative book projects.

An open-source project like basil.js is successful if it is used in the outside world and joined by other developers. We already have received submissions both in the form of design projects and pull requests on Github. Additionally, it will be important to see which kind of aesthetical use and innovation this project will see in its near future.

Acknowledgements. Many thanks to Stefan Landsbek for contributing the initial JavaScript architecture, Prof. Michael Renner for making the support through the Visual Communication institute possible, Philipp Adrian for contributing the backend system of the basiljs.ch website, the be:screen GmbH that provided valuable technical assistance, Ken Frederick for contributing a user interface package and other additions as well as our students for giving us error reports and brilliant inspiration in our beta stage.

Special thanks go to Ben Fry and Casey Reas for starting Processing, the processing.js team for providing an open-source Processing port to JavaScript [7] that we partly ported to basil.js and Jürg Lehni for his Scriptographer [8] project that gave us a starting point of inspiration for scripting within the Adobe Creative Suite.

References

1. Fry, B., Reas, C.: Processing. Software platform and website, http://www.processing.org
2. Büscher, B., Hoffmann, C., von Herrmann, H.-C.: Ästhetik als Programm, Max Bense / Daten und Streuungen. Diaphenes, Zürich (2004)
3. Nake, F.: Teamwork zwischen Künstler und Computer. FORMAT Nr 11, 38–39 (1967)
4. McLuhan, M.: Understanding Media. Extensions of Man. McGraw-Hill, New York (1964)
5. Wittgenstein, L.: Tractatus Logico-Philosophicus (1921)
6. Groß, B.: Tools and Authorship in Computational Design. MA Dissertation, Royal College of Art, London (2012)
7. Processing.js, an open-source port of Processing to JavaScript, http://processingjs.org/
8. Lehni, J.: Scriptographer. Scripting environment for Adobe Illustrator, http://scriptographer.org/