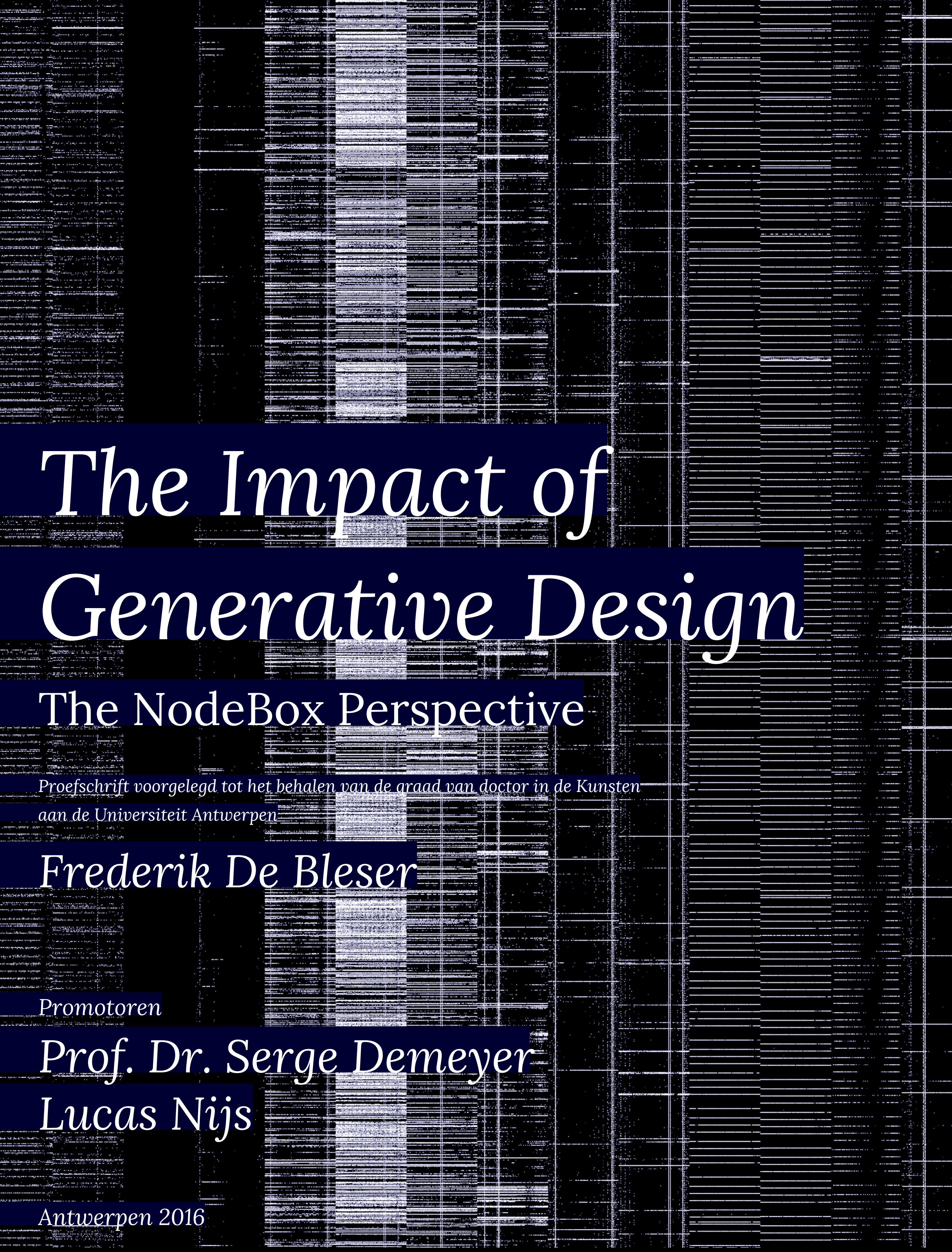


The Impact of Generative Design

The NodeBox Perspective

Proefschrift voorgelegd tot het behalen van de graad van doctor in de Kunsten aan de Universiteit Antwerpen

Frederik De Bleser

Promotoren

Prof. Dr. Serge Demeyer

Lucas Nijs

Antwerpen 2016

Universiteit Antwerpen

SINT LUCAS ANTWERPEN
ST LUCAS SCHOOL OF ARTS ANTWERP

Contents

Abstract

English

In this PhD, we study the impact of generative design tools on graphic design and art and how such an approach differs from traditional digital tools such as Adobe Photoshop. It poses the question if designers are more efficient, explore a broader approach, discover new aesthetics and if they are self-reliant.

We study our own approach and make art projects using generative tools developed in-house. We study the approach that others have taken through masterclasses and workshops, both with students and professional designers.

This research helps to understand the scope of generative design tools and can help shape the future of commonly used tools.

Dutch

In dit doctoraat onderzoeken we wat de impact is van generatieve software op grafisch ontwerp en kunst, en hoe een dergelijke aanpak verschilt van klassieke digitale gereedschappen zoals Adobe Photoshop. We stellen de vraag of ontwerpers efficiënter zijn, een bredere aanpak kunnen hanteren, een nieuwe esthetiek ontdekken en of ze binnen zo'n omgeving zelfredzaam zijn.

We bestuderen onze eigen aanpak en maken kunstprojecten met zelfgemaakte generatieve software. We bestuderen de aanpak van anderen door masterclasses en workshops met studenten en bevragingen van professionals.

Uit het onderzoek kunnen we de reikwijdte van software voor generatieve vormgeving begrijpen en mee de toekomst bepalen voor de digitale gereedschappen die vormgevers dagelijks gebruiken.

Contributions

Software

NodeBox 1 (2004) is a Mac OS X application that lets you create 2-dimensional visuals using Python programming code. Graphics can be animated or made interactive and can be exported to PDF or as QuickTime movies. NodeBox 1 is described in Section 3.4.

NodeBox 2 (2010) was a cross-platform, visual node-based software application used in creating generative designs and data visualisations. It has since been superseded by NodeBox 3. It is described in Section 4.2.

NodeBox 3 (2012) is a cross-platform, visual node-based software application used in creating generative designs and data visualisations. It uses lists as its primary abstraction and list matching to create variation. Its functionality is described in Section 4.4.

DataPipe (2014) is a utility application designed to sample, filter and aggregate large datasets. It is described in Section 4.3.

NodeBox Live (2016) is a web-based visual node-based software application. It is a port of the conceptual model of NodeBox 3 to JavaScript. NodeBox Live is described in Section 4.5.

OpenType.js (2014) is a JavaScript library that can parse and read TrueType and OpenType fonts. The rationale and implications for this library are described in Section 4.6.

Logic Layout (2016) is a graphical vector editor build around random ranges and constraints. We describe Logic Layout in Section 5.3.

Writing

Gravital: natural language processing for computer graphics (2013). Chapter for the book "Creativity and The Agile Mind: A Multi-disciplinary Study of a Multi-faceted Phenomenon" (*De Smedt*, 2013). Described in Section 3.6.3.

Artwork / Exhibitions

Traveling Letters (2012) – part of the "Traveling Letters" exhibition organized in Lahti, Finland. Computer Installation. Technique: Scala, Processing. (Figure 1)

arial.wtf (2013) – part of the "Traveling Letters – Typo Jazz" exhibition organized in Vilnius, Lithuania. Print on paper, 70cm x 100cm. Technique: JavaScript, OpenType. (Figure 2)

Frequensea (2015) – exhibition organized in SHOWROOM, Antwerp in March 2015, in collaboration with Lucas Nijs, Pieter Heremans and Lieven Menschaert. Print on aluminium, projection, installation, virtual reality. Technique: Custom software, HackRF. (Figure 3)

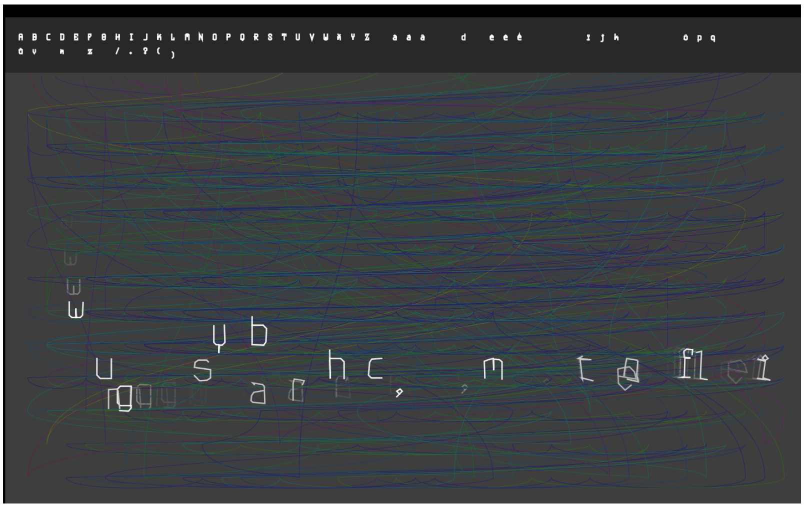


Figure 1: "Traveling Letters", Frederik De Bleser, 2012

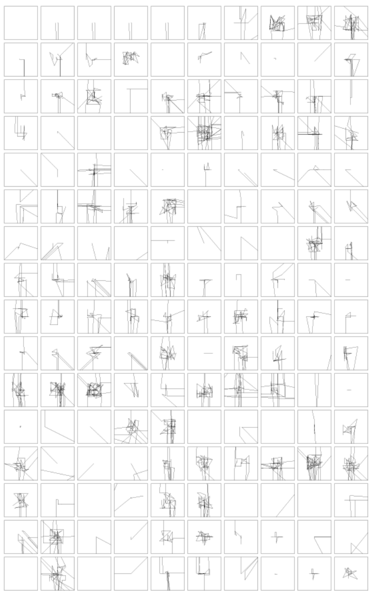


Figure 2: "arial.wtf", Frederik De Bleser, 2013

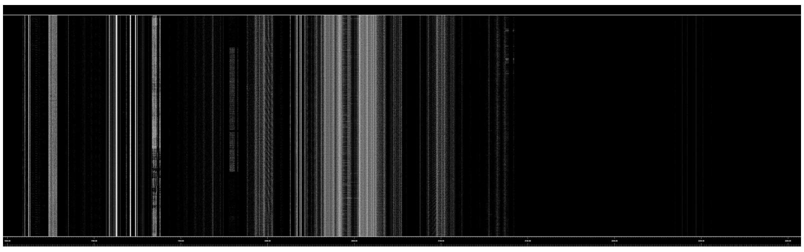


Figure 3: "Frequensea", Frederik De Bleser, 2015

Acknowledgements

A PhD thesis is an endeavor you can not do by yourself, although it sometimes feels like it, especially when you're holed up in a cave trying to make the deadline. That's why I'd like to acknowledge and thank the people who have helped me on this journey.

First off I'd like to thank my promotors Prof Dr. Serge Demeyer and Lucas Nijs, without whose guidance, support and sometimes less-than-gentle prodding there wouldn't have been a PhD. Their advice, remarks and hands-on approach have helped me create a PhD of which I can be proud.

My internal jury members Prof. Dr. Hans Vangheluwe and Prof. Dr. Kris Luyten for their input and interesting conversations. My external jury members, Benedikt Groß and Prof. Dr. Michele Lanza, for taking time to review my thesis in depth and provide helpful critique.

I'd like to thank Dr. Tom De Smedt, whose advice as a was very helpful in guiding the overall structure of the PhD. Of course all the rest of the Experimental Media Research Group: Lieven Menschaert, Stefan Gabriëls, Pieter Heremans and Ludivine Lechat.

I also like to thank Werner vander Meersch and Livin Mentens for their collaboration on projects. Also, special thanks to the PhD coordinators Ruth Loos and Kim Gorus.

I also like to thank the people who have helped me show my work: Prof. Ausra Lisauskiene for giving me multiple opportunities to present my work, Pia Clauwaert and Michel Van Beirendonck for exposing our work in SHOWROOM, and Hugo Puttaert for letting EMRG take the stage at Integrated. Also the people of Bocoup, for giving me the opportunity to present at OpenVisConf.

All the students in workshops throughout the years whose kind words, less kind words and utter frustrations helped shape NodeBox to what it is today.

I also like to thank Erik van Blokland and Just Van Rossum of LettError, whose DrawBot project helped kickstart NodeBox 1.

And last but not least I'd like to thank my family: my wonderful girlfriend Roselien as well as my daughter Erlin and son Linus. They had to endure a very stressed out friend and father. I'm also very grateful to my mother, father, mother-in-law and father-in-law for taking care of the kids when I needed to make a deadline.

1 Introduction

If you're reading this, you're probably a designer. Or a computer scientist. Or maybe you would identify as someone who falls between the two disciplines. We've experienced difficulty ourselves in trying to articulate to our friends or family what it is we do. Our field of generative design sits on a precipice between art and science, connecting the world of ideas with the world of ones and zeroes. This often results in an exciting academic or professional life, but can be quite awkward at dinner conversations.

Generative design and data visualization can be found in everything from mundane applications like personalized direct mailings to visualizations of our vitals on our smartwatches. They entertain passers-by of Deutsche Bank in Hong Kong (Figure 4) or visualize taxi flows in New York City (Figure 5). They can help illustrate complex issues such as gun control through storytelling (Figure 6) or break out of the virtual world and become physical objects (Figure 7).

Building these new kinds of applications requires new tools. John Maeda's Design By Numbers (*Maeda*, 2001) and Fry and Reas' Processing (*Reas*, 2007) elevated generative design tools from niche hacks to apps used by a broader public of designers. The terminology of "sketches" helped establish the medium as not just a production tool but an exploratory environment in its own right. Generative tools eschew the old metaphors of programs like Photoshop. They do not have a brush, or pencil or eraser. Instead they expose the fundamental building blocks of the machine and make them available for creative purposes. They give control over the logical flow of computer programs and the mathematical concepts underneath the graphics.

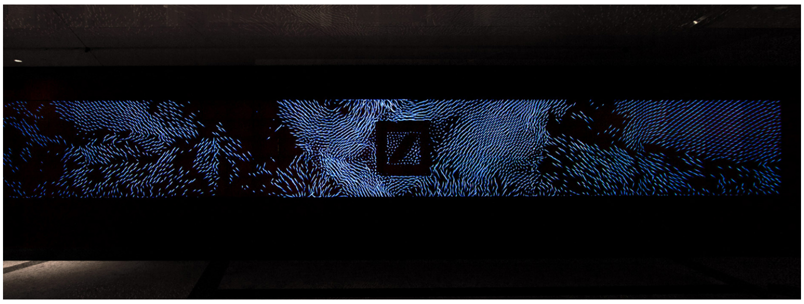


Figure 4: "Deutsche Bank Realtime Visualization", Universal Everything, 2011

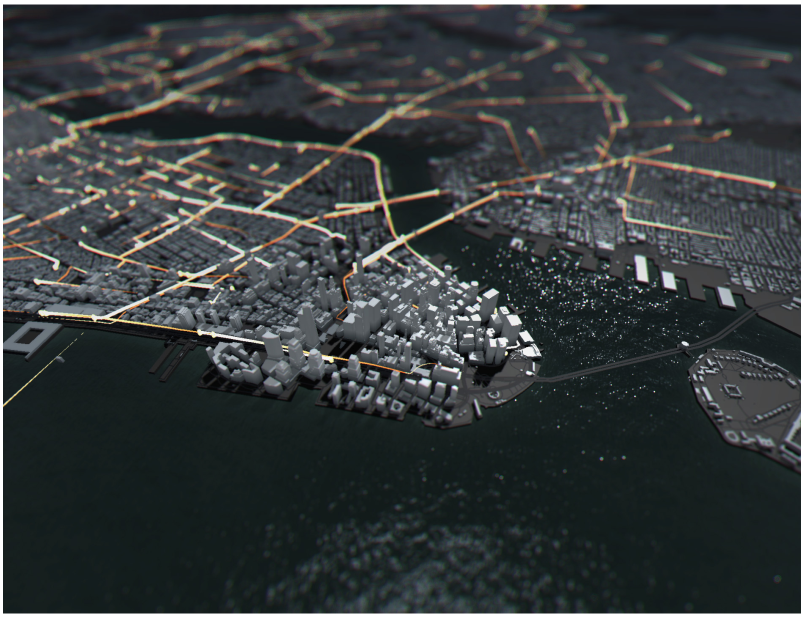


Figure 5: "Taxi, Taxi!", Robert Hodgin, 2016

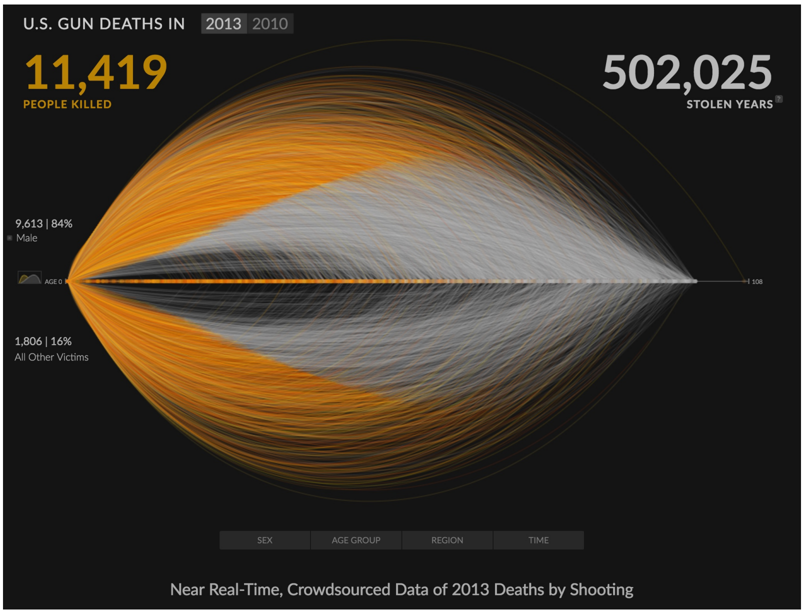


Figure 6: "U.S. Gun Deaths", Periscopic, 2013

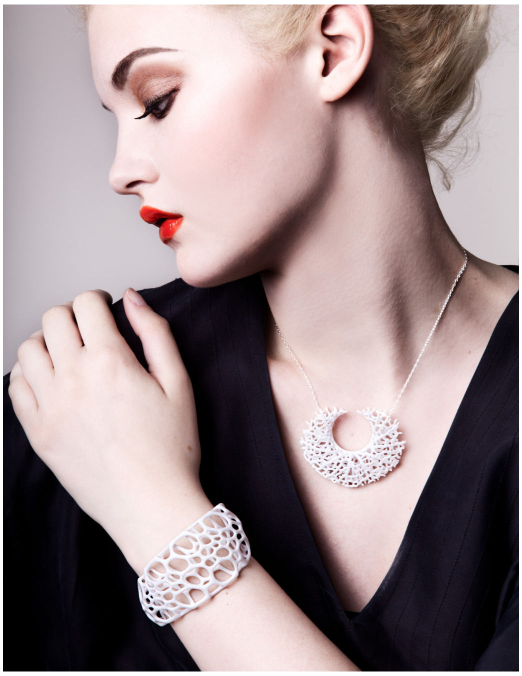


Figure 7: "Cell Cycle Jewelry", Jessica Rosenkrantz and Jesse Louis-Rosenberg, 2009

As part of the Experimental Media Research Group, we've been researching new approaches to graphic design since 2004. Our aim is to promote the computer from a production tool to a creative partner that helps extract, process and present information. We've developed NodeBox, an open-source tool used for automating graphic design production tasks. Newer versions have combined a programming approach with a visual node-based approach. This thesis will evaluate the different versions of NodeBox, note their respective advantages and disadvantages, and describe how we mitigated issues.

The tools we use are not a side issue. They are a fundamental part of the creative process. The Italian design studio TODO, who won the Silver European Design Award in 2010 for their work Spamghetto, wrote about NodeBox: "Spamghetto wouldn’t be a reality without their wonderful software tool for generative graphics." *(Olivero, 2010)*

All tools sit between the user and the hardware of the machine. Generative design tools, by their very nature, gravitate closer to the side of the machine. Their conceptual model might be unfamiliar to designers, raising the barrier to entry.

Our hypothesis is that *a generative design approach is a useful addition to a designers' toolbox*. Its use can accomplish four different goals:

G1 – Efficiency Generative design tools allow designers to become *more efficient.* Designers and artists are faster at completing tasks and are not arbitrarily restricted by what a tool allows them to do.

G2 – New Aesthetics Generative design tools allow designers to create results that have not been seen before. Designers and artists create work that they would otherwise not be able to create.

G3 – Broader Approach A generative design approach allows designers to try out more things before settling on a solution. Designers and artists can integrate new and familiar tools to extend their possibilities.

G4 – Self-Reliant Generative design tools are easy-to-use. Designers and artists can use them without any outside help.

This thesis is structured around three important programming paradigms. Each paradigm represents a different approach in thinking about computing.

Chapter 2, Context, will describe the requirements for generative tools and how we evaluate our approach.

Chapter 3, Imperative Programming, will cover the paradigm that is closest to the physical nature of the machine. It describes how loops, conditions and data structures can be used for generative design, examines NodeBox 1 and evaluates this approach through visual works and usability studies.

Chapter 4, Functional Programming, introduces visual programming, and explain how three iterations of NodeBox gradually lower the barrier to entry based on input from users.

Chapter 5, Constraint Programming, proposes a new tool, Logic Layout, which obviates the need for any kind of programming, visual or otherwise, while still retaining important benefits of generative design.

2 Context

The use of digital tools in the arts has a long history. However, it wasn't until recently, with the creation of tools like Processing and NodeBox, that these tools become widely available to an audience of non-technical artists.

In 2004 we started the Experimental Media Research Group (EMRG) from the graduation project of Dr. Tom De Smedt and Frederik De Bleser, under the auspices of Lucas Nijs. EMRG is associated with the St Lucas School of Arts of the Karel de Grote-Hogeschool (Antwerp, Belgium). Our vision was to make the power of the computer available to a wide audience of designers and artists. The group develops a number of software tools and libraries, most notably NodeBox, a tool for generative graphic design, and Pattern, a library for natural language parsing and web mining.

Through workshops, teaching design students and professionals and through online communities we spread word of our tools. But just getting those tools out there doesn't mean everybody will use them. For example, our research has shown that 67% of surveilled students have heard of Processing but only 29% have used it (see Section 4.7.4). Clearly, there is a gap between awareness and usage.

The goal of this thesis is to evaluate the impact of tools for generative design. Specifically, we want to measure if users are more efficient, create work that is aesthetically new, if they handle a broader approach and if the tools they use make them self-reliant. We do this by observing how students and professionals work, we ask them through questionnaires and in depth interviews, and we perform experiments using instrumented software (see Section 4.7).

In this section we will give an overview of terms and concepts used in graphic design, programming, and how we evaluate our approach.

2.1 Generative Design

Generative design is a method whereby we use an algorithm to generate an image. The design is controlled by a set of rules that determines its outcome. This basic principle has many different applications. Here are some examples:

1. Given a database, design a product catalog using a predefined layout scheme (see Section 3.5.3.1). Here we use the computer pure as a production tool. We give it a very strict ruleset and a dataset. In essence, the computer just automates the boring, repetitive work we would have to do by hand. This approach is used in direct mailing to personalize the communication.

2. Describe a procedure to design a logo that includes some randomness. For example, the ChampdAction logo (see Section 3.5.3.3) randomly connects lines between random points in the outline. The randomness is limited to predefined bounds. This gives the computer some wiggle room and generates a (possibly infinite) set of variations.

3. Use the computer as an idea generator. For example the Prism algorithm (*De Smedt*, 2007) generates a color palette for any given word. Here we can use the "wisdom of the crowds" to go beyond randomness and towards a system based on knowledge. The job of the algorithm is no longer to execute instructions to generate a result that we could create by hand as well (as tedious as it might be), but to help generate ideas and surprise us.

4. Allow the computer to operate "autonomously". This is often used where we know how the solution will look, but not the exact steps to get there. Approaches here are declarative programming, evolutionary algorithms or neural networks. The Ottobot algorithm (see Section 3.5.3.2) is an example of this approach.

Throughout this thesis we will look at examples for all these different applications. We will examine if using tools for these different applications make designers more efficient (G1), allow them to create a new aesthetic (G2), let them explore a broader approach before settling on a solution (G3) and become self-reliant (G4).

2.2 In History

Artists have used the "generative design" model long before the invention of the computer. Many artists in the Renaissance (like Raphael, Verrocchio) ran large workshops with assistants executing the work based on drawings and instructions by the master. Most famously, Andy Warhol's Factory had several assistants producing different variations of his silk-screen multiples based on instructions by Warhol.

The appeal of using programming for design and art is simple: it allows you to leverage the speed of the computer to automate repetitive tasks, thereby freeing the artist to concentrate on the ideas instead of the execution. It's like having a large workshop with assistants, all carrying out your instructions, for a fraction of the cost and time.

2.3 Data Visualization

In the age of big data there is an opportunity for graphic designers to apply their design knowledge to the visualization of abstract information. The job of the designer – converting abstract concepts into visual representations – maps very well onto the field of data visualization. However, the field represents a unique challenge since the subject matter, the data, is usually large and often dynamic (*Tufte*, 1983).

Work by Bret Victor proposes three choices available to graphic designers interested in visualization (*Victor*, 2013). They can *use* a set of ready-made visualizations, available in spreadsheet software and online tools. However, since it's the task of graphic designers to come up with creative ideas, using "out of the box" solutions is not feasible. They can *draw* their visualizations using pencil and paper or software like Adobe Illustrator. This provides immediate control but connecting each data point to a visual representation is a manual, error-prone process. Lastly, they can *code* their visualization using programming tools. This avoids the inflexibility of spreadsheets and the static nature of drawing since new data can be fed into the visualization continuously.

For large data sets, programming a visualization is the only realistic option. Specific tools for data visualization have been created (most notably D3 (*Bostock*, 2011)). We will focus on the use of NodeBox for data visualization and how the different goals set out in the introduction apply to different iterations of the software.

2.4 Thinking in Metaphors

From the earliest cave paintings to modern design applications, we have used tools to express our creativity. In the real world these tools are pencils, markers, and before the age of desktop publishing, rulers and glue. In the digital age, these tools have digital analogues. The very naming of the most known graphic design application – Photoshop – comes from the physical place that allowed you to develop your photos. The tools in Photoshop are metaphors for the physical objects they are replacing: the brush, the eraser, the stamp, the dodge tool (used in the darkroom to lighten areas of the image) (*Galai*, 2015).

These metaphors are useful when creating a new tool because they encourage adoption. They allow users transitioning from the analog world into the digital world to take their knowledge with them. Ever used an eraser? Now the eraser is an icon in the toolbar. However we've long passed the point were those metaphors make sense. Just as the save icon is represented by a floppy disk, a technology that has been obsolete for decades, people using Photoshop today have never seen a physical dodge tool.

A bigger problem is that those metaphors limit us to think in their terms. The available tools and metaphors in design applications establish a language, a way that allows us to express our intent through tools. And even though that language is malleable in the digital world (e.g. we never run out of paint to fill the page with), it still fundamentally limits how we think about solving a design problem. As Wittgenstein said: "the limits of language are the limits of my world" (*Wittgenstein, 1994*).

Some design problems like data visualization require a completely different approach to design. If we want to visualize, say, broadband penetration in the United States, it is no longer useful to think in terms of physical metaphors. We've reached the limits of the metaphoric language we can use.

Pioneers like John Maeda have been promoting a different way of thinking about computers. Not as a way of carrying over and extending physical metaphors, but as an opportunity to develop entirely new metaphors. His software, Design By Numbers, was an important milestone in producing an application that could be used by non-programmers to use the computer in entirely new ways (G2).

Of course we *still* think in metaphors. Only now these metaphors closely map to the physical reality of the computer. Instead of dealing with pens and erasers we think with variables, conditions, loops and objects. Underneath they still get translated to machine code instructions. We hypothesize that thinking with these new metaphors can allow designers to be more efficient (G1), more creative (G2), help broaden their approach (G3) and thus are a valuable addition to a designers' toolbox.

2.5 Speaking the Same Language

To give instructions to the computer we need to speak its language. We need to agree on a shared form of communication that both the user and the computer can understand. This language can be text-based or visual. And because it is a set of coded instructions that *programs* the computer to behave in a predetermined way, we will call it a *programming language.*

Just as there is no single human language, there is no single programming language. They range from low-level (close to the physical reality of the computer) to high-level (close to the user's conceptual model), from mainstream (Java) to obscure (Piet). And just as our choice of metaphors decides the final outcome of the work, so does the choice of programming language.

In this PhD we will study three different programming paradigms: imperative programming, functional programming and constraint programming. For each we will demonstrate related work and show our own applications within that space. We will examine how each paradigm relates to the four goals set out in the introduction.

We recognize that our categorization is overly broad: in reality there is a lot of overlap and cross-pollination between paradigms. We decided to use these paradigms as a helpful framework to compare different approaches to programming.

2.6 Evaluating our Approach

During the PhD we made observations of users using the program (both in person and using screen casting software), we had them fill out questionnaires, conducted in depth interviews, performed experiments and evaluated visual results. To validate our hypothesis we will need to look at how each of these instruments can be used to validate our four goals.

Orthogonal to these measurements are the *scales* we can use to rate the overall fitness of a system. We integrated the Dreyfus scale, Fred Brook's distinction between Accidental Complexity and Essential Complexity, and Soloway's method of plan composition. These scales will be used to make observations about the system as a whole.

2.6.1 Instruments

The NodeBox workshops were an opportunity to evaluate our tools in real-world situations. They are intensive one- or two-week workshops designed to give students an introduction in programming, new media arts and digital sketching. The results of these workshops were posters, movies and interactive projects. The model for these workshops was based on the experimental type design workshops by Lucas Nijs (*Nijs*, 1998), originally using FontLab and Macromedia Director. The NodeBox workshops retained the timing and "flow" of these workshops while using new tools.

We used a number of instruments to gage the impact of generative design. During workshops we made **observations** from the students working with the software: we noted which aspects they were delighted about, where they struggled, and where they got completely stuck and needed help. At the end of the workshop we used the *System Usability Scale (Brooke*, *1996)*, an industry standard usability **questionnaire** that contains questions like "I thought the system was easy to use", and scales from "strongly agree" to "strongly disagree". In addition, we provided a **feedback form** where users could list advantages and disadvantages of the software.

In some workshops we conducted **in depth interviews** with participants. These were recordings, ranging from 15-30 mins, talking in depth about the usability of the software and applicability in their professional work. In later workshops we also used **A/B testing** to compare the efficiency and usability of textual and visual programming approaches (see Section 4.7).

2.6.2 Four Goals

To validate our hypothesis we will need to look at how each of these measuring instruments can be used to validate our four goals.

G1 – Efficiency To measure efficiency we recorded screencasts during certain workshops to see how fast users could complete a given task. To compare the efficiency of textual and visual programming we set up an A/B experiment (see Section 4.7).

G2 – New Aesthetics To evaluate whether users could produce new and surprising results, we looked at the *visual results* produced at the end of each workshop. We also looked at *feedback forms* and *in depth interviews.*

G3 – Broader Approach To determine if users were able to explore more, we *observed* how students used the software during the workshop. In addition, our *in depth interviews* explored how usage of the software altered their creative process.

G4 – Self-reliance To measure self-reliance we measured the usability of the software using the System Usability Scale (*Brooke*, 1996) *questionnaire.* In addition, users often indicated usable and less-usable aspects of the system on *feedback forms.* We also *observed* students during the workshop to see where they would get stuck, and questioned them about it during *in depth interviews.*

2.6.3 Accidental Complexity vs Essential Complexity

Fred Brooks, in his book "No Silver Bullet", distinguishes between two different types of complexity: accidental complexity and essential complexity (*Brooks*, 1987). **Essential complexity** is caused by the problem to be solved. It emerges from the requirements of the system: if the users want the program to do 5 different things, all of these 5 things are essential, and we can't reduce this further. **Accidental complexity** relates to problems introduced by the infrastructure around the problem: syntax, compiler invocations, translation between the language to machine code,....

The invention of higher-level programming languages made a significant improvement in the area of accidental complexity. Python manages the memory for us, reducing the burden to write programs in it.

Whenever we discuss an approach we will use Brook's framework to distinguish between accidental and essential complexity. For example in Section 3.6.3 we will argue that language syntax is accidental complexity. It has no bearing on the problem to be solved: it is only there to tell the compiler what machine code to output. Program state (discussed in Section 3.6.5) is another example of accidental complexity: the careful management of the current fill and stroke color, the transformation matrix and the font require significant mental processing even if they do nothing to solve the fundamental problem.

The benefit of reducing accidental complexity is that users have more time to focus on the task at hand (G1). In addition, any complexity has the risk of creating a situation where the user is stuck, reducing their self-reliance (G4).

We will see that reducing accidental complexity is a balancing act. For example, replacing textual programming with visual programming will solve issues with syntax but introduces new complexity related to closeness of (visual) mapping (*Green*, 1996). In addition, removing complexity can also limit expressivity, inadvertently creating a situation where we can't solve the essential problem anymore.

2.6.4 Plan Composition

Teaching the syntax and semantics of a programming language is not enough. The real problems novices have with programming (both textual and visual) is being able to "put the pieces together": to compose a bigger program out of smaller language constructs (*Spohrer*, 1986).

To solve a problem, humans employ template-like solutions, or *plans*. In programming, a plan consists of a series of code fragments joined together to solve the problem. Plans are implemented using *language constructs*: simple statements, loops, conditions, and even bigger groups like classes and modules. Learning to program is mastering the problem-solving skill of coming up with a plan (*Soloway*, 1986).

Using language constructs incorrectly and plan composition are the two major causes of programming errors for novices (*Ebrahimi*, 1994). We will talk about language constructs and syntax when evaluating NodeBox 1 (see Section 3.4). We evaluate plan composition by looking at how a system helps its users with planning. For example, nodes and dataflow in NodeBox 3 are an effort to make plan composition explicit (see Section 4.4.8.4) whereas Logic Layout attempts to work around plan composition entirely (see Section 5.1).

2.6.5 Dreyfus Model of Skill Acquisition

The Dreyfus model is a model of how students acquire skills. It was developed in the 1980's by the brothers Stuart and Hubert Dreyfus (*Dreyfus*, 1980).

The model outlines five discrete stages through which students pass on their way from novice to expert. The model is *situational*: it applies to a specific skill, not to the person as a whole. You can be proficient in programming but a novice in line dancing, for example. The stages are **novice** (beginners with little to no experience), **advanced beginner** (able to perform simple tasks on their own but have difficulty troubleshooting), **competent** (conceptual model of problem domain, can troubleshoot), **proficient** (need larger conceptual framework, can self-correct) and **expert** (primary source of knowledge in the field).

We use the Dreyfus model to help us rank students when performing A/B experiments (see Section 4.7.1). When developing the software the model helps in determining which type of documentation or interface is needed at which skill level (see for example Section 4.5.4).

3 Imperative Programming

In this chapter we will describe imperative programming, which is the dominant programming paradigm for generative design and programming in general. We will show how generative design works through a tutorial, describe NodeBox 1 and show visual results by students, commercial projects and our own work. Finally we will evaluate this approach and describe its advantages and drawbacks.

3.1 Introduction

Imperative, or procedural programming, is the dominant programming paradigm. It uses statements (actions) that describe each step the computer has to perform. The computer will execute a series of commands (assignments, loops, conditions and calls), thereby changing the internal state or drawing something on the screen. General-purpose programming languages like C/C++, Java, or JavaScript are all imperative programming languages.

Imperative programming focuses on the *how*: how the program should get to the final result, specified step by step. It is comparable to a cooking recipe: a list of instructions that should be followed in order to reach the end result.

3.2 Background / Related Work

In this section we want to show a number of imperative programming languages and tools that inspired the development of NodeBox. We selected projects that were important from a historical perspective, that have been used to create influential generative work, or that have an important educational purpose.

3.2.1 Logo

Logo is a programming language designed for education, created by Wally Feurzeig and Seymour Papert. It is well known for its "turtle graphics" in which line graphics were produced by an on-screen cursor or a small robot called a "turtle". Students were encouraged to reason about the turtle's motion by imagining what they would do if they were the turtle.

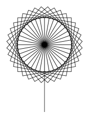
The turtle moves with commands relative to its own position, e.g. `RIGHT 45` will spin the turtle to the right 45 degrees.

First we define a routine to draw a square:

```
TO square
REPEAT 4 [FORWARD 30 LEFT 90]
END
```

Then we draw this square multiple times, each time rotating a bit:

```
 FORWARD 100
 REPEAT 36 [LEFT 10 square]
```



The educational aspect of Logo, specifically the vision of Papert as described in his book Mindstorms (*Papert*, 1980), played an important role in the development of NodeBox.

3.2.2 PostScript

PostScript is a computer language for creating vector graphics. It was created by John Warnock, Charles Geschke, Doug Brotz, Ed Taft and Bill Paxton in 1982 (*Press*, 1985). It was developed in tandem with the first laser printer as a standard for describing the contents of a page.

PostScript is a complete programming language of its own. It contains instructions for specifying geometry in terms of straight lines and Bézier curves. Letter shapes (glyphs) are described as a combination of lines and curves as well, which can then be rendered at any resolution.

An interesting application of generative design through PostScript is Beowolf, a digital typeface designed in 1989 by Just van Rossum and Erik van Blokland from LettError (*van Blokland*, 1991) (see Figure 8). This typeface exploited the fact that PostScript was a full programming language by including random instructions in the description of the glyph shapes. Each time the printer would output a glyph as part of a word, it would create a different variation of the letter. Color laser printers would even output four different variations of each letter, for each of the CMYK (Cyan / Magenta / Yellow / Black) color layers in the printer.

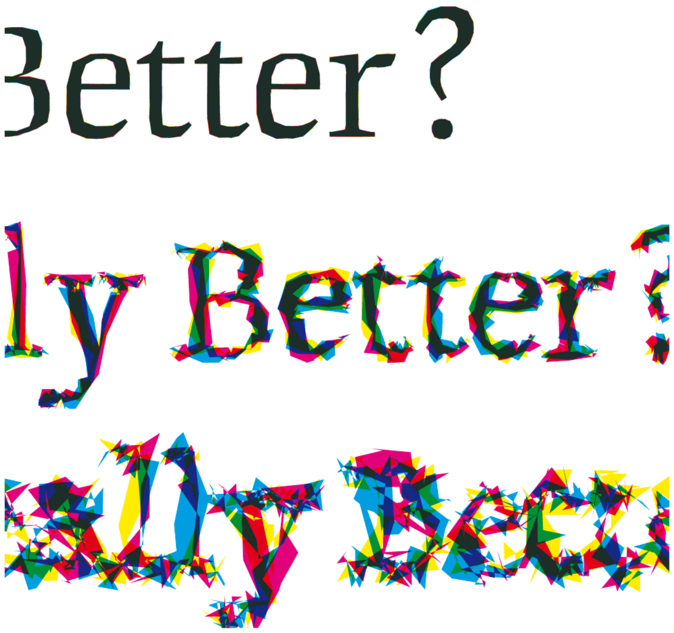


Figure 8: Beowolf by Just van Rossum and Erik van Blokland

PostScript, and its concepts of a state machine, was instrumental in designing the command API for NodeBox 1.

3.2.3 Design By Numbers

Design By Numbers was a software application created by John Maeda at the MIT Media Lab, first released in 1999 (*Maeda*, 2001). It was aimed at allowing designers, artists and other non-programmers to easily start computer programming.

The programming environment is kept deliberately simple. It contains a 100 by 100 canvas that only supports black, white and shades of gray. By removing all non-essential complexity, it boiled generative design down to a minimum viable product: a setting that allowed room for experimentation within the bounds of the medium. The book Design By Numbers demonstrates the broad range of possibilities, even within these limits.

Design By Numbers was purposely created as an introductory tool into generative design, but its approach is universally applicable and incited a whole range of new generative design tools. Most famous is Processing, created by two of John Maeda's alumni, Casey Reas and Ben Fry.

Both the book and software for Design By Numbers showed that by fully embracing the metaphors of the machine, a new kind of aesthetic was possible (G2).

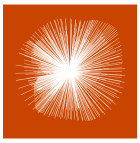
3.2.4 Processing

Processing is a computer language and integrated development environment (IDE) that builds on the work of Design By Numbers to create an environment fruitful for new media arts, visual design and data visualization (*Reas*, 2007).

Processing was designed as a tool to get non-programmers started with programming with as little overhead as possible. Processing documents are called "sketches", emphasizing the processing of using the computer as a digital sketchpad. Clicking the "Run" button executes the sketchs and shows the visual output. Processing is build on the Java language, but uses a simplified syntax and an easy API for graphics programming, inspired by PostScript and OpenGL.

Here's a simple visual program in Processing:

```
void setup() {
    size(400, 400);
    stroke(255);
    background(192, 64, 0);
}

void draw() {
    line(200, 200, mouseX, mouseY);
}
```



This program draws a line from a point in the middle in the screen to the position of the mouse. Because the draw method is called repeatedly, and previous results aren't automatically erased, this results in an ad-hoc drawing program for radial shapes.

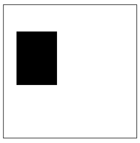
Additional projects were spun off from Processing, most notably Arduino, an open-source electronic prototyping platform and p5.js, which ports Processing to the web. The popularity of Processing made graphic designers aware of both the existence and importance of generative design.

3.3 Tutorial

Before we discuss our contributions we want to give a short introduction into using imperative generative software. This tutorial will use NodeBox but the concepts explained here also apply to other imperative programming languages like Processing or p5.js.

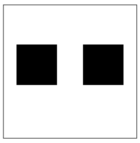
NodeBox has a set of built-in commands that render elements to the screen. For example, you can draw a simple rectangle using the `rect` command:

```
rect(10, 20, 30, 40)
```



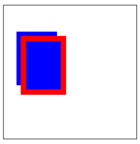
The numbers in this command specify a position, going from left-to-right and top-to-bottom:

```
# This rectangle will appear to the left.
rect(10, 30, 30, 30)

# This rectangle will appear to the right.
# Note that the value of the first parameter is larger.
rect(60, 30, 30, 30)
```



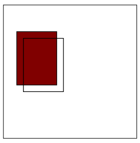
NodeBox remembers the current "fill" and "stroke" style. Think of the fill and stroke operations as grabbing a crayon: every shape you draw afterwards will be drawn with that crayon.

```
# The colors are specified as R/G/B values
# in a range between 0.0 and 1.0.
fill(0, 0, 1)
rect(10, 20, 30, 40)

stroke(1, 0, 0)
strokewidth(4)
rect(15, 25, 30, 40)
```



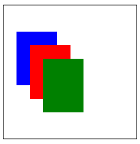
You can turn off the fill color using the "nofill" or "nostroke" commands:

```
fill(0.5, 0, 0)
stroke(0)
rect(10, 20, 30, 40)
# Turn off the fill
nofill()
rect(15, 25, 30, 40)
```



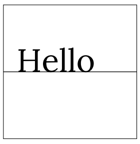
NodeBox uses a "painters model" of rendering (See (*Ferraiolo*, 2000)). Each successive operation "paints over" earlier operations. In other words, items that appear first in the program are layered in the back. To move items to the front, re-order them so they appear at the end of the program.

```
fill("blue")
rect(10, 20, 30, 40)

fill("red")
rect(20, 30, 30, 40)

fill("green")
rect(30, 40, 30, 40)
```



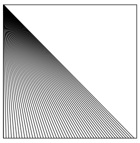
To draw text, use the `text` command. The vertical position of the text specifies the baseline. Just as with colors, the `font` and `fontsize` can be specified beforehand and will be remembered for all subsequent commands.

```
baseline = 50
fontsize(24)
text("Hello", 10, baseline)
stroke(0)
line(0, baseline, 100, baseline)
```



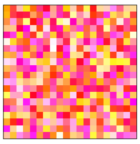
The power of NodeBox comes from repeating a set of instructions using *loops*. We use the `for` loop to instruct NodeBox to place multiple lines on the page, even though we only have a single line command:

```
stroke(0)
strokewidth(0.1)
for i in range(0, 100, 2):
    line(0, 0, i, 100)
```

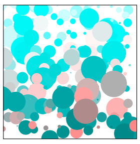


To create variation we utilize the `random` function. This instructs NodeBox to create a different result every time it runs through the loop.

```
for x, y in grid(20, 20, 5, 5):
    fill(1, random(), random())
    rect(x, y, 5, 5)
```



Random values can take the place of any other value in the system, resulting in complex compositions.



```
from math import sin, cos
grid_size = 10
v = random()
c = random()
for x, y in grid(10, 10, grid_size, grid_size):
    fill(sin(v + y * x / 100.0), cos(c), cos(c), random())
    s = random(grid_size)
    oval(x, y, s, s)
    fill(cos(v + y * x / 100.0), cos(c), cos(c), random())
    dx = random(-5, 5)
    dy = random(-5, 5)
    ds = random(-20, 20)
    oval(x + dx, y + dy,ds, ds)
    c += 0.01
```

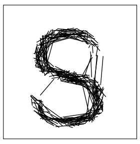
We don't have to be content with the built-in functions of the language. Instead, we can *expand* the language by creating `abstractions`. These allow us to build new language elements out of existing ones.

Here we create a "tree" primitive out of simple lines, which we use to create a forest.

```
background(0, 0.2, 0)

def tree(x, y):
    beginpath()
    moveto(x,    y)
    lineto(x-2,  y)
    lineto(x-2,  y-10)
    lineto(x-10, y-10)
    lineto(x-5,  y-20)
    lineto(x-10, y-20)
    lineto(x,    y-35)
    lineto(x+10, y-20)
    lineto(x+5,  y-20)
    lineto(x+10, y-10)
    lineto(x+2,  y-10)
    lineto(x+2,  y)
    endpath()

nofill()
for y in range(0, 120, 3):
    fill(0, random(-0.2, 0.2) + y/120.0, 0)
    tree(random(100), y)
```

The `textpath` command turns text into Bézier paths. This allows us to use text as a `template shape`, for example by using the `contains` function to see if a point falls within its bounds. In combination with our distance function we can re-create the "spider web" technique used in the ChampdAction logo (see Section 3.5.3.3).

```
def distance_squared(pt1, pt2):
    return abs(pt1.x-pt2.x) + abs(pt1.y-pt2.y)

fontsize(100)
p = textpath("S", 10, 90)

stroke(0)
for i in range(50000):
    pt1 = Point(random(100), random(100))
    pt2 = Point(random(100), random(100))
    if p.contains(pt1) and \
       p.contains(pt2) and \
       distance_squared(pt1, pt2) < 25:
        line(pt1, pt2)
```



3.4 NodeBox 1

During our studies in graphic design we were frequently presented with assignments that could benefit from a generative approach. This included repetitive work, work that would be displayed online, or interactive installations. Often we would create ad hoc solutions: custom software that was designed solely for the purpose of completing the assignment. Our approach drew the attention of fellow students who were interested in making similar work. Unfortunately the technical capabilities needed to create such solutions were out of their reach.

In 2002 Just van Rossum of LettError developed an application called DrawBot (*van Rossum*, 2002). This application used a similar visual interface than Processing but used the Python programming language instead of Java. The application ran on OS X and used the Quartz and Cocoa APIs for drawing. We saw the benefits of this integrated environment that coupled an easy-to-use programming language with a visual canvas that showed the output of your code. During my internship at LettError I worked with Just on the development of the application.

In 2004 we forked NodeBox from DrawBot. NodeBox introduced a number of breaking changes to the API, bringing it more in line with the syntax of PostScript. We replaced object-oriented wrappers around Cocoa objects (like `NSColor`) with simple functions (like `fill()`) that change the state of the underlying drawing context.

The aim of NodeBox 1 was to be a single environment for graphics programming. It includes features for working with vectors, images, adding animation and interactivity. The application produces images in high quality PDF format. The basic vocabulary of NodeBox 1 consists of less than 100 commands, which can be extended through libraries.

3.4.1 Python

NodeBox (and DrawBot) uses the Python programming language, both for our users and internally. Python is a high-level, dynamic language focused on readability and terseness. Using a high-level language means that we can reduce the amount of accidental complexity, since we don't need to worry about concepts like memory management. In addition Python has a number of other nice properties, such as significant whitespace, which means that students programs are easier to read and easier to debug for themselves as well as their teachers (G1, G4).

3.4.2 Command Language

On top of the built-in Python keywords NodeBox provides a set of graphics primitives. These are commands that draw shapes to the screen.

The commands provide a wrapper around Mac OS X's API, called Cocoa. This API is accessible from Python but requires significant effort to use. For example, here's how to draw a circle using Cocoa and PyObjC:

```
from AppKit import *
clr = NSColor.colorWithDeviceRed_green_blue_alpha_(1.0, 0.0, 0.0, 1.0)
clr.set()
path = NSBezierPath.bezierPath()
path.appendBezierPathWithOvalInRect_( ((10, 20), (50, 50)) )
path.fill()
```

For comparison, here is the same using NodeBox's drawing API:

```
fill(1.0, 0.0, 0.0, 1.0)
oval(10, 20, 50, 50)
```

Our goal in designing the API is to reduce accidental complexity by being as direct as possible and reduce surprise, while still being powerful. For example: the `rect()` command draws a rectangle, but also returns a Bézier path that can be further manipulated. The API consists of less than 100 keywords.

3.4.3 Vector Graphics

There are two conceptual modes in which you can use NodeBox. You can use NodeBox as a simple direct drawing machine, in which every command immediately produces output on screen. In addition, every drawing command also returns the Bézier curves so you can further manipulate it.

Manipulation of the paths happens using the "bezier" library, written by Dr. Tom De Smedt and Prof. Dr. Florimond De Smedt. This library can calculate arbitrary points on a Bézier curve, providing users with fine-grained control over paths. Our users rely on this feature during experimental type design workshops to produce dynamic type faces (G2) (see Figure 9).

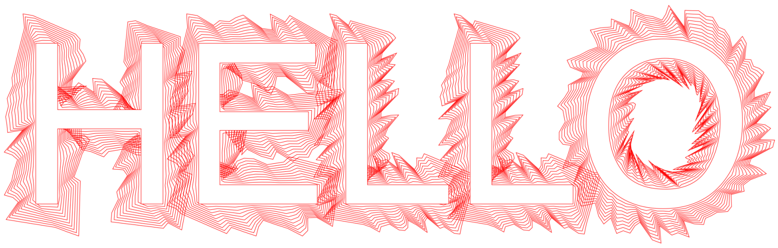


Figure 9: An example of Bézier manipulation. For each contour we calculate the surface normals, then randomly extend these to create wavy outlines.

3.4.4 State Machine

NodeBox keeps a set of drawing state around to figure out how to draw shapes or text. It has a notion of the "current" color, font, transformation, text alignment and so on. Drawing commands will take this state into account when placing their elements on the screen.

Each of these state variables has a sensible default state, such as a black fill color, a default font size of 24, etc. `push()` and `pop()` operations allow you to save and restore the current state.

3.4.5 Transformations

NodeBox uses the traditional concept of a current graphics transformation, represented as an affine transformation. This allows for the current canvas to be translated, rotated or scaled. Affine transformations operate around the origin of a Cartesian coordinate system (the top-left corner in NodeBox) which is often an unnatural way to think about transformations.

We introduced the concept of "center" transformations where the origin lies in the center of a given shape. This makes rotating and scaling shapes easier (G4). While this technique is useful, it becomes more difficult to transform groups of elements because each element will rotate around its own center. To solve this we could introduce grouping but this introduces additional complexity. For these use cases we generally advice to use traditional (what we call "corner-mode") transformations.

3.4.6 Randomness

Computers are created to execute the instructions exactly as you specify them. This predictability is a desirable property of an automated system. For artists and designers, unexpected results emerging from randomness and chance can stimulate creativity (*Freeman*, 1997) (see also Section 5.2.3). That is why we added the option to add randomness to generative designs. This randomness gives the computer some "creative liberty" to generate designs within the given bounded scope (see Figure 10).

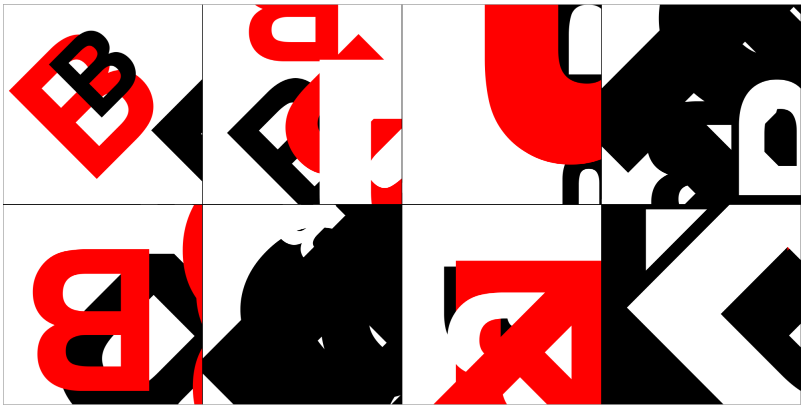


Figure 10: A number of variations of the same script (http://bit.ly/1qUhyQi)

NodeBox uses Python's `random` module, which implements a pseudo random number generator. This algorithm produces an approximation of true random numbers. More importantly, we are able to *seed* the algorithm with a given value so that it always produces the same series of random numbers. This seed is controlled in NodeBox when tweaking a value using the throttle (described below).

3.4.7 Repetition

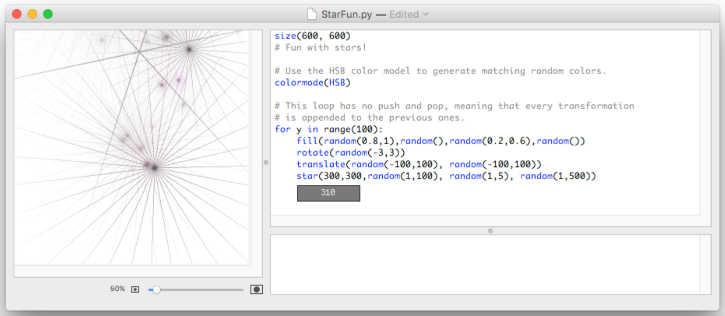
Another crucial aspect of generative design is that of the *loop*. If computers could only do what we instructed, once, there wouldn't be much need for them: we'd rather do the task by hand, faster and easier. However, once we can tell the machine to repeat a set of instructions, the speed of the machine proves beneficial.

NodeBox uses standard Python loop constructs such as `for` and `while` to repeat a block of statements. In addition we created a spatial construct called `grid` which is a two-dimensional loop that loops over a grid of X/Y positions. Each value in the loop returns a certain coordinate of this grid.

3.4.8 Tweaking

The typical workflow when developing NodeBox scripts is to make a change to the code, then to press a keyboard shortcut to run the code and view the results. This is a nuisance when using code for creative purposes where it often takes a number of tries to find the "right" number for a given parameter (where "right" depends on the artists' subjective, aesthetic feel).

We added the ability to tweak any value in the code without needing to manually re-run the code. This allowed users to interactively manipulate the values for colors, sizes, loop ranges and positions and immediately get visual feedback of their changes. A visual indicator, called the "throttle" shows the new value of the number (see Figure 11).



```
size(600, 600)
# Fun with stars!

# Use the HSB color model to generate matching random colors.
colormode(HSB)

# This loop has no push and pop, meaning that every transformation
# is appended to the previous ones.
for y in range(100):
    fill(random(0.8,1),random(),random(0.2,0.6),random())
    rotate(random(-3,3))
    translate(random(-100,100), random(-100,100))
    star(300,300,random(1,100), random(1,5), random(1,500))
```

Figure 11: The NodeBox 1 throttle, here manipulating the position of the stars.

Value tweaking made it efficient for users to explore the solution space (G1). This allowed them to try more experiments before settling on a solution (G3).

3.4.9 Print

NodeBox is built on top of the Quartz rendering layer of OS X. This allows us to export to PDF while retaining all vector information and text outlines for further modification in traditional design applications such as Adobe Illustrator (see Figure 12). This sets it apart from Processing, which is more focused on fast screen output. Due to the way Processing handles curves, they are exported in PDF as small straight lines, resulting in quality loss.

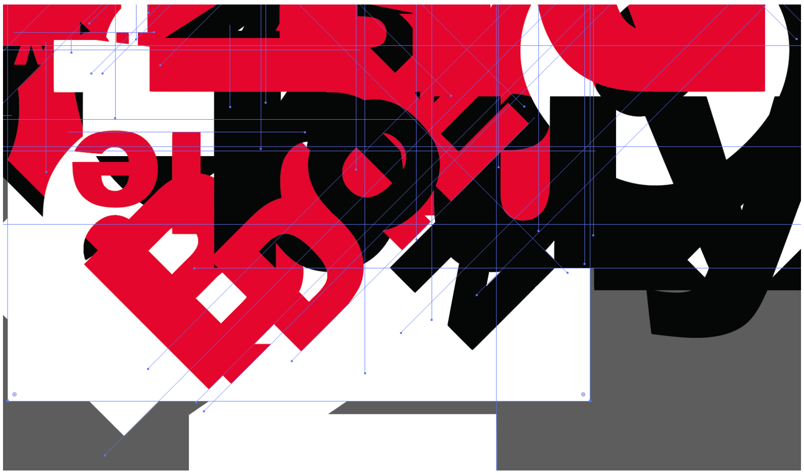


Figure 12: NodeBox project exported as PDF and opened in Illustrator. Note that all text remains modifiable.

We added support for calibrated colors, native CMYK color support, and real-world units such as centimeters and inches.

3.4.10 Libraries

The core of NodeBox adds a small number of built-in commands to the Python runtime. These commands can be extended through the use of libraries. Python has good support for working with external modules. We extended Python's import facilities to give NodeBox libraries direct access to the internal state of the application through a command called `ximport`.

The NodeBox community built a large number of libraries. We'd like to acknowledge the work of Dr. Tom De Smedt who developed a large number of libraries for language parsing, pixel manipulation, emergent systems and color processing. Some of these libraries form the basis of the Python Pattern library (*De Smedt*, 2012).

Libraries provide users with well-tested code in areas that are outside of the scope of the original program. The Core Image library, for example, allows users to create real-time image manipulation using a simple API. This has been used to create the "Evolution" project (Figure 13).

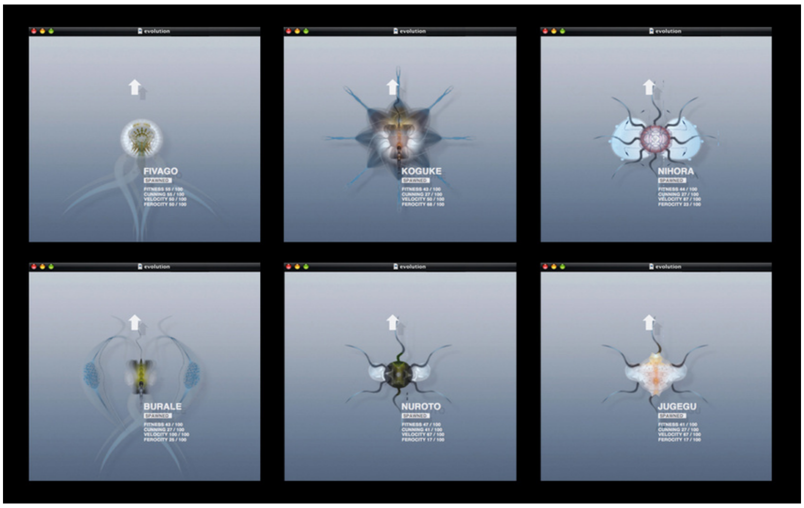
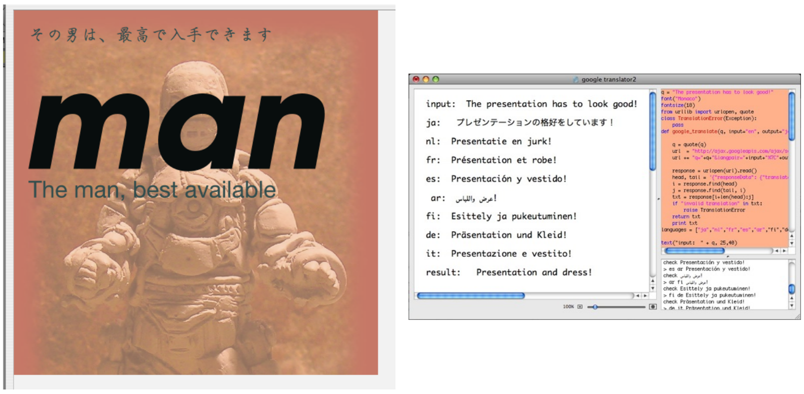


Figure 13: "Evolution", Ludivine Lechat, Tom De Smedt and Frederik De Bleser, 2007

3.4.11 Data Visualization

Simple examples in NodeBox start out with loops around predefined variables, e.g. repeat this block 25 times. These projects can look very compelling while still being based on purely aesthetic preferences. Often however, we want to build projects around external data. This data can come from databases (e.g. to make a product catalog), web pages (e.g. through parsing), images (e.g. to calculate color distribution) or even sound or other external interfaces.

An example of data visualization using external data was a project by Hannu Aarniala created in 2009 that used Google Translate to repeatedly translate a sentence from English, over to Japanese, Dutch, French, Arabic and so forth before translating back to English. This transformed the text, often distorted its original meaning. For example the sentence "The best a man can get" (the slogan for Gillette) was transformed into "The man, best available" (see Figure 14). In this project lists were used to represent the sequence of languages to translate.



```
input:  The presentation has to look good!
ja:   プレゼンテーションの格好をしています！
nl:  Presentatie en jurk!
fr:  Présentation et robe!
es:  Presentación y vestido!
ar:  عرض والملابس!
fi:  Esittely ja pukeutuminen!
de:  Präsentation und Kleid!
it:  Presentazione e vestito!
result:   Presentation and dress!
```

```
q = "The presentation has to look good!"
font("Monaco")
fontsize(18)
from urllib import urlopen, quote
class TranslationError(Exception):
    pass
def google_translate(q, input="en", output="j

    q = quote(q)
    url  = "http://ajax.googleapis.com/ajax/s
    url += "q="+q+"&langpair="+input+"%7C"+ou

    response = urlopen(url).read()
    head, tail = '{"responseData": {"translat
    i = response.find(head)
    j = response.find(tail, i)
    txt = response[i+len(head):j]
    if "invalid translation" in txt:
        raise TranslationError
    return txt
    print txt
languages = ["ja","nl","fr","es","ar","fi","d

text("input:  " + q, 25,40)
```

```
check Presentación y vestido!
> es ar Presentación y vestido!
check عرض والملابس!
> ar fi عرض والملابس!
check Esittely ja pukeutuminen!
> fi de Esittely ja pukeutuminen!
check Präsentation und Kleid!
> de it Präsentation und Kleid!
```

Figure 14: "The Man, Best Available", Hannu Aarniala, 2009

3.5 Visual Results

To show the validity of this approach we show a selection of works created by students during workshops, commercial work and our own work.

3.5.1 Workshops

Here we show the results of work produced with NodeBox 1 during workshops. Students were introduced in the software and the basics of programming, and received guidance and tutoring. Results were presented at the end of the week.

Evil Font

This project was created in the 2007 Lahti workshop by Valtteri Vitakoski. The concept was to created a parametric font that had an "evilness" slider. Dragging the slider would make the font look more "evil" (Figure 15).

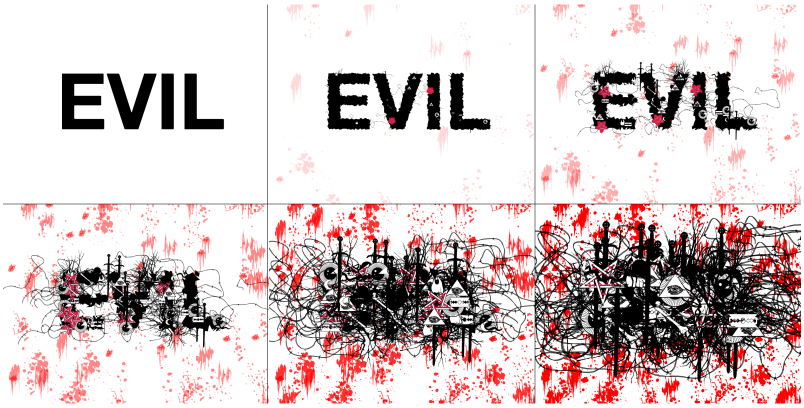


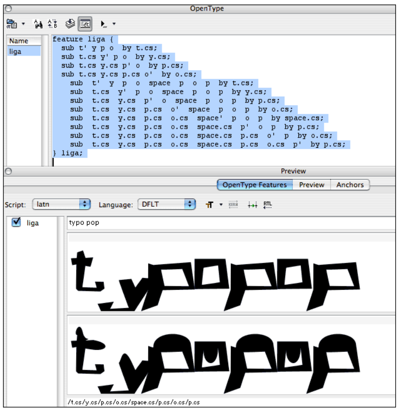
Figure 15: "Evil Font", Valtteri Vitakoski, 2007

The script itself is less than 100 lines of NodeBox 1 code. It draws a piece of text, then randomly places image elements out of a predefined folder structure onto the type. The amount of evilness defines how many images are placed.

This project shows the benefits of a *hybrid* generative approach. We call this a hybrid approach because not everything is generated by a code. The code chooses from a library with predefines shapes and images. By extending the library with more images, we can create more variation in the output. The programmatic approach helps to place the images at the correct locations, a procedure that would be tiresome to do by hand for multiple pieces of text.

Comic Sans Meme

The project designed a typeface that would change appearance based on the content of the text. Specifically, it would replace typical "business-like" words (like "paradigm" or "innovative") with Comic Sans (Figure 16). It was created during the 2007 Lahti workshop by Per-Oskar Grönroos.



```
feature liga {
  sub t' y p o  by t.cs;
  sub t.cs y' p o  by y.cs;
  sub t.cs y.cs p' o  by p.cs;
  sub t.cs y.cs p.cs o'  by o.cs;
    sub  t'  y  p  o  space  p  o  p  by t.cs;
    sub  t.cs  y'  p  o  space  p  o  p  by y.cs;
    sub  t.cs  y.cs  p'  o  space  p  o  p  by p.cs;
    sub  t.cs  y.cs  p.cs  o'  space  p  o  p  by o.cs;
    sub  t.cs  y.cs  p.cs  o.cs  space'  p  o  p  by space.cs;
    sub  t.cs  y.cs  p.cs  o.cs  space.cs  p'  o  p  by p.cs;
    sub  t.cs  y.cs  p.cs  o.cs  space.cs  p.cs  o'  p  by o.cs;
    sub  t.cs  y.cs  p.cs  o.cs  space.cs  p.cs  o.cs  p'  by p.cs;
} liga;
```

Script: latn
Language: DFLT
[✓ liga]
typo pop
/t.cs/y.cs/p.cs/o.cs/space.cs/p.cs/o.cs/p.cs

This project explores highly innovative typographic design. We have formed a vortex of creative collaboration and are currently researching the tactical realities of the marketplace. We are soon coming to a critical conceptual design phase where we tackle the brief with a full-scale design engagement.

Figure 16: "Comic Sans Meme", Per-Oskar Grönroos, 2007

To create a working font, Per-Oskar "abused" the glyph substitution features of OpenType. This feature is often used to replace multiple letters with a ligature (a combined letter shape). NodeBox 1 helped to generate the hundreds of ligatures rules needed to replace words such as "paradigm" with their Comic Sans letter shapes (a task that would be unfeasible to do by hand).

Organic Dots

This project was created by Veronika Schmidt during the 2008 workshop at Aalto University, Helsinki. The concept was to visualize a form of modern ornamentation taken to the extreme. For that Veronika combined organic forms in endless multiplications. This method required a digital method to combine manually created elements into a finished poster.

Veronika combined a number of elements such as leaves and dots using NodeBox into more complex shapes. These shapes were combined in Adobe Illustrator to produce the finished result (Figures 17 and 18).

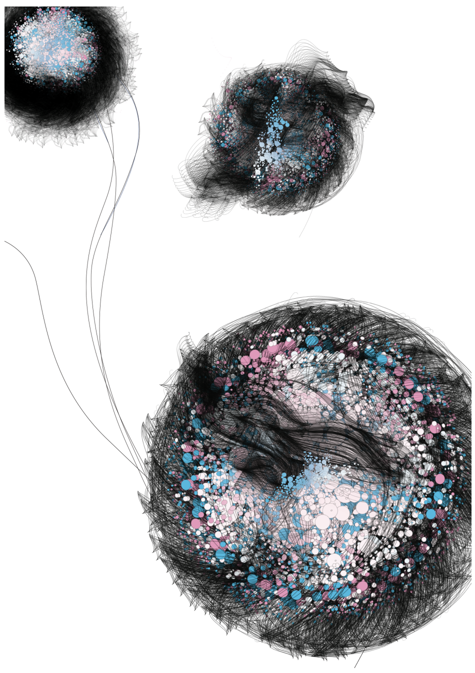


Figure 17: "Organic Dots", Veronika Schmidt, 2008

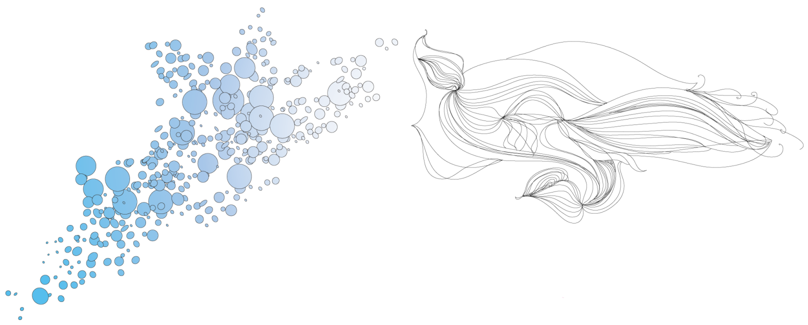


Figure 18: The individual elements that were combined to produce the final work

Oscillating Lettershapes

The goal of the "Oscillating Lettershapes" project was to define a fully generative typeface (Figure 19).

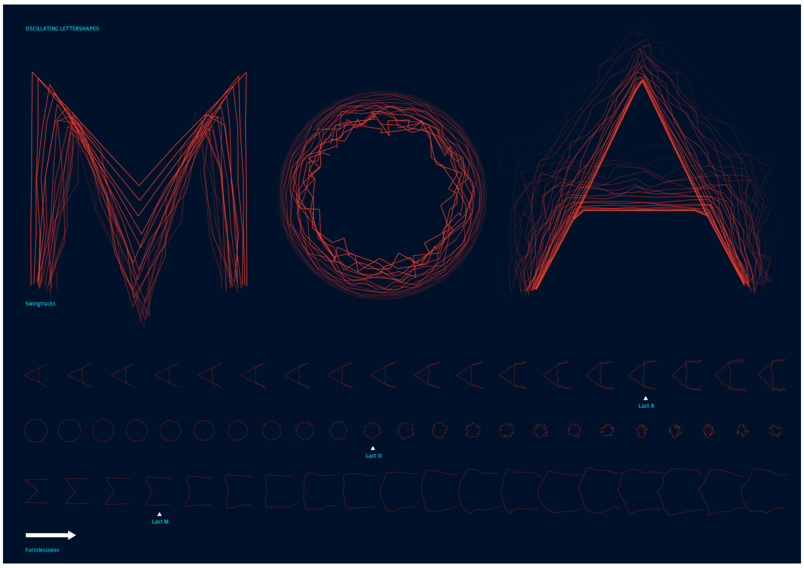


Figure 19: "Oscillating Lettershapes", Jonatan Hildén, 2008

For this Jonatan Hildén defined each letter as a set of programmatic instructions. For example here are the instructions for the letter "M":

```
beginpath(a1x+N1/2,a1y)
lineto(b1x+N2, b1y+N2)
lineto(cx, cy+N2*3)
lineto(b2x-N2, b2y+N2)
lineto(a2x-N1/2, a2y)
```

Since the shape was defined programmatically it could be modified at will. Jonathan designed a number of filters (like `rotate`, `sine` and even `anger`) that would modify the shape with a certain factor.

This project demonstrates a fully data-driven approach, where the line between what is data and what is code (like the shapes of the letters) is blurred.

3.5.2 Commercial work

NodeBox has been used successfully in a number of commercial projects. Here we list some projects created by agencies or individuals outside of the research group.

Spamghetto

Spamghetto (*Olivero*, 2010) was created by the TODO creative agency located in Turin, Italy. The project was a reaction on the flood of spam emails filling our inboxes. Instead of seeing spam as something to be deleted and removed, the designers harvested all the spam to create a generative wallpaper using NodeBox (Figure 20).

Spamghetto was designed by Giorgio Olivero, Fabio Cionini, Fabio Franchino, Andrea Clemente, Andrea Pinchi, Riccardo Mongelluzzo, Elena Fonti and Carlo Syed. In 2011 Spamghetto received the Silver award in European Design.

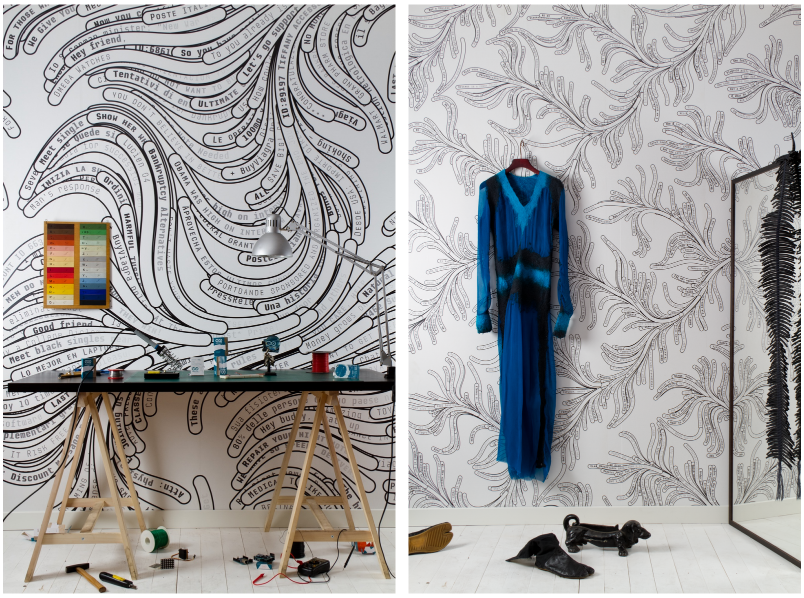


Figure 20: "Spamghetto", TODO, 2010

Lunar Calendar

Produced by Dimitre Lima, the Lunar Calendar is an ongoing visualization showing the phases of the moon during a year. The project was awarded Bronze in the Data Visualization "Information is Beautiful" Awards in 2012 (Figure 21).

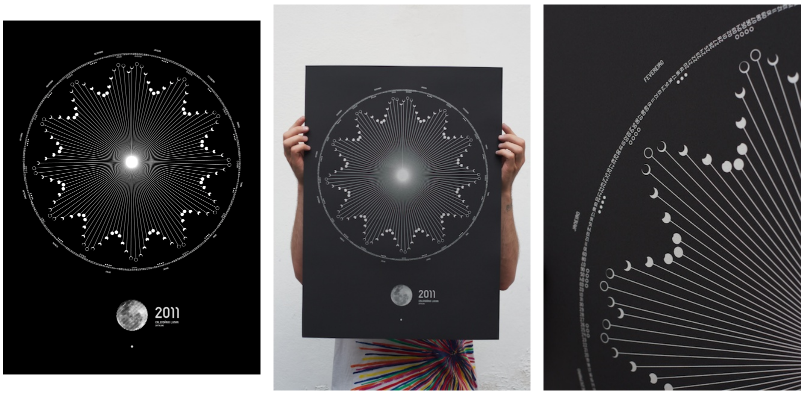
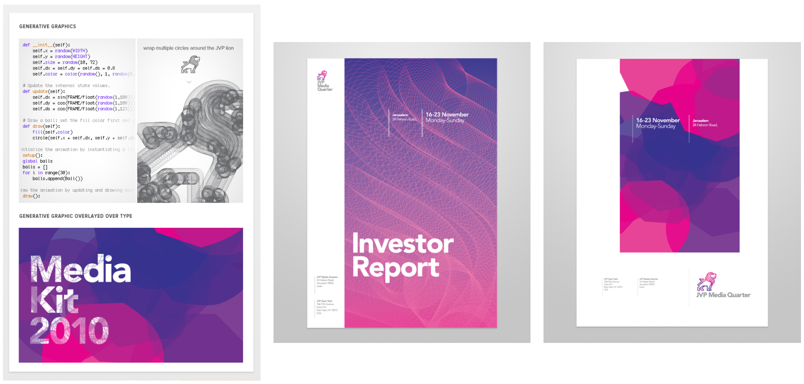


Figure 21: "Lunar Calendar", Dimitre Lima, 2011

The project is ongoing: the artist is still creating new calendars every year. Once the algorithm is done, it doesn't take much effort to generate a poster for a new year.

JVP Brand Identity

JVP is a Israeli venture capital firm. Design agency Designit were asked to re-think the brand identity but retain the existing logo mark. Designer Guy Haviv used NodeBox to create a series of generative graphics that used the original logo as a base, producing generative shapes along the curves of the logo (Figure 22).


GENERATIVE GRAPHICS
wrap multiple circles around the JVP lion
GENERATIVE GRAPHIC OVERLAYED OVER TYPE
Media Kit 2010
16-23 November Monday-Sunday
Investor Report
JVP Media Quarter

```
def __init__(self):
    self.x = random(WIDTH)
    self.y = random(HEIGHT)
    self.size = random(10, 72)
    self.dx = self.dy = self.ds = 0.0
    self.color = color(random(), 1, random(0,

# Update the internal state values.
def update(self):
    self.dx = sin(FRAME/float(random(1,100)))
    self.dy = cos(FRAME/float(random(1,100)))
    self.ds = cos(FRAME/float(random(1,123)))

# Draw a ball: set the fill color first and d
def draw(self):
    fill(self.color)
    circle(self.x + self.dx, self.y + self.dy

nitialize the animation by instantiating a lis
setup():
global balls
balls = []
for i in range(30):
    balls.append(Ball())

raw the animation by updating and drawing each
draw():
```

Figure 22: "JVP Brand Identity", Guy Haviv, 2010

3.5.3 Own Work

As described in the introduction, NodeBox 1 was created as a solution for design issues we had, both in school and commercially. We used NodeBox 1 ourselves on a couple of commercial and personal projects.

Trunk and Co

In 2009 we designed and produced the catalog for Samsonite's youth brand, called Trunk & Co. The brief was to create a catalog of all available products in 5 different languages.

We stored all the data for the catalog in a XML file. This file contained the different products and included the translations for all the text. We used NodeBox to generate PDF files for all of the different languages (Figure 23).

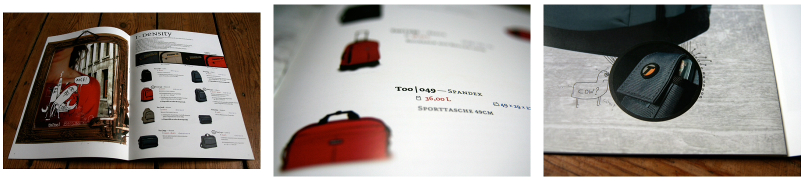


Figure 23: "Trunk and Co Catalog", Frederik De Bleser and Tom De Smedt, 2009

By using NodeBox we could automate the repetitive work of creating multiple versions of the catalog (G1). In addition, because the output of the software did not require manual post-processing, we could keep tweaking the design until the last minute, enabling us to try out more design variations (G3).

Ottobot

Ottobot is a program that creates NodeBox programs. It was initially constructed in 2009 as a grammar file for the Kant Generator Pro, a program written by Mark Pilgrim as part of his "Dive Into Python" book (*Pilgrim*, 2000) that generates text that superficially resembled the philosophical work of Immanuel Kant.

Kant Generator Pro takes its input from a grammar file that contains multiple phrase sets. Each phrase in the phrase set contains references to other phrase sets. By randomly choosing a phrase from each phrase set, recursively, the system can generate entire sentences, paragraphs or pages.

While experimenting with this software we realized it could generate any kind of language, as long as it followed a strict grammar. Therefore, the computer language we used for NodeBox could also be captured in a grammar file. In other words, we could describe the syntax of NodeBox programs and have the system generate a syntactically valid program that we could then execute by NodeBox. Figure 24 shows some typical output of the algorithm.

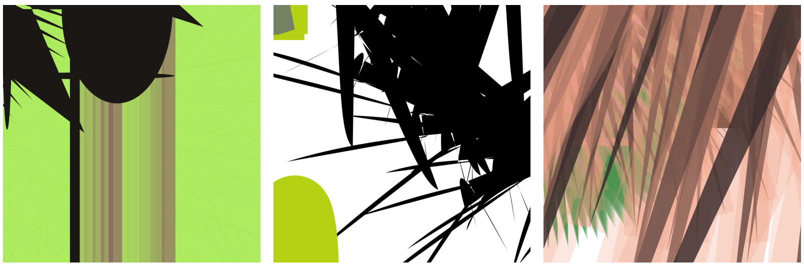


Figure 24: Typical output of the Ottobot algorithm

Sporadically the algorithm generated output that surprised even us, creating results that were completely unique (see Figure 25).

Because Ottobot generates executable NodeBox programs it can generate programs that itself contain randomness (through NodeBox's `random` function). This extra step of indirection means that one program generated by Ottobot can produce many different variations.

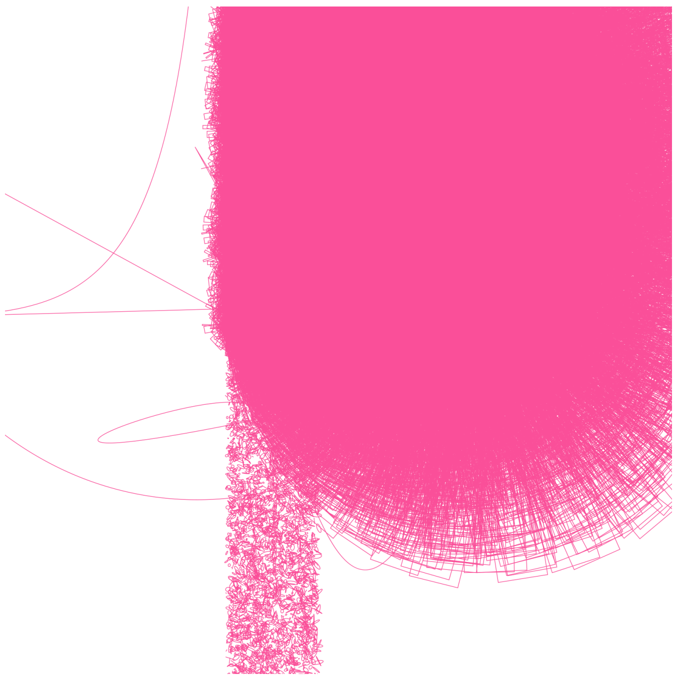


Figure 25: "Ottobot", Frederik De Bleser, 2009

ChampdAction Logo

We designed the logo for ChampdAction, an experimental music ensemble. The logo represented the ever changing nature of the ensemble and their often eccentric approach.

The logo picks random points within the contours of an existing typeface. If these fall within a certain distance of each other, they are connected; otherwise they are discarded. By manipulating the maximum connection distance we can manipulate the readability of the logotype (see Figure 26).

We installed about 40 versions of the logo on the ChampdAction website, with different gradations of "wildness". When users clicked around on the site, gradually more and more distorted versions of the logo were displayed. The logo and website were developed during a one week intensive workshop were a team of designers created the brochure as well.

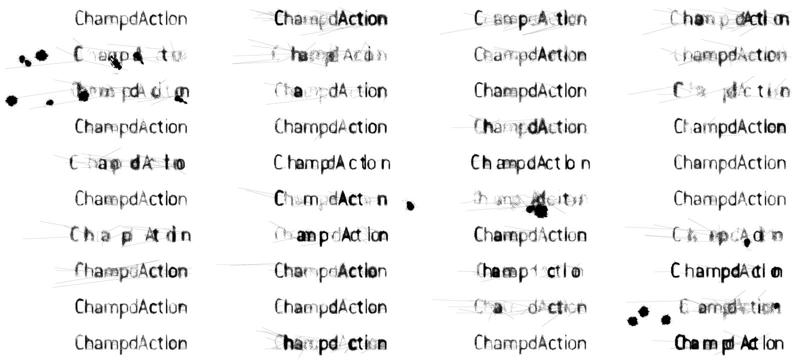


Figure 26: "ChampdAction Logo", Frederik De Bleser, 2007

Word Maze

Word Maze was a project designed around the theme of concealment and discovery. It consists of three movies each containing a word that is initially invisible and becoming more and more clear, although never at once. The words are generated using a pixelated font generator that is combined with a modified recursive backtracking algorithm to produce a maze that is gradually built up and torn down (Figure 27).

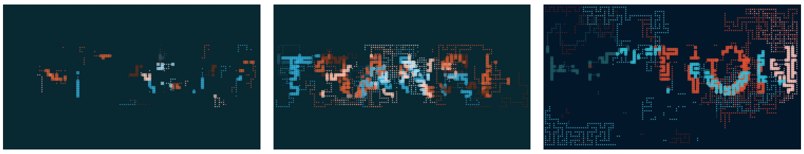


Figure 27: "Word Maze", Frederik De Bleser, 2011

Word Maze was exhibited at the "Travelling Letters" exhibition in 2011 in Vilnius, Lithuania.

3.6 Evaluation

To evaluate our hypothesis and find out if users are more efficient, more exploratory, more creative and self-reliant, we will look at observations made during workshops held with graphic design students from 2004 until 2009.

3.6.1 Workshop Format

We prefer to teach NodeBox 1 to new students during a one- or two-week workshop. During this time they learn to use the software to create a poster or interactive prototype.

We found a one- or two-week workshop to be the best way to teach new generative software. This concentrated period isolates students from distractions and being together with other enthusiastic students allows them to accomplish things that would otherwise take months (*Nijs*, 2007). We have occasionally given the course over a semester, but this requires a lot of repetition at the beginning of each lecture as students tend to forget what they learned in the previous weeks.

To encourage interaction and feedback we plan daily presentations in the evening so students can show their work in progress, see what others have created and provide feedback to each other. We suggest students work in small groups, which we found improves overall results. We speculate this happens because they can bounce ideas off each other and help each other when they get stuck (instead of waiting for the teacher).

The intensity of the workshop has limitations as well (*Sork*, 1984). Because we need to present a lot of material in a short time period, it is difficult to take corrective action when a learner falls behind. We mitigate this issue by trying to have multiple teachers for each workshop so that one can teach while the other helps students that have momentarily fallen behind. The deluge of new material can result in fatigue or information overload. We've learned from experience to detect this general mood and stop introducing new material. There is only so much you can teach during a workshop so we focus on the aspects of the software that are necessary to complete the assignment.

We've organized NodeBox workshops since 2004 in Belgium, Finland, Lithuania, Poland, Italy, Canada and France.

3.6.2 Setting Expectations

A challenge with the workshop format was setting expectations for the students. Since the tools are completely new for them, they can't judge what they will be able to make for themselves in the allotted time. This reduces their capability of "dreaming big".

One key insight is that we now do all the teaching up-front, often in the first two days, instead of teaching in the mornings and letting the students work in the afternoon, which is what we did before. This way, students have a good understanding of what is possible using the tool, and can set their expectations accordingly.

An unexpected side effect of this is that students would often come up with ideas that closely mimicked the examples shown in the introduction.When questioned about this, students were often unaware that they did this, citing that one particular example looked cool or is what they wanted to do. We found that this behavior mimics stage two of the *Dreyfus* model (*Dreyfus*, 1980): students have just encountered the software and are asked to follow a new set of rules exactly (otherwise the program punishes them with syntax errors). They do not yet have a holistic view of the software and thus will tend to think within the bounds of what they have just learned.

We mitigate this issue by also showing work made in previous workshops. Often this work will be based on similar initial ideas and will create the impression that "their" idea has already been done. While this can feel temporarily discouraging it forces students to go beyond their very first ideas, think beyond the capabilities of the software and focus on a specific project they want to execute.

3.6.3 Language Syntax

Students struggle a lot with the level of preciseness that the language expects from them. This is often clear in simple mistakes such as forgetting to close a brace. This is a minor issue for trained programmers, but a cause of major frustration for designers. It can cause them to be stuck for several minutes or even give up entirely.

As an example, a frequently occurring bug in Python is forgetting the ending column after a `for` statement:

```
for i in range(10) # Note the missing ":" here
    print i
```

Syntax is an example of *accidental complexity*. It bears no relation to the problem that needs to be solved, but is merely a way of expressing the steps the computer has to take to solve the problem. In other words, programming code is not an intuitive interface for artists. Instead, artists should be able to enter into a conversation with the computer through natural language, much like a chat session (*Winograd*, 1971).

To solve this issue we initiated the Gravital project in 2007. Its aim was to create a solver that could take in a graphic design problem in natural language, convert it to an intermediate representation that could be edited, and output a visual result. This research project ran for two years and resulted in the development of a number of natural-language processing modules and a visual-node based version of NodeBox, described in the next chapter. This work also resulted in a peer-reviewed publication that was included in the book "The Agile Mind" (*De Smedt*, 2013).

3.6.4 Error messages

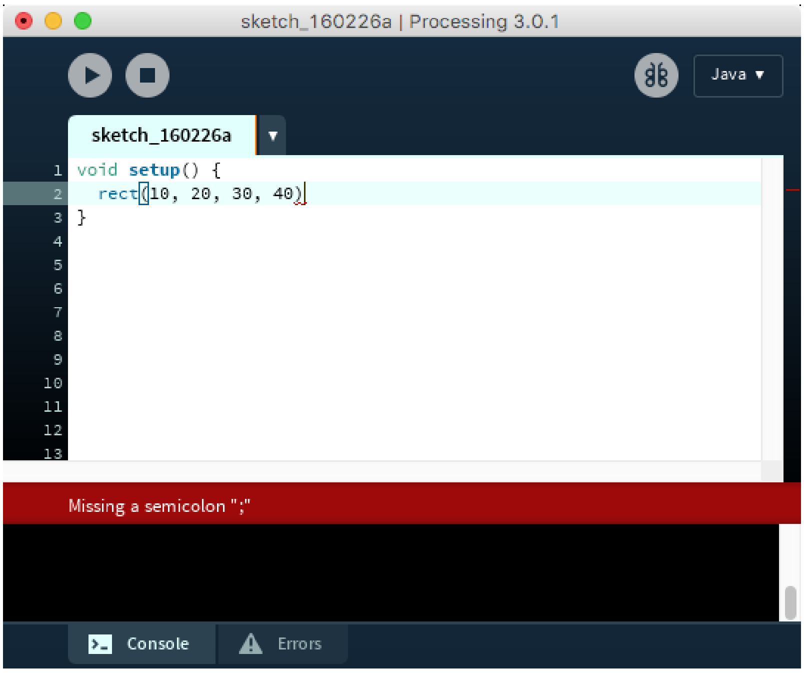
Whenever the program encounters an error developers rely on error messages to help debug the issue. We found that the design of error messages is a crucial step in helping students with their issue. This finding corroborates available research (*Brown*, 1983).

One major issue is that users often ignore errors (*Bargas-Avila*, 2007). In workshop environments, we observed that when an error shows up, students will search for the quickest way to get rid of the error without reading it. We speculate this comes from years of conditioning that errors in applications relate to issues outside the users' control, and the best solution is to try again later. This is in line with *Dreyfus*'s finding that novices just want to accomplish an immediate goal, ignoring everything that doesn't lead to task completion.

To mitigate this effect we deliberately introduced errors in our programs when training the students. We showed them how to read the errors by looking for the line number. By examining the code on that line (and sometimes some of the surrounding lines), they could often find simple syntax errors themselves.

Python errors often contain a stack trace, a listing of the execution frames. We found this output not very useful in NodeBox. The programs themselves were not very long and often contained only a single level of nesting. In addition, the stack trace contained information about the execution of NodeBox itself which made the errors unnecessary convoluted.

We argue that error messages, taken directly from a computer programming language like Python, are unusable for beginners and students. We suggest that programming environments for designers should wrap the errors in a more usable format, focussing on the line number and a more readable error explanation. The Processing 3 environment (described in Section 3.2.4) is a good example of error handling: it underlines the erroneous lines in red and simplifies error messages to a single human-readable sentence (see Figure 28).


sketch_160226a | Processing 3.0.1

```
void setup() {
  rect(10, 20, 30, 40)
}
```

Missing a semicolon ";"

Figure 28: Example of Processing 3 error message.

In conclusion, errors are a necessary evil in programming. To demonstrate this, one student created a project during a workshop where she created a book of over 700 pages showing all the error messages she encountered during the making of the book itself (Figure 29).

Figure 29: "NodeBox Errors", Elina, 2004

3.6.5 Invisible state / Statement ordering

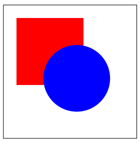
NodeBox 1 is designed as a state machine. Internally it contains information about the current fill and stroke color, the current transform, the current font name and size, and more. All of theses state variables will impact drawing operations. When drawing a rectangle NodeBox will use the current fill and stroke information to color the object, as well as the current transformation matrix to place the object. This approach mirrors similar drawing APIs such as PostScript, OpenGL or Processing.

Although the state plays a crucial role in the operation of NodeBox, it is not visualized throughout the program. There is no way to know what color will be used to fill a rectangle apart from running the program. With each drawing command, the user has to make the mental exercise of figuring out all of this state to predict the effect of their drawing operation. This implies that, to debug programs, users will need to replay state transitions in their head. Being able to visualize the state of the program in your head appears to be a core competency of skilled programmers (*Graham*, 2007). The better you are in reasoning about the current state of the program, the faster you can detect bugs.

For novice programmers this often caused confusion when re-ordering code to change the visual ordering of elements on the screen. Because the drawing operations themselves depend on the state, re-ordering the code would often break the drawing code.

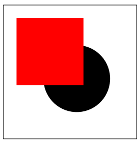
Here's a simple example:

```
# Draw a red rectangle
fill("red")
rect(10, 10, 50, 50)

# Draw a blue ellipse
fill("blue")
oval(30, 30, 50, 50)
```



In this example, because the oval statement appears *after* the rectangle, the oval is drawn on top of the rectangle. If we want the oval to appear below, we would need to place the statement before. However, if we forget to also move the associated fill statement, the oval would be displayed in black (the default starting color):

```
oval(30, 30, 50, 50)

fill("red")
rect(10, 10, 50, 50)

fill("blue") # Has no effect
```



Note that the final fill statement sets the color, but nothing gets drawn afterwards.

Although the issue is quite clear in this simple example, the problem was exacerbated when using functions. Functions that would draw shapes would need to set the internal state, affecting later operations. This meant that most functions weren't "pure", i.e. they had side effects. This problem permeates all environments that depend on global state (*Victor*, 2012).

This is another example of *accidental complexity*. The fact that state changes need to be carefully coordinated together with drawing statements has no effect on the problem to be solved. We will see in Section 4.1.1 that functional programming can help with getting rid of this complexity.

3.7 Lessons Learned

3.7.1 Function casing

In programming, the space character has special meaning. It is used to separate multiple statements. This means the space can't be used to separate multi-word function names, ie. we can't write "image size" as a function.

The reason we want to separate out the words is to avoid multiple interpretations of a name. There are a two common solutions to solving this issue: we can use a different character to represent the space, such as an underscore (e.g. `image_size`) or we can "camel-case" the word, which means we use upper-case letters in the middle of the name to denote a word break (e.g. `imageSize`). Both systems are used pervasively in programming. In fact, Python uses both schemes, due to legacy reasons, in different parts of their built-in API.

Although research found camel-cased identifiers to be more readable (*Binkley*, 2009) we've opted to use a third scheme, which is not to make a word distinction at all. (e.g. the "image size" command is written as `imagesize`) Because our list of keywords is small and we can easily choose a different combination if there is a potential for confusion, we found that multi-word separators introduce an unnecessary extra level of complexity.

3.7.2 Variable shadowing

The API of NodeBox is intentionally simple. One way we developed this is "a rect is just a rect". In other words, if you want to draw a rectangle, the code to write this should look like this:

```
rect(10, 20, 30, 40)
```

Because the functions are not name-spaced (e.g. we don't use `nodebox.rect` or `nbRect`), our API effectively introduces new keywords for the Python language. This set of keywords has expanded over time; right now there are about 50 functions that NodeBox predefines. Those keywords, just as any other keywords in the language, can cause issues with variable names. Because Python doesn't enforce typing, this can cause bugs.

We occasionally encountered scripts that used the identifier `size` as a variable name, despite `size` being a NodeBox command.This caused scripts to fail with often cryptic errors (e.g. `'int' object is not callable`).

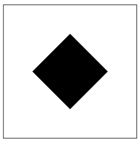
Although we can't change this behavior without fundamentally changing the way Python works, we mitigated this issue by using syntax coloring to distinguish built-in commands from custom identifiers. In other words, if a word is blue, it is a command and shouldn't be used as a variable name.

3.7.3 Transformations

To express rotation, translation and scale, NodeBox uses affine transformations, represented as a 3 by 3 matrix. Users don't have to know the underlying representation of the affine transformation, but they have to be aware of the ordering of the transformations. E.g. a translate operation followed by a rotate operation doesn't have the same effect as a rotate operation followed by a translate operation.

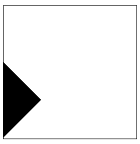
To illustrate, here we first do the translate, then the rotate:

```
translate(50, 50)
rotate(45)
rect(-20, -20, 40, 40)
```



Here we switch the rotate and translate operations:

```
rotate(45)
translate(50, 50)
rect(-20, -20, 40, 40)
```



Another issue is the origin point of the transformation. By default, this origin point lies at the origin of the canvas, i.e. the top left. This implies that when doing rotations, everything rotates around the top-left corner. This is counter-intuitive and against the expectations of our users (G4).

To mitigate this we've created an additional, "center" transformation mode, which uses the center of the object's bounding box as the origin of transformations. Now when rotating a shape, the shape pivots arounds its own center. This proved to be much more intuitive and we've made it the default setting. It still causes issues with objects that are visually grouped, since they all rotate around their own origin point. In those cases, we instructed students to use the "corner" transformation mode. Another solution would be to provide a way to group objects together and rotate around the group center. This is something we've integrated into newer versions of NodeBox.

3.7.4 Implementing Value Tweaking

To implement value tweaking (described in Section 3.4.8) we converted the position of the mouse to a text position in the code panel. This gave us the character under the cursor. We expanded from that character to include digits, periods and minus signs on both sides of the number. Once we found the number, we replaced it with a "magic variable" whose value we change whenever the user drags the mouse. When the user releases the mouse, the old number is replaced by the new value of the magic variable.

While this approach worked well in most cases, it broke down when the sign of the number changes from positive to negative or vice versa. Specifically, we encountered three distinct cases:

- If an addition goes from positive to negative the sign should change:

```
random(alpha+8) -> random(alpha-8)
```

- If a subtraction goes from negative to positive the sign should change:

```
random(alpha-8) -> random(alpha+8)
```

- *But* if a negative number turns positive, the sign should vanish:

```
random(-8) -> random(8)
```

To find the difference between the first and last case, we use the Python parser to build an abstract syntax tree (AST). Within the AST, these use cases are represented by three different classes: `Addition` (binary addition), `Sub` (binary subtraction) and `UnarySub` (unary subtraction, or a negative number). We find the location of our magic variable in the AST and apply the correct replacement when inserting the new value.

When using the throttle with random numbers, the values change for every run, causing the output to flicker. We solved this by re-using the same random seed for each run when dragging the throttle.

3.7.5 Data structures

When teaching NodeBox we began by introducing the language, then explaining the built-in drawing commands. After that we would explain variables, conditions and loops. However it took a while to understand that to use the program effectively, users also needed an introduction into data structures. More so than loops or conditions, data structures are the core expressive tool of the language.

This was an insight that took some time for us to develop, especially because data structures are so ingrained in a programmer's brain that it doesn't really make sense to think you would teach it to beginning programmers. In fact, all programming books we've encountered start with teaching the basic concepts like statements, variables, conditions and loops. While these are important, students can only start to express their problem in terms of a solution once being taught data structures.

We found that data structures (and specifically lists) are the key building material for developing interactive projects. They provide a language construct that bridges the students' idea and the machine. The letters in a word are a list, the points in a Bézier path are a list, search results return a list of websites, the pixels in an image are a list, and so on. By teaching this concept early on, students would be able to turn their ideas into a conceptual framework for talking to the computer, even though they might still struggle with the syntax. This key insight is what led to the development of NodeBox 2 and 3.

Through most of the course, we only taught lists as the basic data structure. We found Python's support and "syntactic sugar" for creating and manipulating very useful. We only taught associative arrays ("dictionaries" in Python) to a few students; we found the Python syntax for them overly ceremonial and often resorted to using objects (which carry their own set of issues). We almost never used classes or other object-oriented features of the language, except for using them as a simple data holder. We never used inheritance and only occasionally used member functions.

3.8 Conclusion

In this chapter we demonstrated that a generative, imperative approach can produce results that we can't easily produce using traditional software applications such as Adobe Photoshop. Through features like the NodeBox throttle, users can rapidly explore the boundaries of the algorithms, leading to a broader approach. We've also seen that users create entirely new aesthetics through the use of algorithms (such as the Spamghetto example). We discussed a number of ways that caused users to get stuck and made suggestions on how to mitigate this (like improving error messages).

Overall we found teaching programming to graphic designers challenging for two reasons. First off, students had difficulties expressing their problems into a "plan" (*Spohrer*, 1986), which is an important cause of failure to program (*Shackelford*, 1993). However, plan composition is an essential skill when designing a computer program. Secondly, they needed to convert that plan into a rigid, formal syntax. There they would run up against syntax errors they were unable to solve themselves. This barrier is arbitrary: language syntax is unrelated to the essential problem to be solved (the design itself). We also noticed a shift of interest towards data visualization which required more data processing operations based on lists.

The combination of these factors led to the research in functional programming and the creation of NodeBox 2.

4 Functional Programming

In this chapter we will describe how we used functional programming concepts and dataflow to build applications for generative design and data visualization. We describe NodeBox 2, NodeBox 3 and NodeBox Live, visual programming tools that use the box-and-wire paradigm and a dataflow execution model. We also briefly touch on DataPipe, a non-visual dataflow application for pre-processing large data sets. We speculate that a visual programming would be a better fit for visually oriented people such as graphic designers. In Section 4.7 we discuss how we evaluated this approach.

4.1 Background / Related Work

Here we will highlight programming paradigms and software applications that were important in inspiring the design of our software.

4.1.1 Functional Programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions, and avoids changing state and mutable data. It is rooted in lambda calculus, a formal system of expressing mathematical logic based on function application introduced by Alonzo Church (*Church*, 1951). As an example take a function that checks whether a string is blank (empty or solely consisting of whitespace). In an imperative programming language (here Java) it would look like this:

```
public static boolean isBlank(String str) {
    if (str.length() == 0) {
        return false;
    }
    for (int i = 0; i < str.length(); i++) {
        if (!Character.isWhitespace(str.charAt(i))) {
            return false;
        }
    }
    return true;
}
```

Here is a similar implementation in Clojure, a functional programming language:

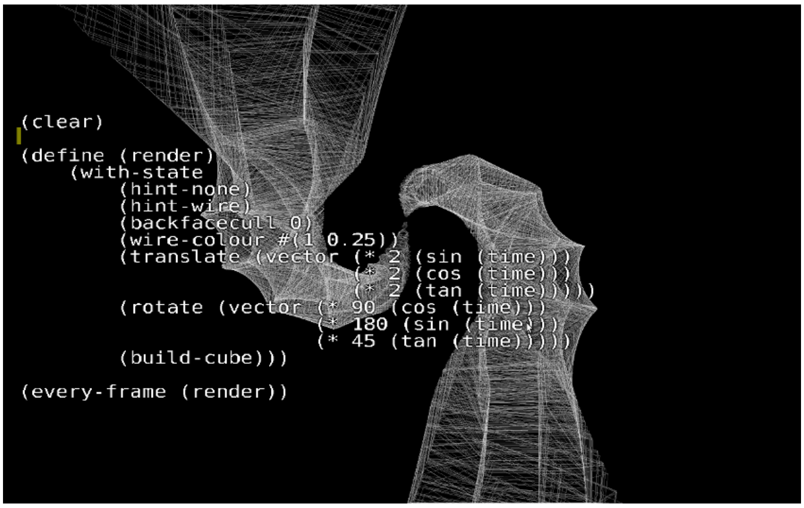
```
(defn blank? [str]
    (every? #(Character/isWhitespace %) str))
```

Although this example seems unrelated to generative design it highlights a number of important distinctions. The functional implementation has no (invisible) state, avoiding a typical error scenario we encountered when observing students working with NodeBox 1. In addition there are no off-by-one scenarios, no boundary conditions and no multiple paths of execution, which are all typical error scenarios for novice programmers (*Johnson*, 1984). We will see how we can use this functional approach to reduce the complexity of visual programming in NodeBox 2, 3 and Live.

Lisp / Clojure

Lisp is the second-oldest high-level programming language in widespread use today. Lisp derives its name from "LISt Processor", using linked lists as the ubiquitous data structures both for working with data and to compose programs out of. Lisp source code is itself made up of lists, allowing Lisp programs to generate and manipulate other Lisp programs through a mechanism called macros.

Fluxus (*Griffiths*, 2005) is a live coding environment for 3D graphics, music and games written in a Lisp dialect called Racket. Fluxus leverages Lisp's read-eval-print-loop to build a live coding environment (Figure 30).



```
(clear)

(define (render)
    (with-state
        (hint-none)
        (hint-wire)
        (backfacecull 0)
        (wire-colour #(1 0.25))
        (translate (vector (* 2 (sin (time)))
                           (* 2 (cos (time)))
                           (* 2 (tan (time)))))
        (rotate (vector (* 90 (cos (time)))
                        (* 180 (sin (time)))
                        (* 45 (tan (time)))))
        (build-cube)))

(every-frame (render))
```

Figure 30: Screenshot of Fluxus

Clojure is a modern dialect of Lisp created by Rich Hickey. Clojure has an extensive collection of operations on collections. In its rationale it cites Alan Perlis: "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures." (*Perlis*, 1982) The **list operations** in Clojure served as an inspiration for the nodes in NodeBox 3.

Haskell

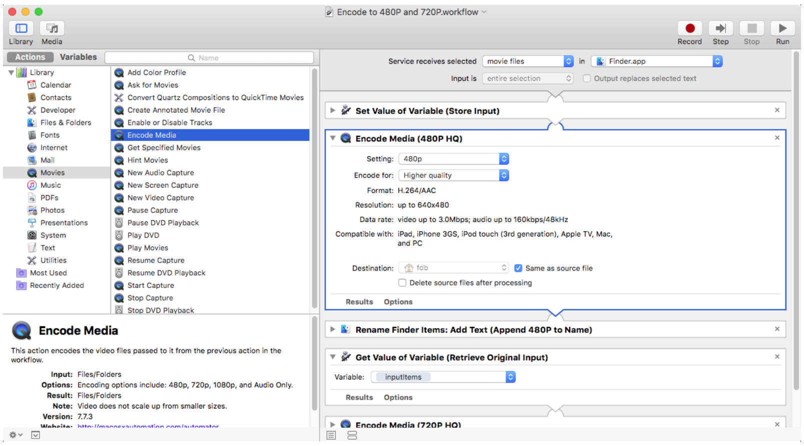
Haskell is a purely functional programming language with strong static typing, named after logician Haskell Curry. It has many features commonly associated with functional languages: lazy evaluation, pattern matching, list comprehension,...

In general, functions in Haskell do not have side effects. To represent operations that deal with state or cause side effects, Haskell uses *monads*. A monad represents a computation expressed as a series of steps. The execution can happen at a later time, or not at all. An example is the *maybe* monad which represent an optional return value: either a single value or `Nothing`, meaning no value. For an in-depth explanation we refer to (*Wadler*, 1992).

The Haskell **state monad** is a mechanism for attaching state to a pure function. A state monad is a function that takes in a state, and outputs the new state. This new state is than fed to the function again on the next execution. This allows for a clean way to model a mutable environment. The state monad served as the foundation for how we implemented stateful nodes in NodeBox 3 and NodeBox Live.

4.1.2 Dataflow / Visual Programming

Dataflow models a program as a directed graph of data flowing between operations. The first description of a dataflow system appeared in 1961 (*Kelly*, 1961). As an example, the OS X "Automator" application uses a linear dataflow metaphor to automate repetitive manual actions, such as encoding a folder of movies to a different format (Figure 31).



Encode Media

This action encodes the video files passed to it from the previous action in the workflow.

Input: Files/Folders
Options: Encoding options include: 480p, 720p, 1080p, and Audio Only.
Result: Files/Folders
Note: Video does not scale up from smaller sizes.
Version: 7.7.3
Website: http://macosxautomation.com/automator

Service receives selected: movie files in Finder.app
Input is: entire selection
Output replaces selected text

Set Value of Variable (Store Input)

Encode Media (480P HQ)
Setting: 480p
Encode for: Higher quality
Format: H.264/AAC
Resolution: up to 640x480
Data rate: video up to 3.0Mbps; audio up to 160kbps/48kHz
Compatible with: iPad, iPhone 3GS, iPod touch (3rd generation), Apple TV, Mac, and PC
Destination: fdb [✓ Same as source file]
Delete source files after processing

Rename Finder Items: Add Text (Append 480P to Name)

Get Value of Variable (Retrieve Original Input)
Variable: inputItems

Encode Media (720P HQ)

Figure 31: Screenshot of Automator

There is no standard specification for dataflow. Therefore, similar concepts have often different names in various implementations. A node is sometimes called a "block", a "process", an "action" or some other name. For an overview of the many variants of dataflow we refer to the book by Carkci (*Carkci, 2014*). Here we will look at the terminology as it applies to NodeBox.

A **node** is a processing element that takes a set of inputs, performs a computation, and returns a set of outputs. The computation can be seen as a black box. "Pure" nodes only operate on the incoming data; the only way those nodes can send and receive data is via their ports, which are the input and output connection points on the node. Sending the same input data always returns the same output data. "Impure" nodes have side effects: they might react to mouse events, make a network connection, or play a sound effect.

Nodes have input and output **ports.** Ports are the holders of data and have a type. You can see the input ports as the arguments to a function, and the output as a return value. Data typically flows from the output ports to the input ports through **connections** (sometimes called arcs or edges).

A source or **generator** node does not have any input ports. It generates data from scratch, or based on external events such as the network, mouse or keyboard. A **filter** node can have many inputs and a single output. It transforms the data, A **sink** node does have one or more inputs but no output. It is a node that is executed for its side effects, e.g. produce sound or render to the screen.

Collections of nodes and arcs are grouped together in a dataflow graph or **network.** This is the "document" of a dataflow application.

Sections of a graph can be grouped together into **subnetworks.** Subnetworks are a similar abstraction to subroutines in imperative programming. A subnetwork is a part of a dataflow program that performs a single function. Subnetworks looks similar to built-in nodes on the outside but are built out of network of nodes.

Subnetworks can take their input from **inlets,** simple connection points that pass the input from outside nodes further on to the internal nodes. Output data can be passed through an **outlet** (or, in the case of NodeBox, the *rendered node.*)

Graphs in NodeBox are always **directed**: that is, the data always flows in the same direction (copy stamping, used in NodeBox 2, is an exception to his). Also the graphs are **acyclic** meaning the connections in the graph can't form loops.

We can make a distinction between *functional* nodes and *stateful nodes.* **Functional nodes** are idempotent: given the same input values, they will always produce the same output values. Since their output is always the same, the results of complex functional nodes can be cached. **Stateful nodes,** on the other hand, retain local state. They might produce different data every time they are executed. Since we want to avoid state we can use *state monads* (explained in Section 4.1.1.2) to feed in the state as a parameter and take the return value as the new state. This creates a feedback system that passes state along between executions.

Representing loops

One of the fundamental decisions we have to make when designing a visual programming language is how to represent a loop. We can try to mimic the imperative programming style using loop brackets (e.g. Scratch) or nested networks (e.g. Quartz Composer). We can re-evaluate parts of the network using different context variables (e.g. Houdini, NodeBox 2). We can have nodes map over lists of data (e.g. Grasshopper, NodeBox 3, NodeBox Live). All these approaches have an impact on expressibility.

Scratch

Scratch is a visual programming language targeted at young people (*Resnick*, 2009). It allows children to create interactive stories, games and animations that can be shared online. Scratch is used succesfully in schools, museums, libraries, community centers and homes. In 2014, the Scratch community had over 3 million members. Scratch is developed by MIT's Lifelong Kindergarten Group.

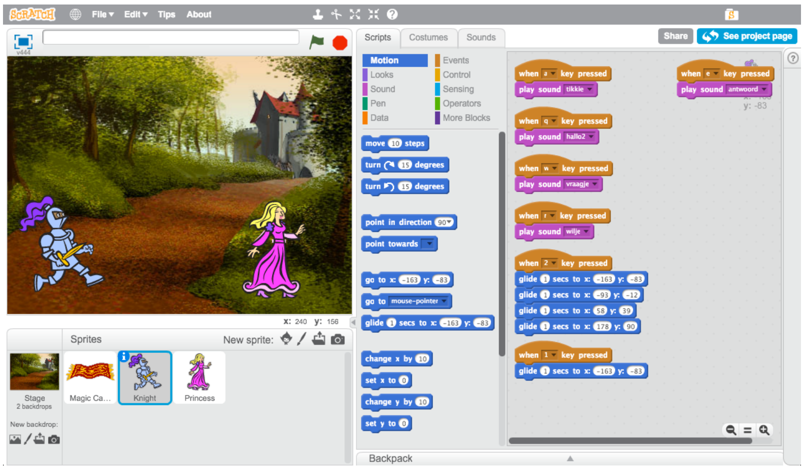
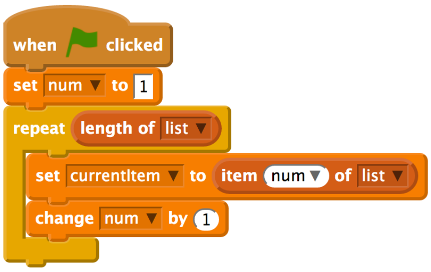


Figure 32: *Screenshot of the Scratch application*

Programs in Scratch closely mimic the structure of imperative programming. Statements are stacked on top of each other. Loops are represented as *brackets* that have a fixed counter or exit condition and contain a set of statements to be executed (see Figure 32).

Scratch avoids syntax issues of textual programming language, but still deals with state issues inherent in procedural programming. This means that anything but simple loops require loop counters. This makes it cumbersome to loop over lists of data, a key requirement in data visualization (see Figure 33).



```
when [flag] clicked
set num to 1
repeat length of list
    set currentItem to item num of list
    change num by 1
```

Figure 33: *Scratch program to loop through the elements of a list*

Scratch is listed here as an approach we *didn't* pursue when designing our visual programming environments.

Houdini

Houdini is a 3D animation application developed by Side Effects Software. It has been used in feature productions such as Frozen, Max Max and the Transformers franchise. Houdini is notable because of its comprehensive procedural approach: it has operators for modeling, particle simulation, animation and audio, compositing, dynamic simulations, shading and rendering (see Figure 34).

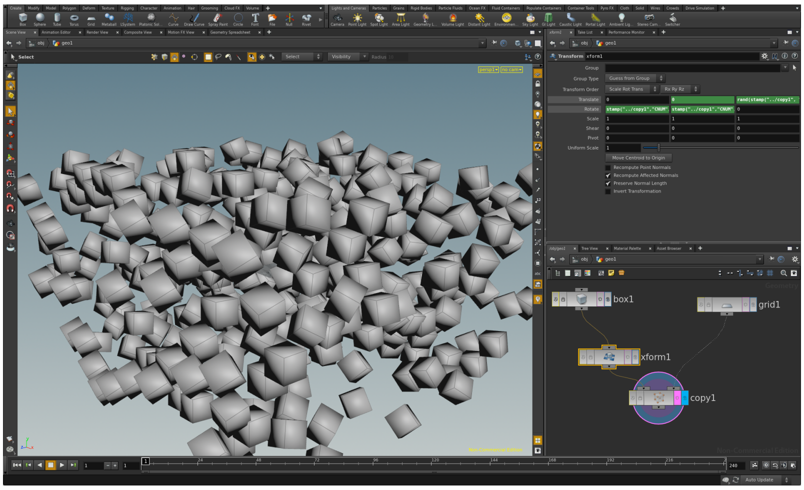


Figure 34: Screenshot of Houdini

Two aspects of Houdini inspired the design of NodeBox 2. First is Houdini's idea of *typed network* where, within a network all nodes are of the same type. This means that every available node in a subnetwork can be combined with any other node. To import differently typed data from another subnetwork you use a conversion node that can *reference* a node within that subnetwork.

The second aspect is Houdini's concept of **copy stamping.** Normally, data in Houdini flows *downstream* through a network of operators. *Stamping* lets users communicate information *upstream* to operators connected to its input. You can use this to vary the copies as they are created. Parameters can use *stamp expressions* to extract data from the copy context. The idea is somewhat similar to that of a loop index, where each time through the loop, nodes can use the current index to change their results.

Grasshopper

Grasshopper is a visual programming language plugin for the Rhino3D computer graphics application (McNeel, 2010). Users can generate entirely new geometry, or mutate existing geometry using a generative node-based approach (Figure 35).

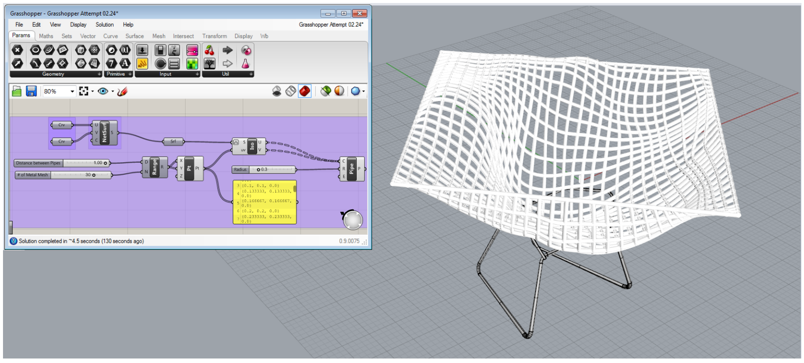


Figure 35: Screenshot of Grasshopper and Rhino3D

Grasshopper uses a **list-matching approach** when combining multiple streams of data (see Figure 36). Data is passed from one node to another using lists of data. If the node doesn't expect a list the elements of the list are *mapped over* the node function, producing a new list at the other side. If we feed in two lists with a different number of elements, Grasshopper chooses how to match the lists. It can use the **longest list** and wrap the shorter lists, it can use the **shortest list**, stopping if that list runs out of elements, or it can **cross-reference** elements, combining every input with every other input.

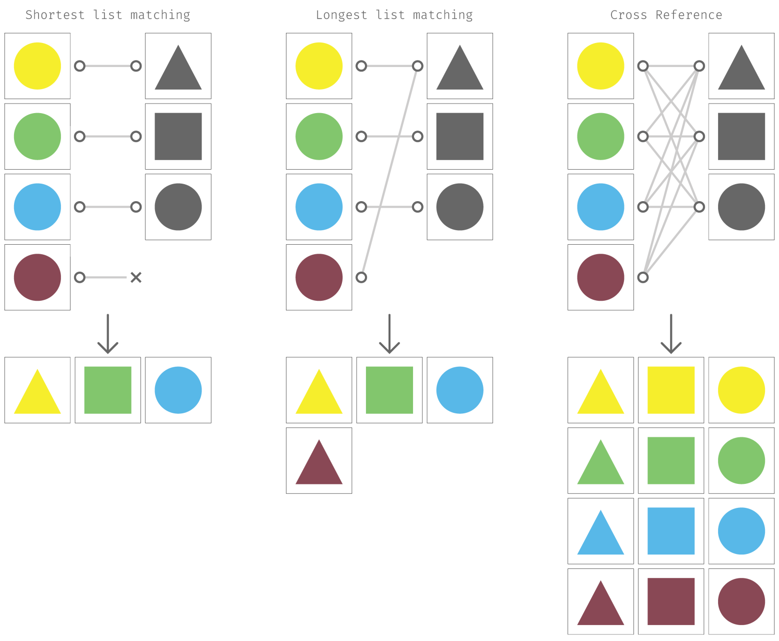


Figure 36: Different methods of list matching in Grasshopper

The list matching approach used in Grasshopper served as an inspiration for NodeBox 3.

4.1.3 Grammar of Graphics

The Grammar of Graphics is a book on data visualization written by Leland Wilkinson (*Wilkinson*, *2006*). It describes a set of grammatical rules for creating graphics based on data. The book shuns chart typologies, instead focusing on the principles that underlie all chart types. It does this by deconstructing typical chart types to its essential components, i.e. a scatterplot as a set of points where the X and Y positions are connected to variables.

This approach is similar to the model we use in NodeBox 3. Both use a pipeline approach to represent pure transformations from data into graphics, implicitly looping over the data. The Grammar of Graphics divides up visualizations in six distinct *statement types*: the *data*, the *statistical transformations*, the *scale*, the *coordinate system*, *geometric elements* and *guides*. In NodeBox 3 these statement types are all represented as different node categories.

4.2 NodeBox 2

NodeBox 2 was designed as a solution for the usability issues encountered with NodeBox 1. Our primary objectives where to make an environment where users didn't need to adhere to an unforgiving syntax and that would allow them to explore more. In addition, we speculated that a program with a graphical interface would be more appealing to designers than a text-based application.

NodeBox 2 is a cross-platform application for generative design and data visualization. NodeBox 2 uses a dataflow approach and a box-and-wire visual programming interface. It is designed to be efficient and interactive: changes to the network are immediately visible, and parameters can be controlled by dragging handles directly in the viewer. The primary focus of the application is on geometric shapes (Bézier paths), but images are supported as well. Even though the primary interaction is through the nodes, the application makes it easy to inspect and edit the underlying source code of all the nodes.

4.2.1 The Data Model

To visualize how NodeBox 2 works, imagine that a NodeBox project is like an assembly line in a factory. Data (raw materials) from the nodes (machines) is carried along connections (conveyor belts) to the next nodes (other machines further down the line). At each node, the incoming data is gradually transformed into a finished product. Each node only does a single thing then passes its data to the next node. The network view visualizes this "conveyor belt", with data flowing from the left to the right.

Each node has zero or more input ports and a single output port. Nodes can be connected together using these ports. Data then flows from one node to the other through the ports. In NodeBox 2, this data is either geometric or image data.

To tweak the behavior of the nodes each of them has a set of parameters. For example a `wiggle` node randomly transforms the points of a shape; the `amount` parameter defines the extent of this random transformation. Parameters have different types: for example an `ellipse` node has number parameters for its position and size, as well as color parameters for its fill and stroke. Parameters can have a set value or be controlled by expressions, described below.

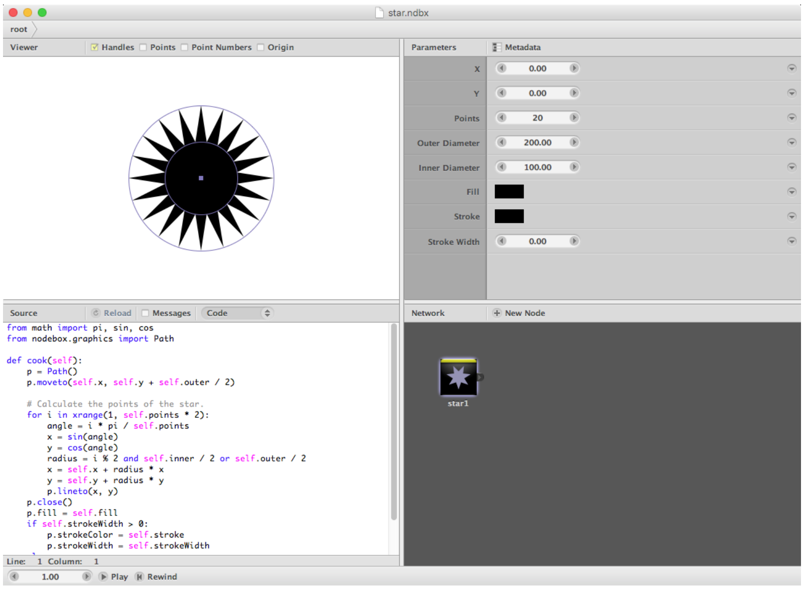
The *rendered node* is the final output of the factory. It is the result that gets displayed in the viewer and that gets exported. However, unlike in a factory it is very easy to change which node is the rendered node. The *rendered node* is more like walking around in the factory and looking at the output of a certain machine. This is useful when errors occur in the network: changing the rendered node allows us to see at which point in the network things went haywire. Users can view the output of the *rendered node* while tweaking parameters of a *selected* node higher up in the chain.

There are two types of nodes: *generator* nodes (which create new shapes) and *filter* nodes (which modify existing shapes). The generator nodes are nodes like `rect, ellipse, line, ...` These nodes create new shapes from scratch without any external input. Compare it to the raw materials that come into the factory. Filter nodes on the other hand take in one or more shapes and change them. Examples are the `translate` node which changes the position of the shape, or the `colorize` node which changes the color of the shape. Generally there are more filter nodes than generator nodes in a network.

Another useful metaphor is that of a river stream. Generator nodes are the sources *upstream* that gradually flow *downstream* through a series of natural filters. The terms *upstream* and *downstream* will be used to refer to nodes that are relatively higher up and lower down from the perspective of a single node.

4.2.2 User Interface

NodeBox 2 divides up the window into four distinct panels. Each panel represents a view of the underlying data model (see Figure 37). At the bottom right is the **network panel.** It shows all the nodes that are part of the network and allows you to create connections between them. When selecting a node in the network panel you can see its parameters in the **parameter panel** at the top right. Here you can tweak the behavior of a certain node: a `star` node has parameters for the amount of points and the inner and outer diameter of the shape. At the top left is the **viewer panel** which shows the visual output of the *rendered node* (the node with the yellow strip on top). The viewer also shows the handles for the *selected node*, allowing you to change the parameters of the node visually. (There is a one-to-one mapping of parameters and handles, meaning they modify the same underlying data). The viewer also has flags for showing the points and their indices, which is useful for data visualization. Finally the **code panel** – at the bottom left – shows the code for the selected node. It shows the user what is going on "under the hood" of each node and allows you to change the implementation. Changes are isolated to a single node instance (i.e. changing the code of one `ellipse` node won't change the behavior of other `ellipse` nodes).



X	0.00
Y	0.00
Points	20
Outer Diameter	200.00
Inner Diameter	100.00
Fill	■
Stroke	■
Stroke Width	0.00

```
from math import pi, sin, cos
from nodebox.graphics import Path

def cook(self):
    p = Path()
    p.moveto(self.x, self.y + self.outer / 2)

    # Calculate the points of the star.
    for i in xrange(1, self.points * 2):
        angle = i * pi / self.points
        x = sin(angle)
        y = cos(angle)
        radius = i % 2 and self.inner / 2 or self.outer / 2
        x = self.x + radius * x
        y = self.y + radius * y
        p.lineto(x, y)
    p.close()
    p.fill = self.fill
    if self.strokeWidth > 0:
        p.strokeColor = self.stroke
        p.strokeWidth = self.strokeWidth
```

Line: 1 Column: 1

1.00

Figure 37: Screenshot of the NodeBox 2 application

At the top of the window is the network path bar. Networks can be nested into subnetworks, and the path bar shows you where you are inside of the network.

The application uses a traditional document interface with options for saving and exporting projects, copy/paste and undo/redo.

4.2.3 Interactivity

We designed the application so that changes to parameters are immediately visible. Where possible we provide handles to control the parameters of the node visually. Users could drag on-screen controls and interactively see their changes in the viewer.

This meant that even users with very little knowledge of the system could open examples and make drastic changes just by selecting node and dragging the handles (see Figure 38). Workshops participants reported they loved tweaking both their own projects, and example projects provided with the application. This allowed them to learn by doing (G4) and arrive at new results by tweaking (G3).

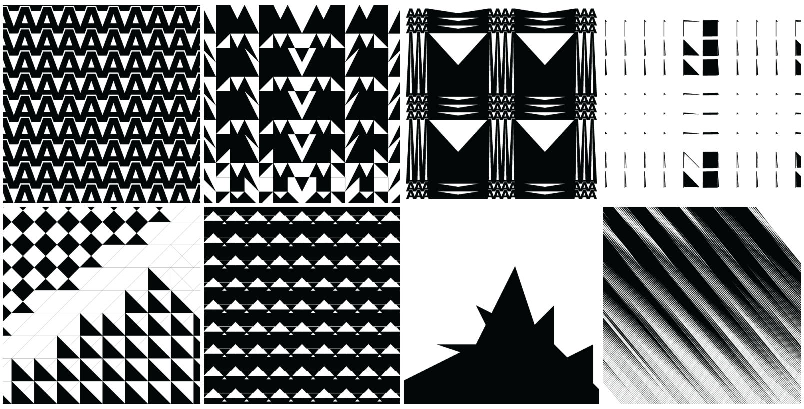


Figure 38: Variations of the same network with only parameter changes

4.2.4 Transformations

In Section 3.7.3 we saw that transformations in NodeBox 1 introduced accidental complexity. One issue was that "regular" affine transformations were difficult to reason about since they started in the top left corner. In NodeBox 1 we partially mitigated this through "center-mode" transforms, but they would cause problems with grouped objects.

In NodeBox 2 we simplified this by placing the origin at the center of the screen. Newly created shapes will be created at the center (e.g. a square of size 100 will run from `-50, -50` to `50,50).` Since all transformations happen around the origin, operations like rotating the shape will work as expected.

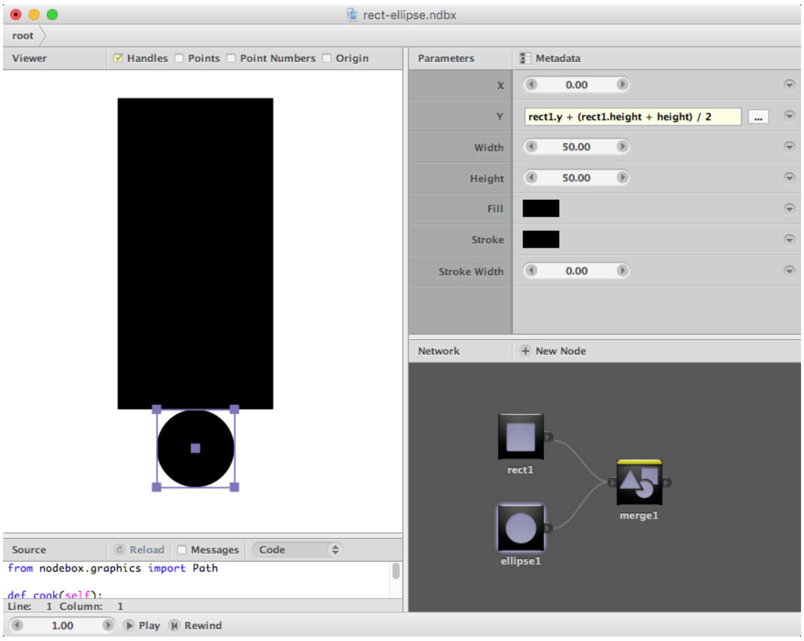
Since the location of the origin is important when reasoning about transformations, we overlaid markers indicating its position on the canvas. This allowed users to know where there composition was in relation to the origin. In addition, we provided nodes like `align` and `fit` which would automatically move their composition back to the center.

4.2.5 Expressions

Expressions make the values of parameters dynamic. They allow parameters to *refer* to each other, perform *math*, control *animation* or *stamp* values. Expressions are similar to Excel formulas.

The most common use for expressions is to link two parameters together. For example, to force the rectangle node to always produce squares, we could link its width and height together by setting the height parameter to the expression `width`. This causes the two parameters to change in sync. We can also refer to the parameters of other nodes by prefixing the name of the node, such as `ellipse1.width` (see Figure 39).

We can combine this with mathematical expressions to create relationships between elements. For example we can force an ellipse to always be below a rectangle by using the expression `rect1.y + (rect1.height + height) / 2` for its Y parameter. A number of common mathematical functions like the sine and cosine are provided under the `math` package.



X 0.00
Y rect1.y + (rect1.height + height) / 2
Width 50.00
Height 50.00
Fill
Stroke
Stroke Width 0.00

```
from nodebox.graphics import Path

def cook(self):
```

Line: 1 Column: 1

1.00

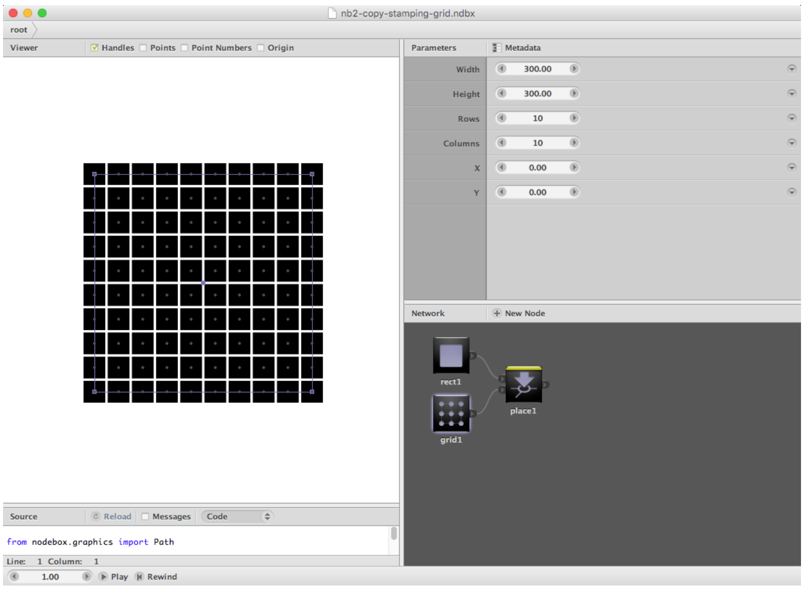
Figure 39: *An example of using expressions*

Expressions are also used to control animation. A special constant (`FRAME`) could be used in expressions to refer to the index of the current frame of animation. This constant could be used in any expression to create an ever-increasing counter. For example, setting the `angle` parameter for a rotate node to the expression `FRAME` would animate the rotation of the shape. The frame constant can be combined with the `wave` function to convert ever-increasing values into back-and-forth values. The wave function supports sine, triangle, square and sawtooth wave types.

Because not all parameters are numeric NodeBox provides expressions for converting to different data types, such as the `rgb` and `hsb()` functions that generate color values or the `trim()` function which removes spaces in strings, useful for text processing. In addition functions are provided for generating random values or restricting a value between a certain minimum and maximum ("clamping").

4.2.6 Copy Stamping

In NodeBox 2 it is trivial to create many copies of the same shape. For example, a grid of rectangles can be obtained by using the `rect`, `grid` and `place` node (see Figure 40). However, what if we wanted each rectangle to have a different size? In the assembly line metaphor, the rectangle is already created and is merely copied on all elements of the grid. There is no way to change its parameters once it has "exited" the machine that created it.



Width 300.00
Height 300.00
Rows 10
Columns 10
X 0.00
Y 0.00

```
from nodebox.graphics import Path
```

Line: 1 Column: 1

1.00

Figure 40: *A grid of rectangles*

To solve this we will need to distort the metaphor of the assembly line to allow for upstream changes of nodes. In other words, instead of only travelling *downstream*, some operations will travel back *upstream*. In our example, the `place` node will travel back upstream to the `rect` node and instruct it to produce a differently sized rectangle for every copy.

To do this, copying nodes (e.g. `place` or `copy`) assign variables in the global context that can be picked up by the upstream nodes (in this case the `rect` node) when they are re-evaluated. In other words, the stamp expression controls the value of a parameter at each iteration through the loop. The intent is to introduce the concept of a *loop index*, where each time through the loop, nodes can use the current index to change their results. Figure 41 shows how NodeBox 2 first evaluates the template geometry, then re-evaluates the `rect` node for every copy, updating it with new width and height values.

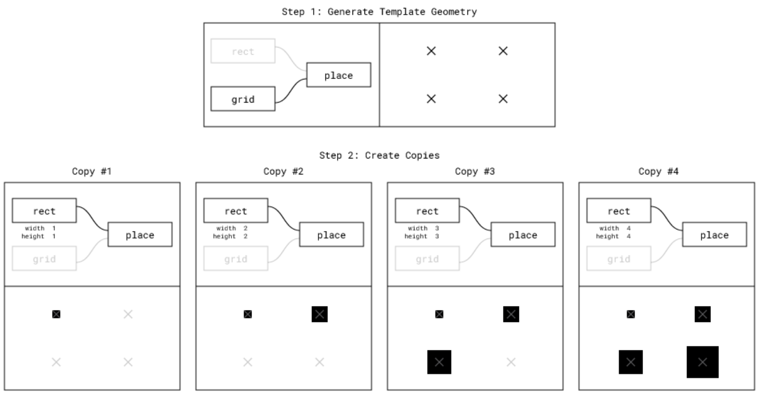


Figure 41: Copy stamping in NodeBox 2

A stamp expression looks like this: `stamp("CNUM", 42).` The key `CNUM` refers to the current index of each copy. By using this type of expressions we can create variations for any parameter of the upstream nodes (see Figure 42).

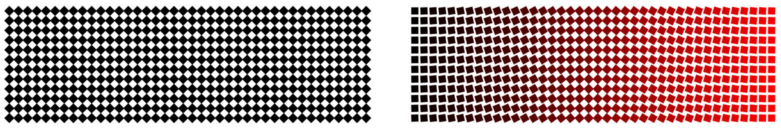


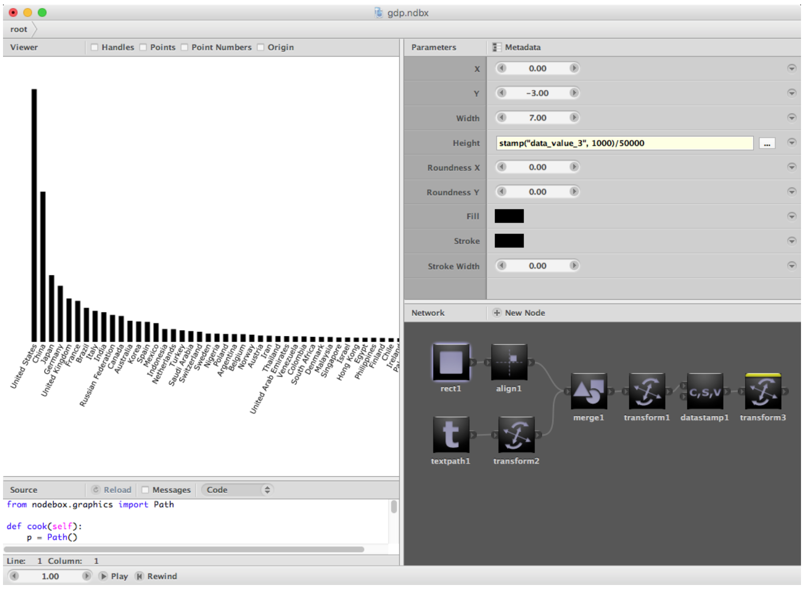
Figure 42: Before and after copy stamping

Copy stamping can allow variations to be created efficiently (G1). Since copy stamping allows control over the iterations of the loop, it can produce designs that are unfeasible to create manually (G2).

4.2.7 Data Stamping

Data Visualizations in NodeBox 2 are produced using the *datastamp* node. The input to the node is a tabular data file (a simplified version of a spreadsheet represented as a comma-separated value file or `CSV`). For each row of the data file, new geometry is created. The datastamp node iterates through the rows, re-rendering all connected upstream nodes and exposing global variables to be used in stamping expressions. Nodes can use these variables to change their font size or use different colors or shapes.

As an example, imagine that we want to make a bar chart based of the GDP of different countries. We create a *datastamp* node that refers to CSV file. The node exposes a number of variables, for example `data_value_0` referring to the data of the first column. The node would re-run the upstream nodes as many times as there are rows in the CSV file. In the height parameter of a rectangle we could then write `stamp("data_value_0", 100)` which would set the height dynamically based on the data (see Figure 43).



Parameters	Metadata
X	0.00
Y	-3.00
Width	7.00
Height	stamp("data_value_3", 1000)/50000
Roundness X	0.00
Roundness Y	0.00
Fill	■
Stroke	■
Stroke Width	0.00

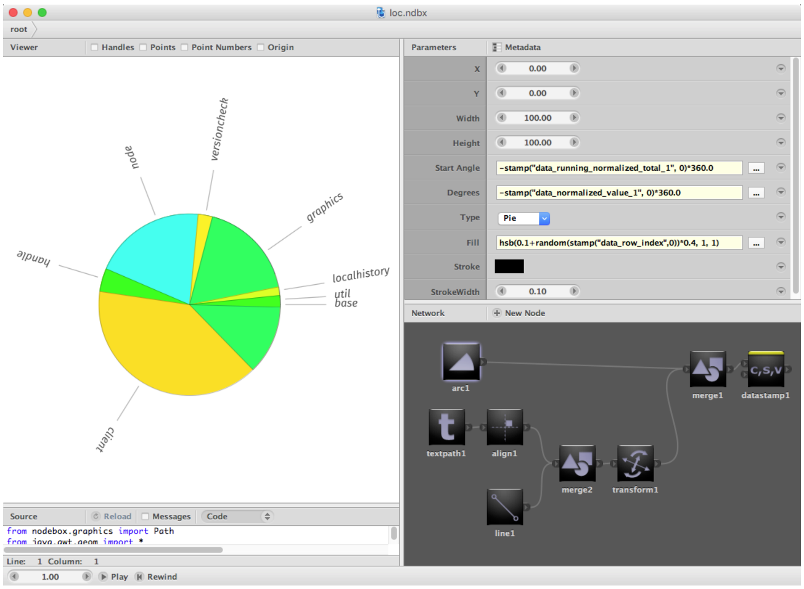
```
from nodebox.graphics import Path

def cook(self):
    p = Path()
```

Line: 1 Column: 1

1.00

Figure 43: NodeBox screenshot showing data stamping

The *datastamp* node provides a number of useful keys: the relative value (the total of the column divided by the current value), the normalized value (the current value within the min-max range of the column), the column total, the "running" total (the accumulation of values up until this row), and the running normalized total (the running total divided by the min-max range). These pre-calculated values are necessary to create different types of visualizations. For example the normalized value and running total are used when creating pie charts, where the total has to add up to 100% (360 degrees) (see Figure 44).



Parameters | Metadata

X	0.00
Y	0.00
Width	100.00
Height	100.00
Start Angle	-stamp("data_running_normalized_total_1", 0)*360.0
Degrees	-stamp("data_normalized_value_1", 0)*360.0
Type	Pie
Fill	hsb(0.1+random(stamp("data_row_index",0))*0.4, 1, 1)
Stroke	■
StrokeWidth	0.10

```
from nodebox.graphics import Path
from java.awt.geom import *
```

Line: 1 Column: 1

1.00

Figure 44: Pie chart showing lines of code in sections of the NodeBox source

NodeBox 2 expects the imported data to be well prepared for use (single header row, no extraneous comments in the spread sheet, no empty cells).

Data Stamping allows user to quickly visualize large data files (G1). However we will argue in Section 4.2.13.2 that the stamping metaphor limits its ease of use (G4).

4.2.8 Node + Code

Early on in the design we made the decision that we didn't want to hide the code. Although we used nodes as the primary metaphor in the interface, we still found it important that users would be able to see the code that made up the nodes. We wanted users to realize that behind every node was a piece of code that made it work, and more importantly, that they could change this code and see the effects immediately (G4).

The expression language, described above, was an intermediate layer between the nodes and the code. Expressions were designed to be easy to use and – especially for simple expressions – easy to understand. They made users understand that a little bit of code could make their generative designs more dynamic. Internally, we jokingly referred to them as the "gateway drug" to coding.

Changes to the code only affected the selected node instance. This meant that users were free to change the code without accidentally breaking the entire application (G4). We integrated simple code changes into the curriculum and online documentation.

4.2.9 NodeBox 1 Compatibility

The functional model of NodeBox 2 works fundamentally different than the imperative model of NodeBox 1. In NodeBox 2, nodes transform the incoming geometry through a set of filters. Each node performs a single operation on all of the data and passes it on to the next node. In NodeBox 1 these steps would be mixed together in a single loop.

To provide users with a bridge from NodeBox 1 to NodeBox 2 we made the Python API backwards-compatible. We built a glue layer on top of the NodeBox 2 graphical objects that accepted NodeBox 1 commands. This meant that most of the NodeBox 1 scripts also ran in NodeBox 2 (G4). These scripts would be represented as *generator* nodes and could be combined with all the other nodes of NodeBox 2.

4.2.10 Creating Custom Nodes

Functionality inside of NodeBox 2 is provided through *built-in* nodes. These are nodes that ship with the application. If there is functionality missing in the application we encourage users to build it themselves by creating a *custom* node. They can either start from scratch or base their work on a built-in node whose functionality is similar to what they want and modify the code. Users can also define handles for their node, either by combining existing handles or creating a new handle from scratch.

When creating custom nodes, users use the same exact API as the built-in nodes. Because our nodes are not "special", users can create nodes with the same expressive power as the built-in nodes. In fact all of the built-in nodes are contained in a regular project that is imported on startup (see Figure 45).

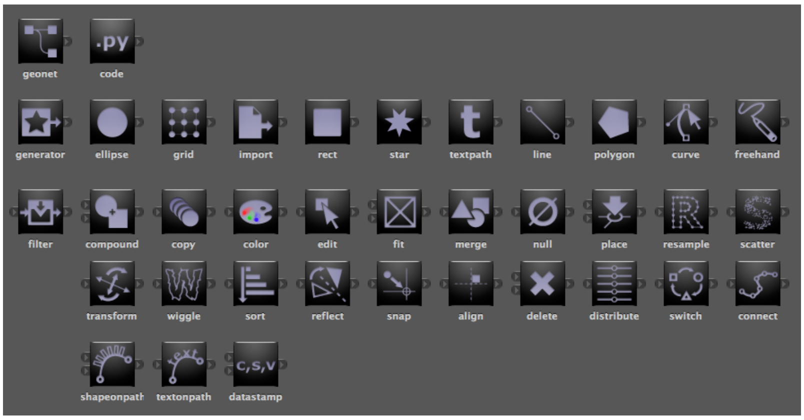


Figure 45: An overview of the corevector library containing all the built-in vector nodes

Users can extend the application in any way they want by calling all available APIs. This integration helps in re-using existing libraries, enhancing efficiency (G1). If they get stuck or unsure how to do something, they can examine the code for the built-in nodes (G4). And because they use the same APIs, users are not restricted by arbitrary constraints.

Because the node code was always visible and served as an example for our users, it was very important that the quality of the code was exemplary. We focused on providing clean, well documented code for all of our built-in nodes.

4.2.11 Visual results

Finish Surnames – Eemeli Nieminen (2010)

In 2010, one of our students from Lahti, Eemeli Nieminen, wanted to use NodeBox to create a data visualization showing the popularity of Finnish surnames. This was the first visualization of a publicly available dataset created using NodeBox 2. The font size of the surnames is scaled according to their popularity (Figure 46).

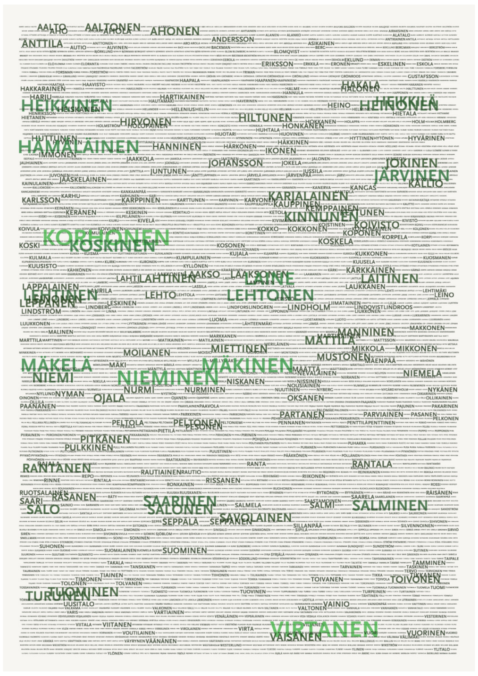
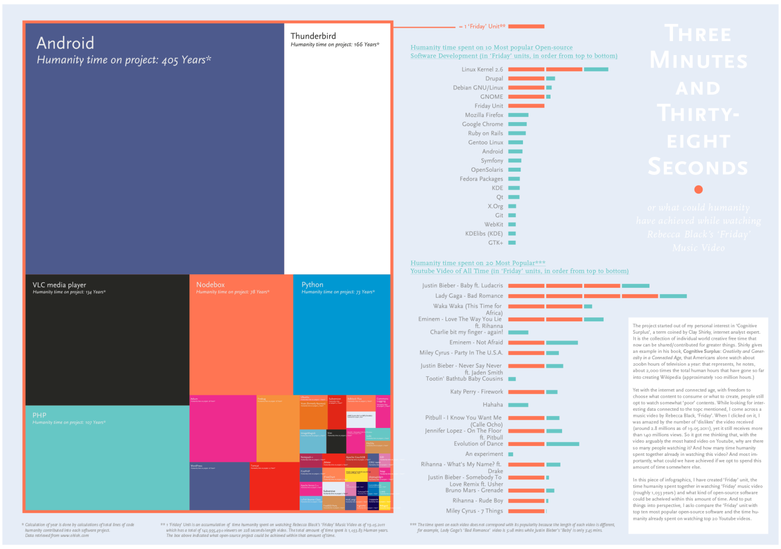


Figure 46: "Finish Surnames", Eemeli Nieminen, 2010

Because NodeBox 2 was primarily made for generative design, initially we didn't have options for integrating external data. Because we could extend NodeBox on-the-fly we designed a precursor to the `datastamp` node during the workshop, and afterwards integrated it into the NodeBox core.

Three Minutes and Thirty Eight Seconds – Sarun Pinyarat (2011)

This project, created in the 2011 Helsinki workshop, started out of Sarun's personal interest in "Cognitive Surplus" (*Shirky*, 2010), the idea that humans now have free time that they can decide to spend however they want. Sarun was fascinated with the fact that people opt to spend that time watching "bad" content, such as the "Friday" music video by Rebecca Black, the most disliked music video on YouTube. The project takes the total time humanity has spent watching this music video (roughly 1,033 man-years) and contrasts this with the amount of effort needed to create popular open-source projects, such as Firefox or Android (Figure 47).



The project started out of my personal interest in 'Cognitive Surplus', a term coined by Clay Shirky, internet analyst expert. It is the collection of individual world creative free time that now can be shared/contributed for greater things. Shirky gives an example in his book, **Cognitive Surplus**: *Creativity and Generosity in a Connected Age*, that Americans alone watch about 200bn hours of television a year: that represents, he notes, about 2,000 times the total human hours that have gone so far into creating Wikipedia (approximately 100 million hours.)

Yet with the internet and connected age, with freedom to choose what content to consume or what to create, people still opt to watch somewhat 'poor' contents. While looking for interesting data connected to the topc mentioned, I come across a music video by Rebecca Black, 'Friday'. When I clicked on it, I was amazed by the number of 'dislikes' the video received (around 2.8 millions as of 19.05.2011), yet it still receives more than 140 millions views. So it got me thinking that, with the video arguably the most hated video on Youtube, why are there so many people watching it? And how many time humanity spent together already in watching this video? And most importantly, what could we have achieved if we opt to spend this amount of time somewhere else.

In this piece of infographics, I have created 'Friday' unit, the time humanity spent together in watching 'Friday' music video (roughly 1,033 years) and what kind of open-source software could be acheived within this amount of time. And to put things into perspective, I aslo compare the 'Friday' unit with top ten most popular open-source software and the time humanity already spent on watching top 20 Youtube videos.

** Calculation of year is done by calculations of total lines of code humanity contributed into each software project. Data retrieved from www.ohloh.com*

*** 1 'Friday' Unit is an accumulation of time humanity spent on watching Rebecca Black's 'Friday' Music Video as of 19.05.2011 which has a total of 142,995,490 viewers on 228 seconds-length video. The total amount of time spent is 1,033.83 Human years. The box above indicated what open-source project could be achieved within that amount of time.*

**** The time spent on each video does not correspond with its popularity because the length of each video is different, for example, Lady Gaga's 'Bad Romance' video is 5:08 mins while Justin Bieber's 'Baby' is only 3:45 mins.*

Figure 47: "Three Minutes and Thirty Eight Seconds", Sarun Pinyarat, 2011

Sun Flowers – Cvijeta Miljak (2011)

The Sun Flowers project visualized the amount of daylight throughout the year in different regions of the world. It shows the sometimes stark differences between summer and winter in northern countries such as Finland (Figure 48).

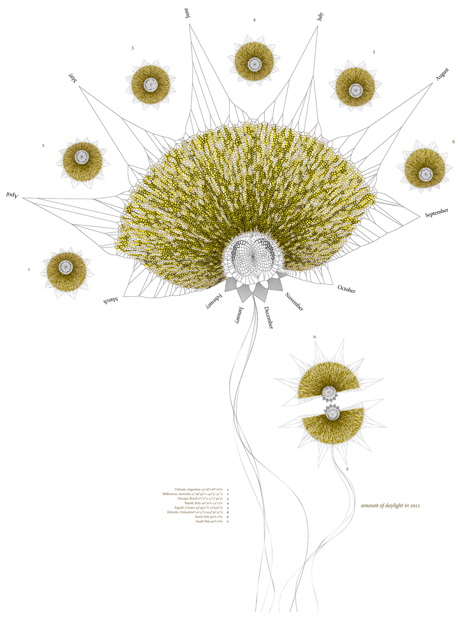


Figure 48: "Sun Flowers" – Cvijeta Miljak (2011)

We Want Sex (Equality) – Alice Lucchin and Stefano Zanini (2011)

This project was done during a commercial workshop organized in Turin together with the TODO design group and the Density Design research group. The work focuses on the difference in gender distribution of the Italian local politicians, visualized in a tree structure. It clearly highlights the fact that, although Italian female politicians are better educated and younger than their male colleagues, they are still totally outnumbered by them (Figure 49).

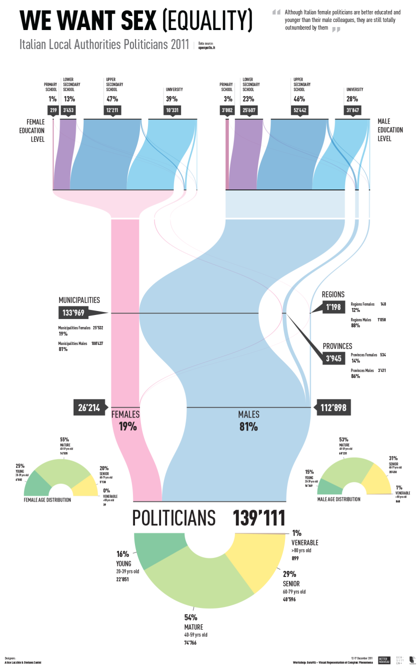


Figure 49: "We Want Sex (Equality)", Alice Lucchin & Stefano Zanini, 2011

4.2.12 Evaluation

To evaluate our approach we used NodeBox 2 during design workshops. We observed how the application was used and we surveyed participants on the user-friendliness and efficiency of the software.

Node-Based Interface

From *questionnaires* we learned that, in general, users found the application – after initial training – to be user-friendly (G4). They appreciated having an overview of all available functionality (the nodes) in one place. They liked being able to export their work in PDF so that they could work with it in other applications.

Users remarked being able to create complex shapes that would be difficult or impossible to make in traditional software such as Adobe Illustrator (G2). They liked being able to modify the parameters and immediately see the results in the viewer. Users reported they spent a long time playing with the parameters just to see which results would appear. The distinction between selected and rendered node, while initially confusing, was reported as a very powerful tool in rapidly exploring variations of a design (G3).

Building Networks

During tutorials we observed that students sometimes had trouble connecting networks as shown by the teacher. For example, they would connect them in reverse (e.g. transform -> rect instead of rect -> transform), even though that connection doesn't make sense from a conceptual point of view. This demonstrates that novices and advanced beginners lack the "big picture". They do not yet have a holistic understanding of the operation of the system (see Dreyfus model Section 2.6.5).

We also observed this behavior when students made their own projects. Even though they understood at the end of the lectures how nodes could be connected, they had trouble figuring out how to translate their idea into a correct set of nodes. This corroborates the findings by Spohrer (*Spohrer*, 1986) on plan composition.

Data Visualization

Later NodeBox workshops focused on using the tool for data visualization. Overall users found it easy to follow tutorials but harder to build up their own visualizations.

We speculate that this relates to the fact that data stamping caused a lot of confusion, making students unsure of what to do next (G4). Because the flow when using data stamping is no longer linear (you have to go back and forth to build up expressions) this can cause difficulties when building up a plan.

Transformations

Due to the location of the origin, we observed much less errors with regards to transformation. We argue that this new method allows users to be less afraid of transformations and become more self-reliant (G4).

Because compositions are created on an essentially infinite plane, some students remarked that they found it difficult to guess the final size of their project. In response, we added a "page" bounding box that is used when the document is exported.

4.2.13 Lessons Learned

Data Preparation

One key component of data visualization projects is the preparation and cleanup of the data, with reports indicating that data scientists spend about 1-3 hours on it *every day* (*King*, 2015). Tutorial material is often deceptive in this regard as the example files are already carefully prepared in the format NodeBox 2 expects. Real-world data (e.g. from governmental websites) is often much messier.

NodeBox 2 doesn't have built-in support for filtering or aggregating the data. We asked users to clean up their data and pre-compute needed aggregate data in Google Docs or OpenRefine before using it in NodeBox 2. This required going back-and-forth between the data and the visualization, as users figured out they needed an additional piece of data that NodeBox 2 couldn't compute for them. Some complex aggregations or data mining operations required programming (often in NodeBox 1). This made users less self-reliant than if the application had built-in facilities for simple data processing (G4).

To mitigate this we built facilities in NodeBox 3 (see Section 4.4) for simple aggregations, and also provided a tool for working with large data sets (see Section 4.3).

Copy Stamping

Throughout NodeBox 2 we used copy stamping and data stamping as a way to change the values of parameters in response to changes in the data. The implementation of copy stamping requires it to re-evaluate upstream nodes for each copy. Because this changes the fundamental nature of the assembly line metaphor (because data travels upstream through the expressions), you lose some of the benefits of dataflow.

The biggest disadvantage is that you can no longer visually inspect each step to see where things go wrong. Stamped nodes, because they rely on downstream nodes to provide global values, have no useful output when you inspect them on their own. Not being able to inspect the output of intermediate nodes makes networks harder to debug. A lot of the debugging needs to happen in your head as you're trying to replay the evaluation of the network. Even for experienced users this causes issues, especially as expressions grow larger.

A second disadvantage of copy stamping is that it is compute-intensive. Because it requires re-executing parts of the node graph that were already executed, the system performs a lot of duplicated work. This performance issue was particularly evident when multiple copy stamping operations were used.

In short, copy stamping in NodeBox 2 created an abstraction that doesn't cleanly decompose. This makes building larger components out of smaller components difficult or impossible due to complexity and performance issues.

We learned that this metaphor was unsustainable in the long term, and moved to a different approach in NodeBox 3 (see Section 4.4).

Expressions

Using expressions in parameters is an important tool to create compelling designs and visualizations. However, we observed that expressions used in data visualizations could grow to be very large. This is a consequence of the fact that we can only work with the raw values from the *datastamp* node, so all of the value conversion needs to happen at the location of use. For example here is the expression for converting temperature to an RGB value:

```
color(
    stamp("data_value_2", 10.0) / 30.0, 0.00,
    1.0 - (stamp("data_value_2", 10.0) / 30.0),
    1.00)
```

Large expressions – in combination with the copy stamping issues mentioned above – became difficult to debug. Users had to resort to temporarily removing parts of the expression and re-executing, which quickly became tedious.

We found that even we, as experienced users, had difficulty reasoning about the expressions in our head. Again this limits the amount of complexity we can introduce in a system.

We partially mitigated this by introducing pre-processed, normalized values and running totals. Even so, expressions are still too large to be easily readable. To reduce complexity further we could do a number of pre-processing steps on the data beforehand. For example, data scales (that map values from *data space* into *visual space*) could be defined outside of the parameter expressions. This would also improve reuse. A well-known approach for this is detailed in the Grammar of Graphics (*Wilkinson*, 2006). In NodeBox 3 we take a similar approach.

Hidden dependencies of expressions

Expressions, because they referenced other nodes, create an additional set of hidden dependencies (next to the dependencies by the nodes, which are easily visualized by their connections). To be able to evaluate the network graph we need it to be acyclic. Inadvertent use of expressions could cause circular dependencies.

If we would continue development on NodeBox 2 we would visualize these hidden dependencies, for example by drawing together with regular connections in the network view, but using a different color. Extensive research has been done around visualizing dependencies in spreadsheets (*Shiozawa*, 1999)(*Hermans*, 2011) and these techniques could be ported to NodeBox 2 as well.

4.2.14 Conclusion

Compared to NodeBox 1 we noticed a general improvement in the quality of the work (G2). We also noticed that students were able to spend more time working by themselves before they got stuck and needed guidance (G4) (we will examine this observation in more detail when we talk about Grasp, in Section 4.7). In addition we saw surprising results that were less linked to the examples bundled with the software. This implies students were free to explore more and discover the boundaries of the software (G3).

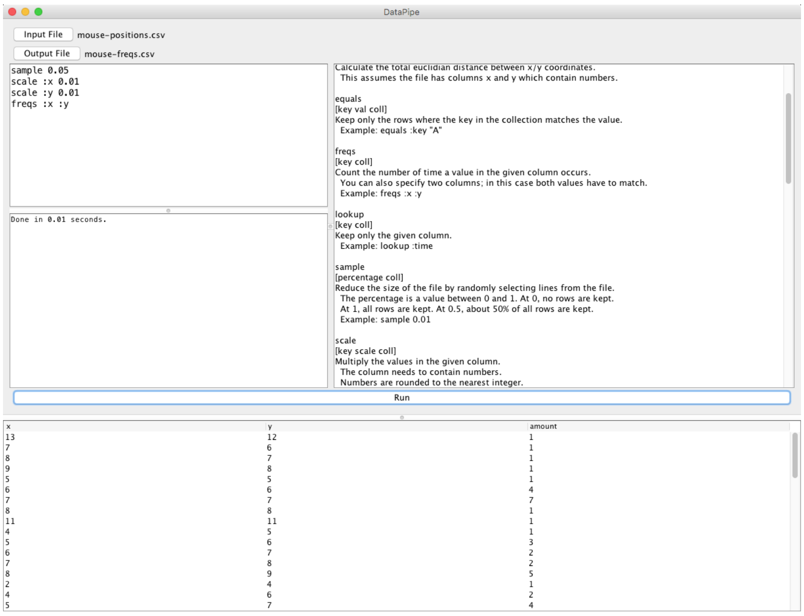
The major challenge with the software was understanding and writing expressions used for copy stamping. Because copy stamping conflicts with the conceptual model of the assembly line, it made reasoning about data flow difficult, not just for beginners but also for experienced users. We discovered that expressions could grow so large they became impractical to debug, limiting the scope of what was possible with the program.

This issue required fundamentally changing the execution model of the application. We decided to abandon the copy stamping approach and move to a new approach where the fundamental data type is a list, and all operations on the network are essentially list operations. That is what we did in NodeBox 3.

4.3 DataPipe

Before data can be used for visualization, it often needs to be filtered, aggregated and mined to produce a "clean" data file that can be used in a visualization program (*Fry*, 2007). DataPipe (*De Bleser*, 2014) is a data processing tool that can work with multi-million row input files and through filtering, sampling and aggregation reduce them down to thousand- or ten-thousand row files, well-suited for visualization in different versions of NodeBox (see Figure 50).

One common use case is to let DataPipe produce a random sample from a larger data file, in essence giving a "lower-resolution" file to work with during development in NodeBox that can be switched out to the "full-resolution" file on the final render. This greatly improves the interactive speed of operations (G1).



mouse-positions.csv

mouse-freqs.csv

```
sample 0.05
scale :x 0.01
scale :y 0.01
freqs :x :y
```

```
Done in 0.01 seconds.
```

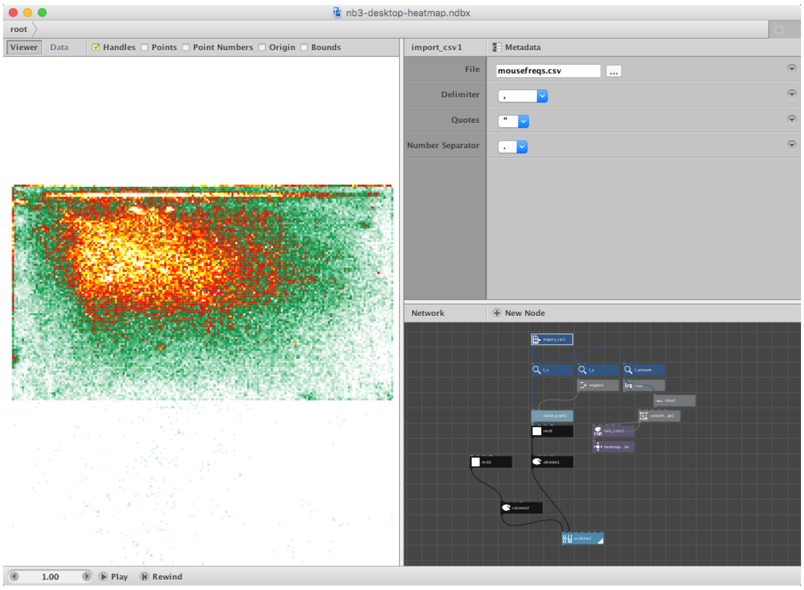
```
Calculate the total euclidian distance between x/y coordinates.
  This assumes the file has columns x and y which contain numbers.

equals
[key val coll]
Keep only the rows where the key in the collection matches the value.
  Example: equals :key "A"

freqs
[key coll]
Count the number of time a value in the given column occurs.
  You can also specify two columns; in this case both values have to match.
  Example: freqs :x :y

lookup
[key coll]
Keep only the given column.
  Example: lookup :time

sample
[percentage coll]
Reduce the size of the file by randomly selecting lines from the file.
  The percentage is a value between 0 and 1. At 0, no rows are kept.
  At 1, all rows are kept. At 0.5, about 50% of all rows are kept.
  Example: sample 0.01

scale
[key scale coll]
Multiply the values in the given column.
  The column needs to contain numbers.
  Numbers are rounded to the nearest integer.
```

x	y	amount
13	12	1
7	6	1
8	7	1
9	8	1
5	5	1
6	6	4
7	7	7
8	8	1
11	11	1
4	5	1
5	6	3
6	7	2
7	8	2
8	9	5
2	4	1
4	6	2
5	7	4

Figure 50: Screenshot of the DataPipe application

DataPipe's user interface consists of a simple command language where chains of commands can be connected together as a simple version of dataflow (see Section 4.1.2). DataPipe was designed to process larger-than-memory streams by chunking the data into usable parts using Clojure's lazy evaluation feature.

DataPipe was used in the "Quantified Self" masterclass held at Sint Lucas Antwerpen in 2014. It was used to visualize mouse and keyboard information gathered over multiple months (see Figure 51).



import_csv1 | Metadata

File: mousefreqs.csv

Delimiter: ,

Quotes: "

Number Separator: .

Figure 51: *Heatmap of mouse positions*

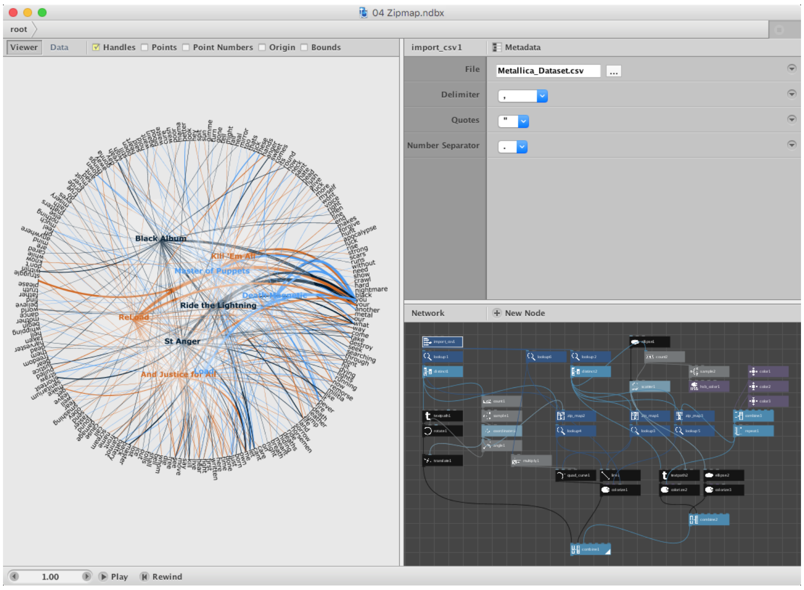
4.4 NodeBox 3

The issues with expressions and copy stamping caused us to re-evaluate our approach with visual programming. In addition we realized that data visualization tools needed to manipulate the data itself, and so a new system should be handle more than just geometric data.

NodeBox 3 kept the cross-platform foundation and general look of NodeBox 2. However, the core execution engine was rewritten to get rid of copy stamping and expressions. We elevated lists as the primary abstraction (see Section 4.4.2) and used the concept of list matching (see Section 4.4.4) to create variations. To support data manipulation and not just geometric manipulation the system supported a wide variety of types through unified ports and parameters (see Section 4.4.3).

4.4.1 User Interface

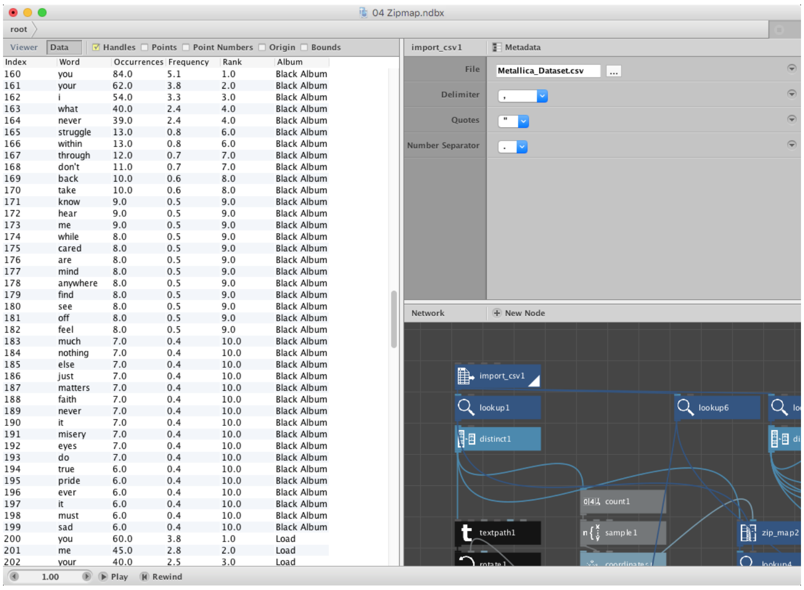
The interface to NodeBox 3 looks similar to NodeBox 2. We changed the orientation of the nodes from horizontal to vertical to accommodate for multiple ports (see Figure 52).



File: Metallica_Dataset.csv
Delimiter: ,
Quotes: "
Number Separator: .

Figure 52: Screenshot of NodeBox 3

NodeBox 3 also adds a "data" view, which explicitly shows the output data of a node in a tabular format (see Figure 53). This allows us to verify that a node created the expected number of elements. The data view was often used in debugging (G4).



File: Metallica_Dataset.csv
Delimiter: ,
Quotes: "
Number Separator: .

Index	Word	Occurrences	Frequency	Rank	Album
160	you	84.0	5.1	1.0	Black Album
161	your	62.0	3.8	2.0	Black Album
162	i	54.0	3.3	3.0	Black Album
163	what	40.0	2.4	4.0	Black Album
164	never	39.0	2.4	4.0	Black Album
165	struggle	13.0	0.8	6.0	Black Album
166	within	13.0	0.8	6.0	Black Album
167	through	12.0	0.7	7.0	Black Album
168	don't	11.0	0.7	7.0	Black Album
169	back	10.0	0.6	8.0	Black Album
170	take	10.0	0.6	8.0	Black Album
171	know	9.0	0.5	9.0	Black Album
172	hear	9.0	0.5	9.0	Black Album
173	me	9.0	0.5	9.0	Black Album
174	while	8.0	0.5	9.0	Black Album
175	cared	8.0	0.5	9.0	Black Album
176	are	8.0	0.5	9.0	Black Album
177	mind	8.0	0.5	9.0	Black Album
178	anywhere	8.0	0.5	9.0	Black Album
179	find	8.0	0.5	9.0	Black Album
180	see	8.0	0.5	9.0	Black Album
181	off	8.0	0.5	9.0	Black Album
182	feel	8.0	0.5	9.0	Black Album
183	much	7.0	0.4	10.0	Black Album
184	nothing	7.0	0.4	10.0	Black Album
185	else	7.0	0.4	10.0	Black Album
186	just	7.0	0.4	10.0	Black Album
187	matters	7.0	0.4	10.0	Black Album
188	faith	7.0	0.4	10.0	Black Album
189	never	7.0	0.4	10.0	Black Album
190	it	7.0	0.4	10.0	Black Album
191	misery	7.0	0.4	10.0	Black Album
192	eyes	7.0	0.4	10.0	Black Album
193	do	7.0	0.4	10.0	Black Album
194	true	6.0	0.4	10.0	Black Album
195	pride	6.0	0.4	10.0	Black Album
196	ever	6.0	0.4	10.0	Black Album
197	it	6.0	0.4	10.0	Black Album
198	must	6.0	0.4	10.0	Black Album
199	sad	6.0	0.4	10.0	Black Album
200	you	60.0	3.8	1.0	Load
201	me	45.0	2.8	2.0	Load
202	your	40.0	2.5	3.0	Load

Figure 53: Screenshot of the NodeBox 3 data view

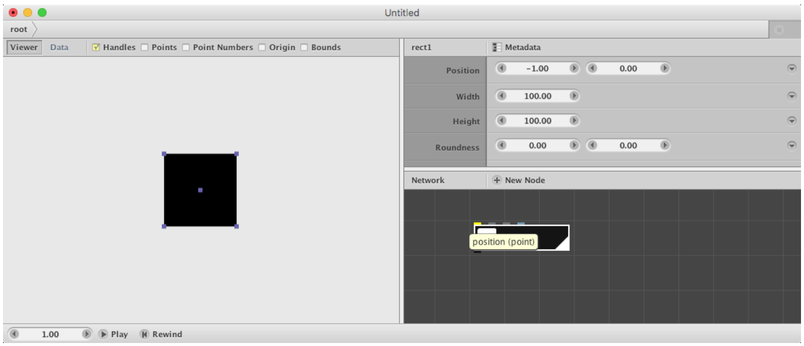
4.4.2 List as primary data structure

In NodeBox 1 we learned that teaching users data structures – especially lists – gave them a tool to express problems in the solution domain (see Section 3.7.5). In NodeBox 3 we made the list the pervasive data structure. We no longer treat the data being processed as a single opaque data structure, but as lists of objects that can be manipulated. In NodeBox 2, the `copy` node generated an opaque `Geometry` object. In NodeBox 3 it generates a list of `Path` objects. This has the advantage that all list operations (`slice`, `shuffle`, `combine`) can be applied to it.

This also applies to non-visual data. When creating a visualization of all the US ZIP codes, each ZIP code is an item in a list; this list can be sorted, reversed, sampled (to visualize only 1%), or aggregated. Lists are also used to represent the pixels in an image, the points in a shape or the words in a piece of text. The list operations are generic and can be applied to any kind of list.

4.4.3 Ports and Parameters

NodeBox 3 unifies ports and parameters. There is a one-to-one relationship between the ports on top of the node and the parameters visible in the parameter pane (see Figure 54).



Position -1.00 0.00
Width 100.00
Height 100.00
Roundness 0.00 0.00

Figure 54: Parameters and Ports show the same values. Note that the first port of the rect node (position) corresponds to the first parameter.

This implies that any parameter can be driven with data from another node, as long as the types are compatible. For example, the height of a rectangle could be controlled by a column of prices coming from a tabular data file. Given multiple prices, this would create a bar chart. However, since we now have many values feeding the port of a single node, we will need to deal with this inconsistency. We solve this by using list matching.

4.4.4 List matching

Loops in NodeBox 3 are implicit. Most nodes only deal with one element at a time (e.g. the `transform` node only transforms a single shape). To create multiple copies of a shape, we feed in a list of values to one of the ports. We then use a list processing construct similar to the `map` operation in functional programming where we *map over* a list of elements, evaluating the node for each item in the list. We assemble the return values into a new list at the other end.

As an example, to create a grid of rectangles, we use a `grid` node to create a list of points laid out in a grid. We then connect this list of points to the `position` port of the `rect` node, which creates rectangles for each point in the input list (see Figure 55).

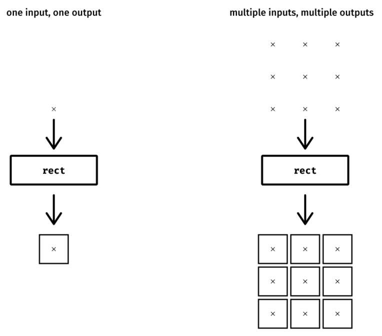


Figure 55: NodeBox 3 executes the node as many times as there are inputs

To create variation, we can feed in multiple lists at the same time. For example, a `text` node used for drawing text could have both a list of strings (the text to display) and a list of positions (coming from a `grid` node). NodeBox will *match* the list of strings and the list of positions, creating a resulting item for every combination (see Figure 56).

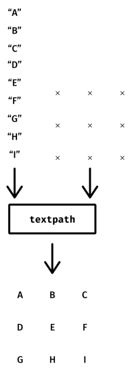


Figure 56: List matching with lists of equal size

In this example the lists have the same length. But what if they have different lengths? What if you have only 2 items of text and 9 positions? In that scenario, there are multiple solutions we can take:

- Match on the **shortest list**: stop processing the lists as soon as the shortest one runs out. This is the scenario we took in the beginning of NodeBox 3. This is also the way that Clojure's map function operates on multiple lists.

- Match on the **longest list**: only stop processing lists when the longest one runs out. For all shorter lists, cycle them. So we have a list with five names `["Alice", "Bob", "Chad", "Dave", "Emmet"]` and two colors `["Red", "Blue"]`, we will generate the following pairs: `Alice - Red, Bob - Blue, Chad - Red, Dave - Blue, Emmet - Red.`

- **Crash**: when the lists are not of the same length, refuse to perform the computation until the users decides what to do. This is the approach the Wolfram Language takes. Although it seems drastic, it solves the off-by-one errors resulting from longest-list matching. It also means that cycling needs to be made explicit. We will discuss this approach when we talk about NodeBox Live, in Section 4.5.1.

NodeBox 3 uses *longest list* matching (see Section 4.4.9.1 in Lessons Learned to find out why we chose this solution). We will repeat the procedure for each item in the longest list. Shorter lists are *cycled*: we repeat their items until we run out. (see Figure 57).

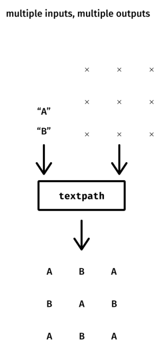
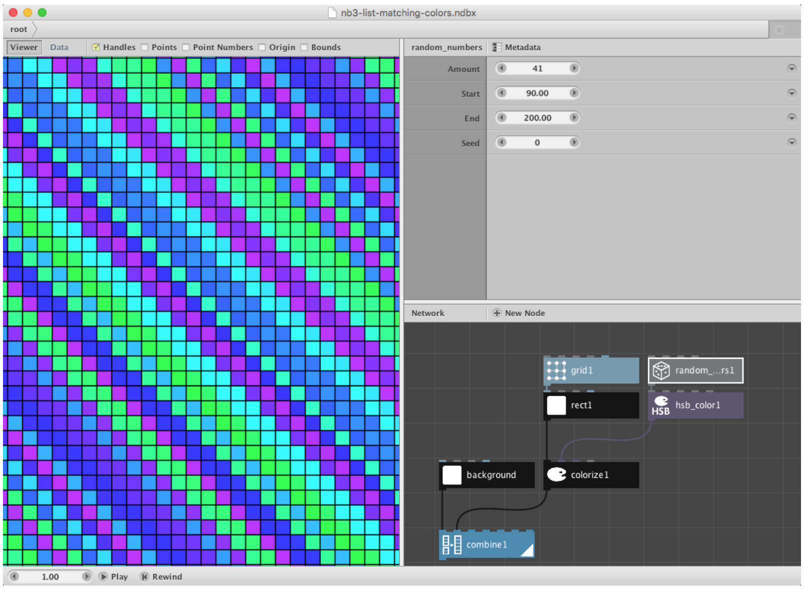


Figure 57: List matching with lists of different size

List matching works across multiple nodes. Because feeding a `rect` node with a list of points creates another list at the other end, we can use that list further down the chain, for example to set colors (Figure 58).



random_numbers: Amount 41, Start 90.00, End 200.00, Seed 0

1.00

Figure 58: List matching across different nodes

4.4.5 Data Visualization

NodeBox 3 redesigned NodeBox 2's data stamping approach. When importing an external tabular data file, such as a `CSV` file, NodeBox 3 returns a list of **associative arrays** (also called *maps* or *dictionaries*). For each row in the file, NodeBox associates the value with the name of the column header. To retrieve data from an associative array we provide the `lookup` node. Given a column name this will generate a list of the values for a single column. This is often used as a first step when visualizing CSV data.

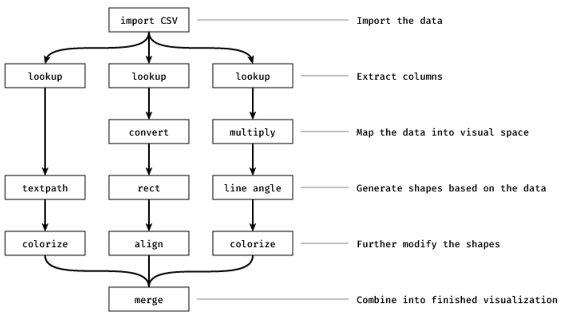

import CSV
Import the data
lookup
lookup
lookup
Extract columns
convert
multiply
Map the data into visual space
textpath
rect
line angle
Generate shapes based on the data
colorize
align
colorize
Further modify the shapes
merge
Combine into finished visualization

Figure 59: Typical NodeBox 3 visualization process

To visualize **categorical data** (e.g. types of accidents) we first figure out the distinct values of a column (using the `distinct` node), then create a *lookup table* by associating them with a specific color or shape using the `zipmap` node (this terminology is inspired by Clojure's standard library). Then we can use this lookup table to transform our data, for example by using color (see Figure 60).

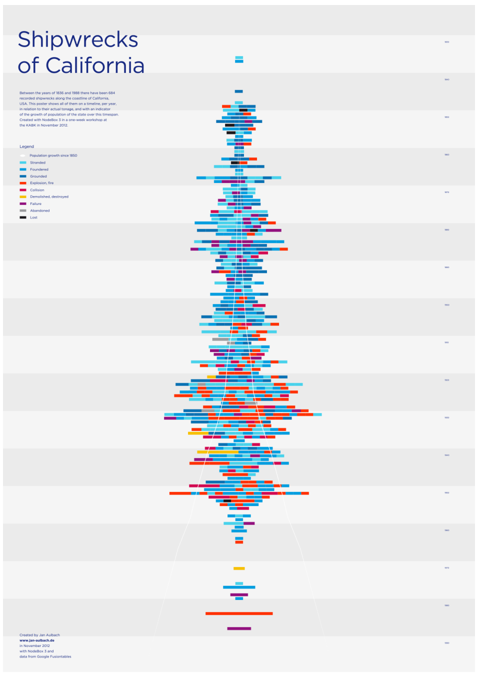


Figure 60: Example showing categories. "Shipwrecks of California", Jan Aulbach, 2012

Over time we've established a pattern (*Alexander*, 1977) that demonstrates how visualizations in NodeBox 3 work (see Figure 59). By teaching this both in workshops and in online learning materials, we can make users more efficient and self-reliant (G1, G4).

4.4.6 Building Abstractions

When projects get bigger, programming systems offer methods of abstraction to split up the complexity and reduce duplicated functionality. In imperative programming, *procedures* are a method of abstraction. In object-oriented programming this role is served by *classes*. In NodeBox 3 this method is called a *subnetwork*. Subnetworks group sets of nodes together into their own isolated space. Once grouped together, subnetworks function similar to built-in nodes (see Figure 61).

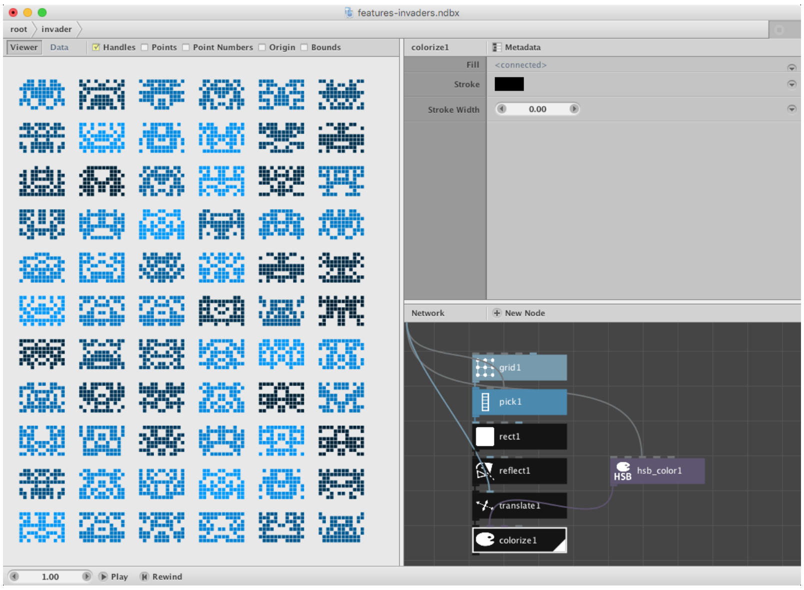


Figure 61: Example of a subnetwork describing a single space invader. Once created, different variations can be produced.

Being able to group together nodes allows users to build complex visualizations faster and with less duplication of effort (G1).

4.4.7 Visual results

Goldberg Variations

This project was created during the 2012 workshop at the Royal Academy of Art in The Hague (NL). It showed the famous works of Bach visualized in NodeBox. The data is based on the MIDI files of the works, using a custom node to import the midi data into NodeBox (Figure 62).

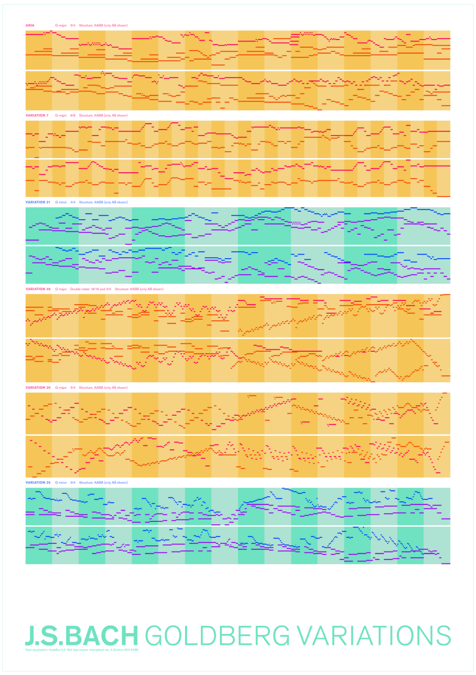


Figure 62: "Goldberg Variations", Anton Sovetov, 2012

Eurovoices

This visualization, created during the 2013 Vilnius workshop, shows the predictability of the voting process of the Eurovision contest. What emerges naturally from this is the typical “block voting” where countries vote for other countries they like out of political reasons, not necessarily because of the quality of the song (Figure 63).

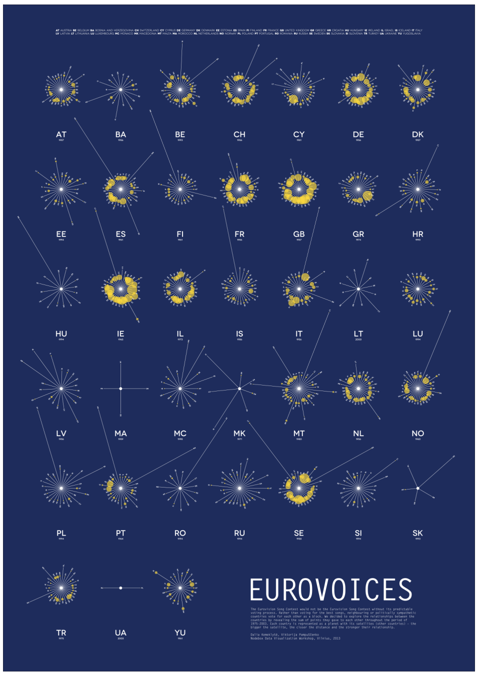


Figure 63: "Eurovoices", Dalia Kemeklytee and Viktorija Pampuscenko, 2013

Fifty years of Film Speech

This project was created during the same 2013 Vilnius workshop. It shows the frequency of the 30 most common words used in english films from 1952 to 2012. In it, you can trace the importance of words like "family" or "sorry".

The visualization was based on over 300 MB of subtitle data. This was then filtered down and aggregated. The designers used NodeBox to generate custom graphs in the shape of sound waveforms, each representing the usage of one word through time (Figure 64).

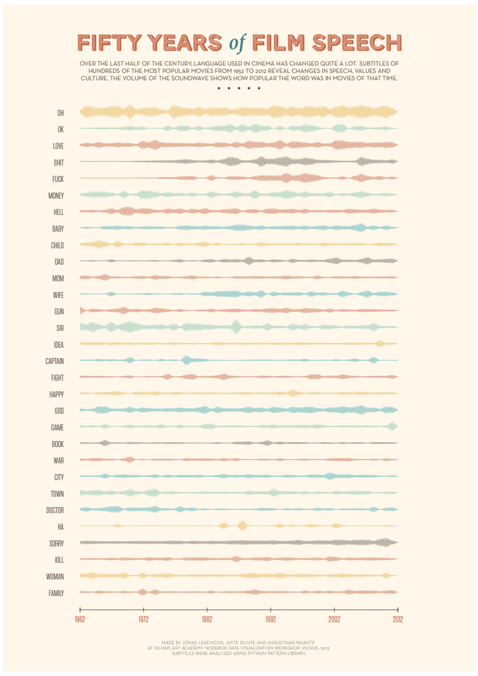


Figure 64: "Fifty Years of Film Speech", Jonas Lekevicius, Juste Ziliute, Augustinas Paukste, 2013

4.4.8 Evaluation

Just as with previous version of NodeBox we used this version extensively during workshops. We observed how students used the application. We conducted in depth interviews with some of the participants. At the end of the workshop we asked users to fill in a usability questionnaire.

Node-Based Interface

Almost no users had worked with NodeBox 2. They compared their experiences with traditional software, or with programming (minority).

Users remarked that the node-based interface allowed them to produce results quickly and change them easily (G1, G3). They found the architecture of the program to be logical, and found that it provided a unique way of thinking, unlike any other application. They remarked that it could easily process large amounts of data. Some users familiar with coding preferred this approach over textual programming (G4).

After initial training they found the node approach logical and found the functions of the nodes clear and helpful. They appreciated the visual interface of the software and remarked that it looked less intimidating than other applications.

Visual Output

Users remarked that through the program they could create unique images that they couldn't produce in Adobe Illustrator or other applications (G2). They found the program fun to play with because they could experiment and get unexpected effects. They liked that some "mistakes" produced cool-looking results.

Debuggability

The visual functional node-based approach makes debugging easier. Users can visually inspect the output of every node in the network, similar to looking at the output of every machine in an assembly line.

We taught a debugging technique during workshops:

1. When the output is not what is expected, examine every incoming node connection.
2. If the previous node is correct, the problem has to be in the current node.

We observed that users could solve the majority of issues using this simple debugging techniques. This made users more self-reliant and reduced workload for teachers during the workshop (G4).

The Right Node

Users remarked that the hardest part was knowing which node to pick next. This correlates with Spohrer's findings on plan composition *(Spohrer, 1986).* In addition, the wording "knowing which node to pick next" indicates a linear step-by-step approach instead of a "big picture" approach, similar to Dreyfus' assessment of advanced beginners *(Dreyfus, 1980).*

Plan composition is *essential complexity*: it is knowing how to convert the problem domain to the solution domain. Gradually building up this skill makes users more competent and self-reliant (G4).

In NodeBox this requires knowledge of all the nodes, what they do and where they can be used. During courses we provide a wide range of task-based tutorials and examples. Each of these will demonstrate a pattern of composing nodes that can be used when solving related tasks (see Figure 59).

4.4.9 Lessons Learned

Different Methods for List Matching

Test versions of NodeBox 3 used *shortest list* matching. To support cycling we introduced the concept of "infinite" lists based on sequences, much like Clojure's lazy sequences (see Section 4.1.1.1). Since their size is unbounded they would always be larger than the input coming from e.g. a CSV file. However, unlike in Clojure, users constantly modify the network, connecting and disconnecting nodes, resulting in intermediate states where only infinite lists were connected to a node. This caused the system to try to process infinite lists. We tried to solve this by having a configurable object limit in the viewer. Ultimately, we decided this was too much of a workaround and we switched to longest list matching.

Off-by-One Errors

List matching can introduce subtle off-by-one errors. Imagine we have a list of 20 preset colors we cycle over. Now imagine we have a list of monthly sales. Normally, we'll have more than 20 sales, so the matching won't be a problem. But if we have less than 20, the list of colors will be the longest list, and we will show some sales multiple times. This exact issue has come up in workshops before and can cause visual artifacts that are hard to spot (see Figure 65).

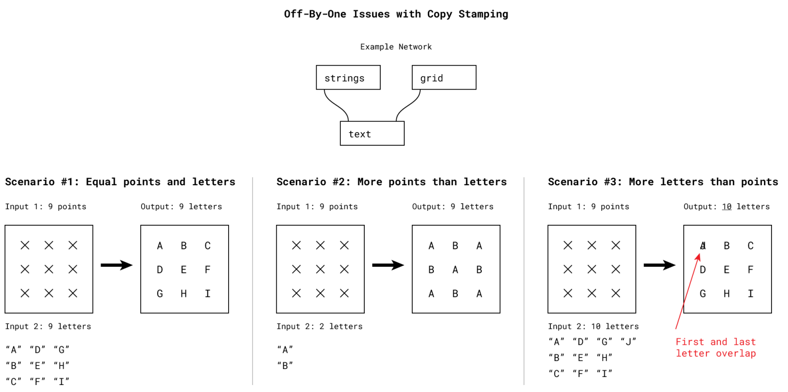


Figure 65: *Example of off-by-one error. Note overlap in last scenario.*

Subnetworks as Abstractions

We introduced subnetworks as an abstraction to split up complex networks and to support unwrapping of nested data structures. However, the only method for re-using subnetworks in a different part of the project (or other projects) is to duplicate them. This violates the *Don't Repeat Yourself* principle (*Hunt*, 2000). By duplicating networks we have multiple independent copies that need to be kept in sync. Changes to one version need to be manually carried over to the other version. This hampers the ability to build clean abstractions and harm efficiency in larger projects (G1).

Custom Code

Although NodeBox 3 added support for integrating external code, we didn't include a built-in editor for this code. We argue that this is a mistake. It made it less direct for users to create their own custom functionality when they needed it, instead opening up another program and linking the code there.

Another disadvantage is that users were not able to look at the implementation of the built-in nodes. This means that even if you were able to write Python, you couldn't learn from examples how to call the NodeBox APIs.

We solved this in NodeBox Live by extracting the "runnable" aspect of both node networks and code into a uniform concept which we called the *function*.

The Web

One common feature request was to be able to publish projects made in NodeBox on the web. Because the application was written in Java there was no simple way to export this to JavaScript. We experimented with rendering on the server but this could not be used for real-time animation.

We then created a version that had an export option where all nodes had a double implementation, both in JavaScript and in Java. We found that this was tedious for two reasons: one, we had to maintain two different versions and keep them in sync. Two, because the desktop and web platforms have fundamental differences it was sometimes impossible to port the node as-is to the web. We decided to port not just the nodes, but the entire application to the browser and called it NodeBox Live (see Section 4.5).

4.4.10 Conclusion

NodeBox 3 retains all the advantages of NodeBox 2 in terms of ease of use. In addition the "list matching" model improved understanding of complex networks and improved debuggability. Because list processing was a core principle built in to the software, it made complex visualizations and generative designs possible. Subnetworks provided a useful approach for working with nested data.

NodeBox 3 is in active use today. A number of challenges that came up (list matching, needless copying of subnetworks) can be mitigated in the current version.

To support web export, we decided to port the entire application, including the user interface to the web. That version is called NodeBox Live.

4.5 NodeBox Live

NodeBox Live is a web-based version of NodeBox 3. The model is functionally identical to NodeBox 3, with a few important exceptions: subnetworks are replaced with functions, code and nodes can be used interchangeably and master lists prevent list matching errors.

4.5.1 List Matching and Master Lists

Our studies and observations showed that users preferred the "list matching" approach over the "copy stamping" approach. Users could build projects faster and with less help than in NodeBox 2 (G4). We've adopted the same list matching engine in NodeBox Live as well.

One issue that came up in NodeBox 3 was that it wasn't always clear how lists would be matched. Users had a preconceived notion in their head of which list would be the "longest list". For example, when working with external data, they assumed that the rows of data would be longer than their list of chosen colors. However, sometimes due to data filtering or user error, that list of colors *would* become the longest list, resulting in visual bugs that were hard to spot and resulted in inaccurate visualizations (see 4.4.9.2).

In NodeBox Live we fixed this by introducing the concept of a "master list". Instead of the system deciding which list to cycle by looking at their sizes, users explicitly set which list is the longest. The final output will never have more items than this master list, and it will never wrap. This helped in debugging visual errors at the cost of introducing extra complexity (G4).

4.5.2 Subnetworks vs Functions

In NodeBox 3 we introduced subnetworks as an abstraction. However we saw in Section 4.4.9.3 that subnetworks could not properly be reused, reducing efficiency in larger projects.

In NodeBox Live, we solved this issue by making *functions* the main abstraction. Functions bundle functionality, just like subnetworks do. However networks use *instances* of these functions so that when the function implementation changes, it is picked up everywhere that function is used. This promotes reuse and improves efficiency (G1).

4.5.3 Nodes + Code

Designing the core abstraction around *functions* made it possible to include JavaScript functions as well. We designed networks so that they could be called in the same way as regular JavaScript functions (multiple arguments in, single argument out). This allowed users to "mix and match" functions based on code and networks. In addition to a network editor, we also integrated a code editor so that JavaScript functions could be created and modified from within the application. This allowed novice users to pair up with users proficient in programming (G4).

Projects can include external JavaScript libraries as well. We used this mechanism to include the Box2D physics library for a workshop project (see Figure 66). By allowing users to extend the program through external libraries they can expand its possibilities, providing them with more functionality and thus broaden their approach (G3).

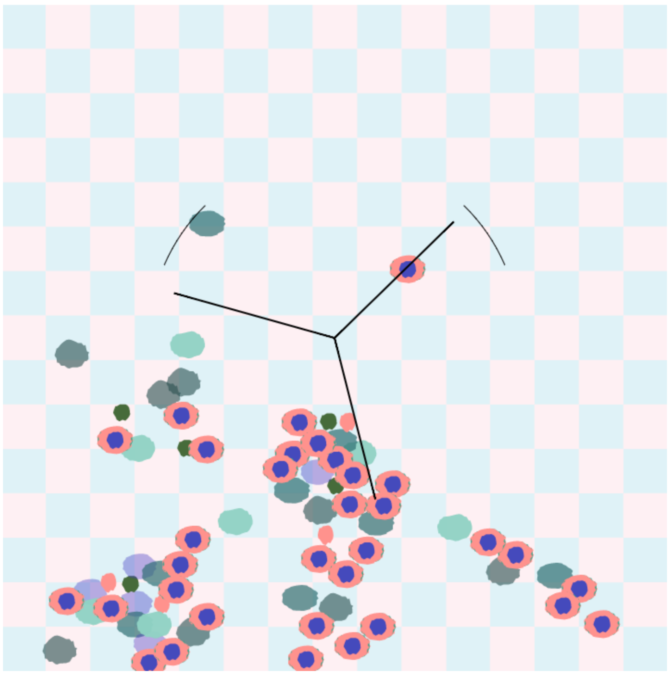


Figure 66: "Raging Revolving Door", Quinten De Meester, 2015

4.5.4 Web Application

As a web application, NodeBox Live can import data from web resources using JSON or CSV. Projects can be exported to web-compatible formats such as SVG and GIF. Projects can also be embedded on other web pages using `iframes` (similar to embedding YouTube movies).

NodeBox Live also exposes a JavaScript API. It allows users to start and stop the animation or change the parameters values of the network (these changes are just applied in memory and not saved). Figure 67 shows an example where conference interactions on Twitter could be explored using an interactive interface.

We provided two specific ways of interaction targeted to two levels of the *Dreyfus* model (see Section 2.6.5). Advanced beginners will likely want to integrate their project on a blog or portfolio website with minimal interaction. There they use the same method as used for embedding YouTube videos. Competent and Proficient users will want to have control over the behavior of the project, and will likely use the JavaScript API.

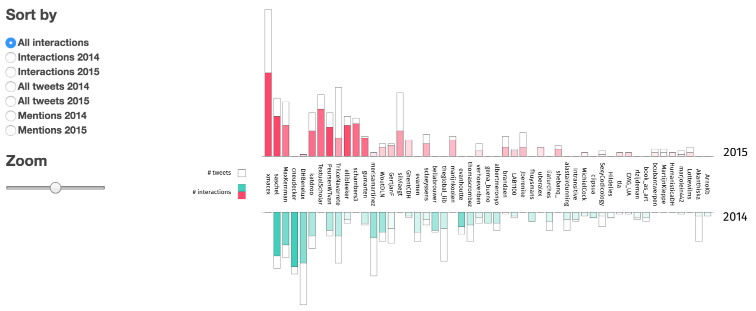


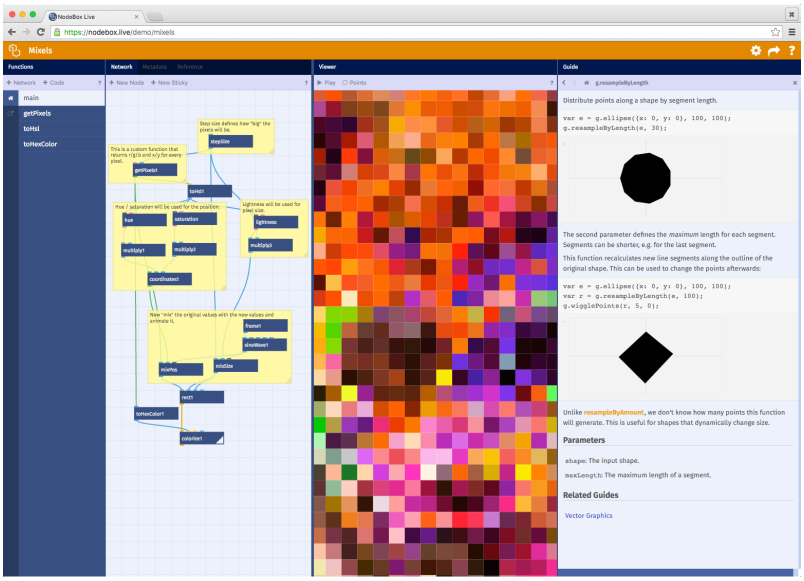
Figure 67: "Digital Humanities Benelux Tweep Interaction", Ben Verhoeven and Florian Kunneman, 2015

This solves one of the big issues with NodeBox 3, namely that projects could only be viewed as static images or non-interactive animations (see Section 4.4.9.5). Embedding allows users to integrate their projects into existing websites or blogs, without having to port the project to a different system (G1). It allows them to build interactive projects that they couldn't do before (G2). And because NodeBox Live projects are built on web standards, all their knowledge of the web is transposable (G4).

4.5.5 Integrated Help

Because our application ran on the web we had the opportunity to integrate the help pages directly into the application (see Figure 68). Every node has a "help" button showing a description and examples. This makes users more self-reliant (G4).

In addition to a reference, the help system provides a categorized listing of all nodes and "thematic" pages that contain patterns for common tasks (e.g. data visualization). This helps in plan composition (see Section 2.6.4).


NodeBox Live
https://nodebox.live/demo/mixels
Mixels
Functions
+ Network + Code
main
getPixels
toHsl
toHexColor
Network
Metadata
Reference
+ New Node + New Sticky
Step size defines how "big" the pixels will be.
stepSize
This is a custom function that returns r/g/b and x/y for every pixel.
getPixels1
toHsl1
Hue / saturation will be used for the position.
hue
saturation
multiply1
multiply2
coordinates1
Lightness will be used for pixel size.
lightness
multiply3
Now "mix" the original values with the new values and animate it.
frame1
sineWave1
mixPos
mixSize
rect1
toHexColor1
colorize1
Viewer
Play
Points
Guide
g.resampleByLength

Distribute points along a shape by segment length.

```
var e = g.ellipse({x: 0, y: 0}, 100, 100);
g.resampleByLength(e, 30);
```

The second parameter defines the *maximum* length for each segment. Segments can be shorter, e.g. for the last segment.

This function recalculates new line segments along the outline of the original shape. This can be used to change the points afterwards:

```
var e = g.ellipse({x: 0, y: 0}, 100, 100);
var r = g.resampleByLength(e, 100);
g.wigglePoints(r, 5, 0);
```

Unlike resampleByAmount, we don't know how many points this function will generate. This is useful for shapes that dynamically change size.

Parameters

`shape`: The input shape.

`maxLength`: The maximum length of a segment.

Related Guides

Vector Graphics

Figure 68: *Screenshot of NodeBox Live with built-in help system*

4.5.6 Visual results

BA4ALL Dashboard

In 2014 we presented at the "Business Analytics 4 All" meeting, a conference on analytics and data visualization for business professionals. We developed a case where we analyzed tweets mentioning Telenet, Belgacom or Voo, the three biggest telecommunications operators in Belgium. For each tweet we calculated the sentiment score, estimated gender and extracted keywords using Pattern (*De Smedt*, 2012). We made a number of visualizations that could be parametrized based on the telecom operator. We then combined all visualizations into an online dashboard (Figure 69).

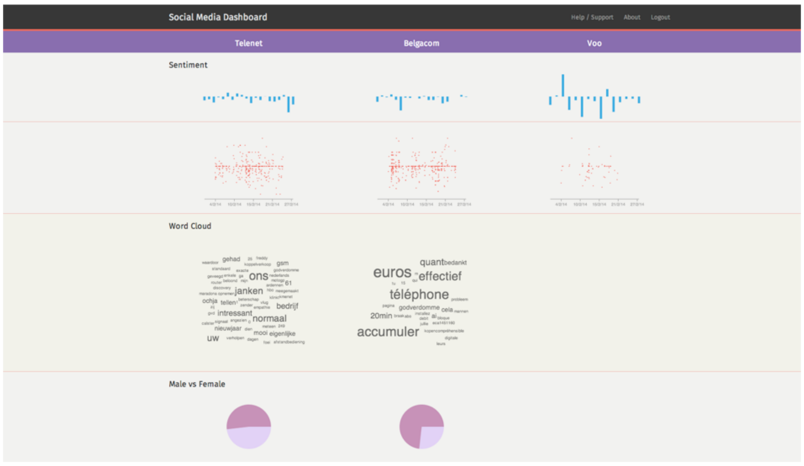


Figure 69: "ba4all Dashboard", Frederik De Bleser and Tom De Smedt, 2014

Metashape

Metashape is a work based on recursive application of the superformula (*Gielis*, 2003). The work was displayed at the 2013 Incubate conference in Tilburg (NL) (Figure 70).

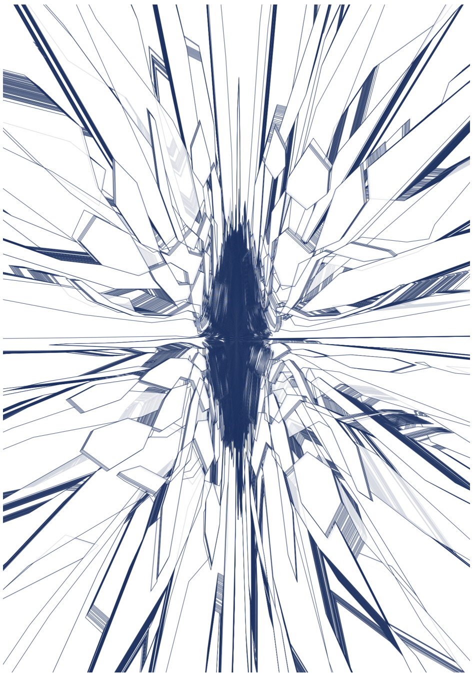


Figure 70: "Metashape", Frederik De Bleser, 2013

4.5.7 Evaluation

We were able to teach preview versions of NodeBox Live during a number of workshops. We observed students and asked them to fill out the usability questionnaire (see Section 2.6.1).

Since the system was built on NodeBox 3, many of the usability remarks for NodeBox 3 were similar. Users liked the strong underlying model for data visualization. They found navigation easy and intuitive. They remarked that the system was a nice alternative to programming.

Function Abstraction

Most *novice* users were unaware of the fact they could write JavaScript code. However, they saw the potential once we showed them tutorial material.

More *proficient* users with experience in JavaScript appreciated that they could easily extend the application with their own functionality. For example users added functionality to work with the current system time. They remarked that it was "just JavaScript" and that they didn't have to find "work-arounds" to let the system do what they wanted (G4).

Integrated Help

Users appreciated the built-in help system. They found it useful to have contextual help both for the interface as well as for the nodes (G4). Some remarked that it could be made more comprehensive.

Web Integration

Users liked the integration with the web. They remarked that they now were able to publish interactive projects on their own site or blog. However users had conflicting thoughts on the centralized web environment. Some users liked that the software was entirely online and that they didn't have to install anything, while other users didn't want to trust their projects to a web application.

4.5.8 Lessons Learned

Undo System

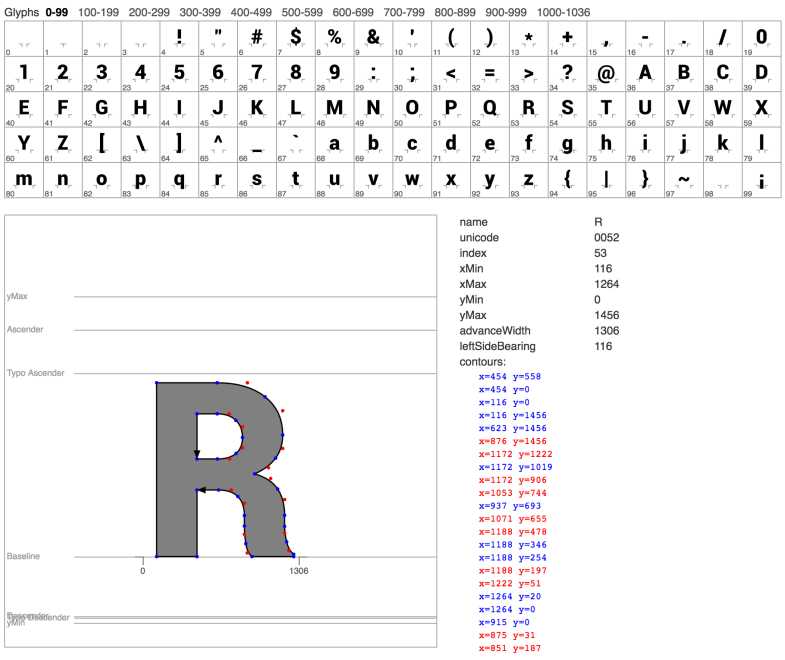
Users complained about a lack of an undo system. Since the system runs online, and can potentially have multiple users editing on the same page, developing a comprehensive undo system is not yet implemented. A properly multi-user real-time system would probably need operational transforms (*Ellis*, 1989).

We are currently designing the first implementation of an immutable data model, similar to that of NodeBox 3, which would help with single-session undo.

4.6 OpenType.js

One hurdle we faced when porting NodeBox Live to the web was good support for typography. Projects like "Evil Font" (Section 3.5.1.1) require access to the outlines of the *glyphs*, the letter shapes of the font. In NodeBox 3 we accessed these glyphs through the Java 2D API. However, HTML5 does not expose a similar API, and no third-party libraries implementing this functionality were available.

We decided that good typographic support was essential for a generative design tool. This led us to the design of OpenType.js, a JavaScript library that can parse and write TrueType and OpenType fonts (*De Bleser*, 2014). The library provides users with low-level access to the outlines of the glyphs and the Unicode-to-glyph mapping. It also provides high level access to convert a string of text to a Bézier path. This high-level functionality is integrated in NodeBox Live using the `textpath` node. The library also provides export functionality to save fonts back out to an OpenType font file (Figure 71).



name	R
unicode	0052
index	53
xMin	116
xMax	1264
yMin	0
yMax	1456
advanceWidth	1306
leftSideBearing	116

contours:

```
x=454  y=558
x=454  y=0
x=116  y=0
x=116  y=1456
x=623  y=1456
x=876  y=1456
x=1172  y=1222
x=1172  y=1019
x=1172  y=906
x=1053  y=744
x=937  y=693
x=1071  y=655
x=1188  y=478
x=1188  y=346
x=1188  y=254
x=1188  y=197
x=1222  y=51
x=1264  y=20
x=1264  y=0
x=915  y=0
x=875  y=31
x=851  y=187
```

Figure 71: OpenType.js Glyph Inspector, showing outline of the "R" glyph

The project is used in a number of popular open-source and commercial tools such as Prototypo (*Mathey*, 2014), FontSelf (*Hoffman*, 2015), Metapolator (*Egli*, 2013) and p5.js (*McCarthy*, 2014).

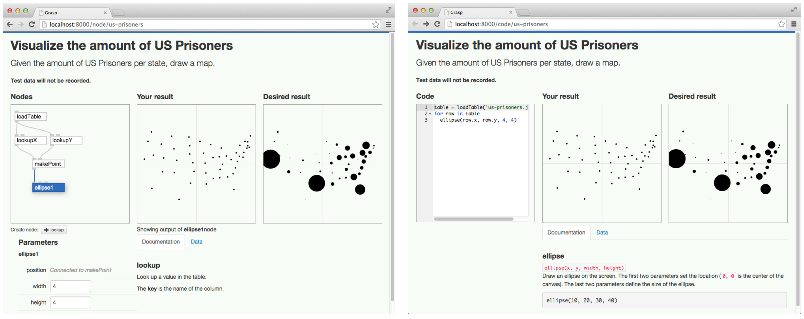
4.7 Comparing Visual and Textual Programming

To evaluate program comprehension we looked at the differences between textual programming tools (such as Processing (*Reas*, 2007) or D3 (*Bostock*, 2011) and visual programming tools like NodeBox 3 or NodeBox Live. There are some empirical studies by Green (*Green*, 1991) (*Green*, 1992) that compare the usability of visual programming languages to textual languages. In general they have not favored visual programming languages. These studies all focus on programmers or engineers solving mathematical problems, such as the calculation of rocket trajectories (*Green*, 1996).

Our study examines if graphic designers, who by their nature work in the visual domain, respond differently to visual programming tools than engineers.

4.7.1 Experimental Setup

We developed a web-based application called *Grasp* that provides participants with a series of exercises bundled together in a survey (see table in Section 4.7.5. Each exercise exists in both a textual and visual programming environment. The environments are modeled on Processing and NodeBox 3 respectively. To avoid bias both environments are different only in the aspects under test. They share the same look, graphical engine and coordinate system (see Figure 72).



Visualize the amount of US Prisoners

Given the amount of US Prisoners per state, draw a map.

Test data will not be recorded.

Showing output of **ellipse1**node

Parameters

ellipse1

position	*Connected to makePoint*
width	4
height	4

lookup

Look up a value in the table.

The **key** is the name of the column.

Visualize the amount of US Prisoners

Given the amount of US Prisoners per state, draw a map.

Test data will not be recorded.

```
table = loadTable('us-prisoners.j
for row in table
  ellipse(row.x, row.y, 4, 4)
```

ellipse

`ellipse(x, y, width, height)`

Draw an ellipse on the screen. The first two parameters set the location (`0, 0` is the center of the canvas). The last two parameters define the size of the ellipse.

```
ellipse(10, 20, 30, 40)
```

Figure 72: The Grasp testing environment showing the textual and visual programming interfaces.

Before they begin the exercise, students fill in a questionnaire asking them to rate their skill level using the 5-point Dreyfus scale (see Section 2.6.5). After that they are shown either a textual or visual environment. Here they are asked to perform a series of operations that will take the system from an initial state (with some existing code) to a desired state. Controlling the initial state means we can focus on the specific concept participants have to comprehend before they move on. For example, the *color the squares* exercise teaches that the system draws shapes using the most recently specified fill color. Participants do not have to know how to draw the shapes since they are already provided, but they do have to think about the ordering of the statements. Finishing the exercise means participants have to understand the concept of graphics state (*Press*, 1985), even though they might not know the concept by its name.

This method is called *gated exercise*. It requires someone to understand a certain concept before he or she can solve it. In other words, the *gate* only opens once a user discovers a certain key – only the key is not a physical object, but a mental concept in their head. A number of puzzle games, notably Portal, use this principle to guide player progression.

We detect if an exercise is finished by comparing the participant's output with the desired output, pixel by pixel. We do not look at the resulting programming code at all, which means any answer is accepted as long as the output matches exactly. If participants are stuck they can *skip* an exercise, allowing them to continue with the next exercise.

At the beginning of the survey participants are asked some demographic questions about their experience with graphic software and coding. Then they are randomly assigned to the textual or visual programming environment and will not come into contact with the other approach. Surveys are conducted with a minimal amount of initial instructions. No outside help is given and students are discouraged from helping each other.

4.7.2 The Programming Environment

The textual programming environment uses CoffeeScript, a procedural programming language resembling Python that compiles down to JavaScript. The visual programming environment uses a dataflow computational model and a box-and-wire representation.

The exercises require the use of variables and loops but not conditionals. There is a small amount of reference documentation on the commands or nodes relevant to the exercise. Data visualization exercises that require outside data show the table containing the numbers.

4.7.3 Concepts in Graphic Programming

Graphic programming requires comprehension of some basic concepts: users should understand the coordinate system, the graphics state (for textual programming), draw order, loops, mathematical expressions and connecting data to visual aspects like position or size. Each exercise tests the comprehension of a separate concept and works as a symbolical gate", opening only when they grasp the concept. Exercises were designed so that it was hard to complete them by random chance.

4.7.4 Survey Details

We detail the results of multiple surveys:

- One survey held in Montréal in April 2013, with 23 students
- One survey held in Valence in October 2013, with 17 students

All raw survey data is available as a MongoDB database dump (*De Bleser*, 2013).

We asked participants to assess their software skills according to the Dreyfus scale into 5 levels, from novice to expert (*Dreyfus*, 1980). On average, participants rate themselves as competent / proficient with traditional graphic design applications (avg. 3.24), as novices with textual programming (avg. 0.57) and having no experience with visual programming (avg. 0.09). These numbers indicate that results will be slightly skewed in favor of textual programming languages.

4.7.5 Results

The results show that a textual programming environment is faster in almost all exercises (see Table 73 and Figures 74 and 75). If we disregard uncompleted exercises due to skipping, participants using textual programming were able to complete the survey in 33.80 minutes and those using visual programming in 38.43 minutes. In other words, **students using code are almost 15% faster.**

Exercise	Concept	Textual	Visual
Align the squares	Coordinate system	02:08 (s=01:23)	02:48 (s=01:44)
Color the left square	Shape modification	03:27 (s=04:13)	04:46 (s=03:20)
Order the squares	Layering	01:59 (s=01:41)	01:30 (s=01:21)
Move the lines	Mathematical expressions	12:40 (s=10:29)	06:19 (s=05:48)
Double the elements	Iteration logic	02:22 (s=02:24)	01:51 (s=01:19)
Visualize points on a map	Connecting data to visual aspects	06:28 (s=05:37)	12:07 (s=10:09)
Visualize information in a bar chart	Connecting data to visual aspects	04:44 (s=05:26)	09:05 (s=06:43)

Figure 73: An overview of the Grasp exercises, average durations, in minutes, and confidence interval (CI)

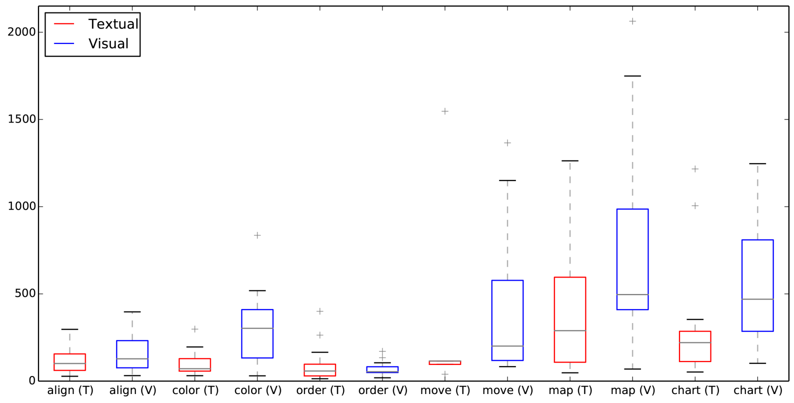


Figure 74: Duration per exercise in seconds

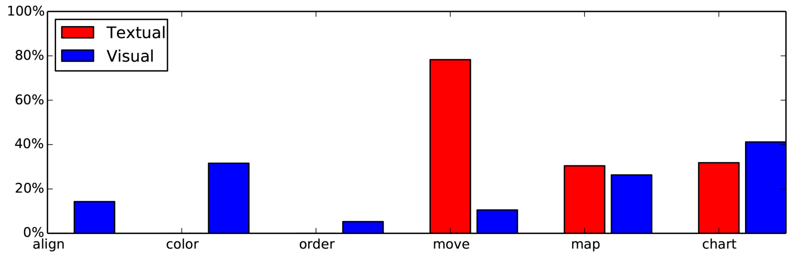


Figure 75: Amount of skips per exercise

For textual programming, exercise 4 ("move the lines") was a clear outlier in difficulty. This exercise required expanding a mathematical expression to use subtraction (from `i * 5` to `i * 5 - 100`). Only 8 out of 40 participants were able to complete the exercise, whereas 19 out of 35 visual programming participants were able to complete the same exercise.

A possible reason could be the "implicitness" of math formulas in textual programming. In visual programming each mathematical operation is represented by a node: this unifies operation ordering and makes it deliberate. In textual programming, math expressions are special, following different ordering rules than function syntax. We observed textual programming participants struggle with the ordering of the mathematical operations, randomly trying out all kind of permutations without finding the answer. It seems that visual programming has a clear benefit here, even though (or perhaps because) math expressions are not as terse.

4.7.6 Conclusion

According to our test a textual programming approach is significantly more efficient than a visual programming approach. For almost each exercise, participants using textual programming completed the exercises faster than participants using visual programming. These findings correspond with results by Green *(Green, 1991)(Green, 1992).* With regard to programming, graphic designers do not think differently from engineers.

However, because participants had some experience with code, and almost none with visual programming (and its dataflow programming model), we expected some learning curve as students struggle with the new paradigm. Remarkably after a few exercises students are already more self-reliant (G4) with regard to mathematical expressions, being able to solve the problem in visual programming but not in textual programming. We conclude that this is a benefit of a unified model where every operation is explicit, requiring deliberate thinking and not randomly shuffling mathematical terms.

Additionally, in our week-long workshops with visual and textual programming languages, anecdotal evidence shows that students using visual programming are much more self-reliant (G4). Further research can determine whether this is because of the lack of syntax or the programming model (imperative vs. functional).

As an added benefit, we found Grasp not only a valuable testing tool but also a useful way of teaching programming to students. By carefully pacing the exercises and gradually layering concepts for comprehension, students retain a deeper understanding through frustration and delight than by being told all the answers.

4.8 Conclusion

In this chapter we proposed a visual, functional approach as an alternative to textual programming. We presented multiple iterations of NodeBox, each with improvements based on real-world feedback. Overall we demonstrated that users could produce remarkable generative designs and data visualizations that they could not produce with traditional applications, without (textual) programming. We argued that a visual functional model helps in debugging complex networks, and removes a lot of accidental complexity associated with syntax and state.

We found that when we tried to take away as much accidental complexity as we could, plan composition remains the primary challenge for generative design *(Spohrer, 1986).* Plan composition implies that, to reach a goal, a number of steps have to be taken in order to reach that goal. This idea is enforced by the metaphor of the assembly factory, where it is instrumental that the machines are connected in the correct order.

There exists a software paradigm that is not bound to the rules of order. Where describing the end result (the "what") is more important than the steps to get there (the "how"). That paradigm is constraint programming, discussed in Chapter 5.

5 Constraint Programming

This chapter will focus on constraint programming languages. We'll introduce constraint programming and examine how it is used for generative design. Then we'll introduce our own contribution: Logic Layout.

5.1 Introduction

Constraint programming is a programming paradigm that describes the relations between variables in the form of constraints. It is a programming paradigm that describes the logic of the computation (the "what") without describing its control flow (the "how"). Instead of listing the sequence of steps the program has to perform (such as in imperative programming), in constraint programming we describe the required constraints the program has to fulfil to be accepted as a solution. As an example, to describe a page layout with constraint programming we would describe the relationships between elements on the page: their relative sizes, positions and ordering. Contrast this with an imperative programming where we need to explicitly position elements and the ordering of the program decides the layer order.

Constraint programming is a form of declarative programming. Declarative programming is an umbrella term used to distinguish them from non-imperative programming languages. It includes functional programming, constraint programming, domain-specific languages and logic programming. Since there is significant overlap between these paradigms some of the related work will fall under the more general term of declarative programming.

One problem that has permeated previous chapters is that of *plan composition* (see Section 2.6.4). Even if – ideally – all accidental complexity is removed, what remains is the composition of a plan. Constraint programming promises a method around that by describing the solution and letting the system figure out the plan on how to get there.

The ordering of the chapters implies that constraint programming is the next evolutionary step in programming and that all programs from hereon should be expressed as constraint systems. This is not our intent. Instead we will see how constraint programming can provide a solution to the four goals set out in our hypothesis. Constraint programs get rid of statement order, a common cause of errors in imperative programming (G4). Because we only define the properties of the end result and not the entire process, the system can generate a range of unexpected results (G2, G3). A well-defined constraint program can make many automated variations that we would otherwise have to make by hand (G1).

Constraint programming has been used in data visualization before (see Lyra, Section 5.2.5). The question we wanted to ask is: can this approach also be used in graphic design, where the variables are not connected to data, but to a subjective opinion of aesthetics?

5.2 Background / Related Work

This section will show examples of related work: both applications (Sketchpad, Lyra) and algorithms (Cassowary).

5.2.1 Design Decisions

Graphic design is essentially a search problem. It set out with a number of requirements, dictated by the client, technical limitations and context, that need to be met for the design to be accepted. Through a creative process, the designer will come to a possible solution that fits the requirements. This is one of many solutions: given a different designer, they will come up with a different solution that also fits the requirements.

This implies that a number of decisions in design can be arbitrary. As long as the requirements are met the design will be accepted. For example, a concert poster has a number of required elements: artist, date, location. As long as the position and size of these elements fall between certain ranges (e.g. so the text remains readable) and the relationships between elements are maintained (artist is more important than location) the exact placement of the elements is irrelevant.

In generative design, this principle defines which properties can be random (e.g. position, size, color) and which elements should be fixed (e.g. name of the artist). We used this technique to develop the dynamic logo of ChampdAction (see Section 3.5.3.3). Instead of defining the exact position of every line an algorithm randomly places lines, constrained by a maximum distance for each line. The algorithm generates random variations of the logo. If we notice that the variations are too chaotic, we can add additional constraints to control the process.

5.2.2 Constraint Programming and Accidental Complexity

Constraint programming can potentially reduce many complexities encountered in imperative and functional programming. For example, in the NodeBox 1 tutorial (see Section 3.3) we talked about the Painter's Model as a metaphor for thinking about how the computer draws elements on screen. Instructions that appear first in the code are drawn below later instructions. As a consequence the Painter's Model marries the *ordering* of statements with the *layering* of elements on the screen. This is an example of accidental complexity: we wanted to express the *relationship* between the elements, but are required to restructure our program to do it. Contrast this with constraint programming, where we can directly express a *is-on-top-of* relationship between the elements and let the execution engine figure out in which order it needs to draw. We'll explore this example more when we talk about the Cassowary constraint solver and Logic Layout.

However some problems are clearly imperative, and benefit from being expressed as such. For example if we have an automation job, the ordering of the steps in the job is predefined and the program should reflect that. A constraint program that describes the interdependencies between the job steps would only complicate the description. In contrast, some problems don't have a clear ordering. When we discuss layout there is no clear plan on which elements to place first; they are all interdependent. Therefore it is our desire to open up the discussion on constraint programming and contribute a practical implementation.

5.2.3 Critique of Constraint Programming

Because CPUs run procedurally but our programs are written declaratively, a transformation step is necessary. To be executed, constraint programs need to run through some kind of engine or solver (such as Cassowary, described below). This implies that the source program and what gets executed on the CPU differ significantly. This makes debugging and tracing of declarative programs difficult *(Pereira, 1986)(Shapiro, 1983)(Wu, 2005)(Freudenthal, 2010).* In other words, because of the large abstraction gap between the program and the machine results can be unexpected.

In exact sciences this property is undesirable. For designers and artists however, unexpected results and randomness are a constant source of inspiration. For example, Leonardo da Vinci used the structure of corroded walls as inspiration for landscapes (*Chastel*, 1990). That is why we speculate that this approach will be especially successful for creatives.

5.2.4 Sketchpad

Sketchpad (*Sutherland*, *1964*) was a revolutionary design application written by Ivan Sutherland in 1963 as part of his PhD thesis (see Figure 76). It introduced many concepts still used in human-computer interaction. It is considered the ancestor of computer-aided design (CAD) and computer graphics. The idea of the graphical user interface (GUI) was derived from Sketchpad.

Sketchpad allowed the user to constrain the geometric properties of the design. For example, you could indicate that two lines needed to be perpendicular, or that they needed to be equal length. The design would then automatically update, taking the constraints into consideration.

The major feature which distinguishes a Sketchpad drawing from a paper and pencil drawing is the user's ability to specify to Sketchpad mathematical conditions on already drawn parts of his drawing which will be automatically satisfied by the computer to make the drawing take the exact shape desired. – Sutherland

Sketchpad's approach of visually defining constraints guided the development of Logic Layout.

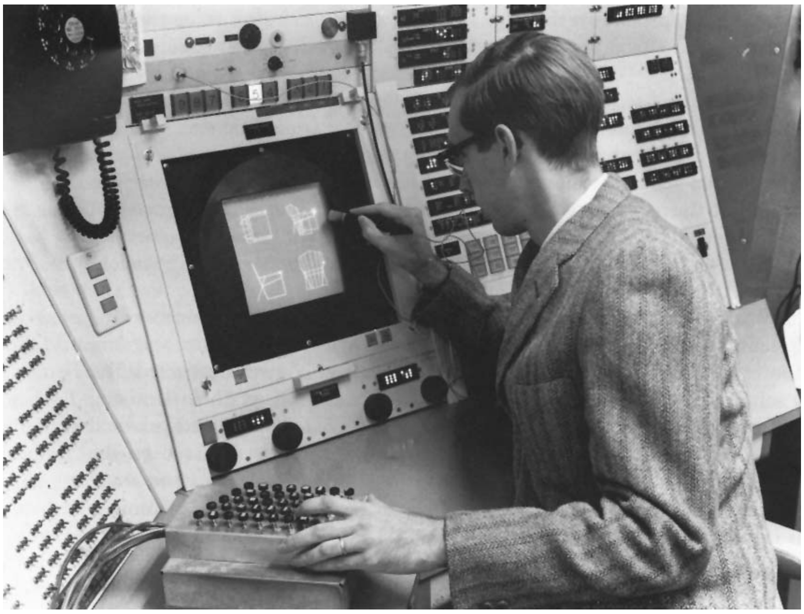


Figure 76: Dr. Ivan Sutherland using Sketchpad, 1963

5.2.5 Lyra

Lyra is an interactive environment for designing customized visualizations without writing code. (*Satyanarayan*, 2014) The design of Lyra is inspired by visualization grammars such as the Grammar of Graphics (*Wilkinson*, 2006), JavaScript-based parametric visualization libraries such as Protovis and D3, and data-driven drawing tools by Bret Victor (*Victor*, 2013).

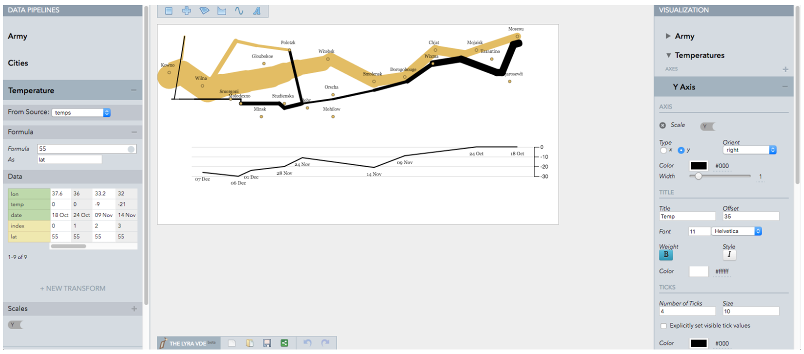


Figure 77: Screenshot of Lyra

Lyra follows the pipeline process described in the Grammar of Graphics. What makes Lyra unique is that it attempts to do this using a purely visual interface (see Figure 77). Visual shapes can be created by dragging and dropping them from the toolbar. Data can be bound to visual shapes by dragging it to the desired property. Lyra served as an inspiration for Logic Layout. Instead of binding shapes to data we researched how they could be bound to user-defined "aesthetic" parameters.

5.2.6 Cassowary constraint solver

Cassowary is an incremental constraint solving algorithm (*Badros*, 2001). It was designed to describe layouts in user interface applications, although it can solve any kind of linear system. Because GUIs frequently need to adapt to changes by the user, the algorithm can solve systems of constraints incrementally and efficiently. It uses the dual simplex method, a popular algorithm for linear programming (*Dantzig*, 1955). Constraints in Cassowary can either be requirements or preferences.

The Cassowary algorithm is used in Auto Layout, the layout engine for OS X and iOS, introduced in 2011 (see Figure 78). It also forms the basis of Constraint Cascading Style Sheets (CCSS), an extension of Cascading Style Sheets (CSS). Constrained CSS, and its successor Grid Style Sheets (GSS), has been included in The Grid, a commercial content management system. In Logic Layout we use the Cassowary algorithm to solve constraints between visual elements on the page.

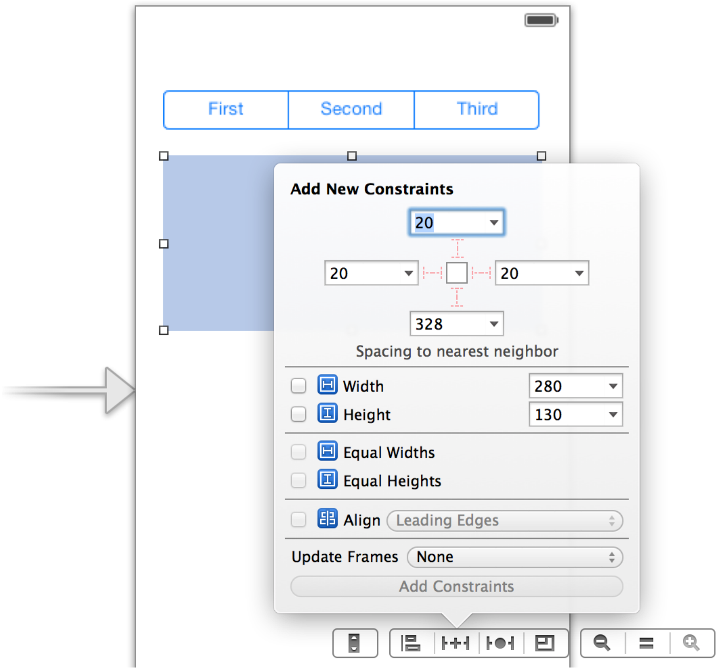
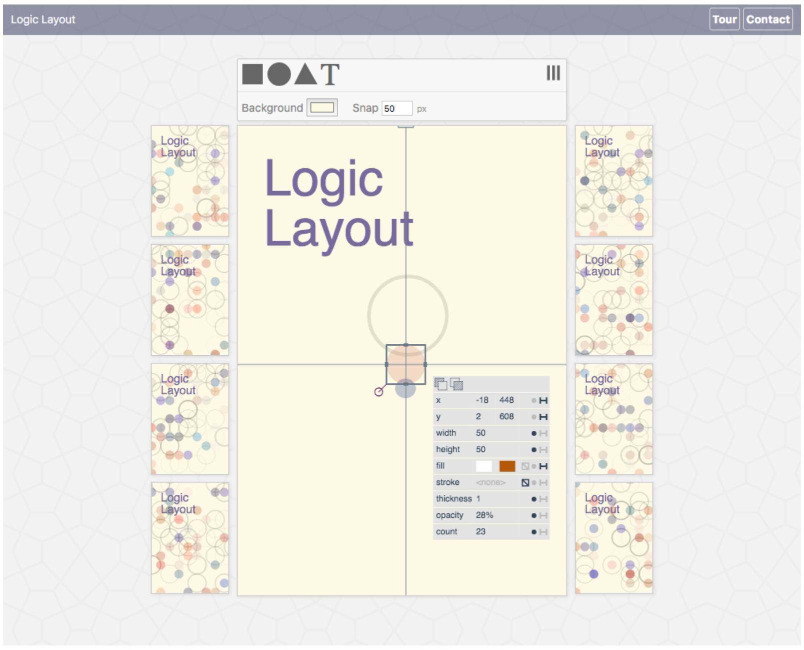


Figure 78: Screenshot of Auto Layout dialog box in Xcode

5.3 Logic Layout

Logic Layout is an interactive design application that uses ranges and constraints to create multiple design variations. It resembles a traditional what-you-see-is-what-you-get (WYSIWYG) design application like Adobe Illustrator. A toolbar contains a number of primitives that can be added to the page. Once added, elements can be dragged across the page and manipulated interactively or using a property panel. Relationships between elements can be created by connecting a line between them (see Figure 79).



Background Snap 50 px

x	-18	448	● H
y	2	608	● H
width	50		● H
height	50		● H
fill	⬜	🟧	● H
stroke	<none>		● H
thickness	1		● H
opacity	28%		● H
count	23		● H

Figure 79: Screenshot of the Logic Layout application

The aim of Logic Layout was to get rid of as many accidental complexity as possible. It no longer requires user to learn textual code syntax (Chapter 3) or come up with a plan (Chapter 4). Instead users place elements using direct manipulation and create constraints through drag and drop. Extra information about elements can be configured in a floating property panel (see Figure 79).

5.3.1 User Interface

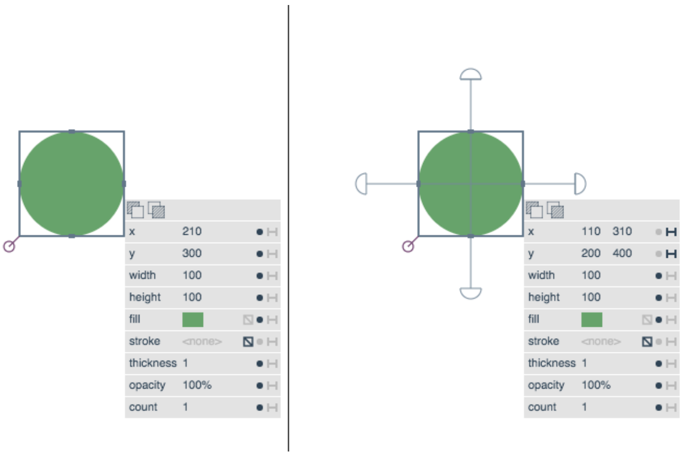
Logic Layout is divided in three logical sections. The toolbar on the top houses shapes that can be dragged onto the canvas, as well as global properties like background color and grid snapping. The central panel contains the viewer where users can directly manipulate graphical shapes. On the left and right side of the viewer are the *variations*, a random selection of possible solutions to the design problem. Because the starting scene doesn't use random ranges, these variations will be identical initially.

To create simple designs, users drag shapes from the toolbar onto the canvas. There they can interactive drag and resize shapes using handles. The property panel allows control over properties like fill color, text size or element ordering.

5.3.2 Random Ranges

In *fixed-mode*, Logic Layout works similar to a traditional vector graphics application like Adobe Illustrator. Properties can be modified either by dragging their handles, or by changing the values in the property panel. We call this *fixed-mode* because we specify a *single* fixed value for each property of the shape (e.g. the text size is 24 points).

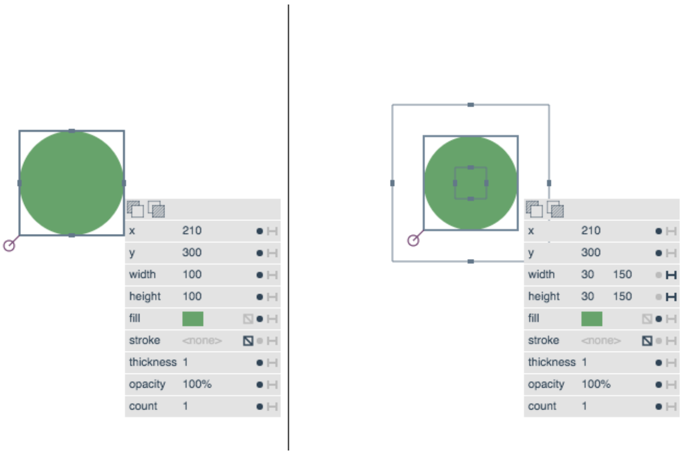
Each property in the system can be set to a *second* mode, called "*random range*". When set to this mode, we no longer specify a single, fixed value for the property, but a *range* of values where the property has to fall between. The system is to free to choose a random value inside of this range (see Figures 80 and 81).



x	210
y	300
width	100
height	100
fill	🟩
stroke	<none>
thickness	1
opacity	100%
count	1

x	110	310
y	200	400
width	100	
height	100	
fill	🟩	
stroke	<none>	
thickness	1	
opacity	100%	
count	1	

Figure 80: Fixed value vs random range. Draggable handles allow you to modify the range of positions.

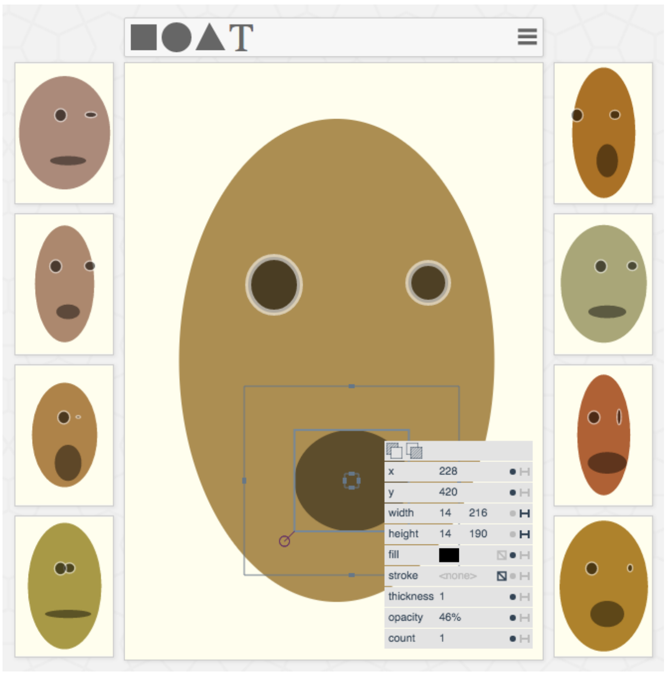


x	210
y	300
width	100
height	100
fill	🟩
stroke	<none>
thickness	1
opacity	100%
count	1

x	210	
y	300	
width	30	150
height	30	150
fill	🟩	
stroke	<none>	
thickness	1	
opacity	100%	
count	1	

Figure 81: Fixed value vs random width / height. The inner and outer box specify minimum and maximum sizes.

Because the element now has dynamic properties we no longer have a single determined outcome, but many possible variations. In each variation shapes will be placed in a different position or have a different size or color, or a combination of all of these (see Figure 82).



x	228	
y	420	
width	14	216
height	14	190
fill	■	
stroke	<none>	
thickness	1	
opacity	46%	
count	1	

Figure 82: "Random faces" created in Logic Layout

5.3.3 Element Count

When we specify a random range for the position of an element, it can appear in many positions, as shown in the variations canvases. By adding a *count* property to an element we're not limited to showing it a single time on the page. The shape can now appear on the page many times (see Figure 83).

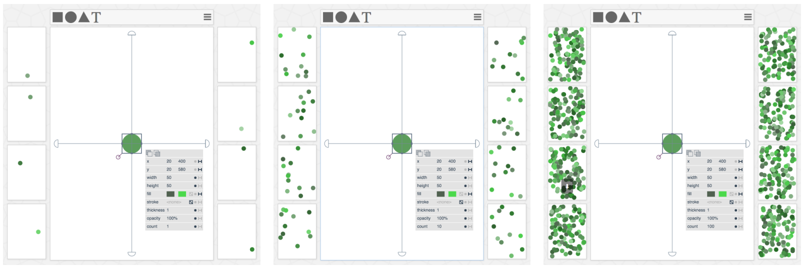


Figure 83: Element count of 1, 10 and 100

The count property can create many elements out of a single element. It turns this one-element composition into a *recipe* for a more complex design. In essence it is an *amplifier* for the manual process, a system that can repeat our actions while allowing for variations through random ranges.

5.3.4 Randomness Generation

The concept of randomness is a vital part in the system. Randomness allows for different variations to be produced. However we want these variations to be *stable*: they shouldn't change without parameters being changed. Because we re-render as much as 10 times per second, JavaScript's built-in random generator would cause the variations to flicker continuously, making it hard to judge a specific variation. In short, even though we need a source of randomness, we want to have a *deterministic* random process that returns the same pseudo-random results every time.

That's why we *seed* the random generator with the same value each time we run the render. We use an RC4-based pseudo random number generator (PRNG) which we seed with a different index for each variation. This causes changes to retain a certain amount of stability while still generating enough diversity for the different variations.

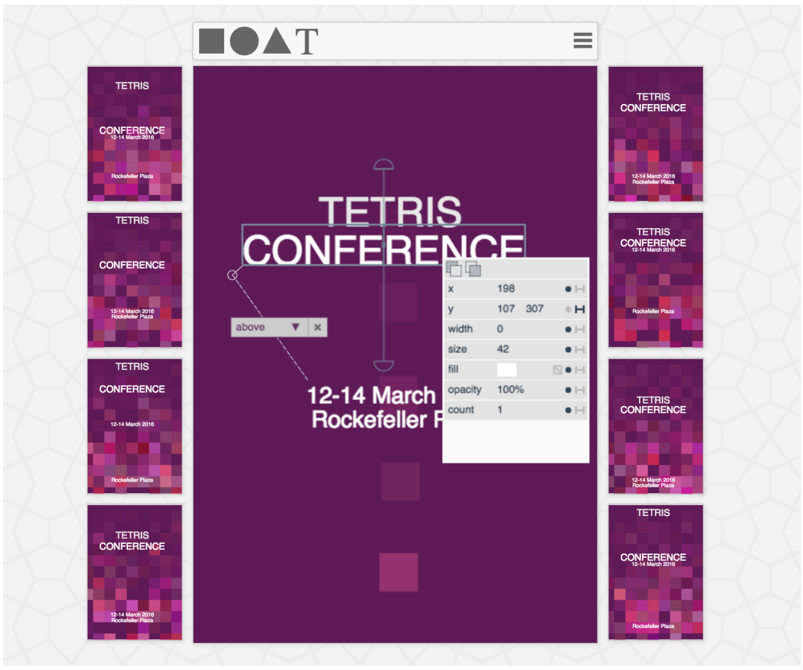
5.3.5 Constraints

So far the system we described deals with the properties of individual elements. However design is about exploring the *relationships* between elements on the page *(Hannah, 2002)*. Examples of those relationships are: text elements shouldn't overlap so they remain readable (see Figure 84). Or elements should follow a certain visual hierarchy: one element always needs to be bigger than another element.

Constraints allow us to specify these relationships between elements. They can indicate that one element needs to be the same size of another element, or that two elements should have the same vertical or horizontal position. It can refer to size, requiring that one element needs to be bigger than another element. It also can refer to color, specifying that elements need to be darker or lighter than their companions.

In Logic Layout we implement constraint solving using the Cassowary algorithm. By default, however, constraint solvers are designed to produce a single, optimal outcome. The solver uses a minimization or maximization procedure that gradually converges to an optimum. This is clearly undesirable in our application since we want many variations that all adhere to the given constraints. We solved this issue by specifying for each variation a random constraint that fell within the bounds of the given random range. We made this a *soft constraint* so that the system is not required to obey it. This solution gave satisfactory results.

The Cassowary algorithm is suitable for Logic Layout since it uses an incremental approach where it doesn't have to full re-solve the linear system for every change. As an additional speed optimization we make a distinction between properties that do not participate in a constraint relationship and elements that do. Only elements that use constraints are included in our constraint solver.



x 198
y 107 307
width 0
size 42
fill
opacity 100%
count 1

Figure 84: *Logic Layout – defining the "above" constraint*

5.3.6 Implementation

Logic Layout is implemented as a web application. It uses HTML, CSS and JavaScript. The system has no explicit backend. Instead, data is persisted in Firebase, a real-time "Database As a Service" (DAAS) that maintains a persistent connection through WebSockets (*Lubbers*, 2010). Apart from Firebase and *seedrandom*, a random number generator, no external libraries or frameworks are used.

The main canvas interface is drawn using the immediate-mode GUI paradigm (*Muratori*, 2005). This makes it possible to create a very dynamic user interface that reacts to complex user events (e.g. shift-clicking one object while dragging another).

Rendering of the variations happens in a separate thread using Web Workers. This keeps the application responsive even for large documents with many elements.

5.4 Visual Results

Our primary goal was to give the same level of ease of use and control you except in a traditional graphics application while augmenting the abstraction to introduce randomness, repetition and constraints. To asses this goal we created a number of visual examples ourselves. Some of these are reproductions of examples used in imperative programming to show that we can create the same results without programming. Other examples are of existing poster designs but augmented with random variations. Finally some examples are completely new.

5.4.1 Pita Style

This example attempts to recreate the typical style of modern design posters such as those found on trendlist.org. It was originally conceived by Lucas Nijs, developed as a JavaScript application, and then ported to Logic Layout. It uses randomized colors, thick borders and abstract geometric shapes. The results (Figure 85) mimic this style.

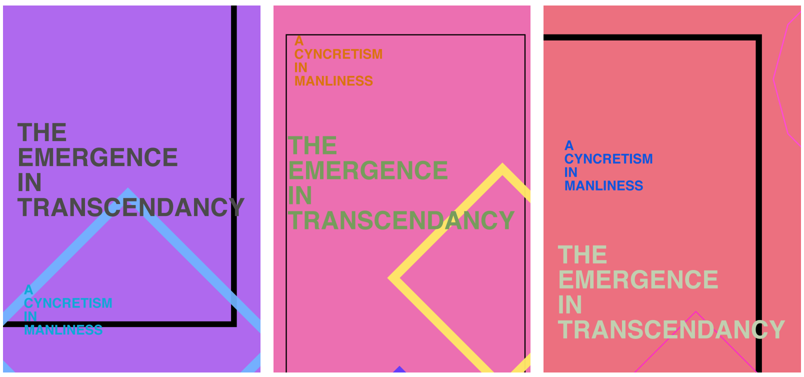

THE EMERGENCE IN TRANSCENDANCY
A CYNCRETISM IN MANLINESS
A CYNCRETISM IN MANLINESS
THE EMERGENCE IN TRANSCENDANCY
A CYNCRETISM IN MANLINESS
THE EMERGENCE IN TRANSCENDANCY

Figure 85: "Pita Style", Frederik De Bleser & Lucas Nijs, 2016

5.4.2 Tempo Moderni

Tempo Moderni is a spoof on Italian movie posters and features "disco lights" in the background (Figure 86). Text elements are constrained through "above" constraints so that they are displayed in logical order. Note that only a single circle has to be present on the layout: all the rest are defined by randomly positioning and giving the object a high count. The snapping feature makes sure the elements fall in a grid.

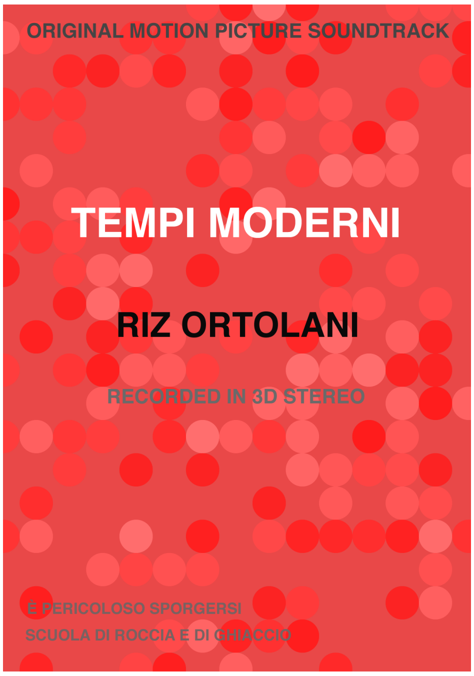


Figure 86: "Tempo Moderni", Frederik De Bleser, 2016

5.4.3 Various Work

Various other sketches created with Logic Layout (Figure 87).

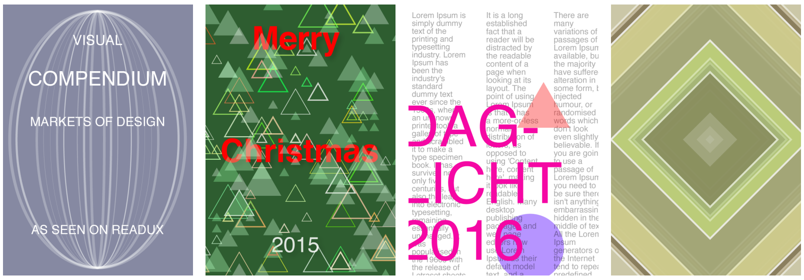


Figure 87: Logic Layout – selection of works

5.5 Evaluation

At its base the application works like traditional vector applications. However, because elements can be randomly placed and constrained, we are able to generate variations more efficiently (G1). This also allows us to quickly explore a broader approach (G3). Lastly, because the application works similarly to applications familiar to graphic designers and does not require programming or plan composition, users should be more self-reliant (G4).

5.6 Conclusions

We've shown that we can take a traditional direct manipulation interface and augment it with ranges and constraints to accomplish a range of generative designs that would normally require a programming approach. By basing our software off of traditional applications, Logic Layout makes generative design more accessible to a broader audience of non-technical people (G4). Through examples we demonstrated the breadth of the tool.

We have many ideas for improving Logic Layout. First off in generative design we can wield more instruments than the static and random value. We foresee options for *sequential values*, so that successive shapes can grow or shrink (or become darker, or move positions). Another feature is to let the system pick from a *set of options*: in the case of colors users could specify a color palette for the system to choose from. Lastly, all of these value types could be stored in a centralized way and *reused* throughout the project (much like variables).

An exciting challenge is to connect the properties of shapes to external data to create data visualizations. Again, instead of choosing between *fixed* and *random* values, users would control the properties of a shape based on the values of a database column. Users could drag the values onto the property of the shape of their choosing. This model matches Lyra with the exception that we can generate many different visualizations where the user can choose the most appropriate one.

Finally we'd like to explore models for interaction, for example by specifying a relationship between our shapes and the position of the mouse.

6 Conclusion

In this thesis we've demonstrated the impact of generative design on artists and designers. We've contributed a number of software applications that allow designers and artists to be more efficient, more creative, explore a broader approach and be self-reliant.

Summary

In chapter 3, we looked at **imperative programming.** We discussed existing tools such as Design By Numbers and Processing, then introduced NodeBox 1. We showed work created with the tool, both by students, design professionals, and ourselves. We discussed how teaching data structures helped students use the tool. We concluded that this approach was extremely powerful, although the syntax of the language presents a barrier that is difficult for designers to overcome.

In chapter 4, we discussed how **functional programming** could help overcome the syntax barrier. First we discussed existing languages and tools such as Clojure and Grasshopper. Then we introduced *NodeBox 2*, a visual node-based language. We showed work created with the tools and described its mechanism for generating variation: copy stamping. We concluded that this mechanism was difficult to use. We presented an alternative approach in *NodeBox 3*. Here the fundamental building block is the list, and variation is created through *list matching*. We saw how that makes debugging easier. *NodeBox Live* brought this approach to the web. It also got rid of subnetworks, instead using the *function* as the main abstraction, which resulted in a hybrid node-code approach. When we compared textual vs visual programming for beginning users, we concluded that visual programming was not necessarily more efficient or easier. However, after the initial learning curve, in depth interviews and visual results indicate an improvement over textual programming for more proficient users. In this chapter we also briefly introduced *DataPipe*, a tool for batch processing of large data sets, and *OpenType.js*, a JavaScript library for parsing and exporting fonts on the web.

In chapter 5, we examined **constraint programming** as a solution to issues with plan composition. We saw that constraint programming provides a way of describing the *what* and not the *how*. We saw existing solutions for layout (Cassowary, Auto Layout) and data visualization (Lyra). We presented *Logic Layout*, a visual editor for generative design that works through direct manipulation. We saw how *random ranges*, *element counts* and *constraints* can help augment a simple drag-and-drop design application to a complex generative design tool.

Technology as an Amplifier

We've all witnessed how Twitter was used in the Arab Spring to widely broadcast messages of dissent. Technology can magnify the impact of our actions. It allows us to communicate in real-time with friends across the world or broadcast messages to millions of followers. We're reminded of Steve Jobs' famous saying that "computers are like a bicycle for our minds". They allow us to take our less-than-optimal way of creating, communicating and sharing and, through the computer, vastly expand our capabilities and reach.

This amplifying principle is the core tenet of generative design: to use the computer as an *amplifier* for the manual process. At its most bare it is about using the minimal set of physical actions (mouse movements and keyboard strokes) to have maximum impact (a successful design or product). We've seen in the preceding chapters how generative tools can multiply the amount of work we can produce compared to a manual approach.

In **imperative programming** we use code as a direct way to provide instructions to the machine. These instructions can contain statements of repetition, allowing us to loop over a list of data, the pixels in an image, or the word in a text. The process of repetition instruction allows us to produce more output than we could possibly produce by hand.

In **visual functional programming** systems, we use the amplifying principle in the same way as it is used in a factory assembly line: by setting up a series of transformations, from the input data all the way to the visual result. Each building block executes simple instructions, but in combination can produce radical transformations. Once the process is built, we can feed it in new data (e.g. through web services) and produce new output, without redoing the manual setup work.

In **constraint programming** we saw how an ordinary drag-and-drop interface (which is marginally faster than the by-hand approach) can be augmented to gain access to the core principles of generative design. Using random ranges and repetition we can create many target elements starting from a single source shape, thereby multiplying our manual effort. In addition, the system produces many variations for us to pick from. We use constraints to specify the relationships between elements, avoiding much of the manual work in picking the interesting designs from a large set of unsuitable outputs.

These amplifying techniques go beyond repetition: we're not required to give the computer a fixed set of instructions and values to follow perfectly every time. We can greatly expand the range of creative possibilities if we integrate randomness into the process.

Randomness as Creative Catalyst

Throughout this thesis we argued that randomness was an important factor in designing generative systems. It allows us to break free from a rigid procedure the computer has to follow and by allowing some leeway, produce results that are sometimes surprising. We've seen that these novel and unexpected solutions can inspire our own creativity.

Randomness is blind. A random procedure only *generates*. There is no evaluation step to determine which results are better. In fact, *we* are the evaluation step. By judging the variations we steer the process, tweaking random ranges or venturing into new directions.

We saw the need for randomness to be controlled through *seeding*. This allows us to explore many variations efficiently and pick the ones we like. In NodeBox 3 and NodeBox Live, this manifests itself through an explicit `seed` parameter which is included wherever we use a random procedure. We recommend this approach to other tool builders.

One of the limitations of randomness is that it has no frame of reference: all values are equally likely. In unbounded randomness, the size of a rectangle can be 5, 42 or 100, but just as well 0, -5000 or 1 million. In contrast, as designers we often know from context which values are suitable. This is both the strength of randomness (by producing unexpected results) and its weakness (by producing many unusable or erroneous results).

Randomness is appealing because it can produce surprising results with very little effort. To go beyond randomness, we can look at machine learning techniques that use varying degrees of knowledge to produce "smarter" results.

Computer Aided Design

The term computer aided design (CAD) started appearing around the mid 1970s. Early CAD software started out as a direct port of the process of manual drafting to the computer. Later the term also encompassed functionality to improve the quality of the design, automatically generate a bill of materials, do automatic layout in integrated circuits and much more.

All of these tasks are *type 1* automations (see Section 2.1): they help in automating production challenges that would be tedious to do by hand. We would like to see the meaning of the term "computer aided design" expanded to encompass all types of automation. In other words, computers *aid* in the design not only by automating production challenges but by helping solve creative challenges as well.

One promising area of research for design automation is machine learning. With the advent of faster processing using the GPU, deep learning systems (which mimic the web of neurons in the human brain) are becoming commonplace for individual use. The techniques are used in image recognition, voice recognition and machine translation. Deep learning algorithms have already been used to create artistic images that mimic existing styles (*Gatys*, 2015).

Techniques such as Bayesian inference have been used to learn and replicate design patterns. Work by Talton for example uses Bayesian grammar induction to learn design patterns, which can be used to synthesize novel designs (*Talton*, 2012).

We see a shift from the reductive technique of trying to "understand the world" by decomposing design in its constituent parts to a holistic process where all parameters are weighted together through algorithms. This weighted technique often results in a "black box" rather than clear, human-readable rules, making it increasingly harder to deduce what the system "knows". Just as we've seen in constraint programming, this can make it hard to debug. However, we argue that this opaqueness does not limit its usefulness.

Hybrid Tools

Throughout the thesis we built up a number of programming tools based on lessons we learned from the previous iteration of the tool. Looking back at this journey, one could argue that constraint programming is the best of all available solutions.

However, constraint programming is not a panacea. It is not the right tool for every job. We see constraint programming as an interesting and largely unexplored paradigm for creative tools. The fundamental algorithms are well-researched and fast enough to be used in interactive applications. We found it to be very useful for layout experimentation, where there is no set order to the steps to perform.

On the other hand, the lack of explicit ordering makes it less suitable for problems that rely on a fixed sequence of steps such as workflow automation. An imperative approach is more appropriate here. In addition we found debugging visual results difficult, especially when the number of constraints became large. Since there are no sequential steps leading up to the final result, we can't examine the output of each operation such as in NodeBox Live. This reminded us of "spaghetti code", a typical issue with novice programmers.

It is best to look at all programming paradigms as tools in a generative designer's toolbox, ready to be wielded at the right moment. Ideally, new tools could be integrated into a hybrid solution where one can switch paradigms as needed to cover a certain aspect of the design. NodeBox Live's function abstraction seems a good fit for combining imperative, functional and constraint programming through a common API.

Beyond Usability Research

Just as the tools for generative design rapidly expand, so should the testing techniques for usability adapt as well. Ideally we would like to compare widely different techniques (based on deep learning, genetic algorithms, and fixed procedures) with traditional applications like Photoshop. However we speculate that even if the task is similar, the interaction with these systems and the resulting output will be so wildly different as to make usability testing unfeasible (*Olsen Jr*, 2007). Instead, we need to look beyond A/B measuring techniques to test different approaches.

To formulate recommendations for current and future systems we use Myers' framework (*Myers*, 2000) to make suggestions around a number of themes.

Threshold and Ceiling

The *threshold* determines how difficult it is to learn the system. The *ceiling* is what can be achieved with the system. Ideally all tools would have a low threshold and high ceiling. In reality however, most tools either cater to beginners (low threshold – low ceiling) or professionals (high threshold – high ceiling). In other words, users will "grow out" of simpler systems and adopt systems best suited for experts.

Since all users will start out at the bottom having a low threshold is important for everybody. Here, reducing accidental complexity (e.g. easy installation, good documentation, simple errors,...) is essential.

Expert users can be accommodated by designing a hybrid approach. This entails giving experts the tools that you use yourself to develop the tool. So, even though the primary interface might be visual (such as NodeBox Live), it also should have a scriptable interface. In NodeBox Live, this means that nodes can be coded both visually and textually. It also means that the system can be accessed through an API, making integration and automation possible.

Path of Least Resistance

Every UI will make some things easier to do than other things. This is a simple application of Fitts' Law (*MacKenzie*, 1992): there is only so much space to place buttons and menu options. In Myers' view – in the context of user interface design – this technique can be used to our advantage to let users do the "right thing".

In design, however, we have to be aware of this. This finding implies that there is a "right way" to do things, which automatically guides designs in a certain direction, either out of laziness or as a way to manage complexity for beginners. We've seen many design students hastily submit their classwork with a superficial adaption of a built-in example. For creative professionals, this property is generally undesirable: it can result in certain shapes or effects being overused, making designs coming from a certain application instantly recognizable.

Ideally the principle of least resistance should be applied to aspects of the tool that are not directly tied to its creative output. For example, the organisation of the project files is something that a tool can handle automatically to avoid common mistakes.

Predictability

Predictable tools, given the same input, always produce the same output. This property is desirable for generative tools as well. On the surface this doesn't seem to integrate well with the principle of randomness generation. That is why we introduce the concept of random seeding in our tools. This still allows the power of random variations but gives control over where and how they are applied.

Moving Targets

Tools are always built *in context*. They are designed to run on a range of machines that consist of physical hardware, and this needs to guide how we develop our tools. This also implies that this context can shift. We witnessed first-hand the shift from the desktop to the web, and adapted our tools to follow.

6.1 Real-World Usage

A good indicator of success is the adoption of the tools in real-world use. NodeBox 3 currently has over 1200 monthly active users (as reported by our software update mechanism). The most active users come from the USA, Great Britain, Germany, France, Italy, Belgium, Russia, Korea, Canada and Poland. The NodeBox community has an active forum with thousands of posts where users help out other users. Because our projects are free software, users are free to make contributions. OpenType.js has over 100 issues reported and more than 60 pull closed pull requests.

Our software is being used to experiment with generative design and create commercial visualizations, such as the "Color of World Flags" project by Nicholas Rougeux (Figure 88).

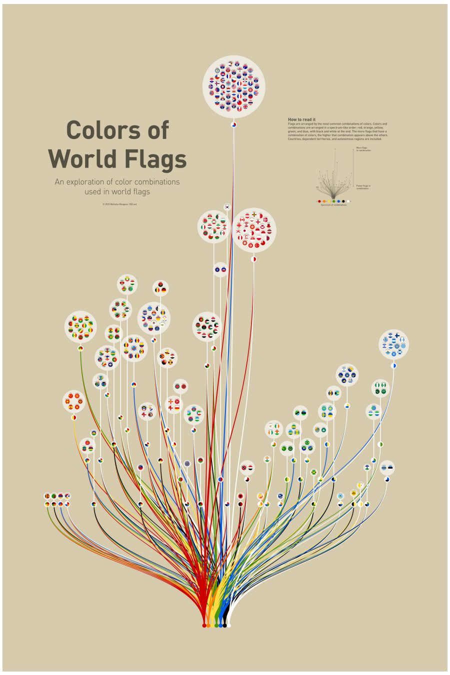


Figure 88: "Colors of World Flags", Nicholas Rougeux, 2015

NodeBox 3 and NodeBox Live are taught by teachers outside of the research group worldwide and is part of the curriculum at a number of art schools, for example the Rhode Island School of Design (RISD).

Learning Process

Understanding the learning process is key in developing tools. We continue to use and advocate the Dreyfus framework to understand the needs of users with different levels of skill. The temptation we have as tool builders is to force users to understand the conceptual model of the software, believing it is essential for understanding the tool, which it is. However, we now know that novices just want to get things done without being overloaded by what they believe is unnecessary theory.

Consequently, tools that are designed to be free-form should also provide built-in template graphics for users to get started. Although the tool can – in theory – produce any kind of graphic, novices need a point of reference to get started. Only after users get over that initial hurdle will they understand or seek the knowledge to progress beyond build-in template graphics.

On the other side of the spectrum are the experts, who need tools that grow with them. This implies planning for people to outgrow the tool at some point. To provide for that, tools should ideally provide documentation on their data format and allow scripting through open APIs so that the tool can be driven from other tools.

Object-Oriented Programming

In this thesis we focused on the imperative, functional and constraint programming paradigms. What about object-oriented programming, which is ubiquitous in the industry?

Programming paradigms can be viewed along multiple "axes". The axis we chose is the imperative vs. declarative axis. Imperative languages focus on the "how", whereas declarative language (of which functional and constraint programming are descendants) focus on the "what". Object-oriented programming falls on the imperative side of this axis. However, its core idea is about the structure it can provide to large programs.

In NodeBox 1 we generally don't have the need for the kind of organizational methods object-oriented programming provides. Typical student programs are small and can do a lot with simple lists and maps. We find that in larger programs, students typically do use classes, so we explain them when needed.

In NodeBox 3 and NodeBox Live we prefer to avoid classes and use universal data structures like lists and maps. This has the benefit that all our data operations (e.g. `filter`, `combine` and `pick`) can work on lists of any type. In addition we found that data hiding through private attributes and getters / setters hinders debugging. By using general-purpose lists and maps we can provide inspection tools that work everywhere.

Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious. — (Brooks, 1975)

Meta Tools

Every tool will come with a path of least resistance, making certain things easy and other things hard. This will ultimately guide what we create with these tools. If your tool provides a star shape but not a heart shape, we will see on average more stars than hearts being created. To wit:

"We become what we behold. We shape our tools and then our tools shape us" – Marshall McLuhan (*Culkin*, 1967)

To overcome this problem it is useful to see generative tools as essentially "meta tools". That is, instead of seeing e.g. NodeBox 1 as *the* tool to create the design, we can see a certain ruleset, *defined within* NodeBox as the tool. In other words we convert the problem space to a solution space first, by defining a domain-specific language within our meta-tool. Once we have this domain-specific language, we can explore solutions within this tool. In essence, we are building different tools for each project.

This way of thinking is a fundamental shift forwards in how we approach tools. Traditional graphic tools such as Photoshop already provide a solution space, and if that space doesn't fit your idea, you can either choose a different tool or a different idea. Often this leads to self-censorship, limiting their ideas to what they deem possible. By regarding tool-building as the first endeavour in every project, designers are no longer required to force their idea in an existing framework of thought. Instead they are free to make up their own rules. That's why we strongly advocate adopting a meta-tool approach. Tool building is too important to be left at the hands of programmers.

7 References

Alexander, Christopher and Ishikawa, Sara and Silverstein, Murray. *A pattern language: towns, buildings, construction.* Oxford University Press, 1977. Cited on page 115.

Badros, Greg J and Borning, Alan and Stuckey, Peter J. *The Cassowary linear arithmetic constraint solving algorithm.* ACM, 2001. Cited on page 147.

Bargas-Avila, Javier A and Oberholzer, Glenn and Schmutz, Peter and de Vito, Marco and Opwis, Klaus. *Usable error message presentation in the World Wide Web: Do not show errors right away.* Elsevier, 2007. Cited on page 63.

Binkley, Dave and Davis, Marcia and Lawrie, Dawn and Morrell, Christopher. *To camelcase or under_score.* 2009. Cited on page 67.

Bostock, Michael and Ogievetsky, Vadim and Heer, Jeffrey. *D3: data-driven documents.* IEEE, 2011. Cited on pages 25 and 135.

Brooke, John and others. *SUS-A quick and dirty usability scale.* London, 1996. Cited on pages 27 and 28.

Brooks, FP. *No silver bullet.* April, 1987. Cited on page 28.

Brooks, Frederick P. *The mythical man-month.* Addison-Wesley Reading, MA, 1975. Cited on page 169.

Brown, Philip J. *Error messages: the neglected area of the man/machine interface.* ACM, 1983. Cited on page 63.

Carkci, Matt. *Dataflow and Reactive Programming Systems: A Practical Guide.* Matt Carkci, 2014. Cited on page 76.

Chastel, André. *Leonardo da Vinci: sämtliche Gemälde und die Schriften zur Malerei.* Schirmer/Mosel, 1990. Cited on page 145.

Church, Alonzo. *The calculi of lambda-conversion.* Princeton University Press, 1951. Cited on page 73.

Culkin, John M. *A schoolman's guide to Marshall McLuhan.* 1967. Cited on page 170.

Dantzig, George B and Orden, Alex and Wolfe, Philip and others. *The generalized simplex method for minimizing a linear form under linear inequality restraints.* 1955. Cited on page 147.

De Bleser, Frederik. *Grasp events database dump.* 2013. Cited on page 138. https://github.com/fdb/grasp-data

De Bleser, Frederik. *DataPipe.* 2014. Cited on page 105. https://github.com/fdb/datapipe

De Bleser, Frederik. *OpenType.js.* 2014. Cited on page 134. https://github.com/nodebox/opentype.js

De Smedt, Tom. *Prism Algorithm.* 2007. Cited on page 24. https://www.nodebox.net/code/index.php/Prism

De Smedt, Tom and Daelemans, Walter. *Pattern for python.* JMLR. org, 2012. Cited on pages 47 and 129.

De Smedt, Tom and De Bleser, Frederik and Van Asch, Vincent and Nijs, Lucas and Daelemans, Walter. *Gravital: natural language processing for computer graphics.* Walter de Gruyter, 2013. Cited on pages 7 and 63.

Dreyfus, Stuart E and Dreyfus, Hubert L. A *five-stage model of the mental activities involved in directed skill acquisition.* 1980. Cited on pages 30, 61, 121 and 138.

Ebrahimi, Alireza. *Novice programmer errors: Language constructs and plan composition.* Elsevier, 1994. Cited on page 29.

Egli, Simon and Crossland, Dave. *Metapolator.* 2013. Cited on page 134. http://metapolator.com/

Ellis, Clarence A and Gibbs, Simon J. *Concurrency control in groupware systems.* 1989. Cited on page 132.

Ferraiolo, Jon and Jun, Fujisawa and Jackson, Dean. *Scalable vector graphics (SVG) 1.0 specification.* iuniverse, 2000. Cited on page 38.

Freeman, Allyn and Golden, Bob. *Why Didn't I Think of That: Bizarre Origins of Ingenious Inventions We Couldn't Live Without.* University of Texas Press, 1997. Cited on page 45.

Freudenthal, Margus. *Domain Specific Languages in a Customs Information System.* IEEE, 2010. Cited on page 145.

Fry, Ben. *Visualizing data: exploring and explaining data with the Processing environment.* O'Reilly Media, Inc., 2007. Cited on page 105.

Galai, Noam. *The Darkroom Techniques Behind the Tools We All Know and Love in Photoshop.* 2015. Cited on page 25. http://bit.ly/1onKvSj

Gatys, Leon A and Ecker, Alexander S and Bethge, Matthias. A *neural algorithm of artistic style.* 2015. Cited on page 164.

Gielis, Johan. A *generic geometric transformation that unifies a wide range of natural and abstract shapes.* Botanical Soc America, 2003. Cited on page 130.

Graham, Paul. *Holding a Program in One's Head.* 2007. Cited on page 65. http://paulgraham.com/head.html

Green, Thomas R. G. and Petre, Marian. *Usability analysis of visual programming environments: a 'cognitive dimensions' framework.* Elsevier, 1996. Cited on pages 29 and 135.

Green, Thomas RG and Petre, Marian. *When visual programs are harder to read than textual programs.* 1992. Cited on pages 135 and 140.

Green, Thomas RG and Petre, Marian and Bellamy, RKE. *Comprehensibility of visual and textual programs: A test of superlativism against the'match-mismatch'conjecture.* 1991. Cited on pages 135 and 140.

Griffiths, Dave. *Fluxus.* 2005. Cited on page 74. http://www.pawfal.org/fluxus/

Hannah, Gail Greet. *Elements of design: Rowena Reed Kostellow and the structure of visual relationships.* Princeton Architectural Press, 2002. Cited on page 153.

Hermans, Felienne and Pinzger, Martin and Van Deursen, Arie. *Supporting professional spreadsheet users by generating leveled dataflow diagrams.* 2011. Cited on page 103.

Hoffman, Franz and Galeran, Joël. *FontSelf.* 2015. Cited on page 134. http://www.fontself.com/

Hunt, Andrew and Thomas, David. *The pragmatic programmer: from journeyman to master.* Addison-Wesley Professional, 2000. Cited on page 123.

Johnson, W Lewis and Soloway, Elliot. *Intention-based diagnosis of programming errors.* 1984. Cited on page 74.

Kelly, John L and Lochbaum, Carol and Vyssotsky, Victor A. A *block diagram compiler.* Alcatel-Lucent, 1961. Cited on page 75.

King, John and Magoulas, Roger. *Data Science Salary Survey.* O'Reilly, 2015. Cited on page 101. https://www.oreilly.com/ideas/2015-data-science-salary-survey

Lubbers, Peter and Greco, Frank. *HTML5 web sockets: A quantum leap in scalability for the web.* 2010. Cited on page 154.

MacKenzie, I Scott. *Fitts' law as a research and design tool in human-computer interaction.* L. Erlbaum Associates Inc., 1992. Cited on page 166.

Maeda, John. *Design by numbers.* MIT Press, 2001. Cited on pages 15 and 35.

Mathey, Yannick and Babé, Louis-Rémi. *Prototypo.* 2014. Cited on page 134. https://www.prototypo.io/

McCarthy, Lauren. *p5.js.* 2014. Cited on page 134. http://p5js.org/

McNeel, Robert and others. *Grasshopper generative modeling for Rhino.* 2010. Cited on page 79.

Muratori, Casey. *Immediate-Mode Graphical User Interfaces.* 2005. Cited on page 154. http://mollyrocket.com/861

Myers, Brad and Hudson, Scott E and Pausch, Randy. *Past, present, and future of user interface software tools.* ACM, 2000. Cited on page 165.

Nijs, Lucas. *Experimental Typography Workshops.* 1998. Cited on page 27. http://designlooksnice.com/workshops.php

Nijs, Lucas. *Ideas from the Heart.* 2007. Cited on page 60. https://www.nodebox.net/code/index.php/Ideas_from_the_Heart

Olivero, Giorgo. *ToDo's Spamghetto wins European Design Awards in Silver.* 2010. Cited on pages 19 and 55. http://bit.ly/todo-spamghetto

Olsen Jr, Dan R. *Evaluating user interface systems research.* 2007. Cited on page 165.

Papert, Seymour. *Mindstorms: Children, computers, and powerful ideas.* Basic Books, Inc., 1980. Cited on page 34.

Pereira, Luis Moniz. *Rational debugging in logic programming.* 1986. Cited on page 145.

Perlis, Alan J.. *Epigrams on programming.* 1982. Cited on page 75.

Pilgrim, Mark. *Dive Into Python.* 2000. Cited on page 57. http://www.diveintopython.net/

Press, Adobe. *PostScript language reference manual.* Addison-Wesley Longman Publishing Co., Inc., 1985. Cited on pages 34 and 136.

Reas, Casey and Fry, Ben. *Processing: a programming handbook for visual designers and artists.* Mit Press, 2007. Cited on pages 15, 36 and 135.

Resnick, Mitchel and Maloney, John and Monroy-Hernández, Andrés and Rusk, Natalie and Eastmond, Evelyn and Brennan, Karen and Millner, Amon and Rosenbaum, Eric and Silver, Jay and Silverman, Brian and others. *Scratch: programming for all.* ACM, 2009. Cited on page 77.

Satyanarayan, Arvind and Heer, Jeffrey. *Lyra: An interactive visualization design environment.* 2014. Cited on page 147.

Shackelford, Russell L and Badre, Albert N. *Why can't smart students solve simple programming problems?* Elsevier, 1993. Cited on page 70.

Shapiro, Ehud Y. *Algorithmic program debugging.* MIT press, 1983. Cited on page 145.

Shiozawa, Hidekazu and Okada, Ken-ichi and Matsushita, Yutaka. *3d interactive visualization for inter-cell dependencies of spreadsheets.* 1999. Cited on page 103.

Shirky, Clay. *Cognitive surplus: Creativity and generosity in a connected age.* Penguin UK, 2010. Cited on page 95.

Soloway, Elliot. *Learning to program= learning to construct mechanisms and explanations.* ACM, 1986. Cited on page 29.

Sork, Thomas J. *The workshop as a unique instructional format.* Wiley Online Library, 1984. Cited on page 60.

Spohrer, James C and Soloway, Elliot. *Novice mistakes: Are the folk wisdoms correct?* ACM, 1986. Cited on pages 29, 70, 100, 121 and 140.

Sutherland, Ivan E. *Sketch pad a man-machine graphical communication system.* 1964. Cited on page 146.

Talton, Jerry and Yang, Lingfeng and Kumar, Ranjitha and Lim, Maxine and Goodman, Noah and Mech, Radomir. *Learning design patterns with bayesian grammar induction.* 2012. Cited on page 164.

Tufte, Edward R and Graves-Morris, PR. *The visual display of quantitative information.* Graphics press Cheshire, CT, 1983. Cited on page 25.

Victor, Bret. *Learnable programming.* 2012. Cited on page 65.

Victor, Bret. *Drawing Dynamic Visualizations*. 2013. Cited on pages 25 and 147. https://vimeo.com/66085662

Wadler, Philip. *Comprehending monads*. Cambridge Univ Press, 1992. Cited on page 75.

Wilkinson, Leland. *The grammar of graphics*. Springer Science & Business Media, 2006. Cited on pages 81, 103 and 147.

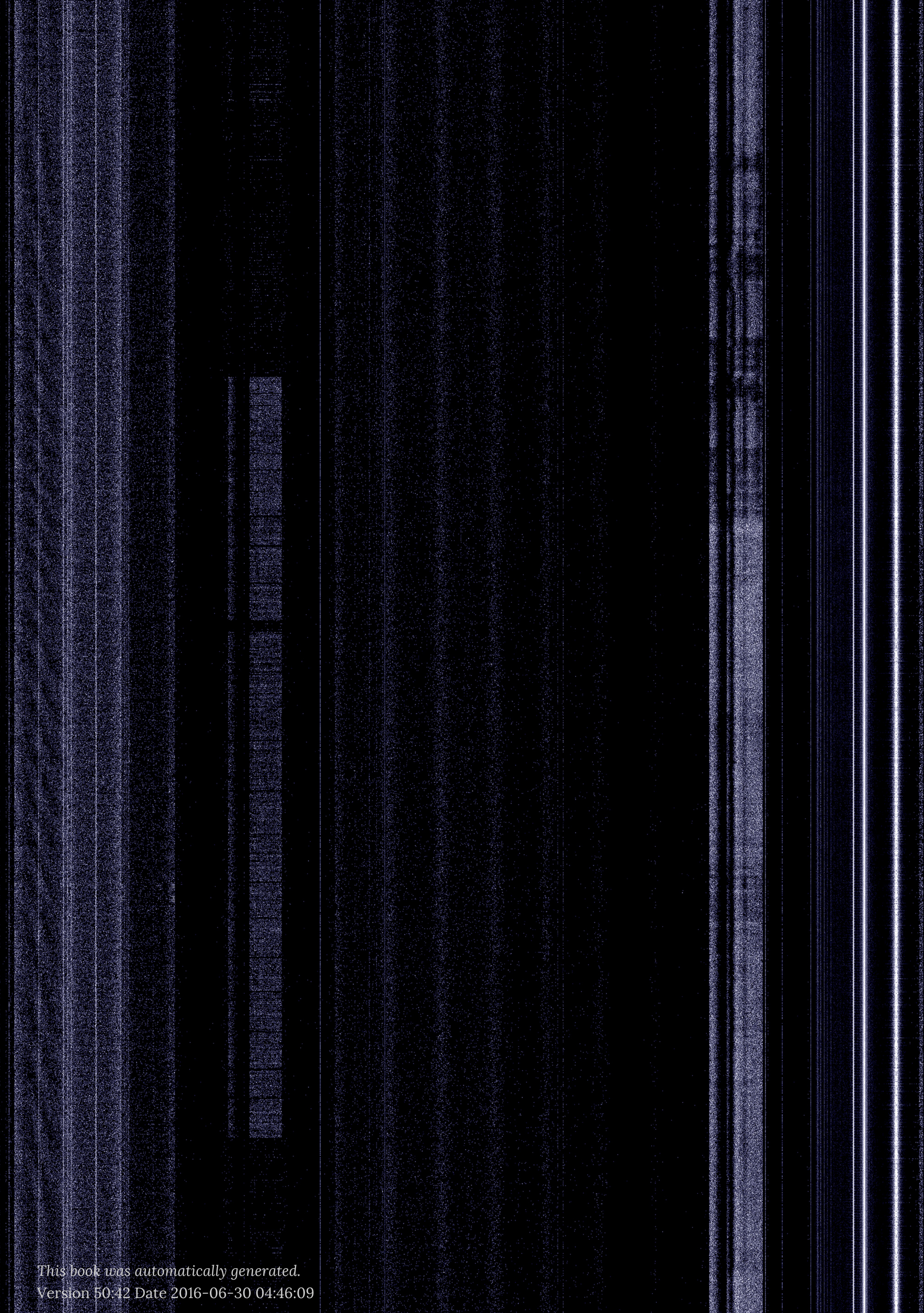
Winograd, Terry. *Procedures as a representation for data in a computer program for understanding natural language*. 1971. Cited on page 62.

Wittgenstein, Ludwig. *Tractatus logico-philosophicus*. Edusp, 1994. Cited on page 26.

Wu, Hui and Gray, Jeff and Roychoudhury, Suman and Mernik, Marjan. *Weaving a debugging aspect into domain-specific language grammars*. 2005. Cited on page 145.

van Blokland, Erik and van Rossum, Just. *Is Best Really Better?* Emigre, 1991. Cited on page 34.

van Rossum, Just. *DrawBot*. 2002. Cited on page 42. http://www.drawbot.com/



This book was automatically generated.
Version 50:42 Date 2016-06-30 04:46:09