+ code

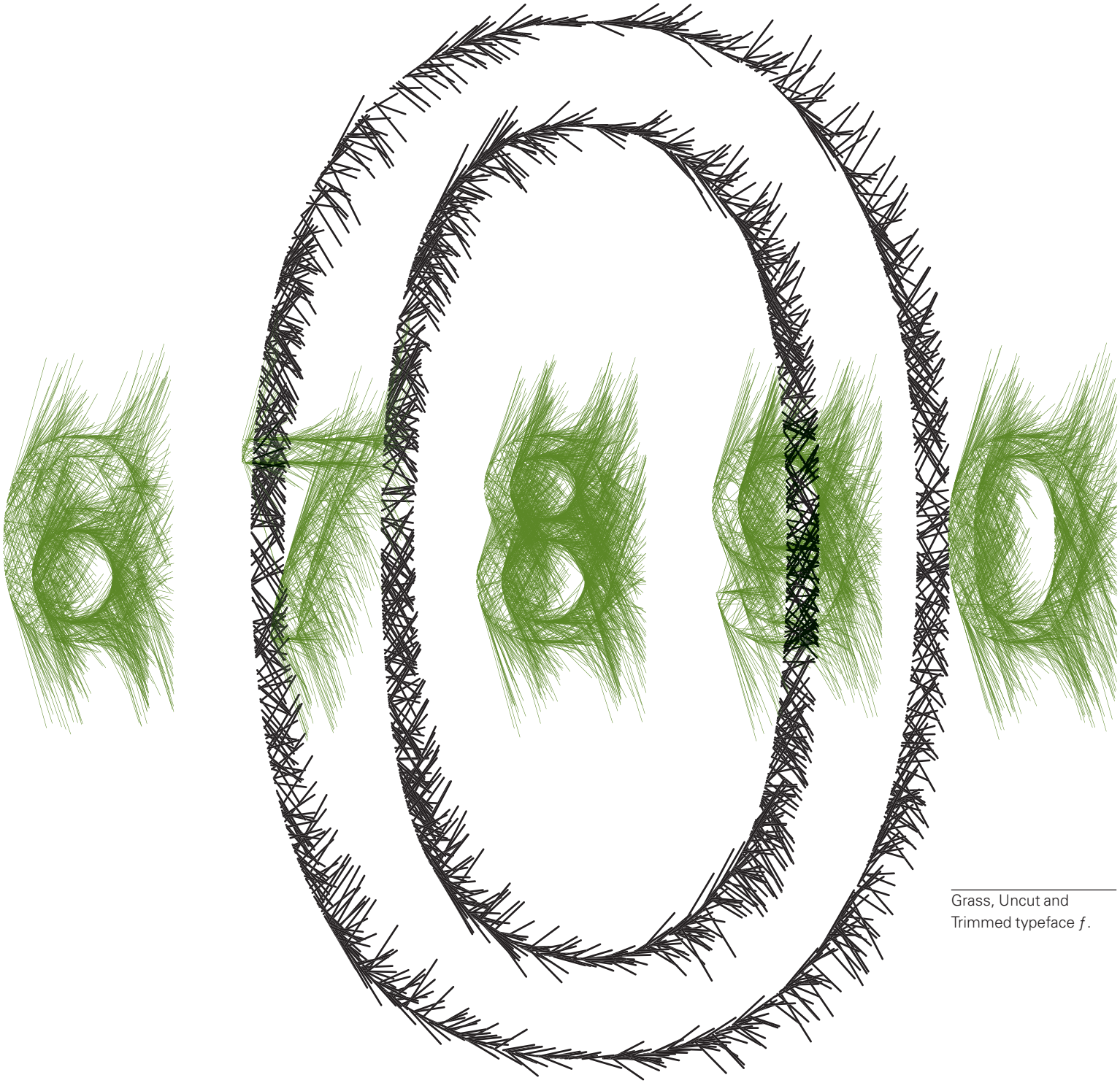
Processing
for Designers

```
PFont font;
doc setup(){
size(8.5,8.5);
textAlign(RIGHT);
page >R< odd {cvr}
```

Yeohyun Ahn.
Viviana Cordova.

> DEFAULT <
end.

1 2 3
4 5 6
7 8 9



Grass, Uncut and Trimmed typeface *f*.

```
PFont font; Universe
doc setup(){
size(8.5,8.5);
textAlign(LEFT);
page >L< even
{IV} page
```


TYPE + CODE
PROCESSING FOR DESIGNERS

ISBN 978-0-578-01143-1

Authors: Yeohyun Ahn, Viviana Cordova
Editors: Jill Ronsley, Jonty Hayes
Book Design: John P. Corrigan
Cover Design: Viviana Cordova
typeandcode Web site design: Kate Harmon

Printed by Lulu.com

www.typeandcode.com
www.processing.org

Dedicated to
MICA GD MFA program

CONTENTS

```
PFont font; Universe
doc setup(){
size(8.5,8.5);
textAlign(LEFT);
page >L< even
{VI} page
```


PREFACE

Type + Code has been created especially for designers and design students in classrooms with limited or no prior knowledge of programming language code. Also, *Type + Code* is geared towards the interests and values of graphic design such as color, form and typography.

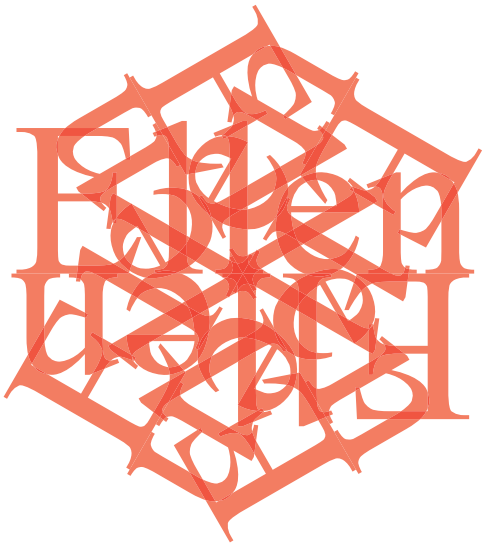
Every chapter has visual examples on beginner, intermediate and advanced levels. Most of the examples will have Processing code as a resource, so students and teachers can have access and use the software for their future projects. You can download the software from the Processing Web site www.processing.org/download.

For more in-depth information within the developing arena, Casey Reas and Ben Fry have created *Processing*, a programming handbook for visual designers and artists.

ABOUT THE BOOK

Type + Code explores the aesthetic of experimental, code-driven typography, with an emphasis on the programming language Processing, which was created by Casey Reas and Ben Fry. This book includes examples using Processing on basic, intermediate and advanced levels. In the beginning of the book, we provide a basic introduction to Processing and typography from letters and patterns to hierarchy, layers, figure/ground and color. We also examine how to perform basic graphic design functions such as rotating, altering transparency and repeating. Later chapters provide inspirational samples created by more structural coding, which express the unlimited possibilities of creating with Processing. These advanced projects use algorithms, a logical sequence based on mathematical principles of computer code, and libraries, which are a method of extending the programming language. This book also includes tutorials to help graphic designers understand the practice of code-driven typography with Processing.

Yeohyun Ahn.
Viviana Cordova.



```
PFont font; Universe
doc setup(){
size(8.5,8.5);
textAlign(RIGHT);
page >R< odd
page {IX}
```


FOREWORD

Ellen Lupton

Around the world, artists and designers are experimenting with code. Rather than delineate the final outcome of a work, designers are creating flexible systems that yield unpredictable results. Often, a few simple conditions and processes combine to produce complex, intricate effects. Tiny elements swarm together to form larger structures. Simple geometric shapes and solids collide, deform and fragment to create new entities. Input harvested from data networks provides bodies of content upon which random operations can be performed.

How do artists create these code-driven works? The open-source software application Processing has found its way into the hands of thousands of designers and artists worldwide. Created by pioneering software artists C.E.B. Reas and Ben Fry, Processing is a language, a tool, a medium and a vital social phenomenon. Conceived especially for visual artists, its elegant interface allows users to quickly envision, test and share results.

Typography is the art of arranging letters in space and time. In this ubiquitous discipline, visual and verbal expression converge, leaving no literate person free from its beauty or tyranny. Typography encompasses the form of individual letters as well as their configuration into words, lines and texts. The alphabet and other writing systems are themselves a kind of code, translating the units of speech (whether sounds or words) into graphic marks. Transforming writing into a reproducible code, typography fixes the idiosyncrasies of the scribe with the uniformity of the machine.

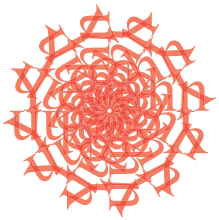
Typeface design has been automated in one fashion or another since the Industrial Revolution, when technologies such as the pantograph enabled designers to manufacture endless variations of a font based on a single core drawing. Similar tools spawned the proliferation of typeface designs during the phototypesetting revolution at mid-century and the massive digital revolution that followed.

Modern designers have long experimented with the forms of the alphabet, whether by radically simplifying its elements down to a minimal core or by allowing those elements to expand and proliferate into ornate or hypertrophied structures. Using code to generate unexpected letterforms is a recent evolution of this rich vein of visual research.

This book was conceived, designed and produced by Yeohyun Ahn, John Corrigan and Viviana Cordova, three graphic design MFA students at Maryland Institute College of Art (MICA). The work began with a series of workshops and thesis explorations initiated at the school; when the team graduated, they continued their research and focused on sharing it with the public. The current volume is the result of several years of sustained effort by these dedicated designers and educators. By creating this guide, the authors hope to inspire other designers to engage with code and consider typography from a fresh perspective. The alphabet is there for remaking.

Type + Code is published by MICA's Center for Design Thinking, a research and publishing unit that works with MICA students and faculty to originate and disseminate ideas about design for a variety of audiences, from general readers to design professionals. The Center is proud to publish this book, our first print-on-demand publication conceived and implemented solely on the initiative of our MFA students.

```
PFont font; Universe
doc setup(){
size(8.5,8.5);
textAlign(LEFT);
page >L< even
{X} page
```

Yeohyun Ahn

I still remember my grandfather showing me how to draw a character with a big oriental brush on Chinese paper. My hometown, Cheongju, South Korea, is famous for calligraphy, and, since it is regarded as such an honor there, being a great calligrapher was always my dream. But even though I had this great interest in calligraphy, I enrolled in the computer science department—on the strong recommendation of my parents, a professor in civil engineering and a former math teacher in high school.

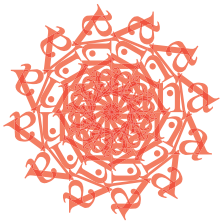
When I came to America as a graduate student, Ellen Lupton, who was my graduate director on the MICA GD MFA program, introduced me to Processing—a programming language for designers. I quickly and easily engaged with the new language, since it had a simple syntax and one could get straightforward and useful samples from its Web site, www.processing.org.

During the past two years, I have created several experimental typographies with Processing, some of them inspired by my childhood interest in calligraphy. Caligraft—computational calligraphy created by Ricard Marxer Piñón—especially enabled me to explore new ways of producing digital calligraphy using Processing. Most of these methods have been included in this book as tutorial samples.

While I got into Processing without much pain, since I have a computer science background, most of my classmates in the graphic design program suffered. They found it hard to figure out what was going on in Processing, because they didn't have any background in programming languages and most Processing tutorials remain programmer oriented. So, I decided to write *Type + Code*. I hope that this book helps designers to understand Processing and makes it easier and friendlier to use.

I would like to appreciate my collaborators, Viviana Cordova and John P. Corrigan. Without their help, I believe, this book would not have been born.

I would also like to express my appreciation for Ellen Lupton and Jennifer Cole Phillips, who introduced me to Processing, taught me how to use it to experiment with typography and supported me in writing this book.



Viviana Cordova

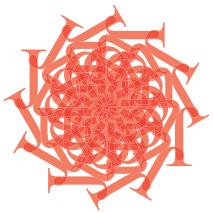
My interest in typography, technology and print design began with fine art classes when I was five years old. I would draw and paint landscapes just to add letters to them. My calligraphy class was my favorite at school. My father also encouraged me to take computer classes, which I really enjoyed—even though my first intensive computer class, which was outside of high school during freshman year, was filled with students five years older than me. I was able to assemble computers at that time, and ever since I've been very interested in technology.

I chose graphic design as a career and concentrated on interactive media, because in this field typography, image and technology work together simultaneously. Thus, computers and design have merged in my present and future interests. I particularly enjoy experimenting with anything new that relates to future educational resources. That is why Processing captured my attention.

There are so many tools for designers—from new software to language programming—but Processing was one of the most accessible to learn. Typography created in a more openly minded way naturally has become a part of my experimental path—as is shown by the various examples in this book.

I worked as a producer after I got my bachelor's degree, and I saw the necessity of constantly learning and updating knowledge according to what is best in one's field. In my case, it was video, editing and motion graphics. In addition, I was responsible for print design within the marketing department. In our multitasking society, we designers have to be prepared to tackle anything that is out there in the real world.

Recently, I graduated from the Maryland Institute College of Art in Baltimore with a master of fine arts in graphic design. Currently I am an adjunct faculty member in the Graphic Design department at Towson University in Maryland. Being involved in graphic design and collaborating with my classmates for this book, I believe that experimental software and programming languages that are open source are necessary as new tools in the twenty-first century. What was experimental ten years ago, the public is now learning and using in their work environment. Therefore, supporting resources such as Processing is extremely necessary for the new generation of graphic designers and artists who want to go beyond their expectations, by creating new ways to visualize their message through language.



John P. Corrigan

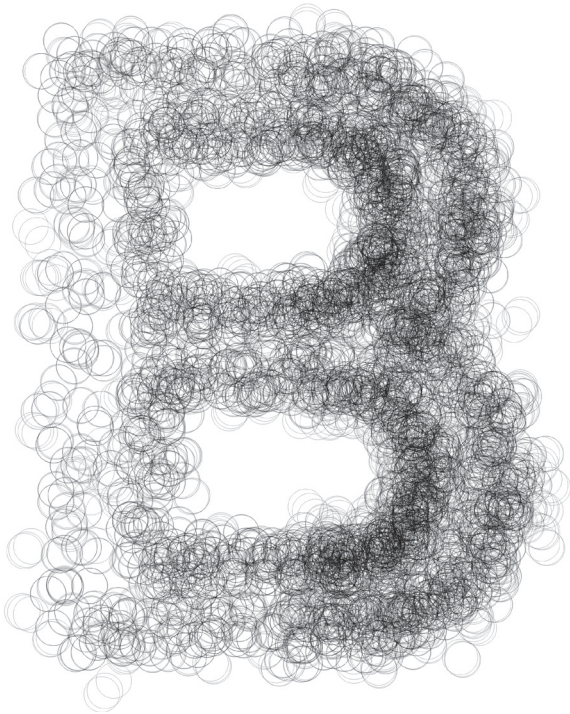
My initial exposure to Processing was through Yeohyun Ahn during my first year of graduate school. I was initially both surprised and confused by her masterful usage of Processing. The more I witnessed, the more I began to see the artistic potential of dynamic typography generated solely by the writing of code. Not limited to static letterforms, Processing can create semi-controlled forms by allowing the program to run with set intervals of operations; the in-between frame sets have their own life, shape and composition. Without knowing the full potential of Processing, I initially had a hard time extending it beyond formal typographic experiments. But, having worked with static Processing files, I now see the potential of Processing-driven forms.

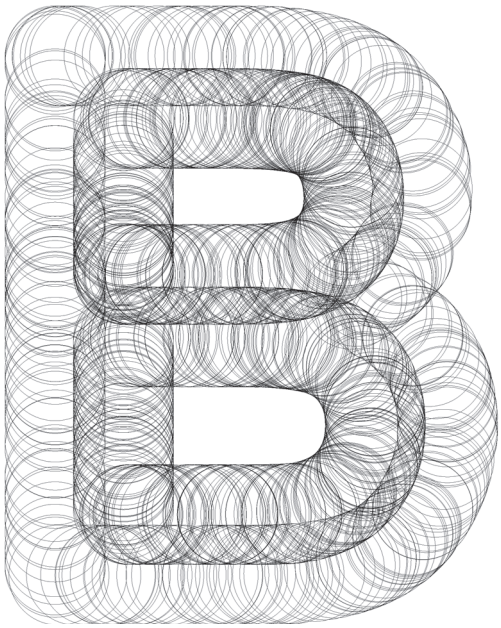
The Processing aesthetic greatly appeals to my design sensibilities. The geometric richness and unexpected nuances in saved-out frames allows a strict constant with an infinite number of possibilities. The seemingly random line gestures are backed by a strict numeric set of parameters. Through this project, I wish to extend the coded language of Processing, through the richness of its infinite variables. Visually, I react to separate and seemingly conflicting code structures, finding a true graphic beauty in the oppositional line qualities.

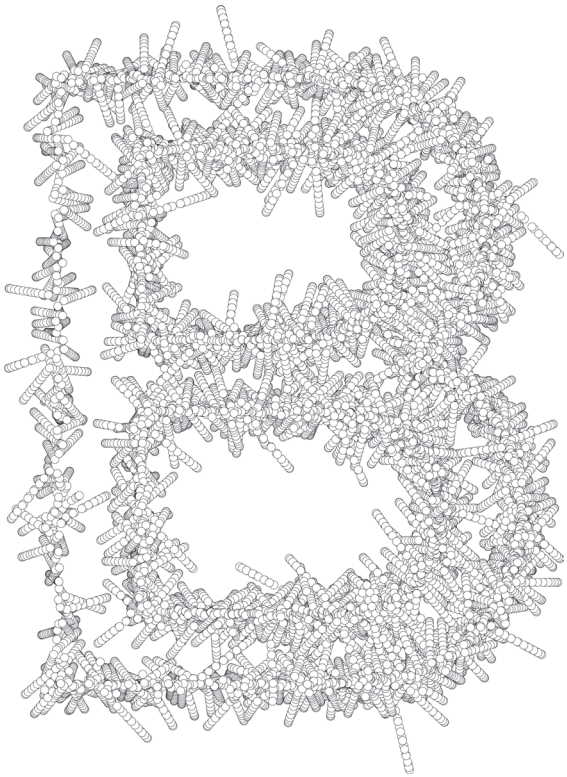
As a typographer and book designer, I was able to translate Ahn's static files into a meaningful and organized set of letterforms and typefaces expressed in print media. By taking what is a known active file format, the translation to print media creates dynamic letterforms for a variety of static print applications. Using Processing to drive data sets and convert them to print applications remains in its infancy, and its aesthetic possibilities are just beginning to be explored. The usage of Processing celebrates a multi-disciplinary design approach—combining interests in animation, advanced mathematics, interactivity and computer-generated graphics.

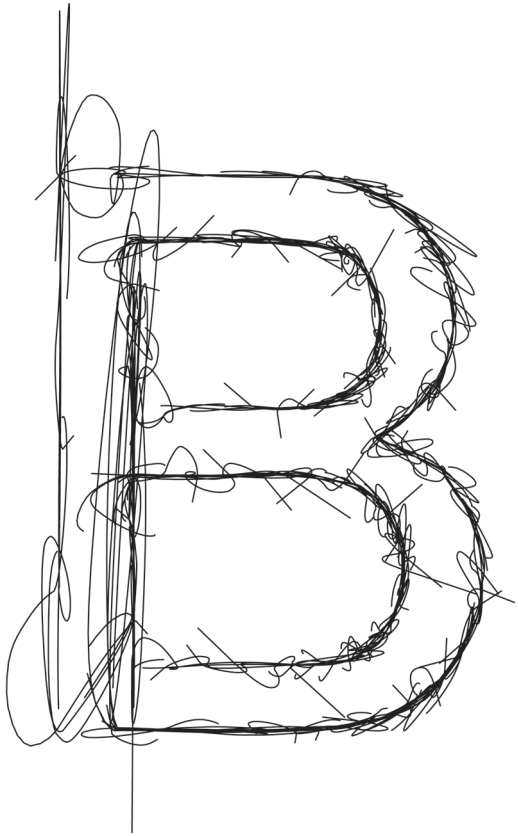
WELCOME TO PROCESSING!

This book begins with basic code, by using statements that are simple and very important to understand such as text( ), fill( ), alpha( ), translate( ), rotate( ) for( ), pushMatrix( ), popMatrix( ) to create simple, unique letters and patterns. From this basis, designers will learn to create their own experimental typography by practicing the tutorials, which will help them to understand the structure and syntax of programming with Processing. As a guide and inspiration, examples of experimental typography and pattern design are included throughout the book.









INTRODUCTION

Yeohyun Ahn

During my first semester in the graphic design graduate program at the Maryland Institute College of Art (MICA), Ellen Lupton, my then graduate director, introduced me to Processing. Created by Ben Fry and Casey Reas, Processing is a programming language for the electronic arts and visual design community. With Processing[1], designers can create posters, typography, information visualization, interactive design, motion graphics, non-linear animation, and so on. The use of coding in Processing helps designers to extend and explore their creativity through algorithm-based and library-oriented numbers. The Web site www.processing.org allows designers to download Processing, take tutorials, present their works and share their codes—all for free.

As part of my research for a proposal—"Contemporary Ornament," for graphic design magazine *PRINT*—I utilized a simple code, ellipse( ), which produced circles in Processing. I then added a semi-controlled mouse event, mousePressed, into my code that, whenever one clicked and dragged on screen in Processing, created logical, but visually unexpected and fresh, intricate pattern designs, using only circles. I was fascinated by the semi-randomness and the complexity generated by ellipse( ) and mousePressed in Processing, since it was original. This was my first time using Processing rather than my habitual application tools such as the Adobe software packages. Later, the work was published in the March 2006 edition of *PRINT*. It was also reinterpreted for the cover art of the book *Graphic Design: The New Basics* in 2008.

My past two years as a graduate student at MICA have been spent in exploring new ways to create with Processing that could also be included in *Graphic Design: The New Basics*. This work, written by Ellen Lupton and Jennifer Cole Phillips in collaboration with MICA students, presents a study of the fundamentals of form and ideas in a critical, logical way, and is inspired by contemporary media, theory and software systems[2]. When I read the draft version, I intuitively realized that it would be very helpful and practical for graphic designers and students majoring in graphic design, since new manuals that reflect contemporary trends in this field are needed.

At MICA, all of my classmates were visually outstanding, professional graphic designers, so I thought about how I could use my particular skills and expertise to contribute to the book. At precisely the right time, Ellen Lupton recommended Processing to me. It has a simple syntax and all of the tutorials are more visually oriented, compared with other programming languages such as Visual C++, Java, PHP, and so on. With a bachelor of science degree in computer science from South Korea, I did not find understanding the syntax of Processing seriously challenging, but exploring new creative ways with Processing would have been considerably harder at the beginning, if I had not met Jennifer Cole Phillips, who is associate director of the graphic design graduate program at MICA.

1 *Processing*. Ben Fry and Casey Reas. www.processing.org.

2 *Graphic Design: The New Basics*. Ellen Lupton and Jennifer Cole Phillips. Princeton Architectural Press. New York, 2008.

During the graphic design graduate studio with her in spring 2006, my task was to generate typographic pattern designs with computer programming languages such as Processing, as part of the research for *Graphic Design: The New Basics.* I later successfully created several experimental patterns in Processing, but, at the beginning, I had no idea of how to start. But her instructions clarified it for me. They were clear and simple: choose one letter, and then duplicate, move, repeat, rotate, overlay, scale, invert, cut and randomize with that one letter. I chose the letter, "Y," which is a part of my name (Yeohyun Ahn) and made a series of sketches by hand. Then I selected the most interesting one: each branch on the letter Y would become a smaller Y for any number of iterations with the same rotating angle.

To visualize my sketch in Processing, I found a perfect algorithm, Binary Tree—a tree data structure in which each node has at most two offspring (branches). By referencing Binary Tree, I defined a function, Ysystem( ) with six parameters: X1, Y1, X2, Y2, Level and Angle. X1 and Y1 indicate the starting position values of the two offspring from each node. X2 and Y2 show the end position values of the two offspring from each node. Level indicates how many nodes will emerge. Angle represents the value of the angle that spreads out from each node. Ysystem( ) was logically duplicated, rotated and overlapped by using these six parameters and then the distinct visual patterns were created. This design, which would have been extremely labor intensive to create in Adobe Illustrator or Photoshop, was presented in 2007 at *School of Thought III* at the Art Center College of Design as a part of "Power of Processing," with Gregory May.

In the fall of 2006, Marian Bantjes, a distinguished illustrator and typographer from Canada, was invited as a visiting artist to Graphic Design MFA Studio III. The project she introduced was to choose two words and express their meanings. I selected "Tension" and "Fear" from a word list. I wanted to explore the expression of these two words with freehand lines by reinterpreting traditional hand drawing-based calligraphy in new code-driven calligraphy with Processing. To visualize my idea in Processing, I was advised by Dr. Ge Jin, who specializes in graphics within computer science. He recommended that I find a related algorithm (a logical sequence in computer science) and libraries—a collection of sub-programs used to develop software in Processing.

Caligraft, created by Ricard Marxer Piñón, crafts computational calligraphy based on the Geomerative library in Processing. Geomerative is a library for Processing that helps forward the tasks of handling vectorial shapes such as the font used[3]. It is constructed using public fonts (such as Arial and Times Roman) as seed fonts with basic shapes such as lines, circles, curves, and so on. To draw the form of Tension and Fear, I chose a simple font style, Arial, as the seed font. Caligraft has variables to enable

changing the font styles, the string of the letters, the speed of deformation, the variation of ink and precision, as well as a drawing function. Through working with variables, I could reinterpret the forms of Fear and Tension. It showed me the possibilities how hand drawing-based calligraphies could innovatively create a new dimension of textual typography with Processing[4]. Examples of Caligraft and use of the library can be found and downloaded at www.caligraft.com.

Several typography projects were created by using algorithms and libraries with Processing during my two years in the MICA GD MFA program. For instance, I designed a three-dimensional virtual typographic tree with the word "Code," in reference to the L-system algorithm, which generates fractals and realistic modeling of a tree, programmed by Jer Thorp. As a teaching intern for Graphic Design MFA Studio in the spring of 2007, I also assisted my classmates on several interactive typography projects. For instance, Viviana Cordova's Genetic Typography, based on the Conway's Game of Life code by Mike Davis, and Ryan Gladhill's interactive sound typography have been assisted by me. I also developed several letter sets by using Caligraft as tutorial samples.

Most of my experimental typography has been included in *Graphic Design: The New Basics*, as was my initial wish. However, on Ellen Lupton's recommendation, I have extended my research to this present Processing tutorial book for graphic designers, *Type + Code*. At the beginning of this project, I wondered if graphic designers would want to use my codes as tutorial sources. I knew that most of my graphic design major classmates at MICA had no clue how to interpret my experimental projects and perhaps thought that my projects went against their graphic designer's ways. However, I changed my mind about sharing my codes as tutorial codes for graphic designers because I wanted to help them understand my projects through *Type + Code*. I hope that it will help extend their application tools beyond the expensive major computer graphic tools, such as the Adobe packages, which rely on a graphic user interface system, to the free Processing language, which is based on directing coding.

Thanks to an introduction from Ellen Lupton, I had the great honor to meet Ben Fry, the co-creator of Processing, and assist in his Processing workshop at the Cooper-Hewitt National Design Museum in March 2007. He told me that my tutorial for typography with Processing would be the number one request from designers. All of the other Processing-related tutorials and examples are, I believe, programmer oriented, so it is still not easy for artists and designers to get into the language.

3 *Caligraft*. Ricard Marxer Piñón. www.caligraft.com.

4 Ibid.

I met Peter Cho, a pioneer in code-driven typography, during my presentation at the third biennial design education symposium *School of Thought III* at the Art Center College of Design in 2007. He pointed out that my tutorials could limit the designer's creativity in the same way as Photoshop filters and Photoshop plug-ins have. These tools may help designers create specific visual effects, but have made visual design routine and unimaginative, with graphic designers just relying on them. So I hope that my readers will use my code as a reference to create their own codes, rather than simply leaning on my tutorial examples.

In spring 2007, I taught a three-week Processing workshop as a teaching assistant for Graphic Design II, directed by Ellen Lupton at MICA. I presented my elementary tutorials for students and gave them a project—create three designs using at least two of these functions in Processing: repeat, rotate, move and random. Some of my students created experimental designs, but most of them just used my tutorial code instead of creating something "new" because they did not know, and could not learn, all of the basic and necessary syntax—including how to define variables, how to define their own statements with parameters and how to use algorithms and libraries—within the limited time period of three weeks.

Therefore, I believe that Processing should be at least a one-semester course in the upper-level classes of graphic design programs or in the freshman classes of graphic design programs for graphic design majors, who are more open to learn new creative ways. In that first semester, students would learn how to visualize their ideas by using the basic and essential syntax of Processing, formulating their own defined statements and using algorithms and libraries.

All of my typography with Processing was extended to interactive pieces by using sound input, video tracking and cell phones in my other classes—Interactive Media II, Interactive Media III and Senior Seminar directed by James Rouvelle, who is co-chair for the Interactive Media Department at MICA. In a possible second semester course, students would learn how to animate their visual designs and make them interactive for other means of communication by using microprocessors, sensors, microphones, video and Web cameras, and so on. And the next version of this book, *Type + Code*, would handle interactivity with typography.

Graphic designers may ask why they should have to learn Processing, because they may be happy enough utilizing Adobe and other packages. My answer is that you should choose Processing for these reasons: if you want to be free and independent in the visual creative process (which is limited by the graphic user interface system of Adobe and other applications); if you desire to explore new, fresh, experimental, powerful typography with codes; and if you want to make your typography interact with

Selected Bibliography:

Graphic Design: The New Basics.
Ellen Lupton and Jennifer Cole Phillips. Princeton Architectural Press. New York, 2008.

Creative Code.
John Maeda. Thames & Hudson. New York, 2004.

Thinking with Type.
Ellen Lupton. Princeton Architectural Press. New York, 2004.

Processing.
Ben Fry and Casey Reas. www.processing.org.

Caligraft.
Ricard Marxer Piñón. www.caligraft.com.

various methods of communication such as sound, video and cell phones, since Processing offers an entry into that world of the physical interface system.

I have been lucky enough to have two dear collaborators, Viviana Cordova and John P. Corrigan, on this book. They are former classmates from the MICA GD MFA program. I invited John to be managing editor and designer and Viviana to be a co-author. As managing editor and designer, John delivered coherent and consistent guidance, whenever Viviana and I were staggering in unclear directions among all the typography, motion graphics and interactivity for the book. He created all of the beautiful layouts and reinterpreted all of the letters generated by Processing into a meaningful and organized typography. As a co-author, Viviana developed parts of the basic chapter and wrote the description in the book for graphic designers. Also, she created the book cover design by using my three-dimensional typographic tree.

The *Type + Code* team was honored to present the book in April 2008 at the AIGA Graphic Design Educator Conference Massaging Media II, in Boston. We invited Kate Harmon as a Web site designer, at the final stage of this project. Now this book is out of my hands. I hope that it will help graphic designers learn the practice of code-driven typography with Processing, as well as guide them in exploring new ways of visual creativity.

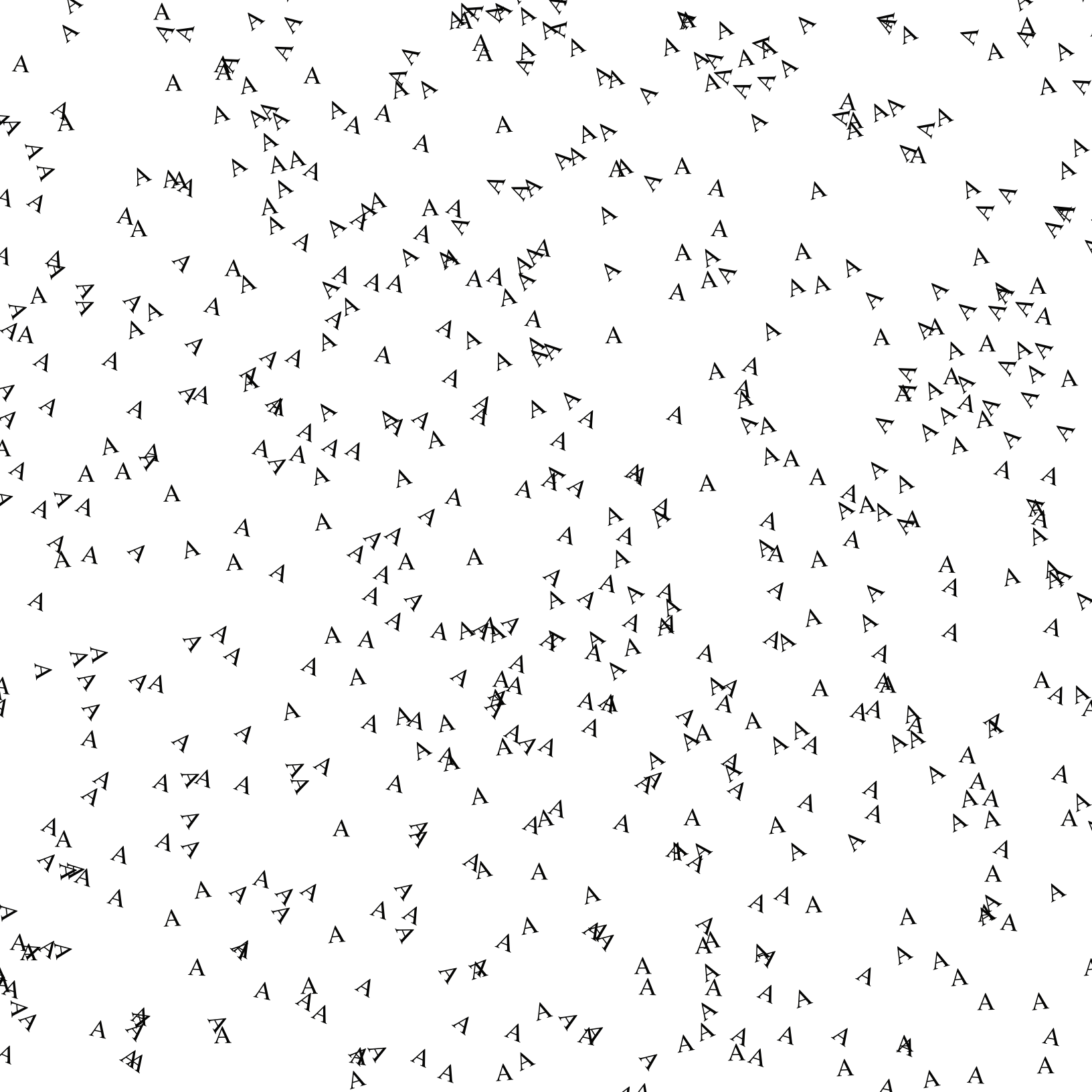
—Yeohyun Ahn

Yeohyun Ahn initiated the majority of the content of *Type + Code* as a graduate student at the Maryland Institute College of Art from 2005–07. She continued developing the content as Visiting Assistant Professor in the Digital Art and Design Program Art Department at Towson University through 2007–08.

Yeohyun Ahn.
Viviana Cordova.

Spike typeface *ƒ*.


ISJTL
KMWN
OYPZQ



BASIC TYPOGRAPHY

After downloading Processing on to your computer from the Web site www.processing.org/download, you can easily start creating. The software installation is self-explanatory. Processing is user friendly, and it has an interface similar to a Notepad (text editor) interface, with the addition of menu properties such as Run to display and Stop to stop. Traditional software has many tools already embedded in the program, often as icons, that automatically perform certain functions intuitively—for example, drawing, selecting and dragging a virtual textbox by using the arrow tool and the mouse. The result is that the designer has total control of the canvas document. Whereas, Processing functions operate in a lower level interface—you need to write code that enables you to access its library, which contains examples using the Processing language. With this in mind, you are ready to begin your first sketch file.

Go into the File menu at the top and choose New. Once your file is saved—by going again to File and choosing Save—you will have a .pde extension at the end of your named file. Inside your sketch, you can start writing code by choosing an example from any of the chapters in this book.

The code structure in Processing is the same for any execution. It begins by loading a typeface, specifying the size of the file, adding determined functions such as color, background, foreground, position, text and others. The structure order does not change, but its infinite options, determined by functions and numerical values, create rich, and sometimes unexpected, visual results. A new method of visualization and communication has arrived: simply by using the code functions of Processing, we can create work that looks as if it has been painstakingly designed.

In the first section, we introduce the statements for basic functions: text( ), fill( ), alpha( ), rotate( ), translate( ), for( ), string( ),if( ), pushMatrix( ) and popMatrix( ). We will be creating letters and patterns from letters. Always remember, adding a value inside any function results in the form of the letters being altered.

The examples in this book use letters to define values. If we use "i" as a value, then with only one statement for( ), we can create a simple syntax code, and are then able to design using add, multiply, subtract, and so on. Using mathematical formulas gives us the power to gain different results according to the variables used. By using functions combined with math formulas, and even algorithms, we are able to design visual graphics with Processing code. Once you have mastered the examples, you can combine them and come up with a whole new visual result. The more you practice and explore various results by testing and previewing your work, the more control you will have over your Processing file.

Designers are able to explore their own creativity using code-driven applications such as Processing, which help create experimental typography. You can simply start by choosing a function and use a value or variable. After writing your code inside your sketch and saving your existing file, you can play it with Run, which is the first option on the left in the main menu of the sketch. A window will open showing the final representation of what you have done so far using your code. To find more guidance in terms of functions, please visit www.processing.org/reference/index.html, which has details of the entire collection of functions for you to explore further.

Yeohyun Ahn.
Viviana Cordova.

PROCESSING BASIC LANGUAGE

Processing is an open source programming language, and its environment has been built for the electronic arts and visual design community. It can be used to express information visually in the form of art installations, academic exercises, posters, video games, animation and music videos. Processing files have a .pde extension at the end of every file; they can be easily exported as Java applets and conveniently uploaded to the Internet. In addition, the program is suitable for those who want their work to be printed in high resolution, because files can be opened in Adobe Illustrator. Once the file has been exported into a vector-based environment, it can be saved as an .eps file; and the Processing files are infinitely changeable, because of their rule-based design. In programs such as Illustrator, users are able to adapt the shape and size of the design and make any other changes they desire.

In addition, an international online Processing community has been growing strongly in recent years. The language and its users have benefited from this burgeoning group of talented artists, designers and developers who are sharing their knowledge by publishing online tutorials and a resource-filled blog at www.processing.org.

1.0

beginRecord(PDF)

Capture image to print

Here is a first tutorial code for graphic designers. Processing provides a library, `processing.pdf.*` to write PDF files directly from Processing. The vector-based PDF can be scaled to any size and output at very high resolutions. It provides open possibilities for graphic designers, who need high-resolution images for print-based publication design.

The screen size function variable is relatively small, `size(200,200)`, because graphic designers do not need to create a large size screen, such as `size(1500,1500)`, because the small size can easily be converted into a PDF and scaled up and down without losing image quality. Increasing the screen size will only cause an unnecessarily longer running time in Processing and will not benefit the recorded PDF file.

```
//import PDF library
import processing.pdf.*;

void setup() {
//size has to be placed always
  //at the beginning
size(200, 200);

//start record after size
beginRecord(PDF, "image.pdf");

//create your design

//end record
endRecord();
}
```

http://processing.org/reference/beginRecord_.html

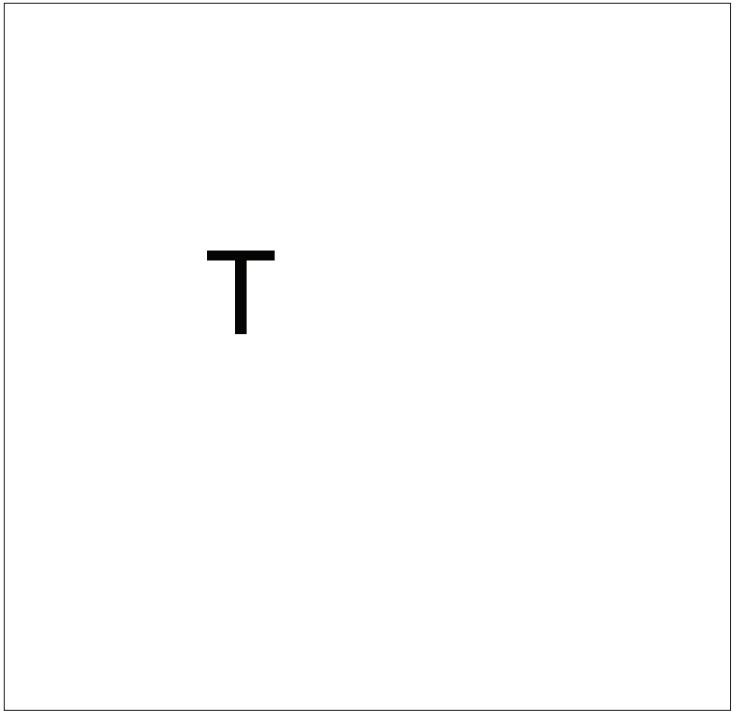
Yeohyun Ahn.
Viviana Cordova.

2.0

text( )

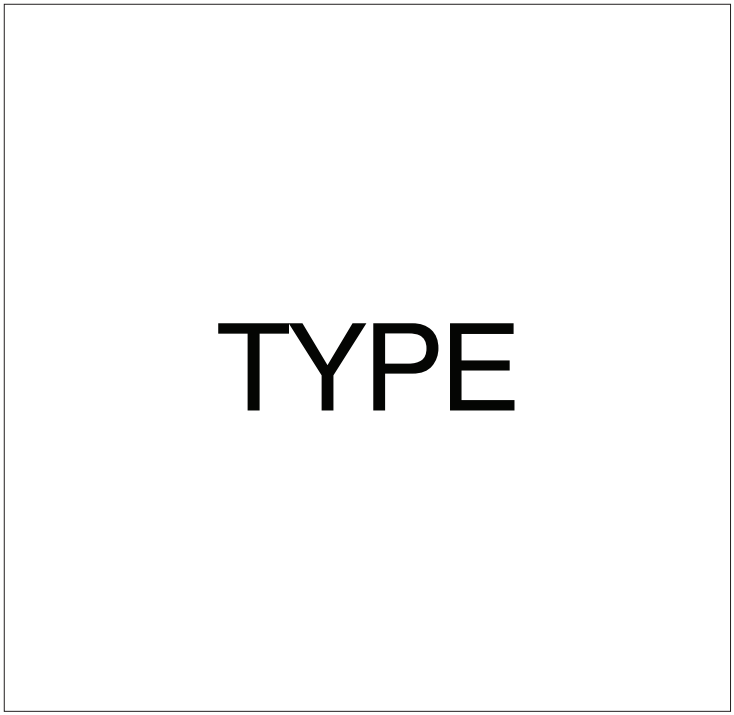
Letter and word

The function text( ) is essential for typography in Processing, since it is used to designate the letters and words that the other functions can transform. The example below shows how the function can be used. First, `myFont = createFont("Univers", 32)` calls for the desired font, in this case Univers at a point size of 32. Secondly, `textFont(myFont)` assigns that font (Univers, 32pt) to the text that we wish to have displayed. While the final function, `text("T")`, calls and displays the content; here it is the capital letter `"T"`.




```
PFont myFont;
void setup() {
//size has to be placed always at //the beginning
size(200, 200);
//create a white background
background(255,255,255);
//loading a font from your computer and size
myFont = createFont("Univers", 32);
textFont(myFont);
//fill the text color to black
fill(0,0,0);
//text written and location for //x:55 and y:90
translate(55,90);
//text function
text("T");
}
```

In case of errors, look for an updated version of the application at the Processing Web site. For more information, please visit www.processing.org/reference/PFont.html.




```
PFont myFont;
void setup() {
size(200, 200);
background(255,255,255);
myFont = createFont("Univers",
  32);
textFont(myFont);
fill(0,0,0);
translate(55,90);
text("TYPE");
}
```

This code changes the letter, `T`, in `text();` to the word `TYPE`.

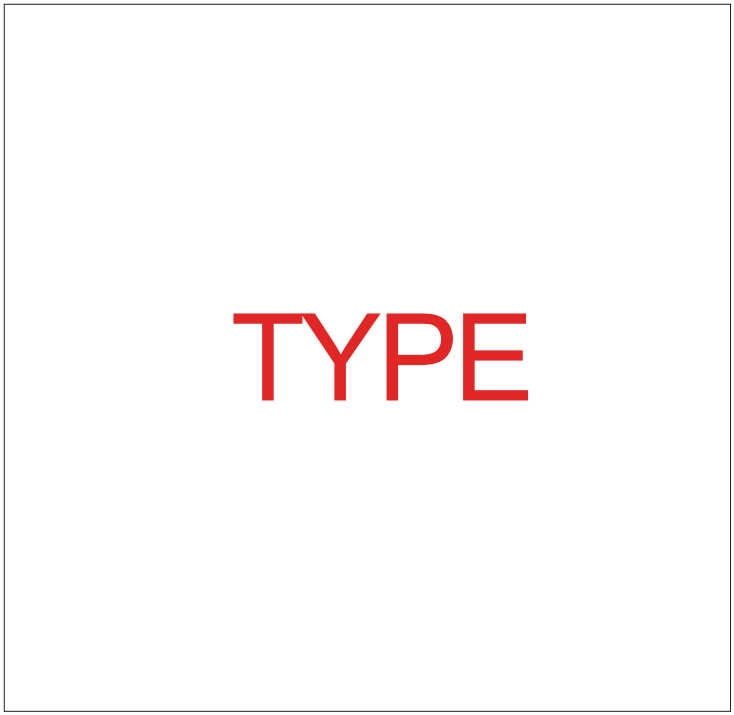
Yeohyun Ahn.
Viviana Cordova.

3.0

fill( )

Color

This allows to one to specify the color of objects within the display window. The numeric values in the function of fill, here for example, `fill(200,10,20);`, define the RGB proportions, which stands for `red, green and blue`, the standardized screen color system, with which you are probably familiar.



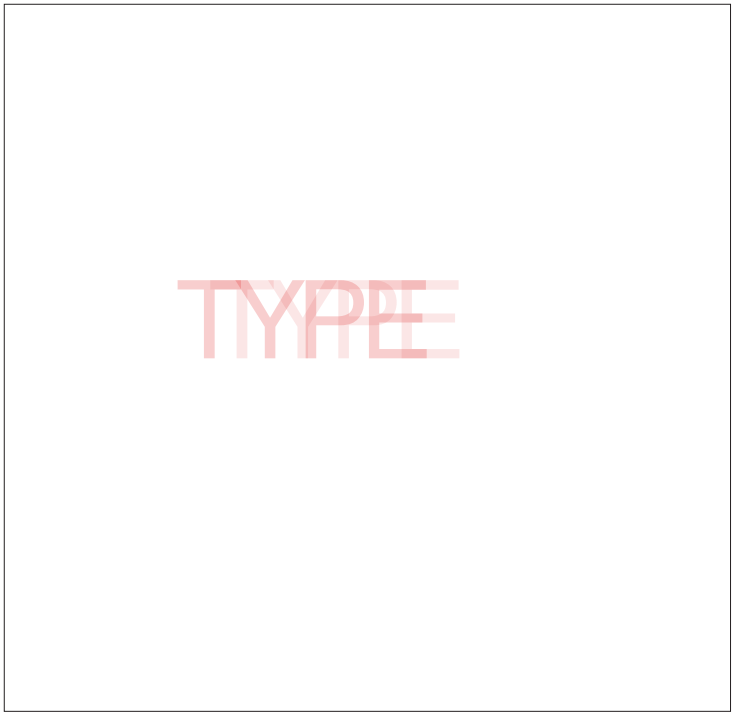

```
PFont myFont;
void setup() {
size(200, 200);
background(255,255,255);
myFont = createFont("Univers", 32);
textFont(myFont);
fill(200,10,20);
translate(55,90);
text("TYPE");
}
```

4.0

alpha()

Transparency

We use transparency to create layers of hierarchy and even motion. By manipulating the alpha value, we are able to control the percentage of transparent parts that visually interact with each other. Using the `alpha()` function, we can create more depth and hierarchy in our work. First, the `color()` function is called to create two color groups `a = color (200,10,20,38)` and color `b = color (200,10,20,58)`. Within the `color()` function the first three values define RGB, while the fourth is the percentage of transparency desired. In this case, color a has `38` and color b, `58`. The `fill()` function then determines color for both a and b. Finally, the `alpha()` function is written for `a alpha(a)` and `b alpha(b)`.




```
PFont myFont;
void setup() {
size(200, 200);
background(255,255,255);
myFont = createFont("Univers",
  32);
textFont(myFont);
//color a grouping is created
//inside color(r:200,g:10,b:20,
  transparency:38)
color a = color (200,10,20,38);
//using fill for color a
fill(a);
//using alpha for color a
alpha(a);
//position x:55 and y:90
translate(55,90);
text("TYPE");
//color b grouping is created
//inside color(r:200,g:10,b:20,
  transparency:58)
color b = color (200,10,20,58);
//using fill for color b
fill(b);
//using alpha for color b
alpha(b);
//position x:-90 and y:0
translate(-90,0);
text("TYPE");
}
```

In this example, the lighter value is coded first with a transparency value of 38. The second has a deeper transparency value of 58 and is then translated -90 from its original position.

Yeohyun Ahn.
Viviana Cordova.

5.0

translate( )

Position

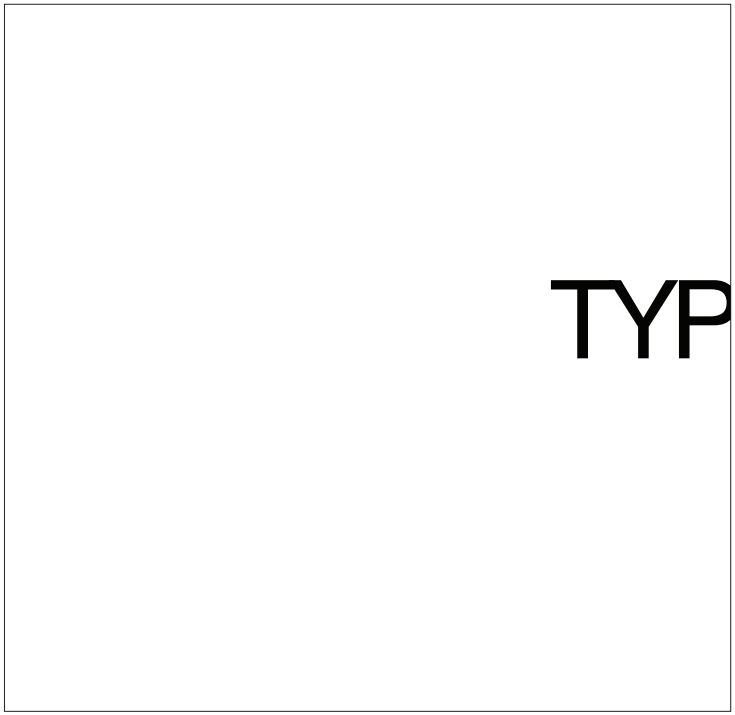
The translate( ) function controls the position of any image, object or, this time, letters shown in our window. The function translate( ) uses x, y axis coordinates to arrange position. (The horizontal axis coordinate is x and vertical axis coordinate is y.) In this case, the word "TYPE" is positioned towards the right side. For 3D work (for example, using 3D libraries such as OpenGL and JAVA3D), you can also add an axis coordinate of z, which is placed after y. This example, however, is based in 2D mode.

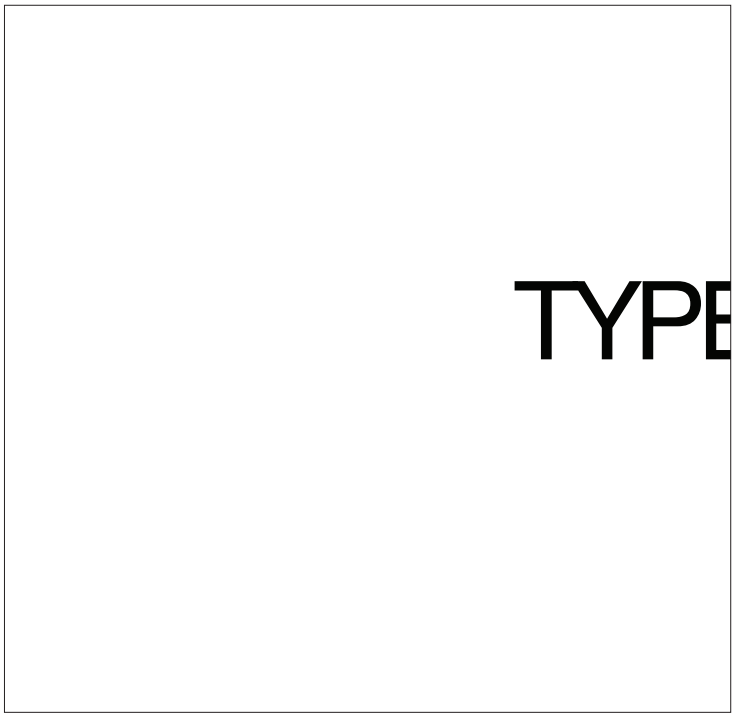
```
PFont myFont;
void setup() {
size(200, 200);
background(255,255,255);
myFont = createFont("Univers", 32);
textFont(myFont);
//position x:142 and y:90
translate(142,90);
text("TYPE");
}
```

In both examples, the position for the y coordinate remains the same, while decreasing the x coordinate from 142 to 125, shifts its position from right to left. Thus, you can determine the exact horizontal and vertical placement by changing the value for x and y .

Top:
`translate(142,90);`

Bottom:
`translate(125,90);`





6.0

rotate( )

Rotation

As its name implies, the rotate( ) function rotates an object or text according to the angle parameter that you assign, such as that specified in the mathematical formula shown below. You can change the number values of this circumference formula to suit your own design needs. In this case, the function `rotate(PI/3.0)` is using pi—often written as π—divided by `3`. The value `PI/3.0` can be represented in degrees (°) as 180°/30°, which is 60°. As a result, the word "TYPE" is at an angle of 60° from the horizontal.

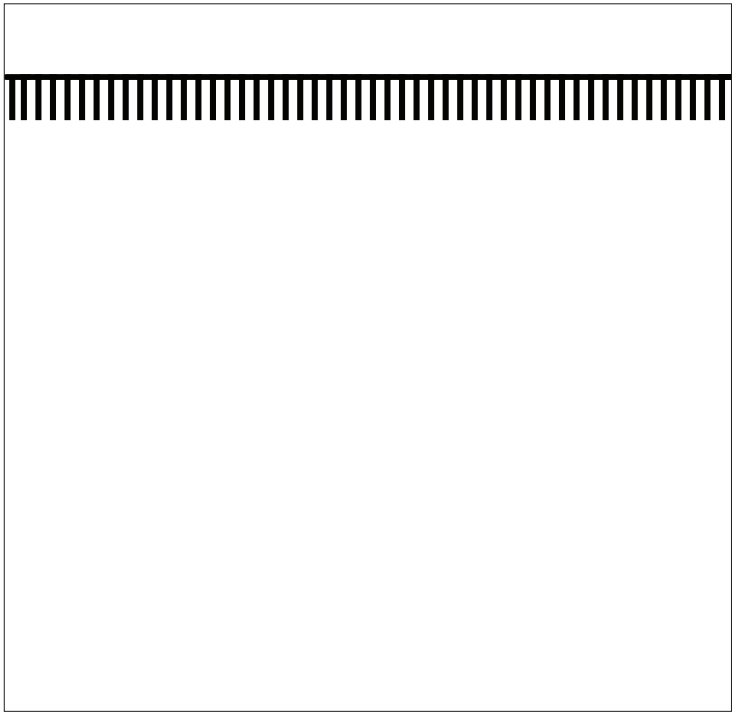

```
PFont myFont;
void setup() {
size(200, 200);
background(255,255,255);
myFont = createFont("Univers", 32);
textFont(myFont);
fill(0,0,0);
translate(50,50);
//rotation uses circumference formula
rotate(PI/3.0);
text("TYPE");
}
```

`PI/3` means 60°
`PI/6` means 30°
`PI` means 180°

By eliminating the circumference value of `3.0` from `rotate(PI)` and adding an alternate value `translate(100,100)`, the word rotates 180° (degrees) from its original orientation, creating the second example.

http://www.processing.org/reference/rotate_.html

Yeohyun Ahn.
Viviana Cordova.



for(int i=0;i<250;i=i+5)

7.0

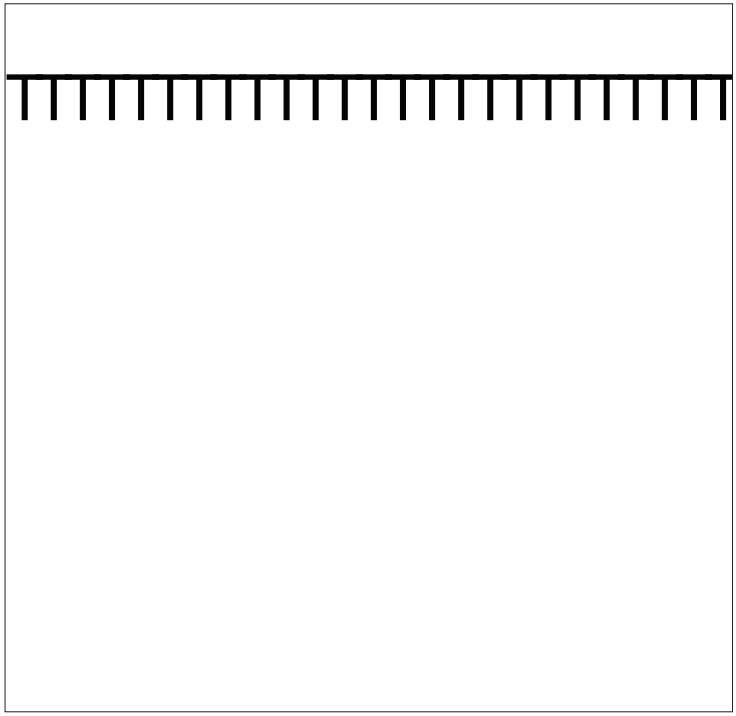
for( )

Repetition

We use the function for( ) to create a series of repetitions, here, a repetitive horizontal line formed by an uppercase letter "T". The distance between each letter is even and controlled by the values we add to for( ). The `for( )` function needs a beginning, middle and end, called: for(initiate; test; update), which are divided by a semicolon `(;)`. In this case, initiate is the beginning point (i equals 0 `i=0`), test sees if the value of i is less than 250 `i<250`, if it is, the function then goes to update, to which we have given the value of i equals i plus 5 `i=i+5`, so the value of i is increased by 5. By using this changing value i, we can have the letter "T" displayed 5 pixels apart across the screen. We need to use the text( ) function to do this `text("T", i,40)`. This calls the letter "T" and uses the variable i to control the x coordinate, while the y coordinate has a constant value of `40`, creating a horizontal line running off the canvas size window, which is 250 by 250 pixels. The y coordinate value places the line of "T"s high up on the canvas.

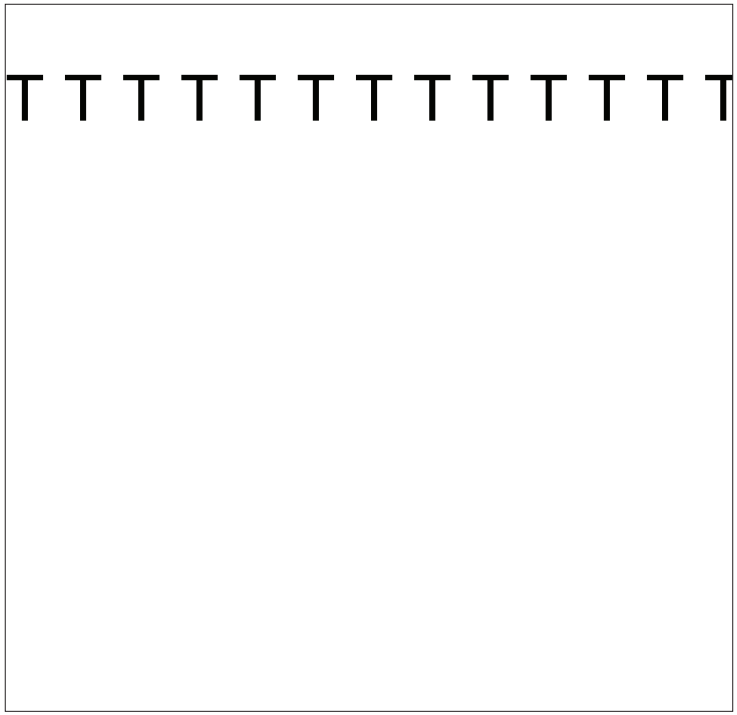
```
PFont myFont;

void setup()
{
size(250,250);
myFont = createFont("Univers",48);
textFont(myFont,10);
background(255);
//for(initiate;test;update)
for(int i=0;i<250;i=i+5)
{
fill(0,0,0);
//textFont(font,size)
textFont(myFont, 22);
//text(letter,i:x ,40:y)
text("T",i,40);
}
}
```

Here the letter "T" is repeated farther apart by changing the value of the increment to i from 5 to 10. The y coordinate remains the same at 40.



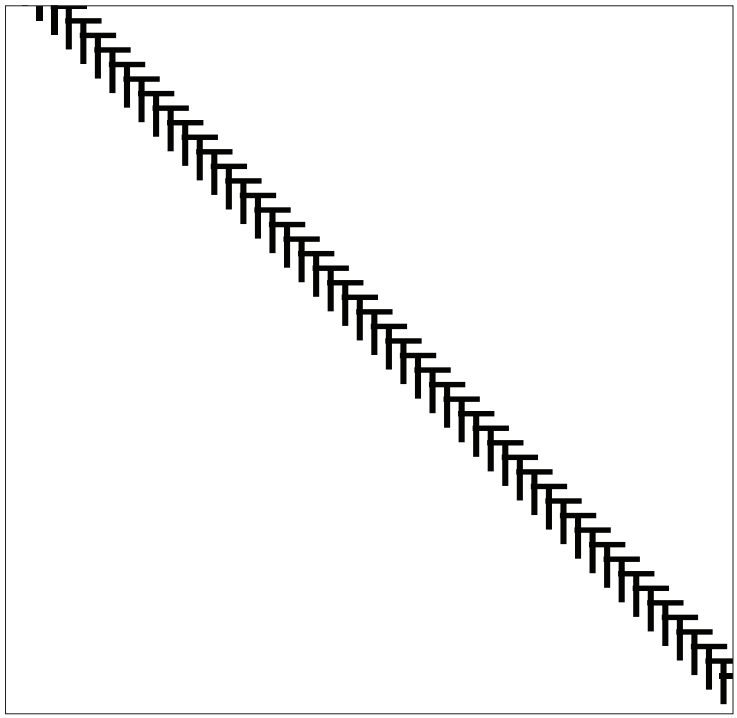
```
for(int i=-5;i<250;i=i+10)
```

The tile has less density, if the increment of the value i is changed from 10, to 20.



```
for(int i=0;i<250;i=i+20)
```

Yeohyun Ahn.
Viviana Cordova.

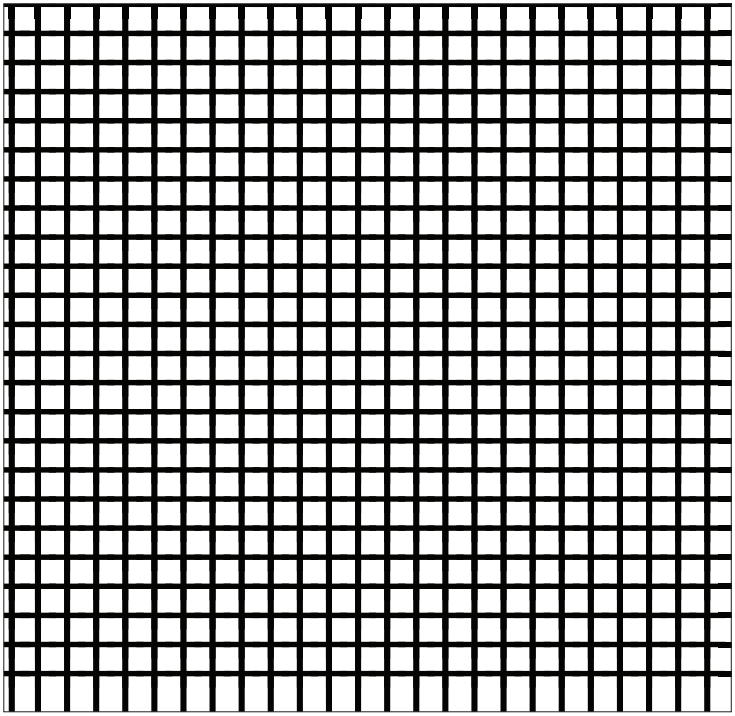


for(int i=-5;i<250;i=i+5)

```
PFont myFont;

void setup()
{
size(250,250);
myFont = createFont("Helvetica",48);
textFont(myFont,10);
background(255);
for(int i=-5;i<250;i=i+5)
{
// Write an additional for( ) statement here with a value j,
  creating a pattern across both the x and y coordinates filling
  the tile completely.
// Examples on following page: for(int j=-5;j<250;j=j+10)
{
fill(0,0,0);
textFont(myFont, 22);
text("T",i,j);
}
}
```

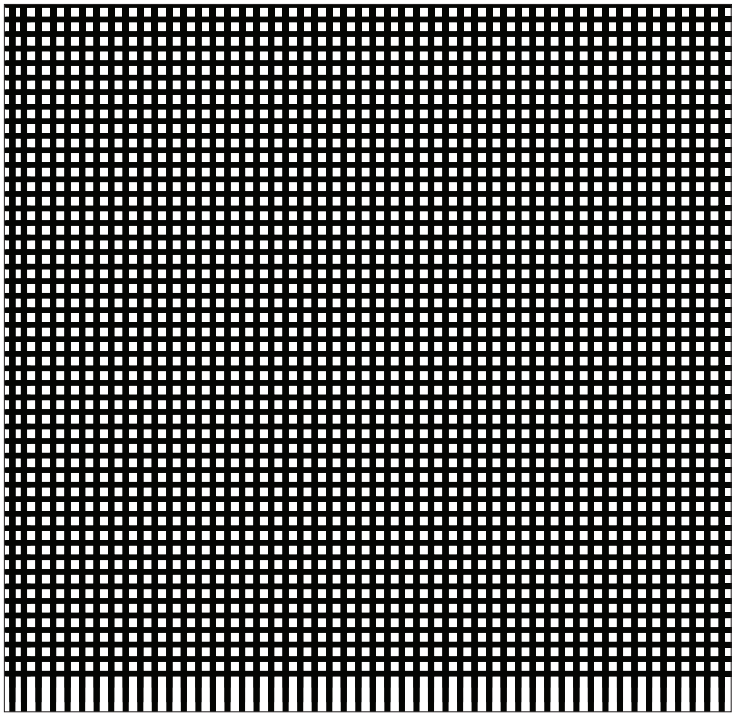
The letter “T” is repeated along the x coordinate, -5, 0, 5,10,15, 20, 25, ... and the y coordinate, -5, 0, 5,10,15, 20, 25, ... creating the diagonal effect.

Adding an additional for( ) statement with the value j, creates a pattern across both the x and y coordinates filling the tile with the letter "T."



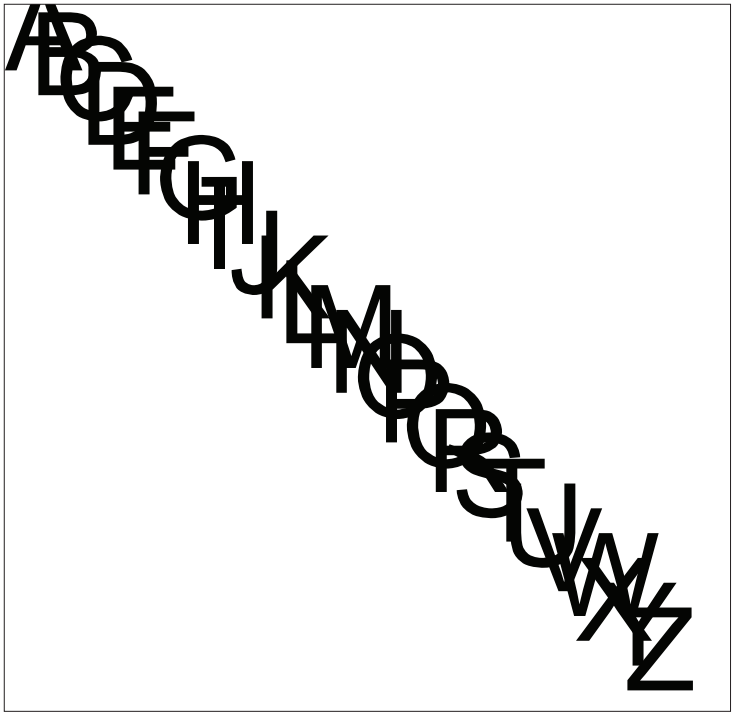
```
for(int i=-5;i<250;i=i+10)
for(int j=-5;j<250;j=j+10)
```

If the increment of the values i and j is decreased from 10 to 5, a more densely populated tile is created.



```
for(int i=-5;i<250;i=i+5)
for(int j=-5;j<250;j=j+5)
```

Yeohyun Ahn.
Viviana Cordova.



8.1

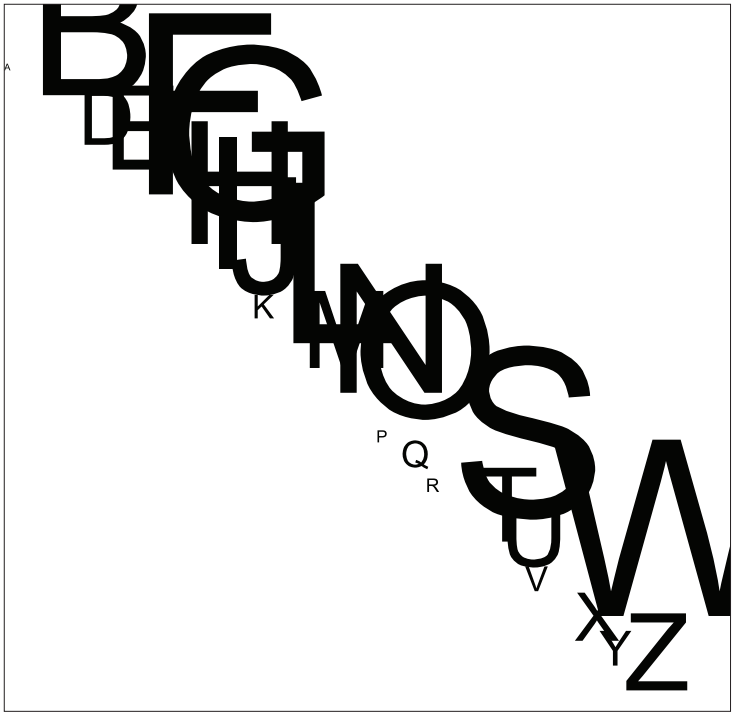
string( )

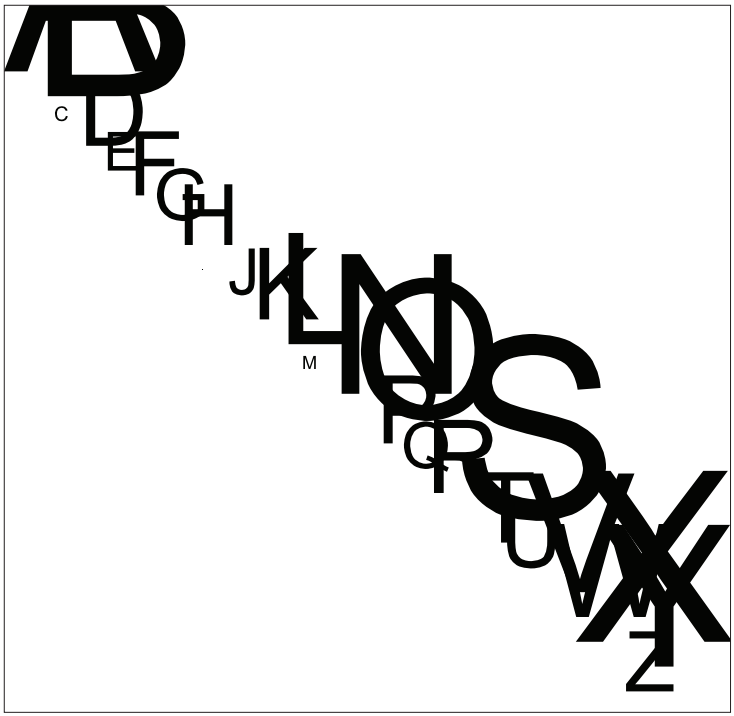
Structured sequence

A string is a sequence of characters. The string( ) function is used to sequence individual letters or entire words to determine how they are manipulated. The previous tutorial codes were explained with the letters and words presented in a non-specific order, in this example we choose to manipulate the letters of the alphabet from A to Z with the for( ) statement. Using `string( )` function is a smart way to comparatively alter the positioning determined within the confines of the for( ) statement.

In this code, the `for( )` statement determines the variables for both the x coordinate position and y coordinate position with regularity (so the sequence of letters is displayed diagonally), while `string( )` is used to place the alphabet from A to Z, defined as `st.charAt(i)`, which can then be utilized within the `text( )` function.

```
PFont myFont;

void setup(){
size(450,450);
myFont =
  createFont("Helvetica",48);
background(255);
String st =
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
translate(10,40);
for(int i=0;i<26;i=i+1)
{
textFont(myFont,70);
fill(0,0,0);
text(st.charAt(i),i*15,i*15);
}
}
```





8.2

string()

Random sequence

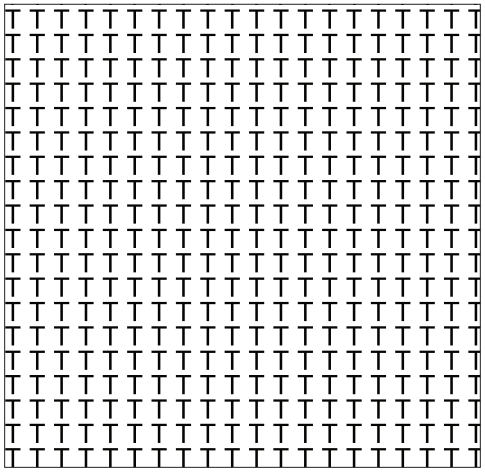
As in the previous example, the for() statement is used to create the variables for the x and y coordinate positions regularly, and `string()` is used to place the alphabet, defined as a sequence `st.charAt(i)`, for the use of the `text()` function. However, here we add the `random()` function, so that the size of the letters are randomized between 0 and 170. (We will see more of the random() function in the next chapter.)

In both code examples, the string() function pulls the entire sequence of letters from the alphabet. The placement of each letter remains consistent, but in this case, the random() function increases or decreases the size. The beauty of the string() function is that, though a sequence is created, it allows each letter to work (and be transformed) independently.

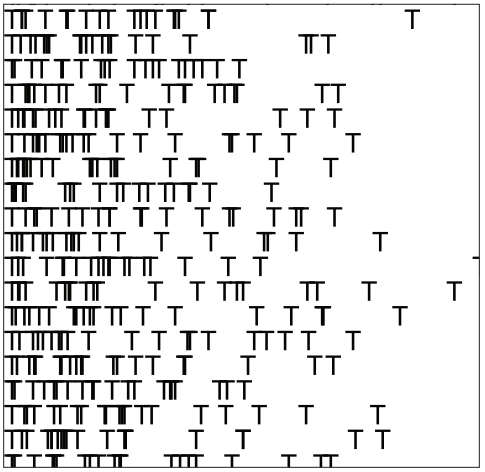
```
PFont myFont;

void setup(){
size(450,450);
myFont =
  createFont("Helvetica",48);
background(255);
String st =
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
translate(10,40);
for(int i=0;i<26;i=i+1)
{
textFont(myFont,random(170));
fill(0,0,0);
text(st.charAt(i),i*15,i*15);
}
}
```

Each time Processing runs this code structure, the random() function generates a unique and varied design, as the two examples on this page show. Notice how the letters remain relatively fixed across the artboard.

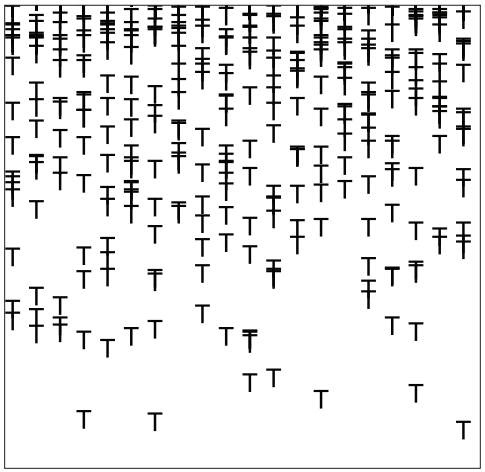
Yeohyun Ahn.
Viviana Cordova.



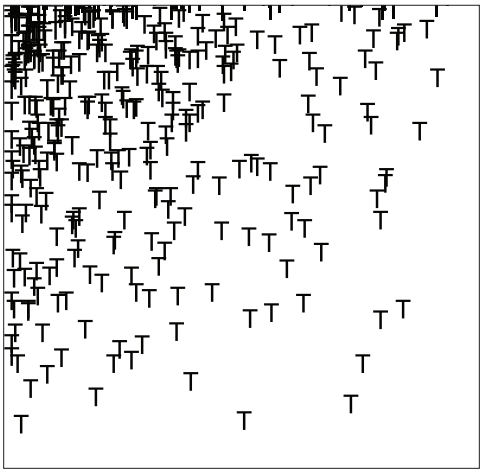
text("T", i, j)



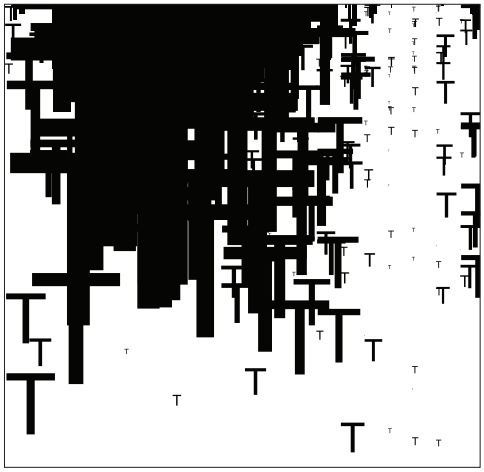
text("T", random(i), j)



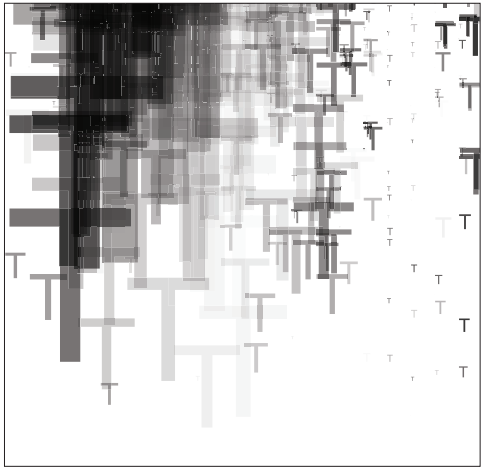
text("T", i, random(j))



text("T", random(i), random(j))



textFont(myFont, random(abs(i-330)))
text("T",i, random(j))



fill(0,0,0,random(abs(i-150)))
textFont(myFont, random(abs(i-330)))
text("T",i, random(j))

9.0

random()

Random

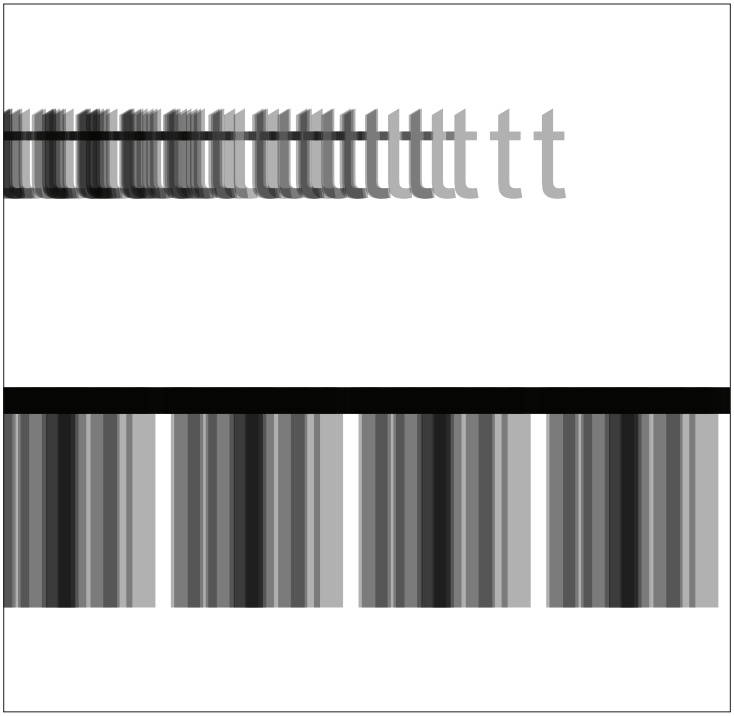
This function is fairly exclusive to Processing. It generates random numbers within a given function. Each time the random() function is called, it returns an unexpected value within the specified range. The random() function can be added to any of the existing basic code structures, which have parameters. On the facing page, we are using an example created earlier (top left) and adding random(). In the top right example, `text("T",random(i), j)`, only the x coordinate uses the random() function. In the bottom left example, it is the turn of the y coordinate to be randomized and at the bottom right, `text("T",random(i), random(j))`, both the x and y coordinates use random().

On this page, the top left captured image uses random() twice: first, in `textFont(myFont, random(abs(i-330)))`, where the `textFont()` function uses the myFont and `random()` functions to determine font size); secondly, in the text() function to randomize the letters' positions along the y axis. The abs() function is used to convert any negative values such as -330 to positive values such as 330. Here it is used to ensure the text size is always positive.

The bottom left design adds a third random() function (which appears first in the code): the fill() function, which utilizes `random(abs(i-150))` to vary the transparency value unexpectedly.

```
PFont myFont;

void setup()
{
size(400,400);
myFont =
  createFont("Univers",48);
background(255);
for(int i=-0;i<400;i=i+20)
{
for(int j=-0;j<400;j=j+20)
{
fill(0,0,0);
textFont(myFont, 22);
text("T", i, j);
}
}
}
```

Yeohyun Ahn.
Viviana Cordova.



10.1

if( )

Hierarchy

Again, we use the `for()` and `text()` functions, but are also going to introduce the function `if()` to create a hierarchy. This function uses the value of i `if(i>300)` where its value needs to be more than 300. The results are that the lowercase "`ttttttt`" (defined in the `text()` function `text("ttttttt",random(i),50)`, where `50` represents its y coordinate) is displayed continuously using `random()` to determine its x coordinate. (The uppercase "`TTTTTTTT`" uses `random()` to fix the x coordinate as well.)

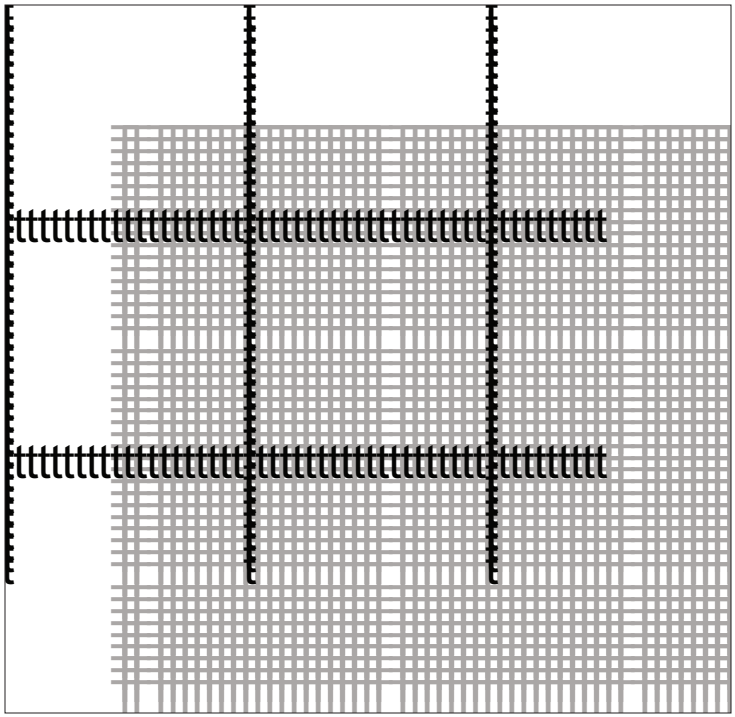
Overall, the repetitive visual of lowercase "t" and uppercase "T," using two different font sizes, 50 and 120, creates a two-level hierarchy. Using `if()` function gives a unifying value for i creating a harmonious movement in the image for both lowercase and uppercase.

```
PFont myFont;
void setup(){
size(300,300);
myFont =
  createFont("Univers",48);
background(250,250,250);
for(int i=0;i<300;i=i+20)
{
if(i>300);
{
textFont(myFont,50);
fill (0,0,0,80);
textAlign(CENTER);
translate(0,0);
text("ttttttt",random(i),80);
textFont(myFont,120);
fill (0,0,0,80);
textAlign(CENTER);
translate(0,0);
text("TTTTTTTT",random(i),240);
}
}
}
```

10.2

if( )

Layers



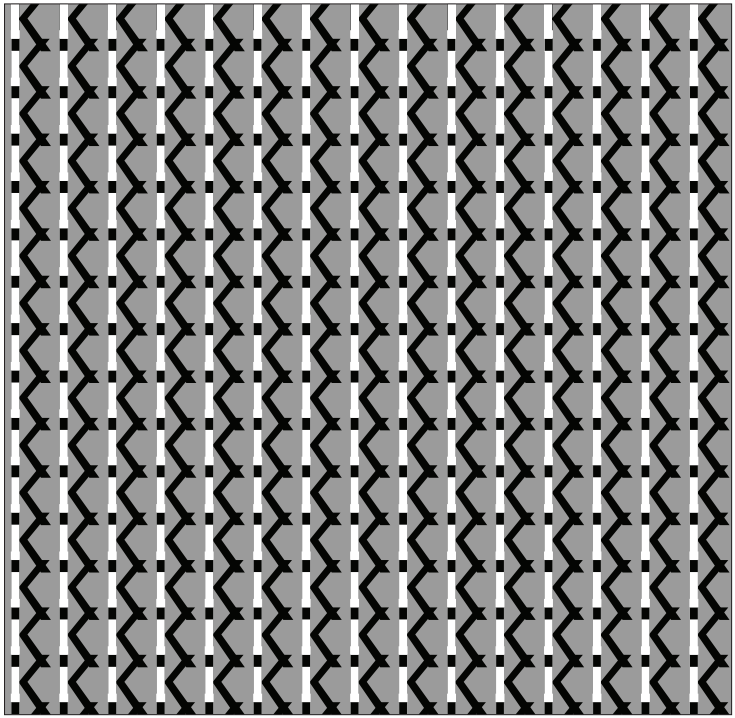
In this example, we are building hierarchy using two layers. One layer is composed of densely spaced "T"s the other a widely spaced grid of "t"s. They are formed by two `for()` functions: `for(int i=-10;i<250;i=i+5)` and `for(int z=10;z<250;z=z+5)`. Each of them contains one value: `i` and `z` respectively. If you wanted to build more layers, the code would need more values than `i`, `z`, and more corresponding `for()` functions.

The if( ) function is used here to "test" the results from the for( ) function and if they meet certain criteria to then execute the code that produces the grid of lowercase "t"s. Namely, it checks the values of `i` and `z`, and if either is exactly divisible by 100, it executes the text( ) function to display a "t." Thus, we get grid lines spaced 100 pixels apart (in both directions).

The final stage of the process is defined by the `else{}` function, which determines what is executed last. In this case, it is the uppercase "T," by using the following functions: `fill()`, `textFont()` and `text()`, which define its color, font and position. The last function, `text("T",i+55,z+55)`, defines the position of the x coordinate as `i` plus `55` and y as `z` plus `55`.

```
PFont myFont;
void setup(){
size(300,300);
myFont =
  createFont("Univers",48);
background(255);
textFont(myFont,80);
for(int i=-10;i<250;i=i+5)
{
for(int z=10;z<250;z=z+5)
{
if(i%100==0 || z%100==0)
{
fill(0,0,0);
text("t",i,z);
}
else{
fill(170,167,167);
textFont(myFont,20);
text("T",i+55,z+55);
}
}
}
}
```

Yeohyun Ahn.
Viviana Cordova.



10.3

if( )

Figure/ground

We will create a figure-ground pattern using the repetition of two letters—uppercase "K" and lowercase "i." The "K" is black and font size 35, while "i" is in light gray and font size 40. The "i" has a line quality presence under the "K," which brings both letters to the front. The "trapped" white spaces in between become the background, helping to balance the repetition of both letters across the canvas.

To achieve this, we use two `for()` functions, `for(int i=0;i<400;i=i+20)` and `for(int j=0;j<400;j=j+20)`, with two variables, `i` and `j`. Also, these variables are used in the `text()` function `text("i", i, j)`, where `i` defines the x coordinate and `j`, the y coordinate.

As in our previous example, the `if ()` statement, `if(i%100==0 || j%100==0)`, tests `i` and `j` by seeing if they can be exactly divided by `100` (thus, the remainder would equal `0`). In addition , both values i and j are being compared using `||` to test whether they are true or false, before executing the code to create the "i"s. The final result is a rewarding pattern using both letters.

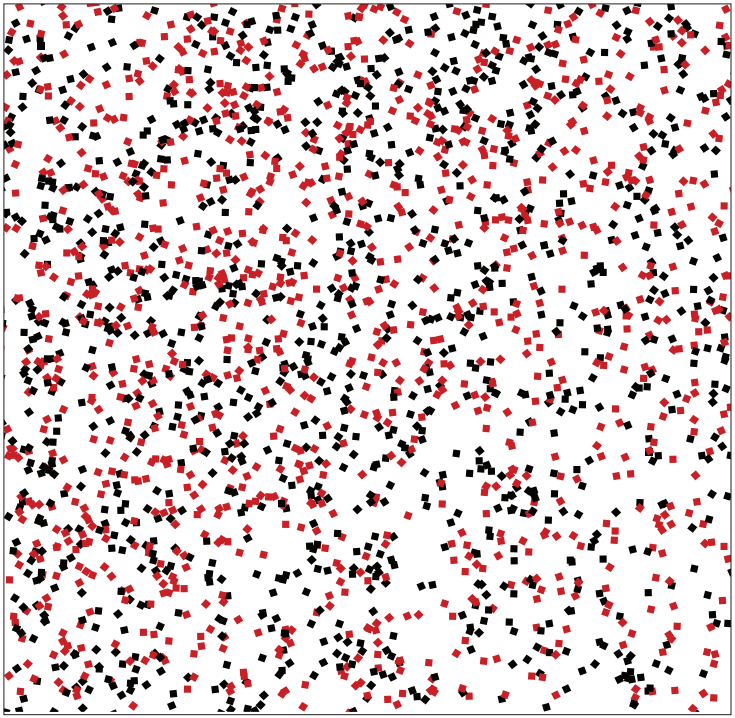
We see an overlap using "i" and "K" throughout the vertical repetition process. In addition, because "i" and "K" have been incorporated using most of the functions such as for( ), if( ) and text( ), it reinforces the idea of balance and harmony that is created by the pattern.

```
PFont myFont;
void setup(){
size(300,300);
myFont = createFont("Univers-
  Condensed",48);
textFont(myFont,10);
background(155,155,155);
for(int i=0;i<400;i=i+20)
{
for(int j=0;j<400;j=j+20)
{
fill(0,0,0);
textFont(myFont,35);
text("K", i, j);
if(i%100==0 || j%100==0);
{
fill(255,255,255);
textFont(myFont, 40);
text("i", i, j);
fill(170,167,167);
}
}
}
}
```

10.4

if( )

Color



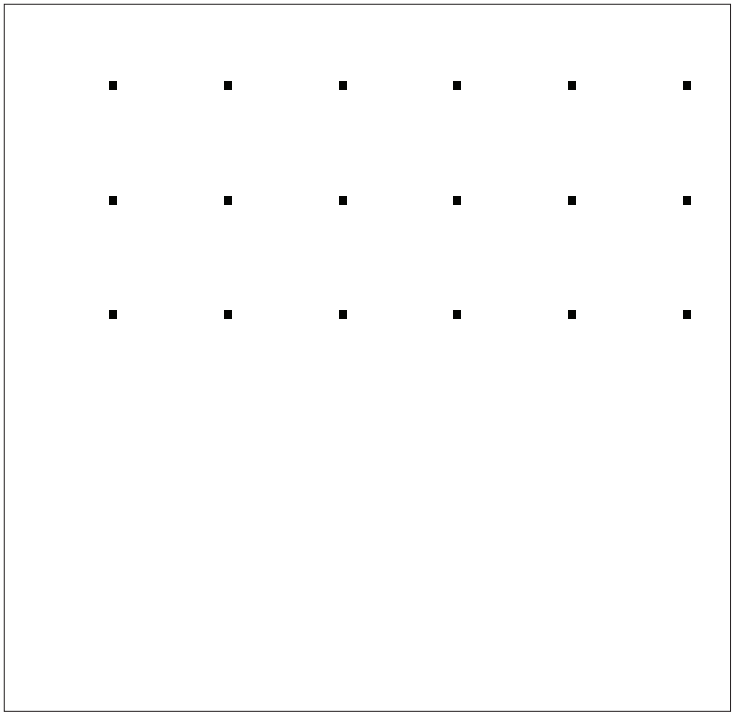
This example of the conditional, if( ), is designed to let us interact with the canvas. After the void setup, size, and font functions, but before the if( ) function, `void draw( )` is needed because the user will be drawing on the canvas, in this case with the mouse. By using the `if(mousePressed)` function, one can use a mouse to interact with the canvas and create a unique captured image. By clicking and/or holding down the mouse button, we create periods around the canvas. Results vary according to the manipulation of the mouse.

At the beginning, we have `int count=0`, which defines a value that counts from 0; therefore, the run file will always start from 0 or blank. As mentioned in previous pages, we are using if( ), for( ) and else, to create hierarchy within our code.

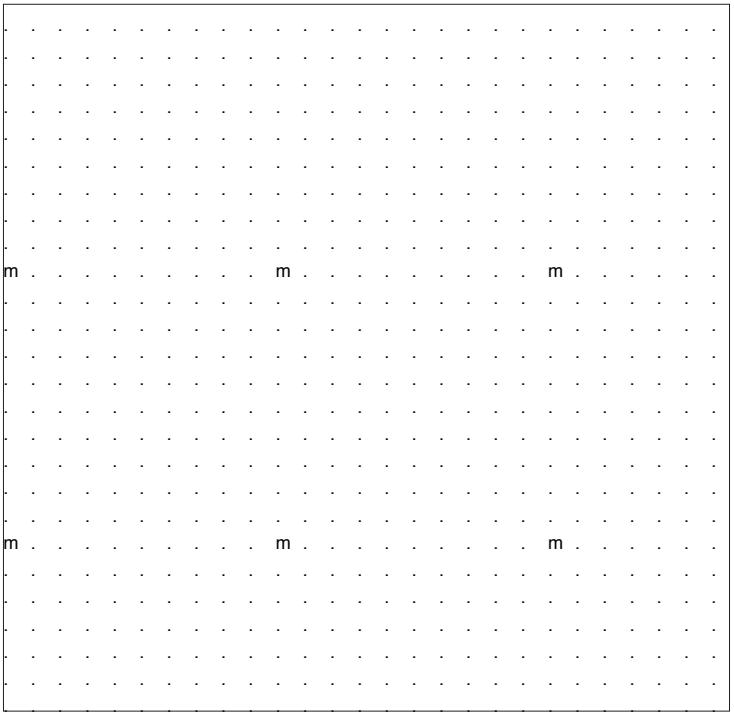
When the code is entered, test it by choosing Run and a window should come up. You will have an empty canvas and you can create your image by clicking/holding down the mouse button. The image will stop capturing once you close the window. To give the basis for the red and black periods, there are two text( ) functions and two levels of color: red `fill(200, 10, 20)` and black `fill(0, 0, 0,255)`. In addition, we are using random( ) for x and y coordinates. Finally, else handles when the code is executed for the second, red, period, "."

```
PFont myFont;
int count=0;
void setup(){
size(500,500);
myFont =
  createFont("Univers",48);
background(255);
textFont(myFont,42);
}
void draw(){
if(mousePressed){
for(int i=300;i<800;i=i+1)
{
if(count%2==0)
{
text(".", random(abs(i-250)+i),
  random(abs(i-250)+i));
fill(0, 0, 0,255);
rotate((PI/3.0)+i);
}else
{
text(".", random(abs(i-250)+i),
  random(abs(i-250)+i));
fill(200, 10, 20);
rotate((PI/3.0)+i);
}
}
count++;
}
}
```

Yeohyun Ahn.
Viviana Cordova.



```
for(int i=-30;i<900;i=i+30){
for(int z=-30;z<100;z=z+30){
  if(i%100==0 && z%100==0){
```




```
for(int i=-30;i<900;i=i+30){
for(int z=-30;z<900;z=z+30){
  if(i%100==0 && z%100==0){
```

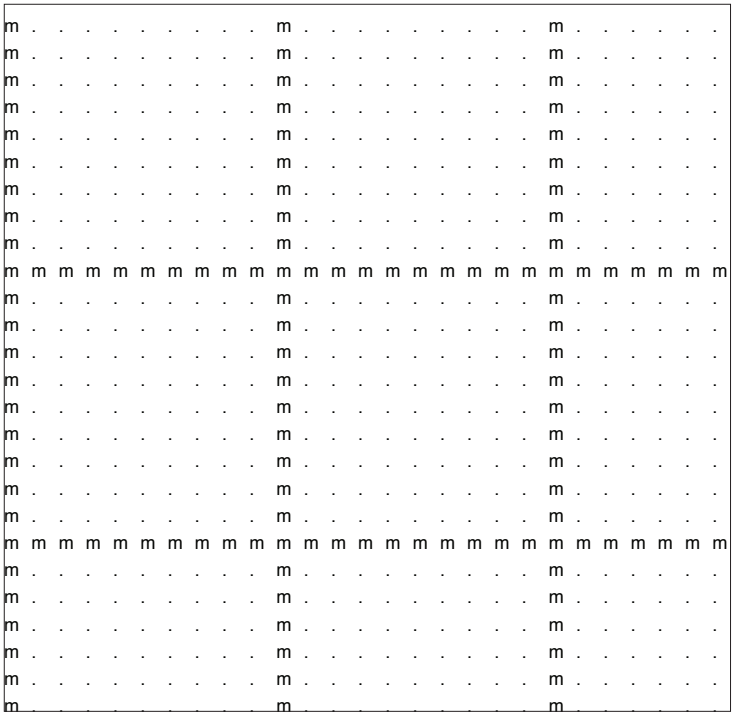
10.5

if( )

Conditional

All the conditional code structures we have looked at in this chapter use a combination of the functions if( ) and for( ). When using both functions, the code will execute each according to its own algorithmic formula. In this set of examples, the text( ) function takes two single letters: lowercase "m" and the period "." While, for( ) is given values: i and z. The if( ) function is considered a conditional function. It is used to create hierarchy within our code and it "decides" which code should be executed first.

In this case, it is the display of the letter "m" along 100 by 100 grid lines, resulting from the if( ) function testing the i and z values (and when they equal multiples of 100 declaring them "true" and displaying "m," in a similar fashion to our previous examples). Secondly, the `else{}` function calls the last or secondary functions such us `fill()`, `textFont()` and `text()` to display the period "." Its position within the canvas creates a secondary layer, placing it in the background.



```
for(int i=-30;i<900;i=i+30){
for(int z=-30;z<900;z=z+30){
  if(i%100==0 || z%100==0){
```

```
import processing.pdf.*;
PFont myFont;
void setup(){
size(800,800);
beginRecord(PDF, "if.pdf");
myFont =
  createFont("Univers",48);
background(255,255,255);
for(int i=-30;i<900;i=i+30){
for(int z=-30;z<900;z=z+30){
if(i%100==0 || z%100==0){
fill(0, 0, 0);
text("m", i,z);
}
```

```
else {
fill(0, 0, 0);
textFont(myFont,20);
text(".", i,z);
}
}
}
endRecord();
}
```

Yeohyun Ahn.
Viviana Cordova.

11.1

for( ) and rotate( )

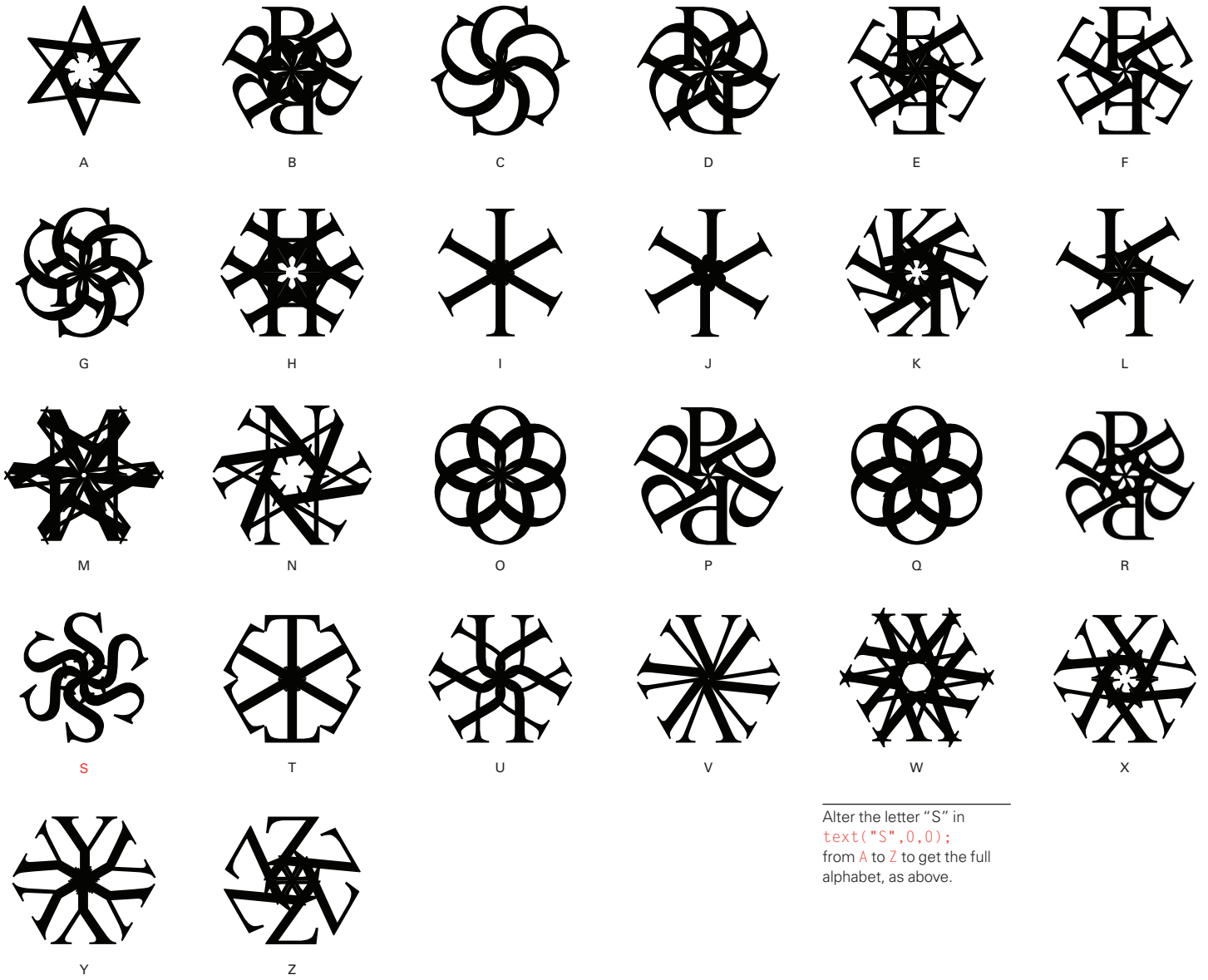
Repetition and rotation

Using repetition with `for( )` and rotation with `rotate()`, a modified serif alphabet has been created (see right). The amount of repetition is set with `for(int i=0;i<6;i=i+1)`— it starts with i equals 0, then tests if i is less than 6, and creates the new value of i as i plus 1. This ensures the letter is rotated six times. Color is set as black `fill(0,0,0)` and the letter has been aligned in the center with `textAlign(CENTER)`. Finally, `pushMatrix()` sets the structure for our finished visual using the value i in `rotate(PI*i/3)` (as the value of i increases as a result of the for( ) function, so the letter is rotated) and `text("S",0,0)`. The function of popMatrix( ) closes this part of the operation.

```
PFont myFont;
void setup(){
size(800, 800);

background(255,255,255);

// String[] fontList = PFont.
  list();
// println(fontList);

myFont = createFont("Times-
  Roman",48);
textFont(myFont,272);
translate(400,400);

for(int i=0;i<6;i=i+1)
{
fill(0,0,0);
textAlign(CENTER);
pushMatrix();
rotate(PI*i/3);
text("S",0,0);
popMatrix();
}
}
```

If you want to change the font styles, please delete // on the code (right) and all of the available fonts will be displayed on the bottom window in Processing. And then, if you want to change the font, for example, Times-Roman to Helvetica, please change the "Times-Roman" in `myFont = createFont("Times-Roman",48)`.

PushMatrix and popMatrix are introduced on page 31.



Alter the letter "S" in
`text("S",0,0);`
from A to Z to get the full alphabet, as above.

Yeohyun Ahn.
Viviana Cordova.

i=1
PI*i

i=2
PI*i/1

i=3
PI*i/1.5

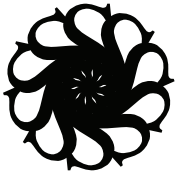
i=4
PI*i/2

i=5
PI*i/2.5

Replace the value of `i` in the `for()` statement and change the value of angle in the `rotate()` statement.

i=6
PI*i/3

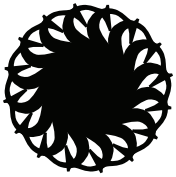
i=8
PI*i/4



i=10
PI*i/5

i=12
PI*i/6

i=14
PI*i/7



i=16
PI*i/8

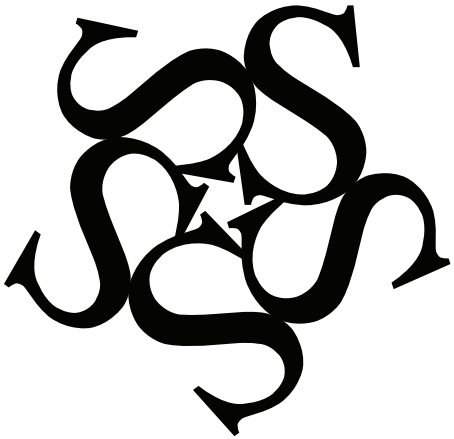
11.2

textAlign( )

Left, center, right

As in previous examples, the `for( )` function is in use again. In this case, the `textAlign( )` function is working in conjunction with `pushMatrix( )`, `rotate( )`, `text( )` and `popMatrix( )`. This formula rotates an uppercase letter "S." The pushMatrix function "stacks" the rotation and text (to allow the mathematical operations involved in the transformation to be performed), and when it is finalized (the operations have been executed) popMatrix brings the result of the stacked functions (in this instance, a rotated "S") back.

```
PFont myFont;
void setup() {
size(800, 800);

background(255,255,255);

// String[] fontList = PFont.
  list();
// println(fontList);

myFont = createFont("Times-
  Roman",48);
textFont(myFont,272);
translate(400,400);

for(int i=0;i<6;i=i+1)
{
fill(0,0,0);
textAlign(LEFT);
pushMatrix();
rotate(PI*i/3);
text("S",0,0);
popMatrix();
}
}
```



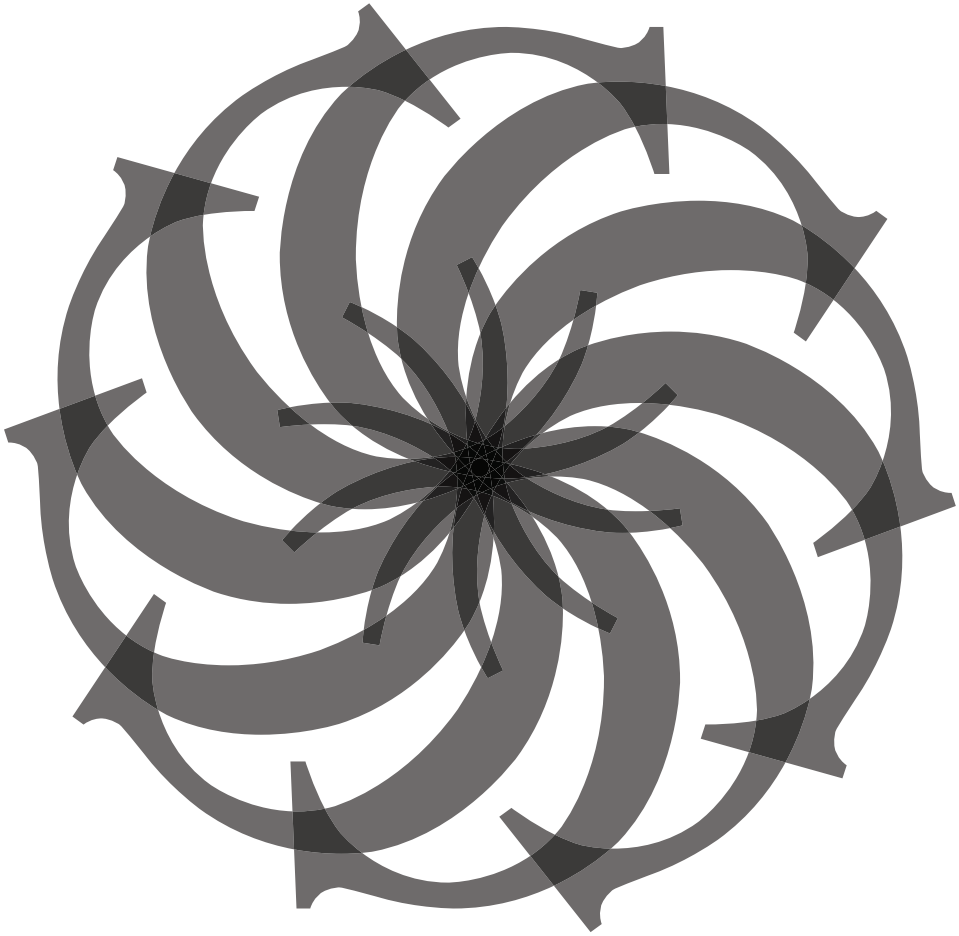
textAlign(LEFT);

textAlign(CENTER);

textAlign(RIGHT);

Change the mood in `textAlign()` to `LEFT`, `CENTER`, or `RIGHT`.

Yeohyun Ahn.
Viviana Cordova.

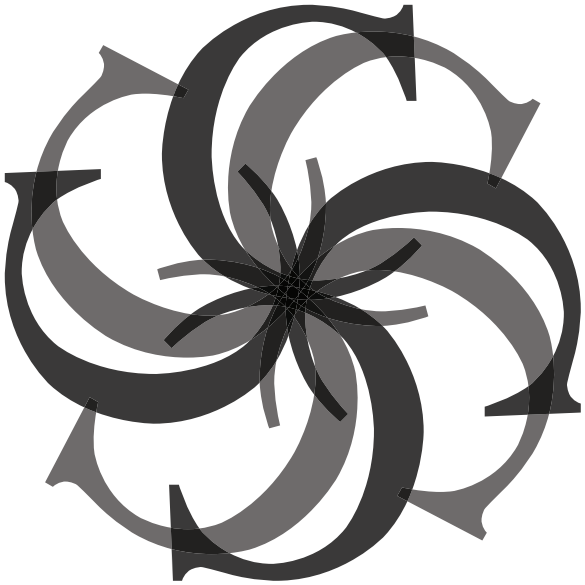


With pushMatrix() and popMatrix()

Comparing the two typographic patterns side by side the design on this page appears consistent, even and symmetrical.

12.0

pushMatrix( )
popMatrix( )



Without pushMatrix( ) and popMatrix( )

These two functions are used together to create regular layering and equal rhythm in conjunction with the `for( )` function. Technically, `pushMatrix( )` saves the current value *to* a matrix stack and `popMatrix( )` restores the prior value *from* the matrix stack. It is not vital to understand how this works, but it is important to grasp how it affects your code.

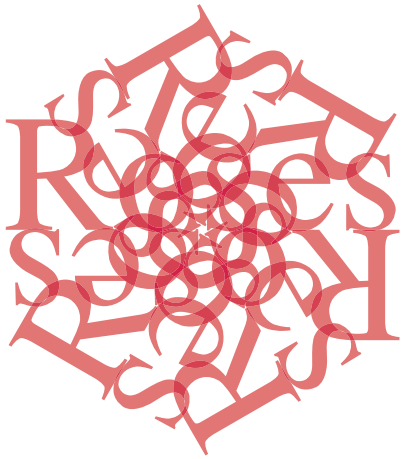
Here, with `pushMatrix( )` and `popMatrix( )`, the value of i in our `for( )` statement changes from 0, 1, 2, 3, up to 11. While the rotation value in `rotate(PI*i/6)` changes from 0, PI*1/6, PI*2/6, PI*3/6, up to PI*11/6, incrementally by PI/6. The facing page shows how using pushMatrix( ) and popMatrix( ) gives us the desired result of this code .

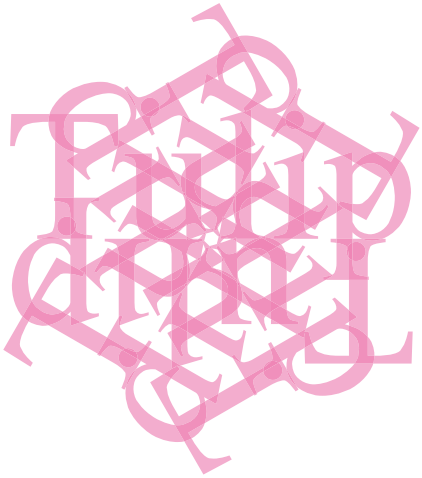
However, without pushMatrix( ) and popMatrix( ), the value of i in the `for( )` statement is changing *cumulatively* from 0, 1,1+2,1+2+3, 1+2+3+4, and so on. This is because the value of i isn't restored to the prior value of i, but is added to the current value of i. Thus, based on this value of i, the rotation angle `rotate(PI*i/6)` is changing from 0, PI*1/6, PI*(1+2)/6, PI*(1+2+3)/6, and so on. As you can see from the example on this page, the result is a very uneven rotation (since the angle is determined by the "irregular" values of i).

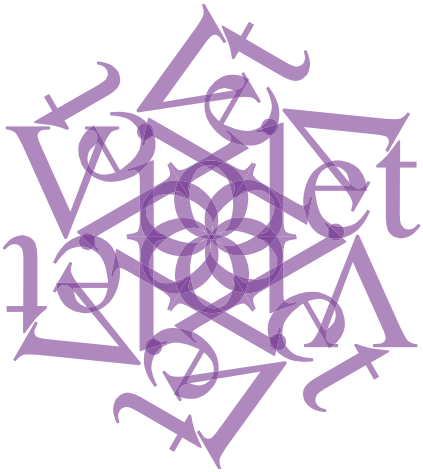
To create constructive structures such as regular repetition, it is suggested to always use pushMatrix( ) and popMatrix( )—especially to make a symmetrical design, because all the values will show up in one layer without overlapping and will not be unevenly positioned.

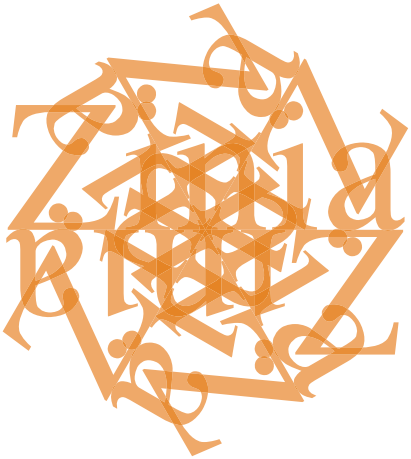
```
PFont myFont;
void setup() {
size(800, 800);
background(255,255,255);
myFont = createFont("Times-
  Roman",48);
textFont(myFont,272);
translate(400,400);
for(int i=0;i<12;i=i+1)
{
fill(0,0,0,120);
textAlign(LEFT);
pushMatrix();
rotate(PI*i/6);
text("C",0,0);
popMatrix();}}
```

Yeohyun Ahn.
Viviana Cordova.



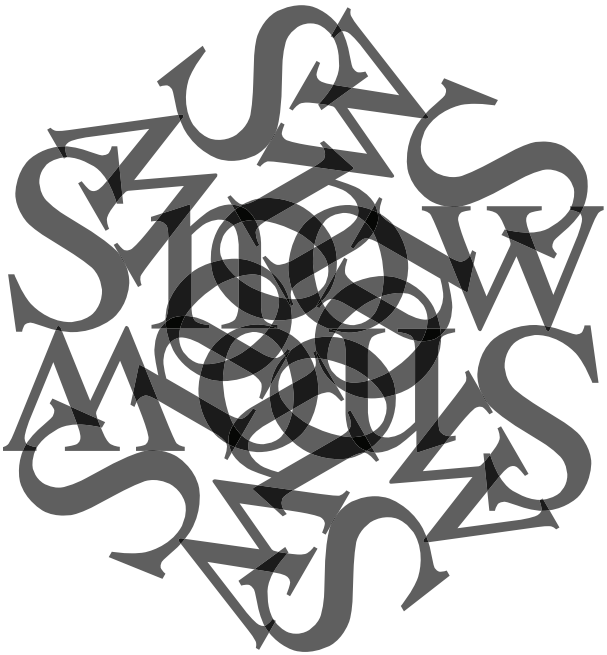






```
text("Roses",0,0);
text("Violet",0,0);
```

```
text("Tulip",0,0);
text("Zinia",0,0);
```



```
PFont myFont;
void setup(){
size(800, 800);

background(255,255,255);

// String[] fontList = PFont.list();
// println(fontList);

myFont = createFont("Times-Roman",48);
textFont(myFont,272);
translate(400,400);

for(int i=0;i<6;i=i+1)
{
fill(0,0,0,180);
textAlign(CENTER);
pushMatrix();
rotate(PI*i/3);
text("Snow",0,0);
popMatrix();
}
}
```

Explore changing the letter "S" (used in previous examples) to single words. Use words that create unique forms with attention to ascenders and descenders, as well as uppercase and lowercase combinations.

Yeohyun Ahn.
Viviana Cordova.

13.0

Letter Design

Graphic designers often have the chance to design a complete set of characters—from A to Z. But how would you create a new character set with reference to an existing font such as Times Roman? In this exercise, we manipulate each letter of the alphabet using the `rotate()` function to create new letterforms for the whole alphabet. The `for()` function uses the i value in the same way as our previous examples: `for(int i=0;i<6;i=i+1)`. The value of i from 0 to 5 enables to duplicate and repeat the letters six times. Black color and semi transparency are applied by using `fill(0,0,0,180)`. This time, after `pushMatrix()`, a much finer rotation is added with `rotate(PI*i/45)`—where i is divided by 45 then multiplied by pi (π). The entire alphabet can be used in the code, which currently displays an uppercase "M": `text("M",0,0)`. Once more, `popMatrix()` closes the operation. Finally, if different font styles such as Mrs Eaves are employed, the visual output can prove even more interesting.

```
PFont myFont;
void setup(){
size(800, 800);

background(255,255,255);

// String[] fontList = PFont.
  list();
// println(fontList);

myFont = createFont("Times-
  Roman",48);
textFont(myFont,272);
translate(400,400);

for(int i=0;i<6;i=i+1)
{
fill(0,0,0,180);
textAlign(RIGHT);
pushMatrix();
rotate(PI*i/45);
text("M",0,0);
popMatrix();
}
}
```

A complete character set can be created with similar values and characteristics by simply changing the letter of designation within the quotation, currently "M."


A


B


C


D


E


F


G


H


I


J


K


L


M


N


O


P


Q


R


S


T


U


V


W

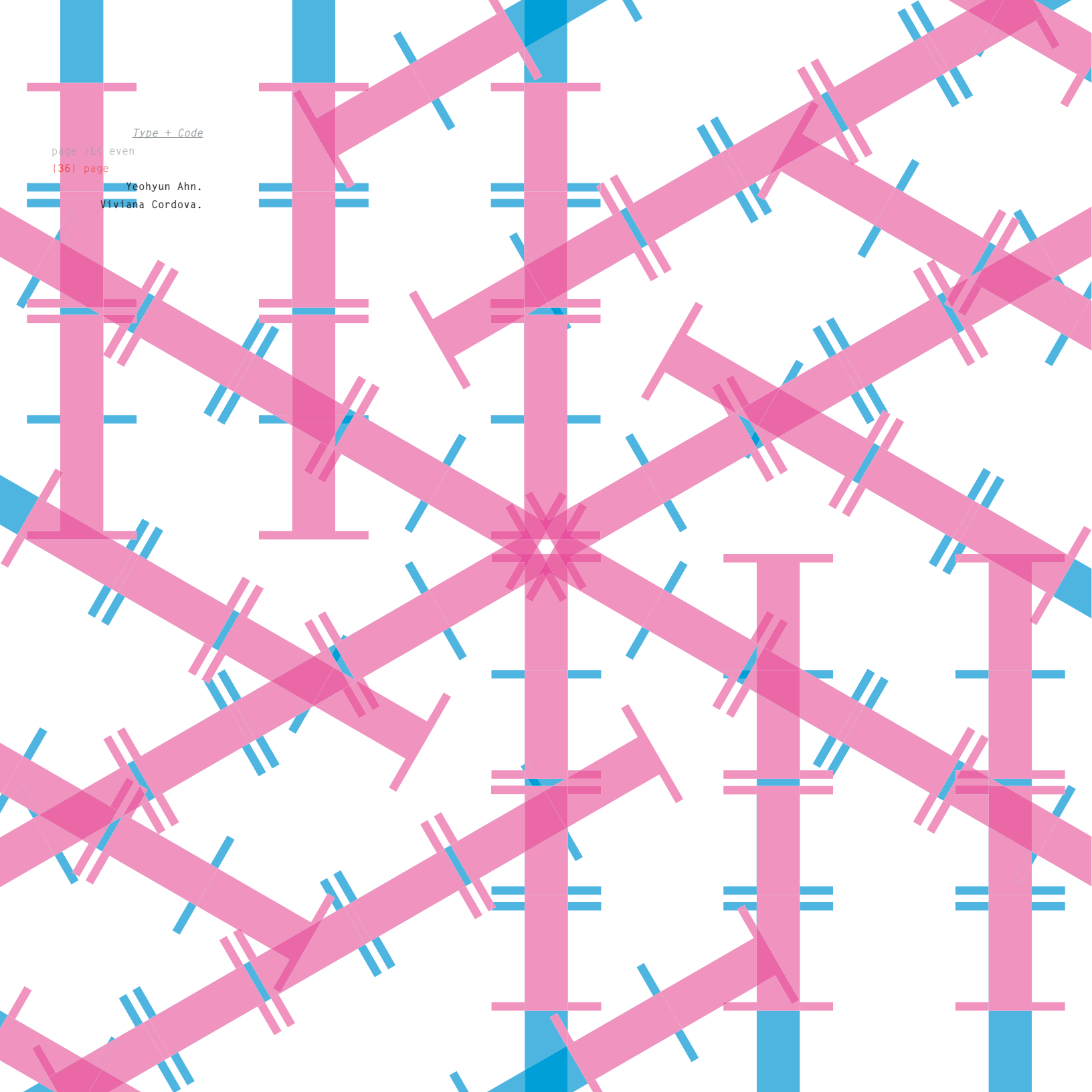

X


Y


Z

Change the mood in `textAlign()` to `LEFT`.
Change the value of i in `for()` statement.
Change the value of angle in `rotate()` statement.

Yeohyun Ahn.
Viviana Cordova.



14.0

Pattern Design

Using hierarchy, color, rotation and other components of graphic design, pattern design requires a *repetition* of elements. Whether we apply them manually or use mathematical formulas, we need to use a multiple.

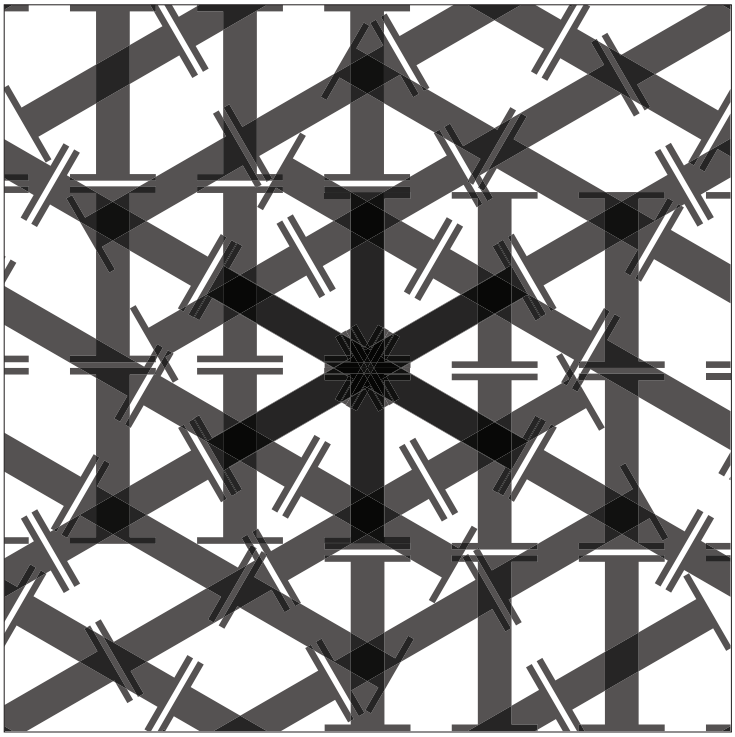
In this example, we are using three `for()` functions, therefore, we need more variables: i, j and k. The `for()` statements are `for(int i=0;i<800;i=i+200)`, `for(int j=0;j<800;j=j+200)`, and `for(int k=0;k<6;k=k+1)`. Transparency is added by `fill(0,0,0,150)`, with a value of 150. Finally, within the `pushMatrix()` and `popMatrix()` statements, `textAlign()`, `rotate()`, and `text()` have been added.

```
PFont myFont;
void setup(){
size(800, 800);

background(255,255,255);

translate(400,400);
myFont =
  createFont("Bodoni",48);
textFont(myFont,290);

for(int i=0;i<800;i=i+200){

for(int j=0;j<800;j=j+200){

for(int k=0;k<6;k=k+1){

fill(0,0,0,150);
pushMatrix();
textAlign(CENTER);
rotate(PI*k/3);
text("I", i, j);
popMatrix();
}
}
}
```

```
import processing.pdf.*;

PFont myFont;

void setup() {
size(800, 800);
beginRecord(PDF, "pattern.pdf");
background(255,255,255);

translate(400,400);
myFont = createFont("Bodoni",48);
textFont(myFont,290);

for(int i=0;i<800;i=i+200){

for(int j=0;j<800;j=j+200){

for(int k=0;k<6;k=k+1){

fill(0,0,0,150);
pushMatrix();
textAlign(CENTER);
rotate(PI*k/3);
text("I", i, j);
popMatrix();
}
}
}
endRecord();
}
```

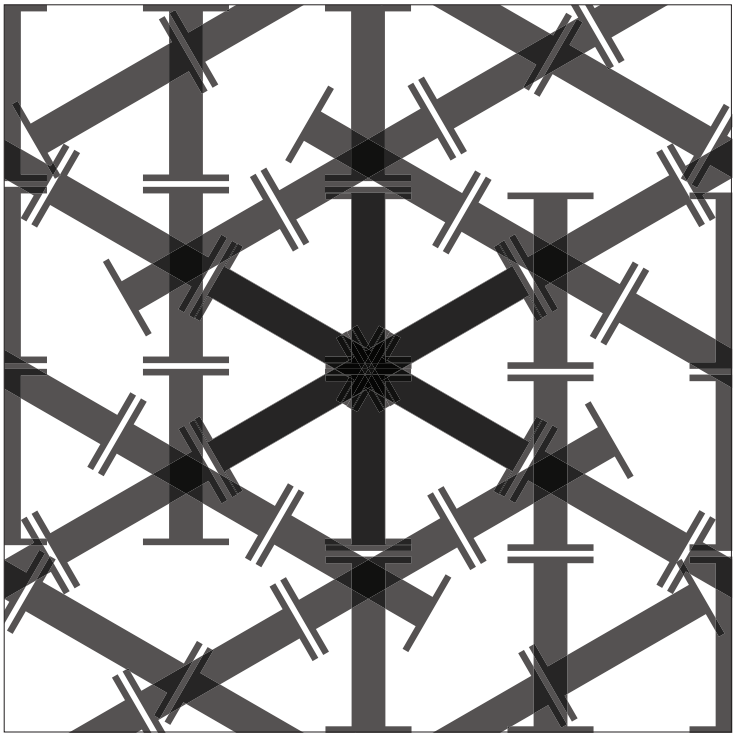
The above version of the code adds: `import processing.pdf.*`, `beginRecord(PDF, "pattern.pdf")` and `endRecord()` to save the output as "pattern.pdf," which can then be imported into a vector-based design application such as Adobe Illustrator where color and transparency can be added.

Yeohyun Ahn.
Viviana Cordova.



```
PFont myFont;
void setup() {
size(800, 800);

background(255,255,255);

translate(400,400);
myFont = createFont("Bodoni",48);
textFont(myFont,290);

for(int i=0;i<800;i=i+140){

for(int j=0;j<800;j=j+200){

for(int k=0;k<6;k=k+1){

fill(0,0,0,150);

pushMatrix();
textAlign(CENTER);
rotate(PI*k/3);
text("I", i, j);

popMatrix();
}
}
}
}
```

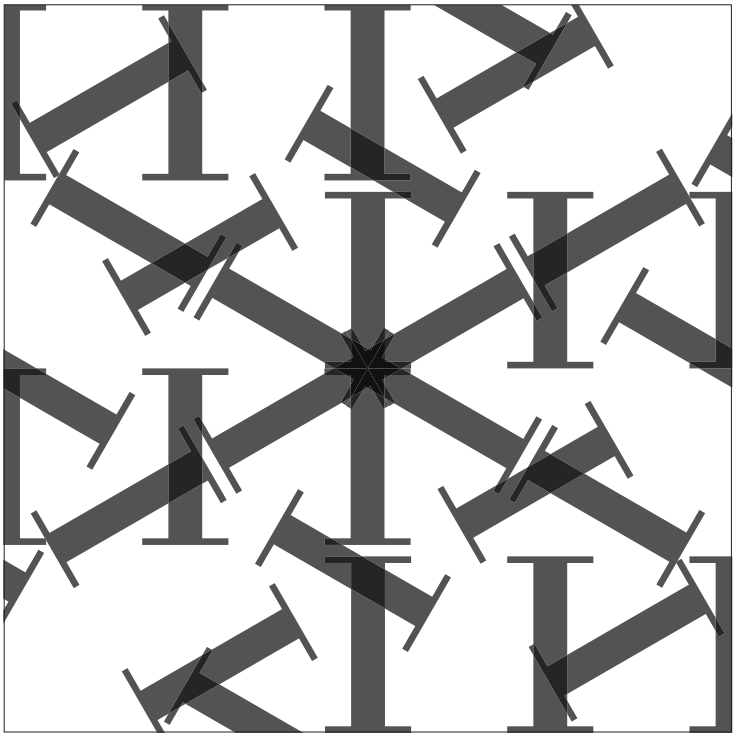
If the increment of `i` is changed from 200 to 140, it creates more complexity and density in this typographic pattern with the letter "I." In `text("I", i, j)`, `i` is the variable that determines the x coordinate position of the letter "I." The sequence of the value of `i` then runs 0, 140, 280, 720.



```
PFont myFont;
void setup() {
size(800, 800);

background(255,255,255);

translate(400,400);
myFont = createFont("Bodoni",48);
textFont(myFont,290);

for(int i=0;i<800;i=i+200){

for(int j=0;j<800;j=j+200){

for(int k=0;k<6;k=k+1){

fill(0,0,0,150);

pushMatrix();
textAlign(CENTER);
rotate(PI*k/3);
text("I", i, j);

popMatrix();
}
}
}
}
```

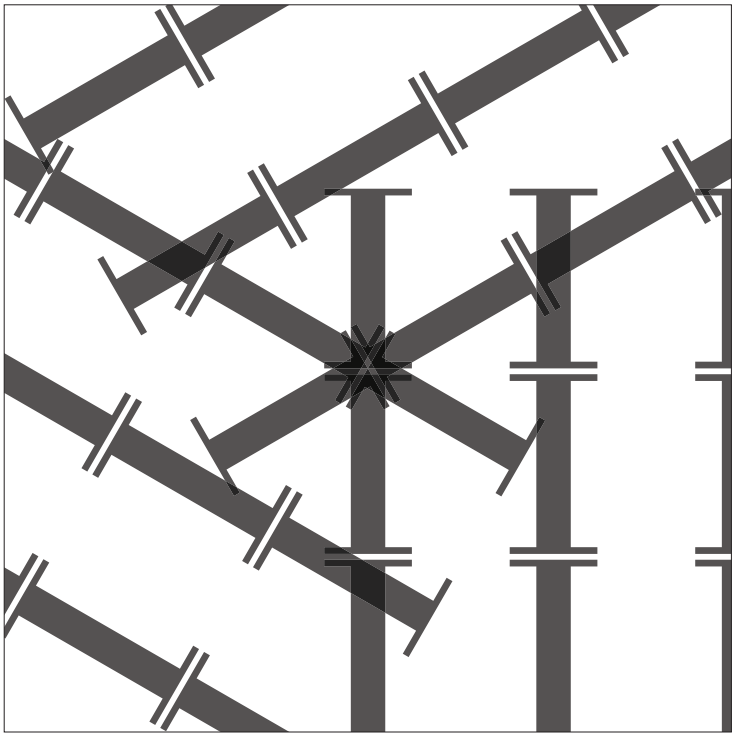
Change the increment of
i to 200.

Yeohyun Ahn.
Viviana Cordova.



```
PFont myFont;
void setup() {
size(800, 800);

background(255,255,255);

translate(400,400);
myFont = createFont(“Bodoni”,48);
textFont(myFont,290);

for(int i=0;i<800;i=i+200){

for(int j=0;j<800;j=j+400){

for(int k=0;k<6;k=k+1){

fill(0,0,0,150);

pushMatrix();
textAlign(CENTER);
rotate(PI*k/3);
text(“I”, i, j);

popMatrix();
}
}
}
}
```

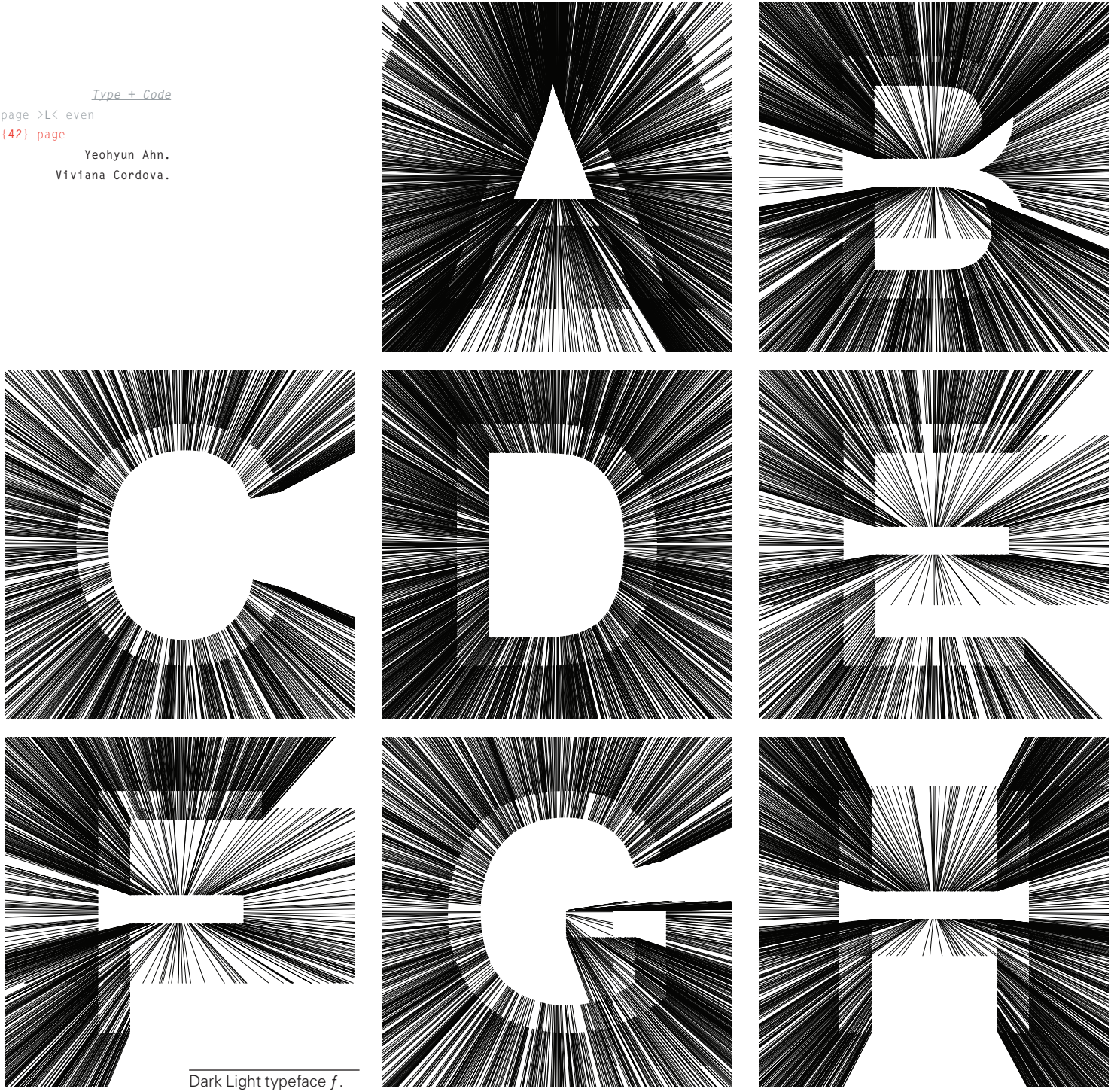
If you change the increment of `j` from 200 to 400, it decreases the complexity and density of this typographic pattern with the letter “`I`.” In `text("I", i, j)`, `j` is the variable that determines the y coordinate position of the letter “`I`.”



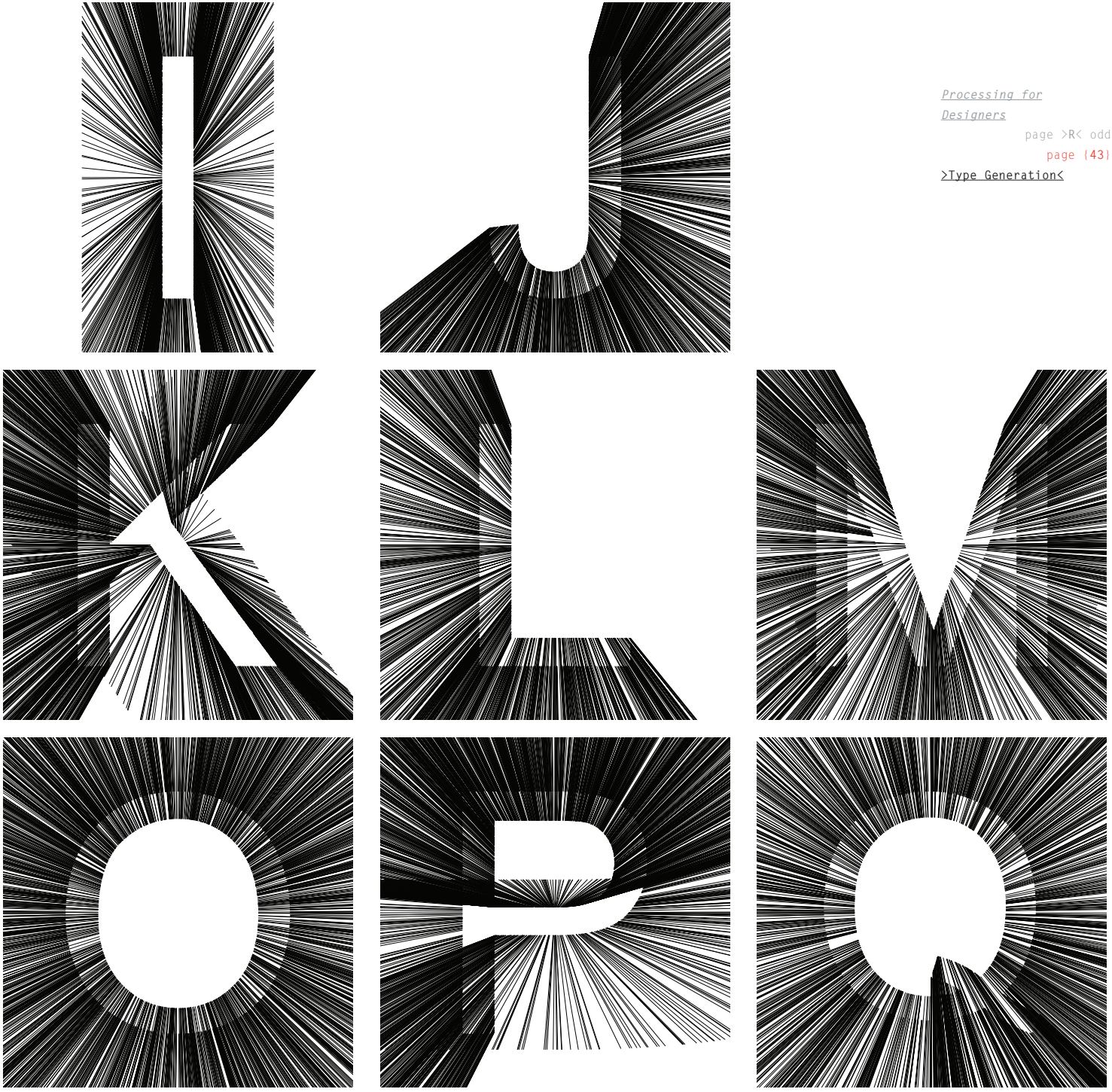
```
PFont myFont;
void setup() {
size(800, 800);

background(255,255,255);

translate(400,400);
myFont = createFont("Bodoni",48);
textFont(myFont,290);

for(int i=0;i<800;i=i+200){

for(int j=0;j<800;j=j+200){

for(int k=0;k<6;k=k+2){

fill(0,0,0,150);

pushMatrix();
textAlign(CENTER);
rotate(PI*k/3);
text("I", i, j);

popMatrix();
}
}
}
}
```

To create another variation on the pattern, change the increment of `k` from 1 to 2. In `rotate(PI*k/3);` `k` is the rotation value. The value of `k` then runs 0, `PI*2/3,PI*4/3, PI*6/3.`

Yeohyun Ahn.
Viviana Cordova.



Dark Light typeface *f*.




code

INTERMEDIATE DESIGN

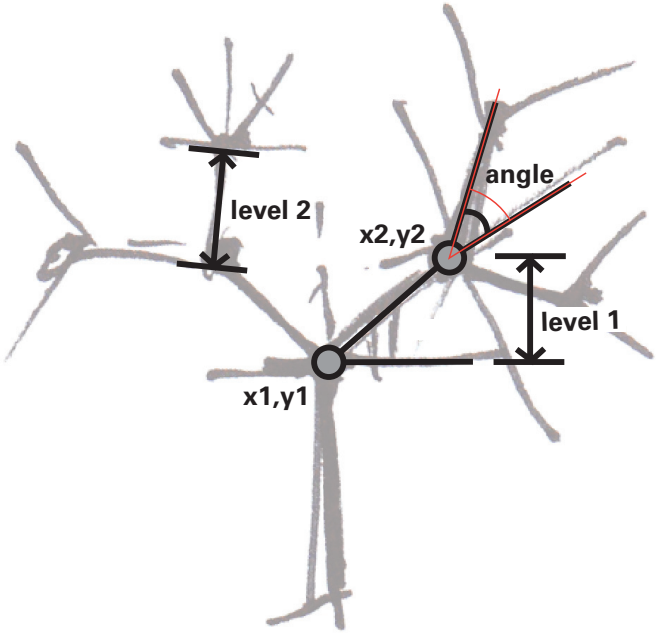
15.0

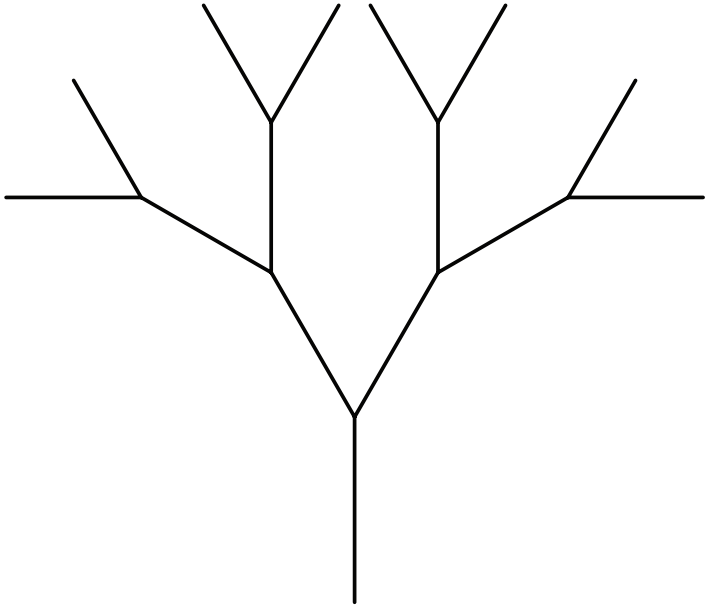
Y–System

The basic concept of this example of repetition—which I completed as part of the GD MFA Studio—was that each branch on the letter "Y" would become a smaller "Y" for a potentially unlimited number of repetitions. To simulate this concept in Processing, I referenced the binary tree algorithm, a basic data structure in which each node has at most two offspring, and defined a function, Ysystem( ). It has six parameters: x1, y1, x, y, angle and level. With a screen size of 800x 800 pixels, it was duplicated, rotated, connected and overlapped, employing the six parameters to visually create three pattern elements. These three elements were repeated in Adobe Photoshop to generate the three final experimental pattern designs.

This three-dimensional virtual typographic tree (left) features the word "code" in reference to the L-system algorithm, which generates fractals and realistic modeling of a tree, programmed by Jer Thorp.

The "code tree" has been duplicated and rotated for the cover design of *Type + Code*.





x1, y1 is defined as the starting position value of the two offspring from each node.

x2, y2 is defined as the ending position value of the two offspring from each node.

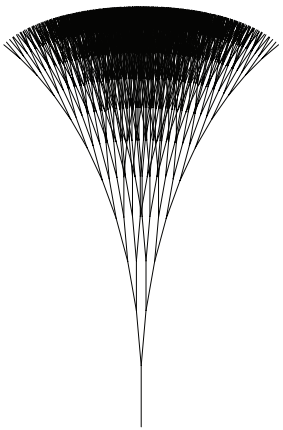
Angle is defined as the value (in degrees) that the "branches" spread out from each node.

Level is defined as how many nodes away from the origin the "branches" have reached. In this case, it is the first level. With further subdivision, the branches and nodes multiply to become a second level. This system could continue infinitely, as levels are added, creating a lively and "organic" branch structure.

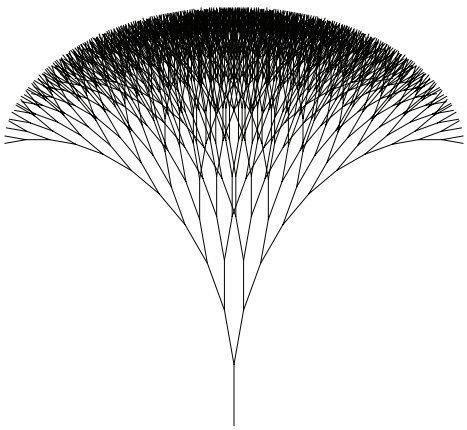
```
void setup() {
size(800, 800);
background(255); }

void draw()
{
Ysystem(400, 600, 400, 550, 30, 3);
}

void Ysystem(float sx, float sy, float ex, float ey, int angle, int
  level)
{
int new_level = level-1; line(sx,sy,ex,ey);
if(level>0)
{float dist = sqrt( (sx-ex)*(sx-ex)+(sy-ey)*(sy-ey) );
float dx = (ex-sx)/dist; float dy= (ey-sy)/dist;
float R = radians(angle);
float new_dist = 0.9*dist*cos(R);
float new_cx = ex+dx*new_dist;
float new_cy = ey+dy*new_dist;float final_dist = 0.9*dist*sin(R);
float dx1 = -dy; float dy1 = dx;
float dx2 = dy;
float dy2 = -dx;
float new_ex1 = new_cx+dx1*final_dist;
float new_ey1 = new_cy+dy1*final_dist;
float new_ex2 = new_cx+dx2*final_dist;
float new_ey2 = new_cy+dy2*final_dist;
Ysystem(ex, ey, new_ex1, new_ey1, angle, new_level);
Ysystem(ex, ey, new_ex2, new_ey2, angle, new_level);
}
return;
}
```

Y-system (x1, y1, x2, y2, angle, level)

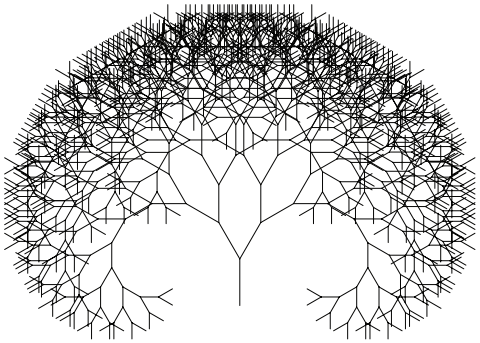
The code utilizes two main functions: draw( ) and Ysystem( ). With the draw( ) function, you can control the output of the Y-system using the six parameters (above).



Ysystem(400, 600, 400, 550, 5, 10);



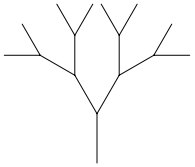
Ysystem(400, 600, 400, 550, 10, 10);

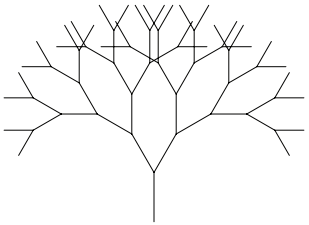


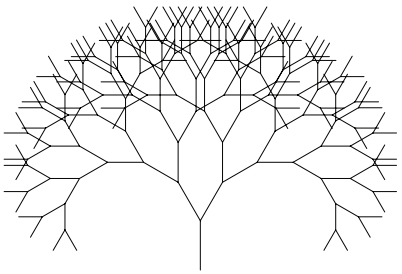
Ysystem(400, 600, 400, 550, 30, 10);

Alter the value of the branch `angle` to condense or expand the graphic.

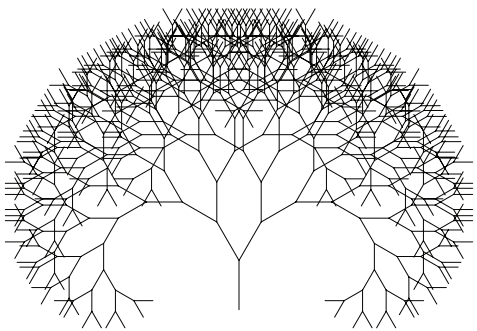
>Y—system<



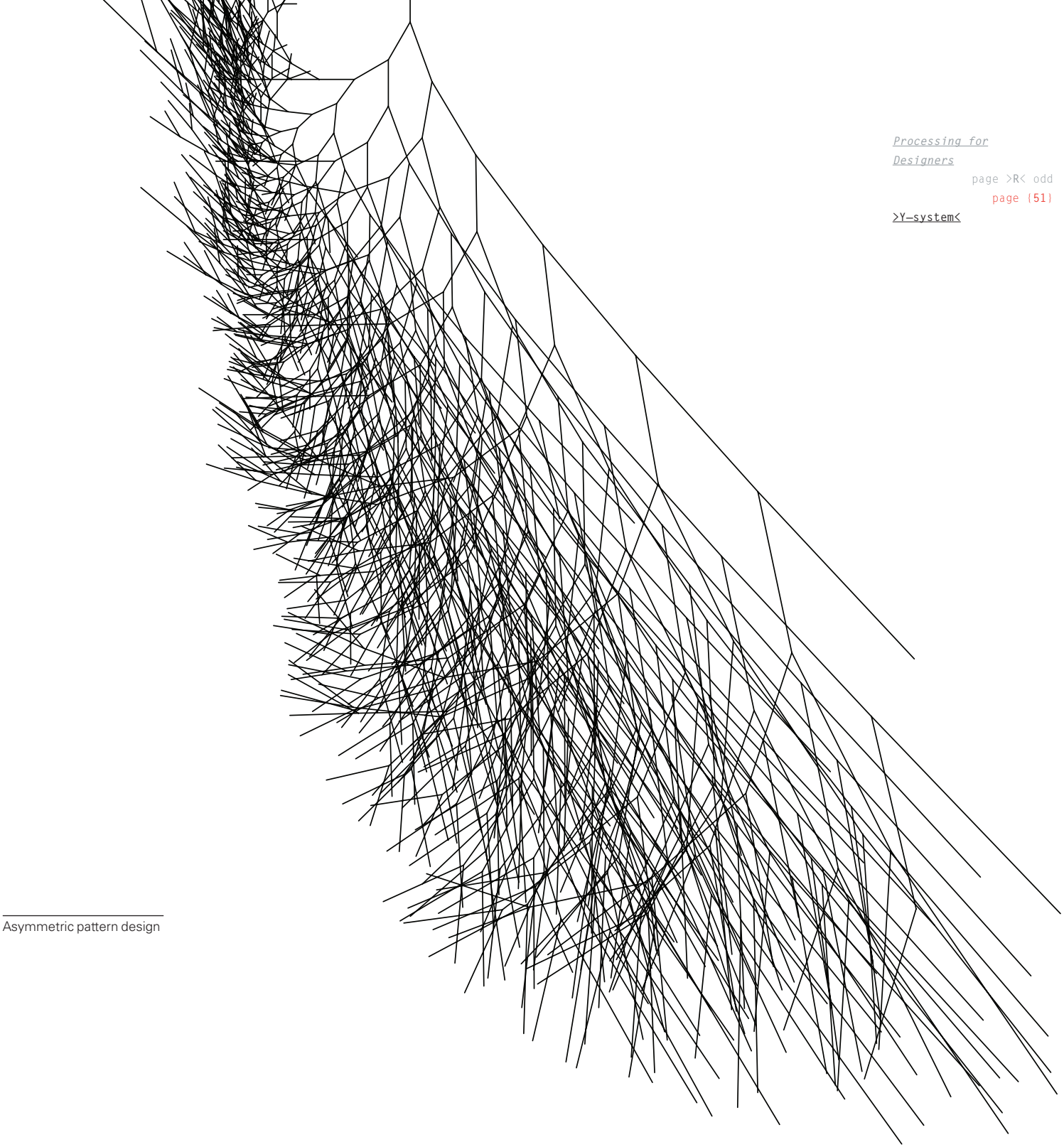




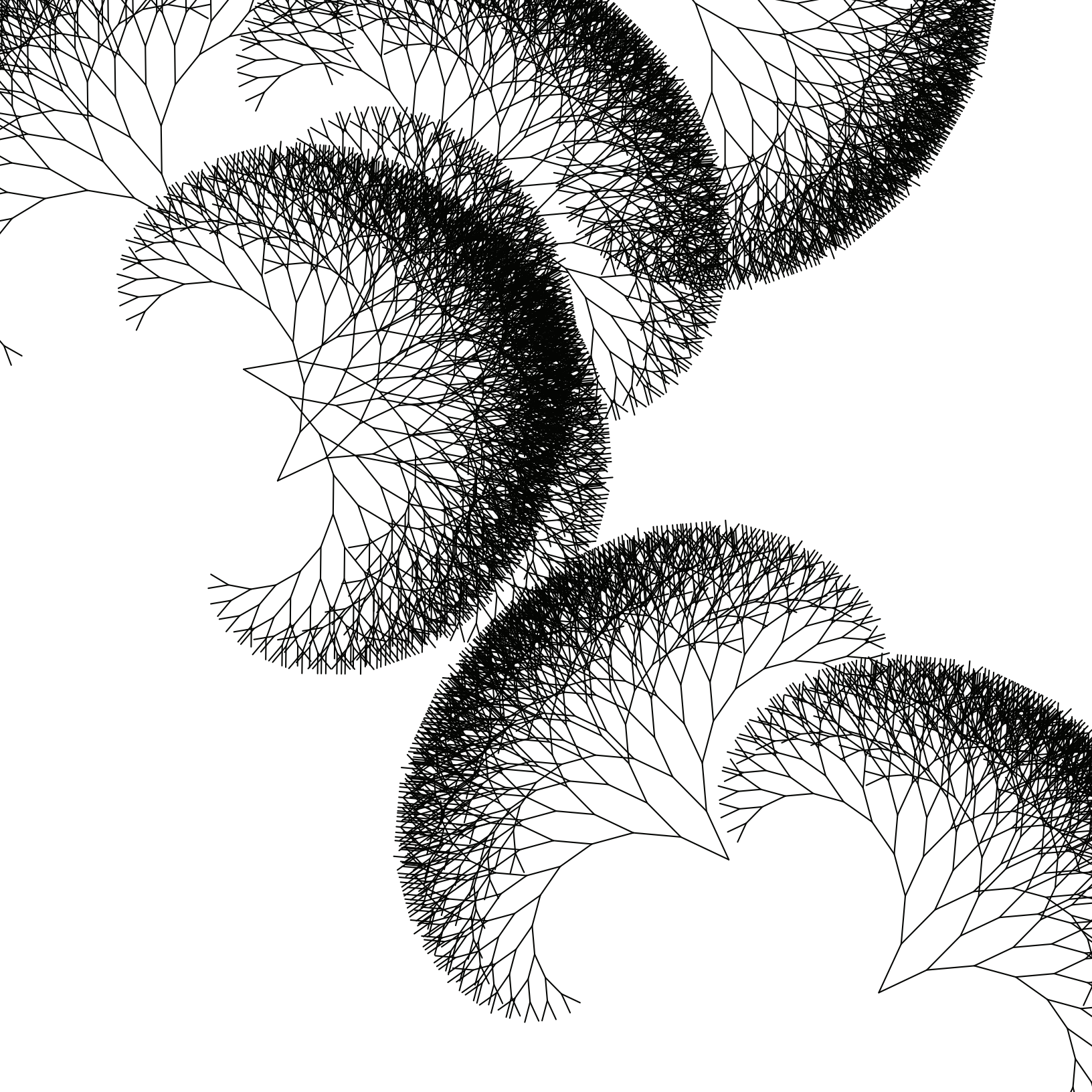
```
Ysystem(400, 600, 400, 550, 30, 1);
Ysystem(400, 600, 400, 550, 30, 3);
Ysystem(400, 600, 400, 550, 30, 5);
Ysystem(400, 600, 400, 550, 30, 7);
Ysystem(400, 600, 400, 550, 30, 9);
```



```
void setup() {
size(800, 800);
background(255);
}

void draw()
{
Ysystem(400, 10, 400, 50, 30, 10);

}

void Ysystem(float sx, float sy, float ex, float ey, int angle, int
  level)
{
int new_level = level-1;
line(sx,sy,ex,ey);
if(level>0){
float dist = sqrt( (sx-ex)*(sx-ex)+(sy-ey)*(sy-ey) );
float dx = (ex-sx)/dist;
float dy = (ey-sy)/dist;
float R = radians(angle);
float new_dist = 0.9*dist*cos(R);
float new_cx = ex+dx*new_dist;
float new_cy = ey+dy*new_dist;
float final_dist = 0.9*dist*sin(R);
float dx1 = -dy;
float dy1 = dx;
float dx2 = dy;
float dy2 = -dx;
float new_ex1 = new_cx+dx1*final_dist;
float new_ey1 = new_cy+dy1*final_dist;
float new_ex2 = new_cx+dx2*final_dist;
float new_ey2 = new_cy+dx2*final_dist;
Ysystem(ex, ey, new_ex1, new_ey1, angle, new_level);
Ysystem(ex, ey, new_ex2, new_ey2, angle, new_level);
}
return;
}
```

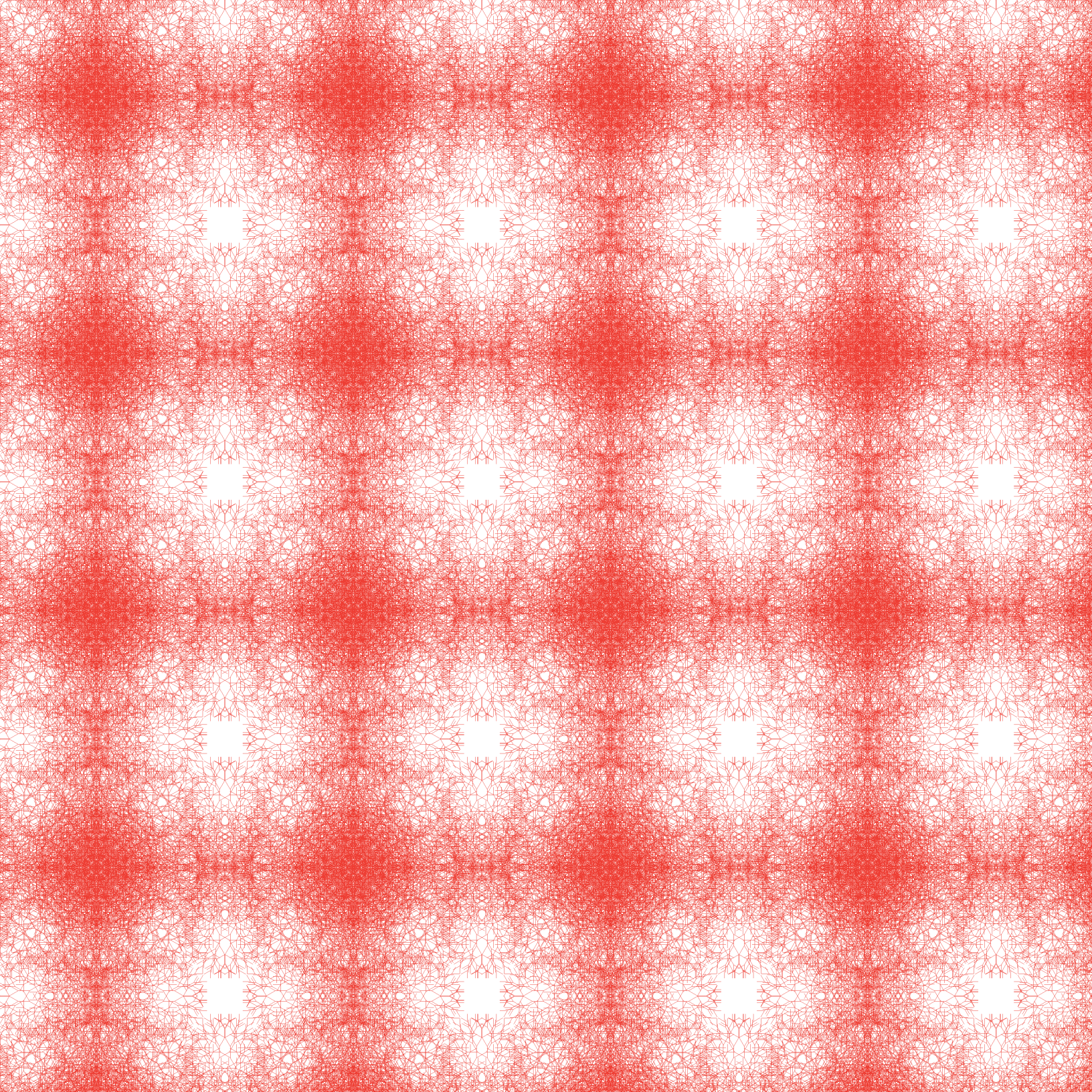


Asymmetric pattern design



```
int i=300;

void setup() {
size(800, 800);
background(255);
}

void draw()
{
translate(random(i),random(i+10));
rotate(PI/3*i);
Ysystem(200, 200, 230, 170, 20, 10);
translate(abs(random(i-10)),random(i));
Ysystem(400, 400, 370, 370, 20, 10);
}

void Ysystem(float sx, float sy, float ex,
  float ey, int angle, int level)
{

int new_level = level-1;
if(level<10)
line(sx,sy,ex,ey);
if(mousePressed) {
if(level>0)
{
float dist = sqrt( (sx-ex)*(sx-ex)+(sy-
  ey)*(sy-ey) );
float dx = (ex-sx)/dist;
float dy = (ey-sy)/dist;
float R = radians(angle);
float new_dist = 0.9*dist*cos(R);
float new_cx = ex+dx*new_dist;
float new_cy = ey+dy*new_dist;
float final_dist = 0.9*dist*sin(R);
float dx1 = -dy;
float dy1 = dx;
float dx2 = dy;
float dy2 = -dx;
float new_ex1 = new_cx+dx1*final_dist;
float new_ey1 = new_cy+dy1*final_dist;
float new_ex2 = new_cx+dx2*final_dist;
float new_ey2 = new_cy+dy2*final_dist;

Ysystem(ex, ey, new_ex1, new_ey1, angle,
  new_level);
Ysystem(ex, ey, new_ex2, new_ey2, angle,
  new_level);
}
}
return;
}
```

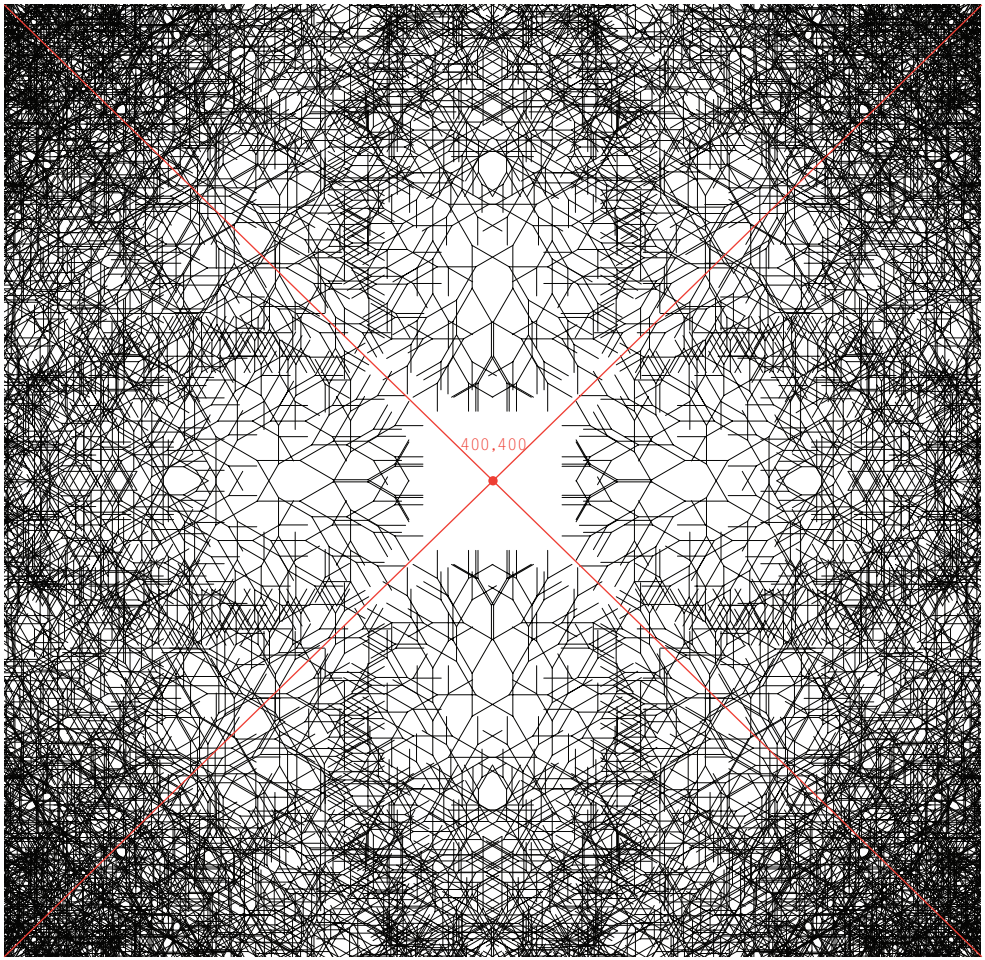
Add translate( ); to move, rotate( ); to rotate, and random( ); to randomize the Ysystem( ).



800,0

>Y—system<

0,0

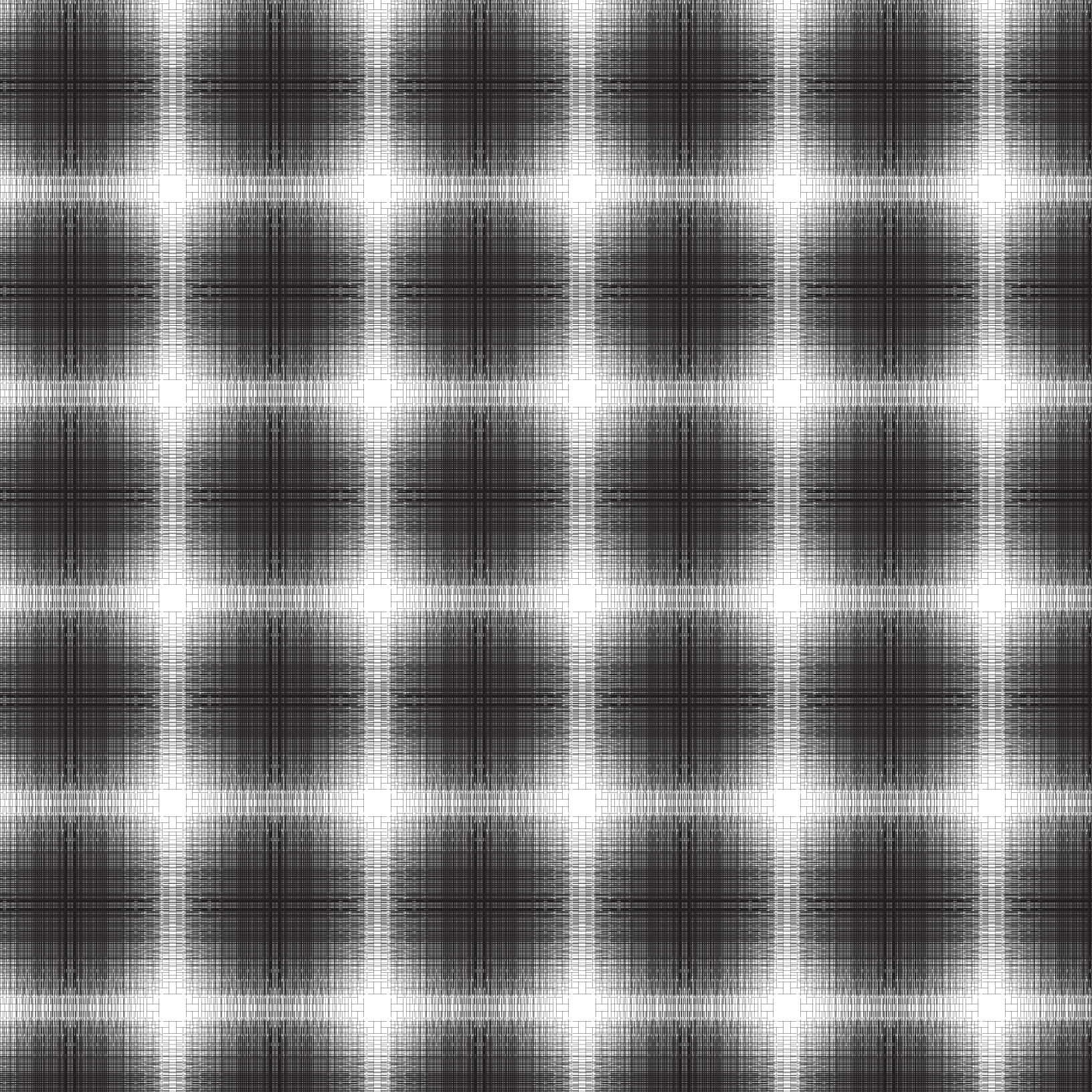


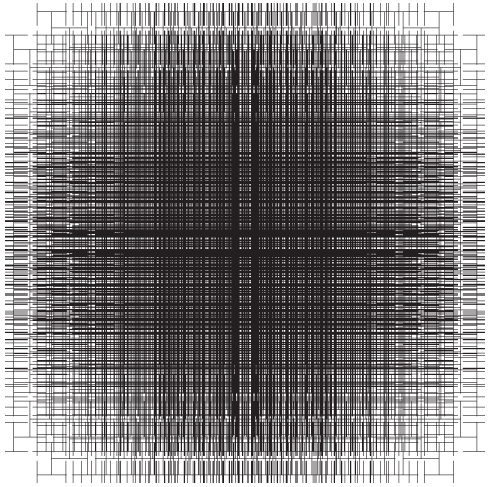
0,800

800,800

Nested pattern designs establish the tone and contrast of the base pattern, and the full page design.

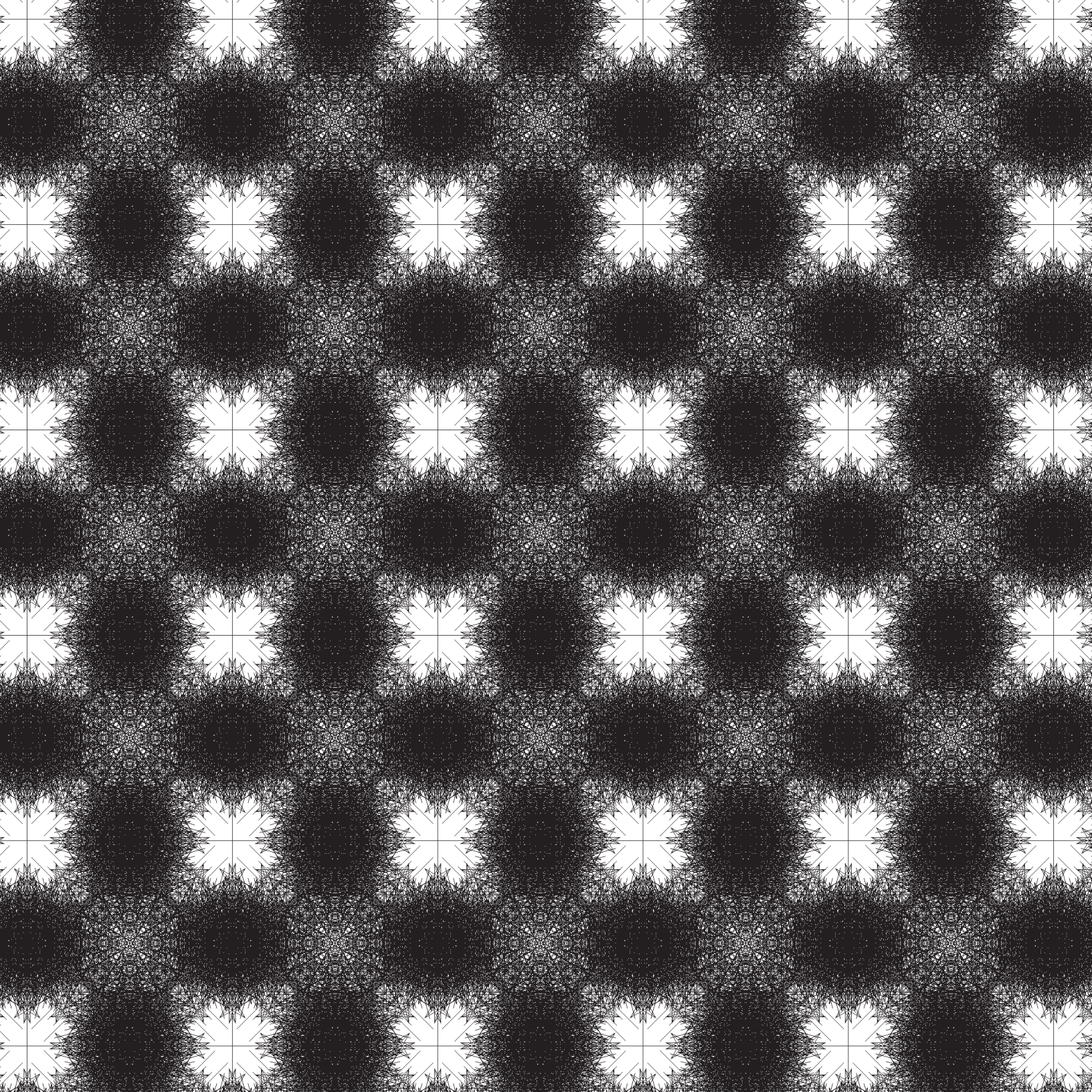
```
Ysystem(400, 400, 400, 490, 30, 13); //up
Ysystem(400, 400, 400, 310, 30, 13); //down
Ysystem(400, 400, 310, 400, 30, 13); //left
Ysystem(400, 400, 490, 400, 30, 13); //right
```

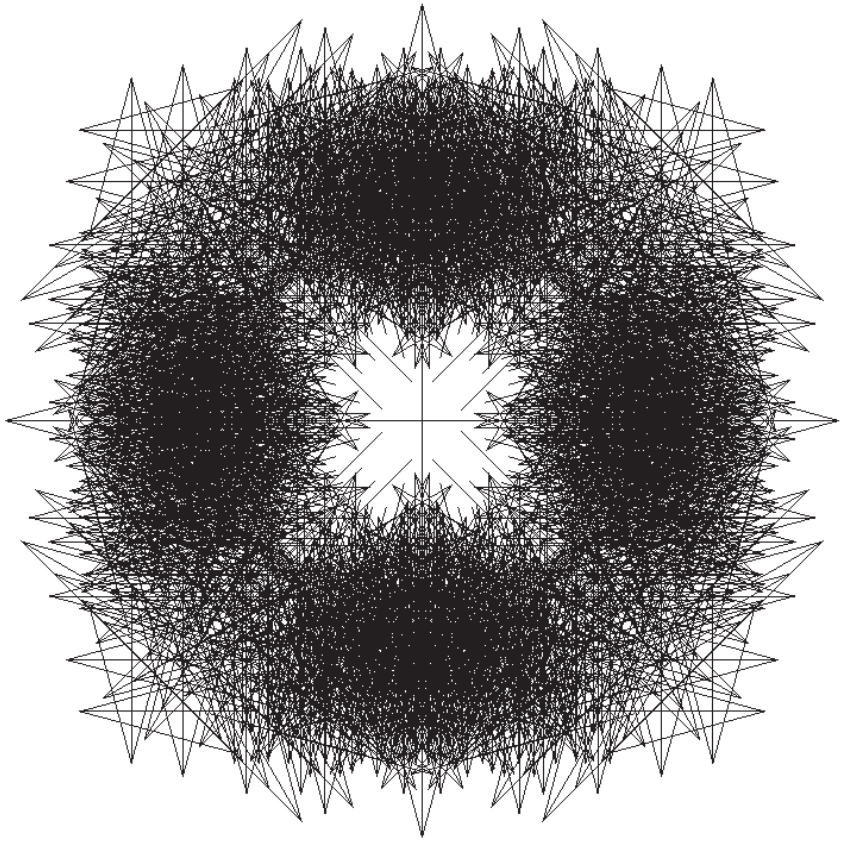




Alter the value of the six parameters by using graphic design principles such as copying, repeating, rotating, moving and connecting.

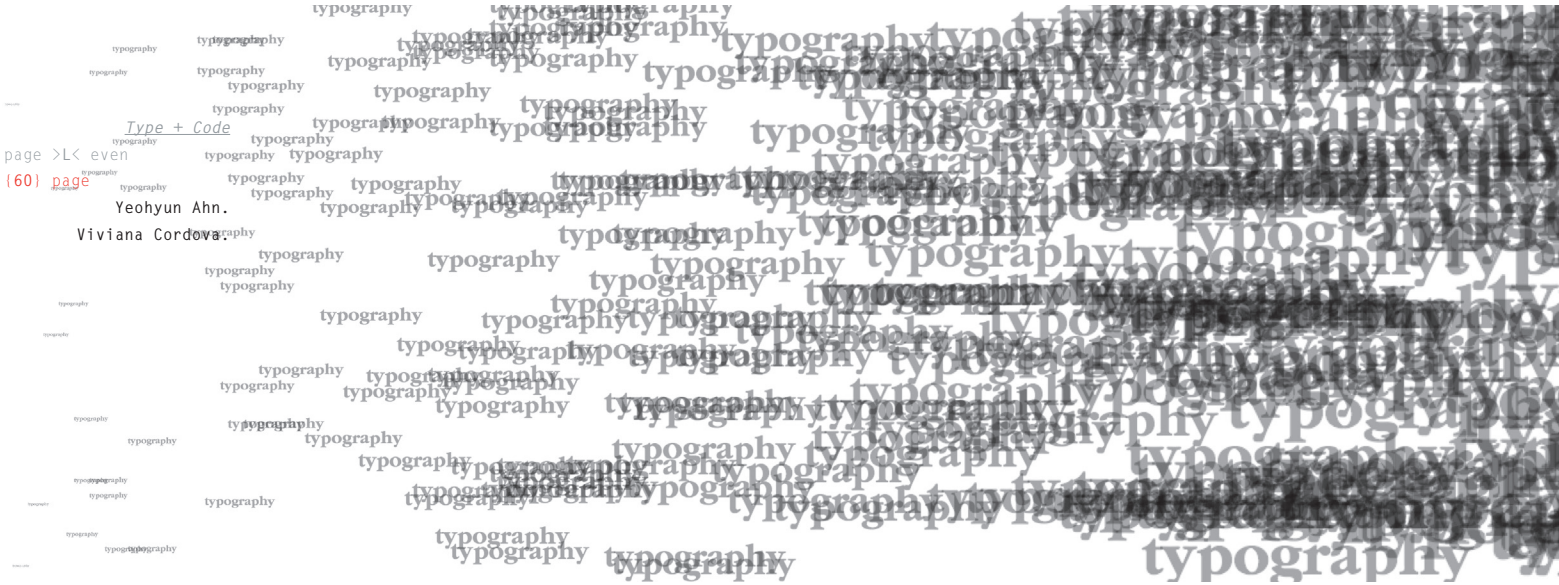
```
Ysystem(400, 400, 400, 495, 90,14);
Ysystem(400, 400, 400, 305, 90,14);
Ysystem(400, 400, 305, 400, 90,14);
Ysystem(400, 400, 495, 400, 90,14);
```



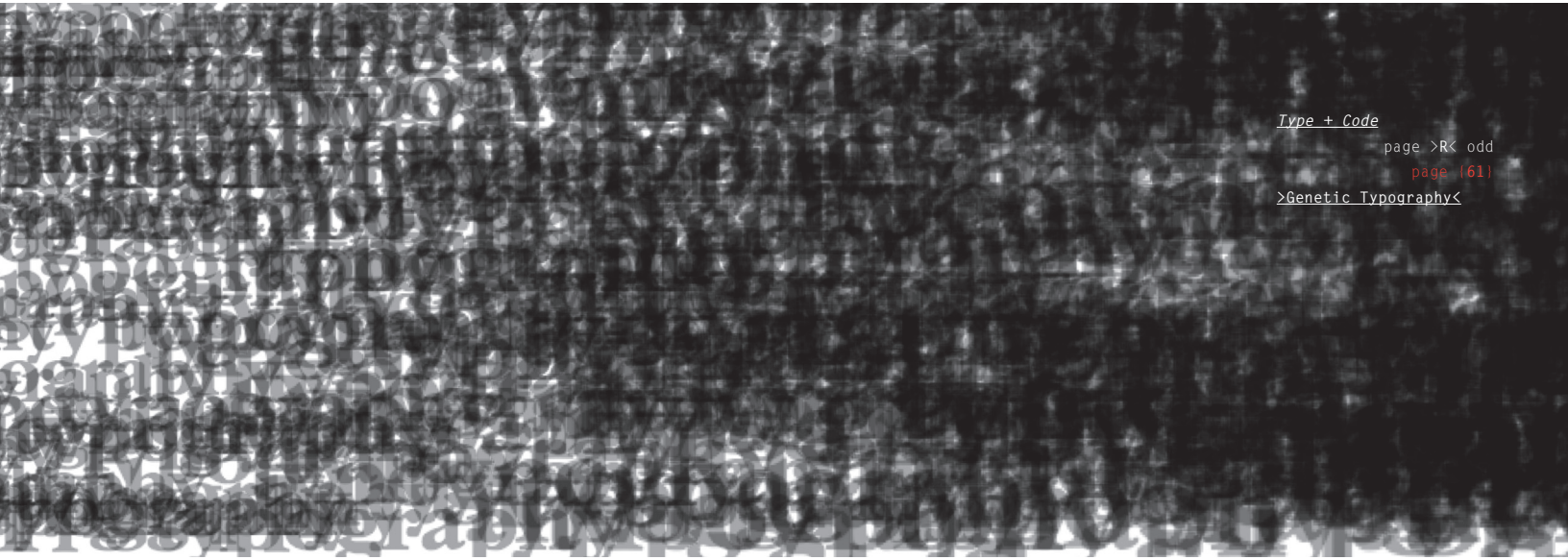


The fully developed pattern on the left uses multiples of stacked and repeated design elements for a full page patterning.

```
Ysystem(400, 400, 400, 750, 165,9);
Ysystem(400, 400, 400, 50, 165, 9);
Ysystem(400, 400, 50, 400, 165, 9);
Ysystem(400, 400, 750, 400,165, 9);
```




```
PFont myFont;

int cellsize = 20;
int COLS, ROWS;
//game of life board
int[][] old_board, new_board;
void setup()
{
  size(8500, 1500);
  smooth();
frameRate(1);
//make sure to upload font
myFont =
    createFont("Garamond-Bold",48);
textFont(myFont, 5);
//initialize rows, columns and set-up arrays
  COLS = width/cellsize;
  ROWS = height/cellsize;
  old_board = new int[COLS][ROWS];
  new_board = new int[COLS][ROWS];
  colorMode(RGB,255,255,255,5);
  background(255);
//call function to fill array with random
//values 0 or 1
  initBoard();
  }
void draw()
{
  background(255, 255,255);
//loop through every spot in our 2D array and check
   //spots neighbors
  for (int x = 0; x < COLS;x++) {
    for (int y = 0; y < ROWS;y++) {
      int nb = 0;
     //Note the use of mod ("%") below to ensure
     //that cells on the edges have "wrap-around"
     //neighbors
      //above row
      if (old_board[(x+COLS-1) % COLS ][(y+ROWS-5) %
ROWS ] == 1) { nb++; }
      if (old_board[ x              ][(y+ROWS-5) %
ROWS ] == 1) { nb++; }
      if (old_board[(x+1)     % COLS ][(y+ROWS-5) %
ROWS ] == 1) { nb++; }
        //middle row
      if (old_board[(x+COLS-1) % COLS ][ y
] == 1) { nb++; }
      if (old_board[(x+1)     % COLS ][ y
] == 1) { nb++; }
        //bottom row
      if (old_board[(x+COLS-1) % COLS ][(y+10)     %
ROWS ] == 1) { nb++; }
      if (old_board[ x              ][(y+10)     % ROWS
] == 1) { nb++; }
      if (old_board[(x+1)     % COLS ][(y+10)     %
ROWS ] == 1) { nb++; }
        //RULES OF "LIFE" HERE
      if     ((old_board[x][y] == 1) && (nb <  1)) {
    new_board[x][y] = 1; }     //loneliness
      else if ((old_board[x][y] == 1) && (nb >  1)) {
    new_board[x][y] = 10; }     //overpopulation
      else if ((old_board[x][y] == 0) && (nb == 2)) {
    new_board[x][y] = 3; }     //reproduction
      else                                  { new_
    board[x][y] = old_board[x][y]; }  //stasis
    }
  }
//RENDER game of life based on "new_board" values
  for ( int i = 0; i < COLS;i++) {
    for ( int j = 0; j < ROWS;j++) {
      if ((new_board[i][j] == 1)) {
        fill(255);
    stroke(0.1);
  //   ellipse (i*cellsize,j*cellsize,cellsize+1,
  //cellsize+1);
    fill(2,0,1,2);
for(int a=0; a<25; a=a+1)

    {
      textSize(i);
    }
     text("typography",i*cellsize,j*cellsize);
      }
    }
  }
  //swap old and new game of life boards
  int[][] tmp = old_board;
  old_board = new_board;
  new_board = tmp;
saveFrame();
}
//init board with random "alive" squares
void initBoard() {
  background(0);
  for (int i =0;i < COLS;i++) {
    for (int j =0;j < ROWS;j++) {
      if (int(random(2)) == 0) {
        old_board[i][j] = 1;
      } else {
        old_board[i][j] = 0;
      }
    }
  }
}
//reset board when mouse is pressed
void mousePressed() {
  initBoard();
}
```



16.0

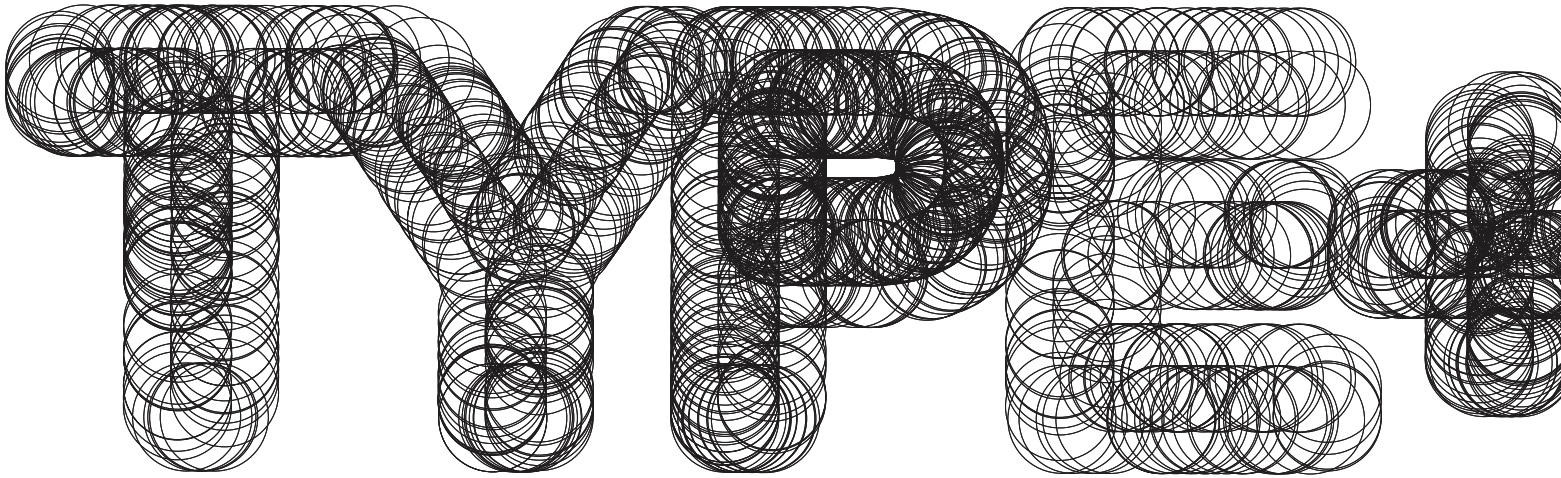
Genetic Typography

Viviana Cordova

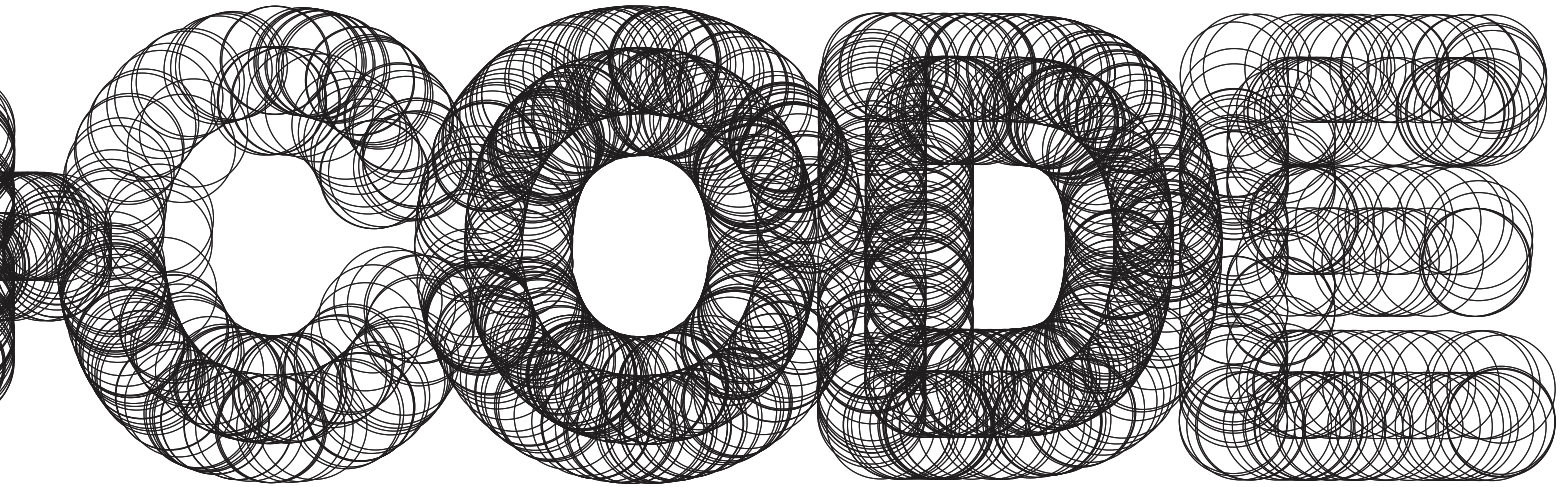
This explosion of type (above) based on the word "typography" uses the Game of Life algorithm—originally devised by mathematician John Conway in 1970— to achieve hierarchy and motion in a 2D environment using Processing. This particular formula helps to activate the canvas in a systematic way, where every single word has a mathematical connection with each other. Also, it helps to balance the reaction between cells dying too fast and when too many are being born. Conway wanted to create a new form of patterns that are live and stable, by evolving a formula that could be infinitely self-generating. He created a balance that will maintain a stable population—like life itself. Using a grid and cells that are typographically visible (the word "typography"), the formula self-generates continuously; and it can be reset by mouse clicking on the window. We use automata theory to create columns and rows that are squared (with the x and y axis coordinates within the canvas) and mathematical elements to self-generate and organize the cells themselves. This gives the result a live status, using the parameters in the formula. It self-generates with a skipping movement as the typography moves across the screen.

Make sure that the font family Garamond is in your font library, for whichever operating system you are using, for example, Mac or PC. Once code is finalized, select Run to test, and please wait three to four minutes for image to show.

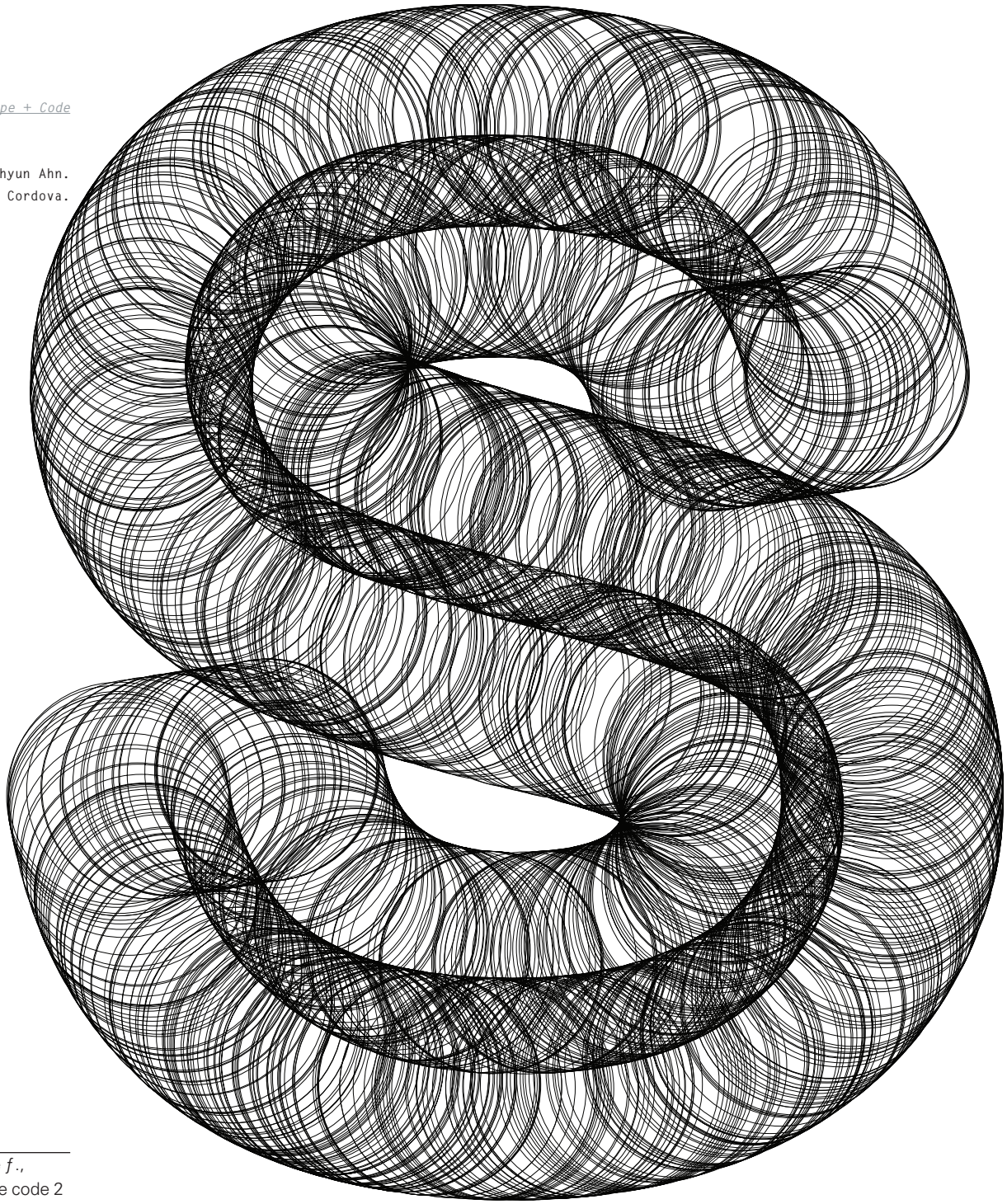
Yeohyun Ahn.
Viviana Cordova.



The code on the following pages is supplied for you to use for your own typographic experimentation. It is meant as a place to start. The collaborators of *Type + Code* hope that with the help of this book, you will be able to identify code strings and use the opportunity to modify and create your own code for expressive means.


CODE

Yeohyun Ahn.
Viviana Cordova.



Slinky typeface *f*.,
Caligraft sample code 2

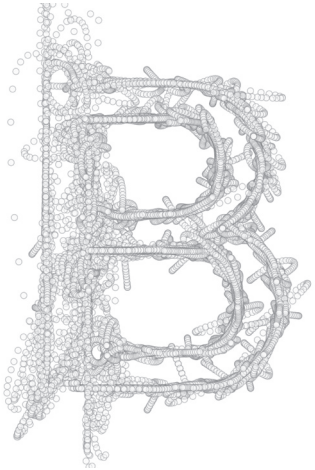
CALIGRAFT

Caligraft (thus named because it crafts computational calligraphy) was created by Ricard Marxer Piñón to provide a new dimension of textual representation through computation in Processing. It was motivated by the abstract generative work produced elsewhere with computers, and aims to create figurative generative typography by using calculus and computation in Processing. Essentially, Caligraft computations alter numeric variables between each control point of a chosen typeface affecting length, distance and position. It is based on the Geomerative library in Processing. Geomerative supports the tasks of handling vectorial shapes such as a font. Caligraft has the GNU, General Public License as published by the Free Software Foundation, GPCJ license, and the Apache license for the part that parses the True Type fonts (.ttf).[1] And it makes use of common system fonts such as Arial and Times found on both PC and Mac platforms. Arial.ttf and Times.ttf are used as seed fonts. Choosing the base font will generate forms with hints of the underlying font structure. This enables the programmer to draw figurative letterforms based on these outlines. The most noticeable variation will appear between serif and sans serif letterforms. The specified font, determined by a single line of code, can easily be altered to create unique letterforms with a variety of results. Caligraft includes several variables and functions to change the font styles, the string of the letters, the speed of deformation, the variation of ink and precision, as well as a drawing function. Initially, I referenced Caligraft to explore how traditional hand drawing-based calligraphy could be reinterpreted by using Processing. The aim was to create a new, innovative dimension of textural typography in computation, utilizing Processing's basic syntax of line(),circle() and Bézier curves. This practice formed part of Marian Bantjes's Word Project in the 2006 Fall Graphic Design MFA studio at MICA. Examples of Caligraft and use of the library can be found and downloaded from www.caligraft.com.

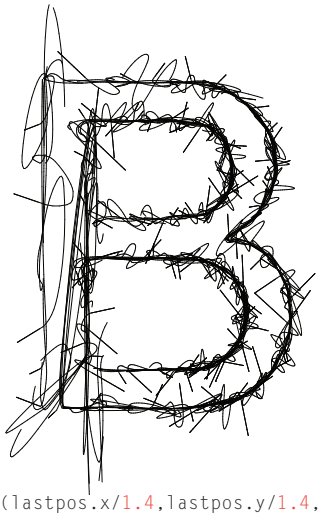
Currently there are three common formats for handling typefaces and fonts. TrueType (.ttf), offers cross-platform accessibility for both Mac and PC where no additional software is needed. OpenType (.otf), offers extended character sets with expanded typesetting capabilities; Adobe Illustrator and InDesign have extended design alternatives for OpenType features. Postscript file formats require a combination of screen fonts and bitmaps (.bmap). Current design and print applications commonly use either OpenType or PostScript type formats, however, Ricard Marxer Piñón makes a case for TrueType fonts, because they are cross platform and contain all of their parts and pieces within a single file. (Previously TrueType also offered advantages for on-screen usage.) Consistent usage of TrueType file formats in Processing may reduce potential errors when rendering Caligraft computations with the Geomerative library.

1 *Caligraft*. Ricard Marxer Piñón. www.caligraft.com/abstract.

Yeohyun Ahn.
Viviana Cordova.



```
ellipse(lastpos.x/1.5,lastpos.y/1.5,8, 8)
```



```
line(lastpos.x/1.4,lastpos.y/1.4,pos.
x/1.4, pos.y/1.4)
```

Replace the highlighted red code in the sample code 1 (on page 68 and also the right-hand example on this page) with the code on the left-hand side on this page. This will give you the Chain typeface, which is a modification of the Scribble typeface. You can see a sample Chain letterform, "B," above left.

17.1

Caligraft
Sample Code 1
Scribble and Chain

CONCEPT:

Reinterpret hand drawing-based calligraphies in code-driven typography, innovating a new dimension of textual typography in Processing.

PROCESS:

1. Choose two words on the word list.

   delight, suspicion, guilt, longing, worry, faith, desire, disbelief, anxiety, doubt, fear, conviction, paranoia, certainty, tension, hope, surprise, grief, lust, melancholy, trust, joy, honor and dismay

2. Express the meaning of them.

Marian Bantjes's Word Project
2006 Fall GD MFA Studio,
Maryland Institute College of Art

To use this code:
1. Arial.ttf is needed
2. Download Geomerative Library from www.caligraft.com.

More details are introduced with sample code 2 in Chapter 17.2.

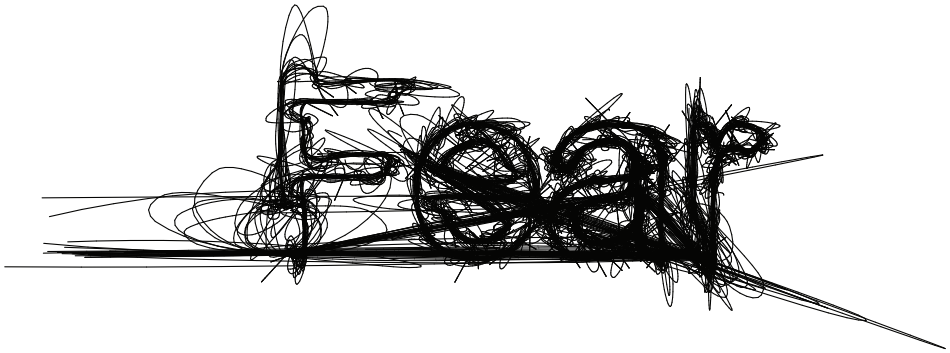
```
import processing.pdf.*;
import processing.opengl.*;
import geomerative.*;
float toldist;
RFont f;
RGroup grupo;
boolean restart = false;
Particle[] psys;
int numPoints, numParticles;
float maxvel;
//---------------------- Runtime properties ------------------------------
// Save each frame
boolean SAVEVIDEO = false;
boolean SAVEFRAME = false;
boolean APPLICATION = true;
String DEFAULTAPPLETRENDERER = P3D;
int DEFAULTAPPLETWIDTH = 600;
int DEFAULTAPPLETHEIGHT = 600;
String DEFAULTAPPLICRENDERER = OPENGL;
int DEFAULTAPPLICWIDTH = 600;
int DEFAULTAPPLICHEIGHT = 600;
//------------------------------------------------------------------------
// Text to be written
String STRNG = "Fear";
// Font to be used
String FONT = "arial.ttf";
// Velocity of change
int VELOCITY = 1;
// Velocity of deformation
float TOLCHANGE = 0.0001;
// Coefficient that handles the variation of amount of ink for the drawing
float INKERRCOEFF = 0.8;
// Coefficient that handles the amount of ink for the drawing
float INKCOEFF = 0.3;
// Coefficient of precision: 0 for lowest precision
float PRECCOEFF = 0.88;
String newString = "";
void setup(){
int w = DEFAULTAPPLICWIDTH, h = DEFAULTAPPLICHEIGHT;
String r = DEFAULTAPPLICRENDERER;
if(!APPLICATION){
// Specify the width and height at runtime
w = int(param("width"));
h = int(param("height"));
r = (String)param("renderer");
// (String) will return null if param("renderer") doesn't exist
if (r != OPENGL && r != P3D && r != JAVA2D && r != P2D) {
r = DEFAULTAPPLETRENDERER;
}
// int() will return 0 if param("width") doesn't exist
if (w <= 0) {
w = DEFAULTAPPLETWIDTH;
}
// int() will return 0 if param("height") doesn't exist
if (h <= 0) {
h = DEFAULTAPPLETHEIGHT;
}
}
size(w,h,r);
frameRate(25);
try{
}
catch(Exception e){
}
```

continued on page 68

Yeohyun Ahn.

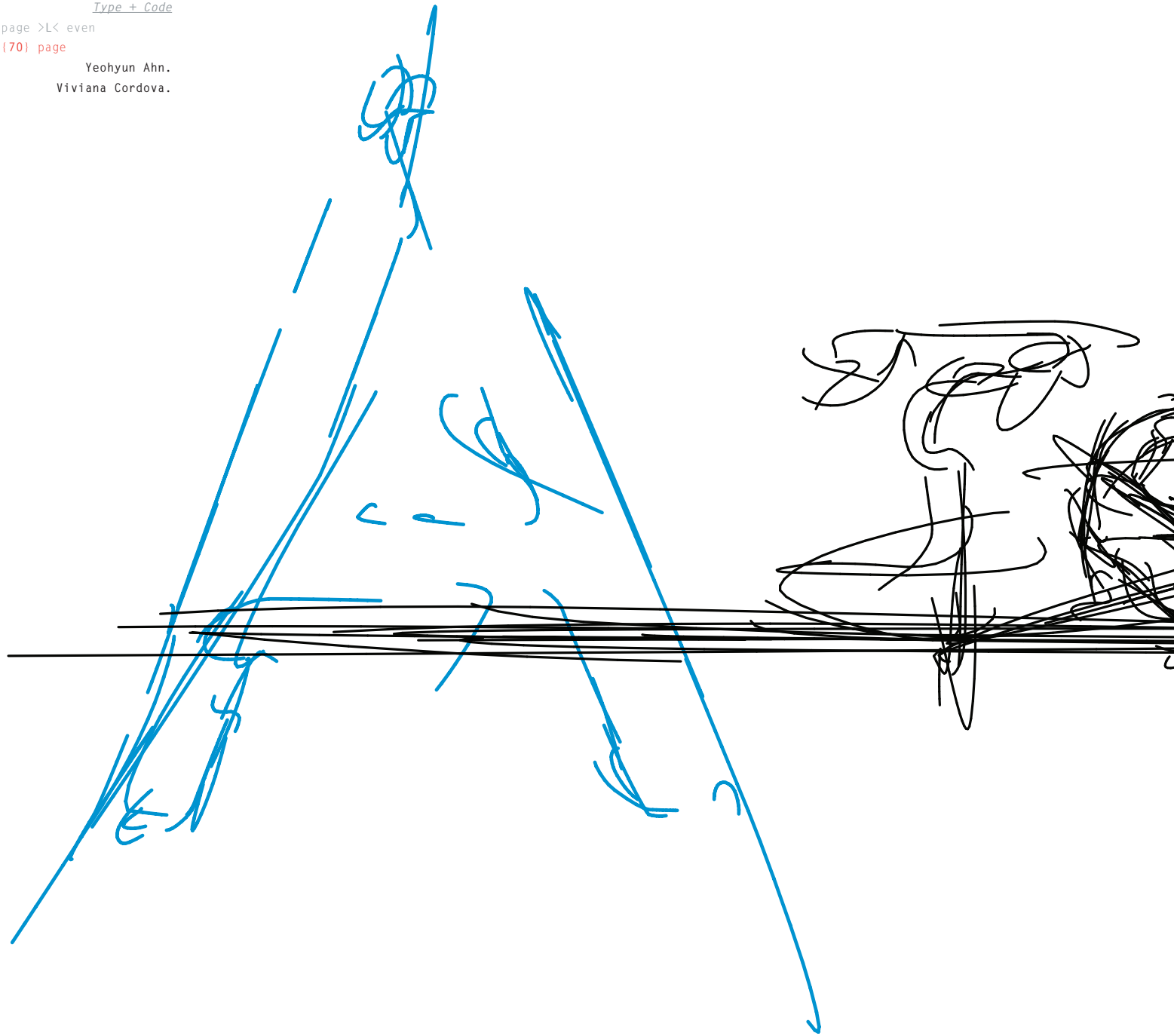
Viviana Cordova.

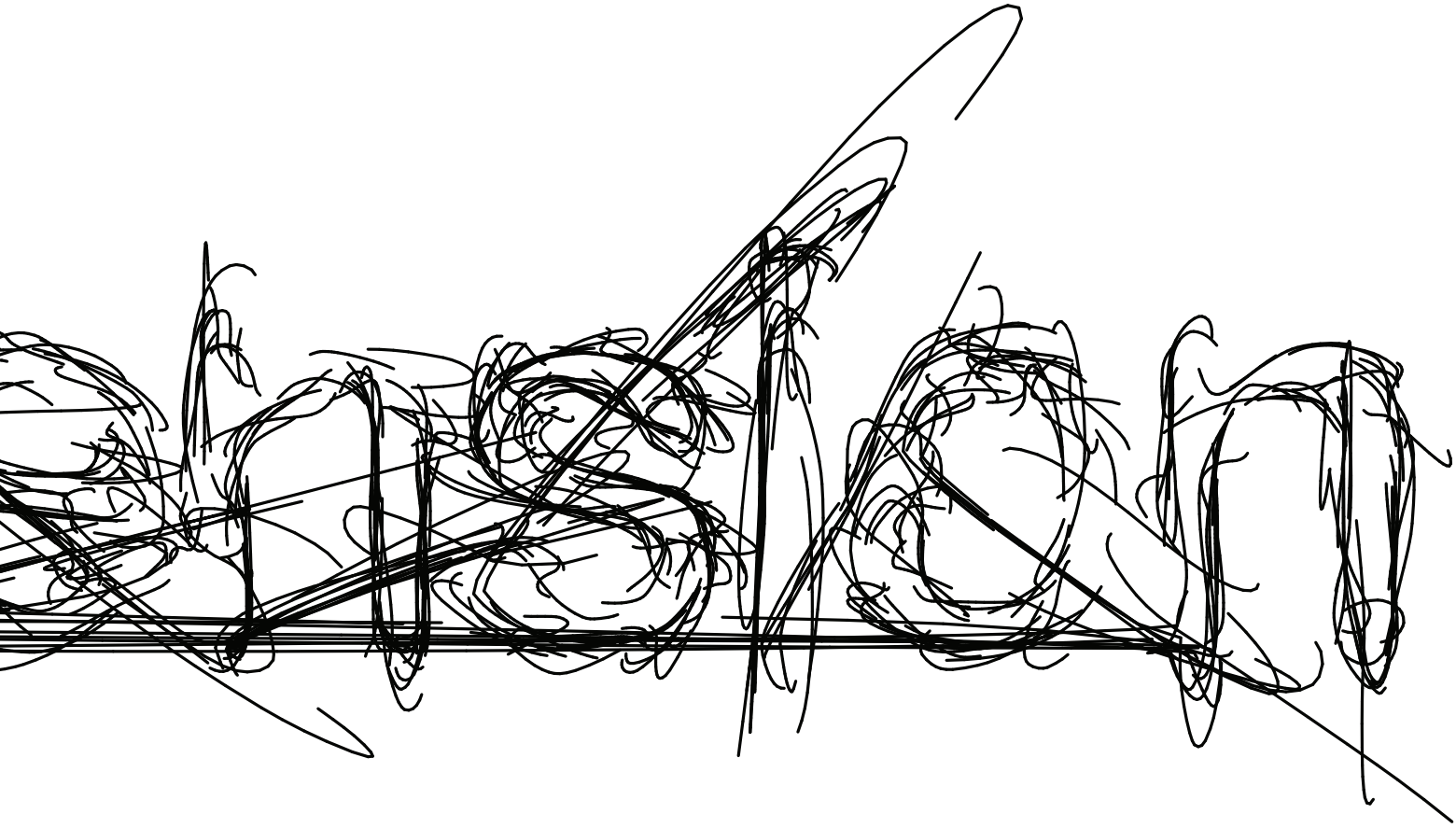
```
background(255);
f = new RFont(this,FONT,372,RFont.CENTER);
initialize();
}
void draw(){
pushMatrix();
translate(width/2, height/2);
noStroke();
for(int i=0;i<numParticles;i++){
for(int j=0;j<VELOCITY;j++){
psys[i].update(grupo);
psys[i].draw(g);
}
}
popMatrix();
if(SAVEVIDEO) saveFrame("filesvideo/"+STRNG+"-##
  ##.tga");
toldist += TOLCHANGE;
}
void initialize(){
toldist = ceil(width/200F) * (6F/(STRNG.
  length()+1));
maxvel = width/80F * INKERRCOEFF * (6F/(STRNG.
  length()+1));
grupo = f.toGroup(STRNG);
RCommand.setSegmentStep(1-
  constrain(PRECCOEFF,0,0.99));
RCommand.setSegmentator(RCommand.UNIFORMSTEP);
grupo = grupo.toPolygonGroup();
grupo.centerIn(g, 5, 1, 1);
background(255);
RPoint[] ps = grupo.getPoints();
numPoints = ps.length;
numParticles = numPoints;
psys = new Particle[numParticles];
for(int i=0;i<numParticles;i++){
psys[i] = new Particle(g,i,int(float(i)/
  float(numParticles)*125));
psys[i].pos = new RPoint(ps[i]);
psys[i].vel.add(new RPoint(random(-
  10,10),random(-10,10)));
}
toldist = 8;
}
void keyReleased(){
if(keyCode==ENTER){
STRNG = newString;
newString = "";
initialize();
}
else if(keyCode==BACKSPACE){
if(newString.length() !=0 ){
newString = newString.substring(0,newString.
  length()-1);
}
}
else if(keyCode!=SHIFT){
newString += key;
}
}
public class Particle{
// Velocity
RPoint vel;
// Position
RPoint pos;
RPoint lastpos;
// Characteristics
int col;
int hueval;
float sz;
// ID
int id;
// Constructor
public Particle(PGraphics gfx, int ident, int
  huevalue){
pos = new RPoint(random(-gfx.width/2,gfx.width/2),
  random(-gfx.height/2,gfx.height/2));
lastpos = new RPoint(pos);
vel = new RPoint(0, 0);
colorMode(HSB);
sz = random(2,3);
id = ident;
hueval = huevalue;
}
// Updater of position, velocity and color
  depending on a RGroup
public void update(RGroup grp){
lastpos = new RPoint(pos);
pos.add(vel);
RPoint[] ps = grp.getPoints();
if(ps != null){
float distancia = dist(pos.x,pos.y,ps[id].x,ps[id
  ].y);
if(distancia <= toldist){
id = (id + 1) % ps.length;
}
RPoint distPoint = new RPoint(ps[id]);
distPoint.sub(pos);
distPoint.scale(random(0.028,0.029));
vel.scale(random(0.5,1.3));
vel.add(distPoint);
float sat = constrain((width-
  distancia)*0.25,0.001,255);
float velnorm = constrain(vel.norm(),0,maxvel);
sat = abs(maxvel-velnorm)/maxvel*INKCOEFF*255;
sat = constrain(sat,0,255);
col = color(hueval,150,255,sat*(toldist/80));
}
}
public void setPos(RPoint newpos){
lastpos = new RPoint(pos);
pos = newpos;
}
// Drawing the particle
public void draw(PGraphics gfx){
noSmooth();
stroke(2);
line(lastpos.x/1.4,lastpos.y/1.4,pos.x/1.4,
  pos.y/1.4);
}
}
```



Scribble typeface *ƒ*.

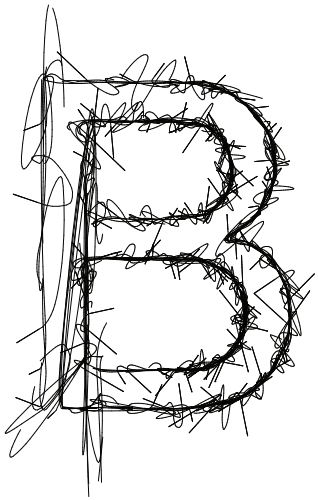
Yeohyun Ahn.
Viviana Cordova.

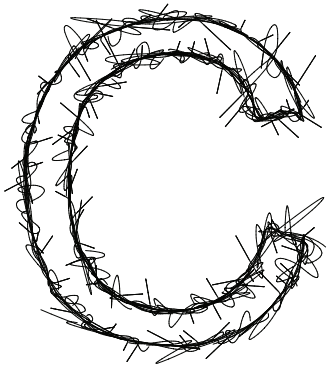


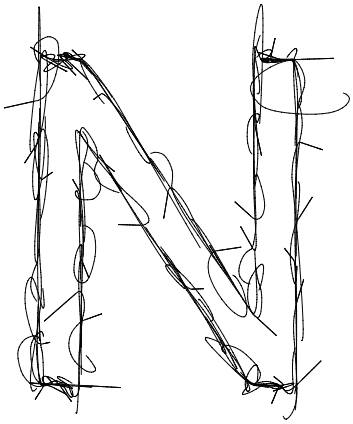


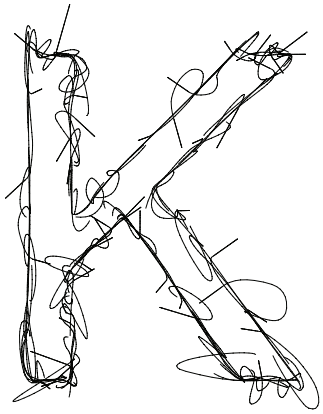
Caligraft "Tension" code is provided so you can gain a better understanding of code strings and structure. If you replace the initial string, as well as several integral variables, the outcome changes.

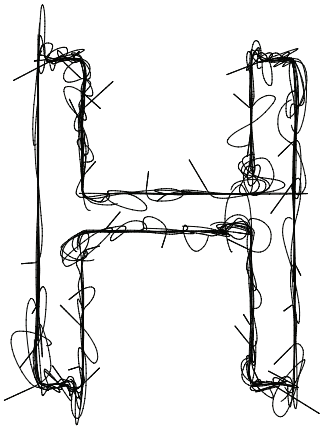
Yeohyun Ahn.
Viviana Cordova.

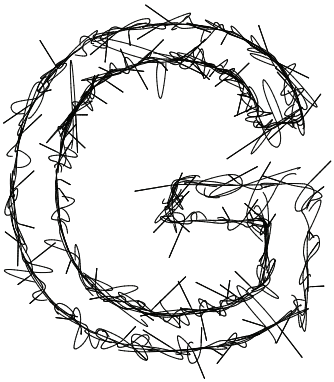


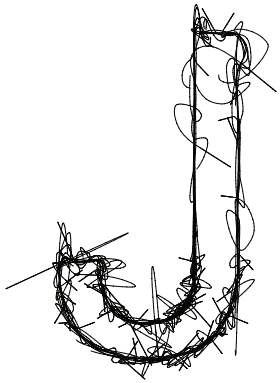


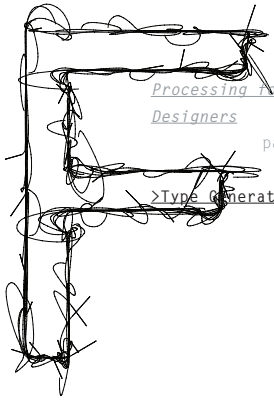


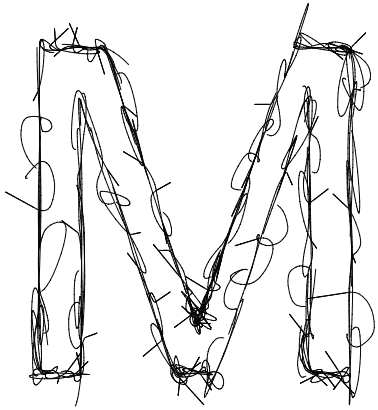




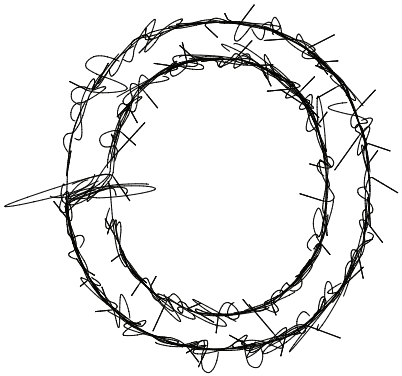


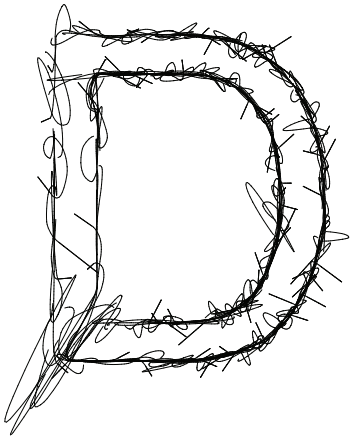




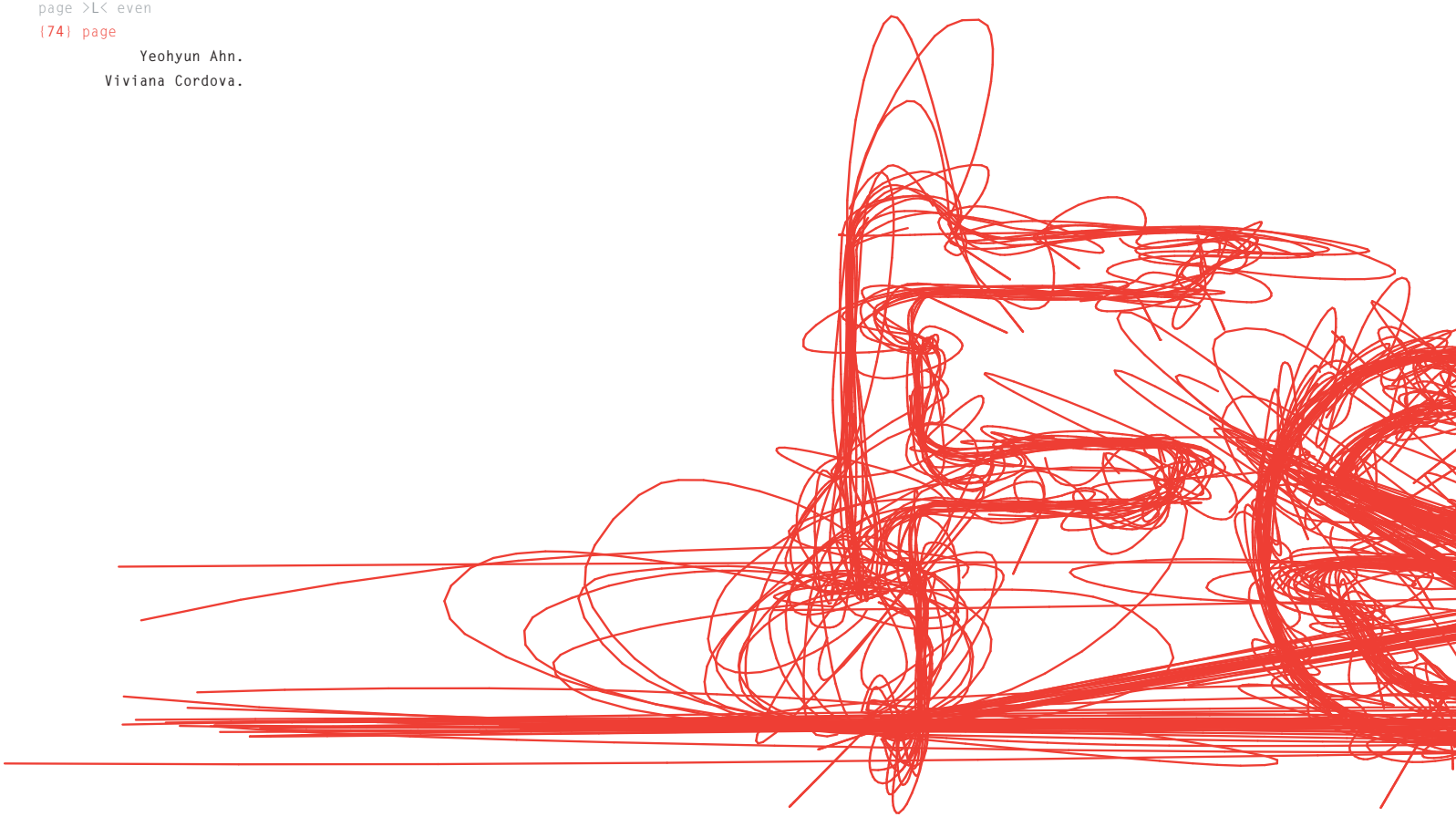


Scribble typeface *ƒ*.

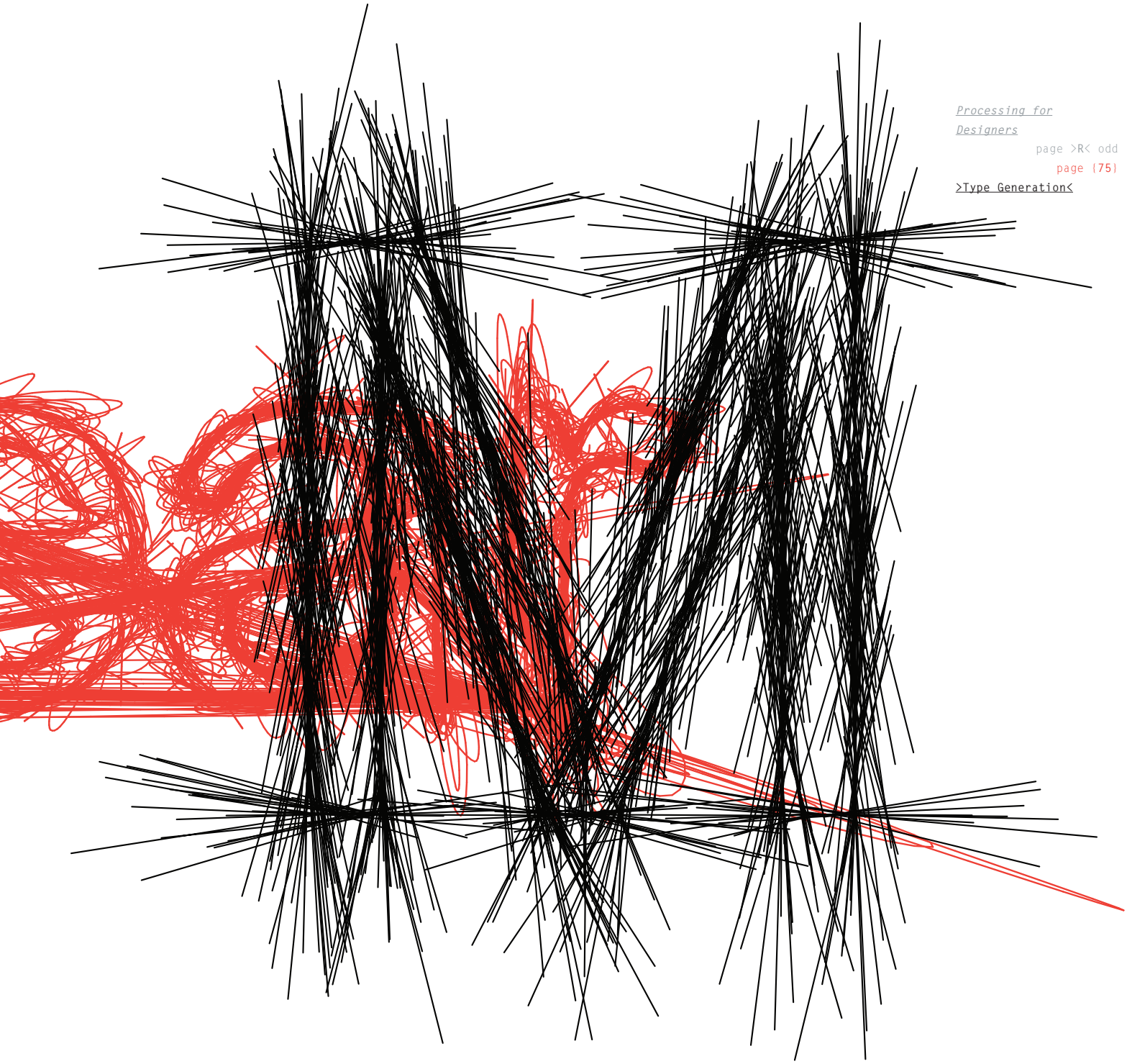




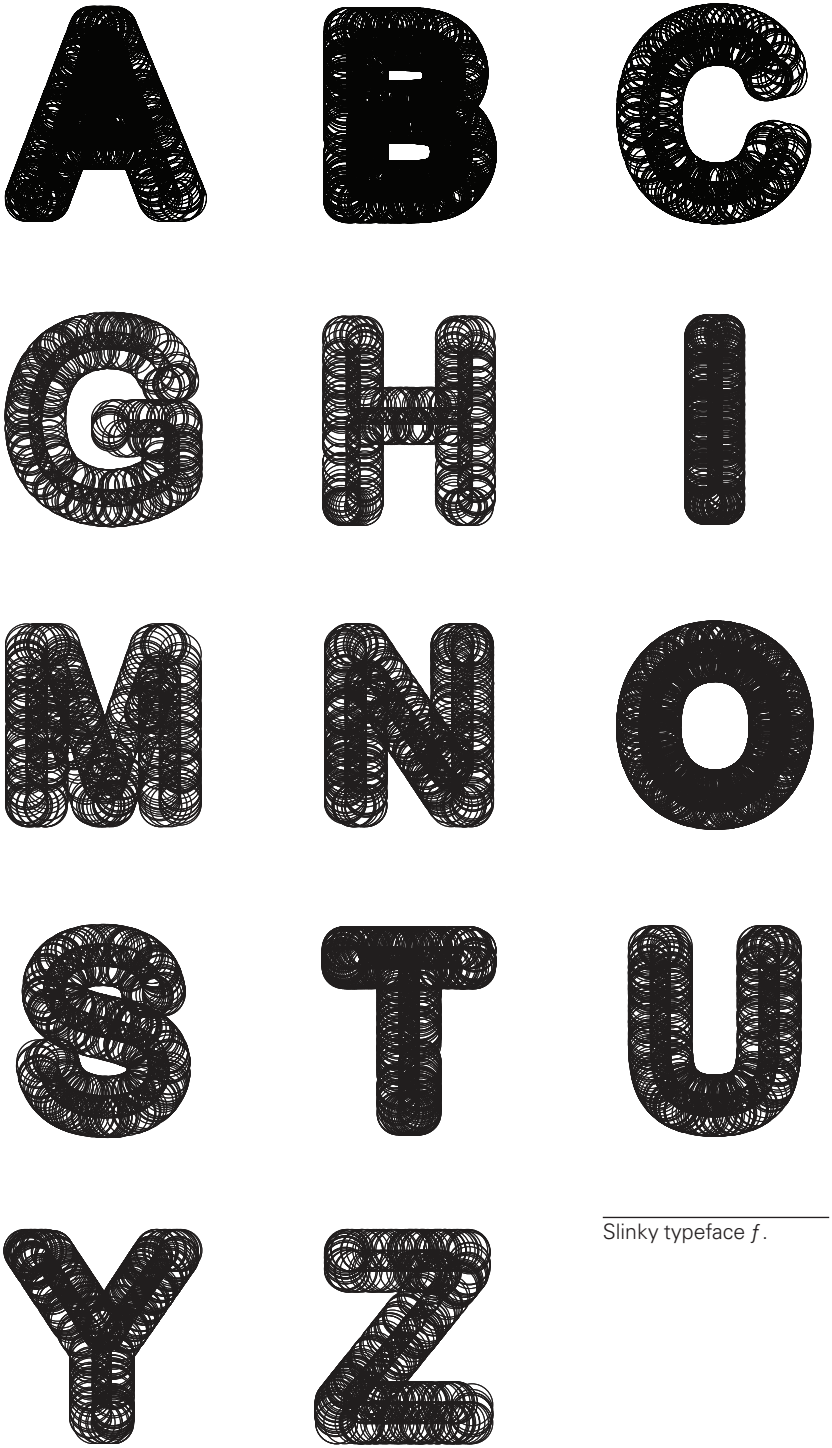
Yeohyun Ahn.
Viviana Cordova.



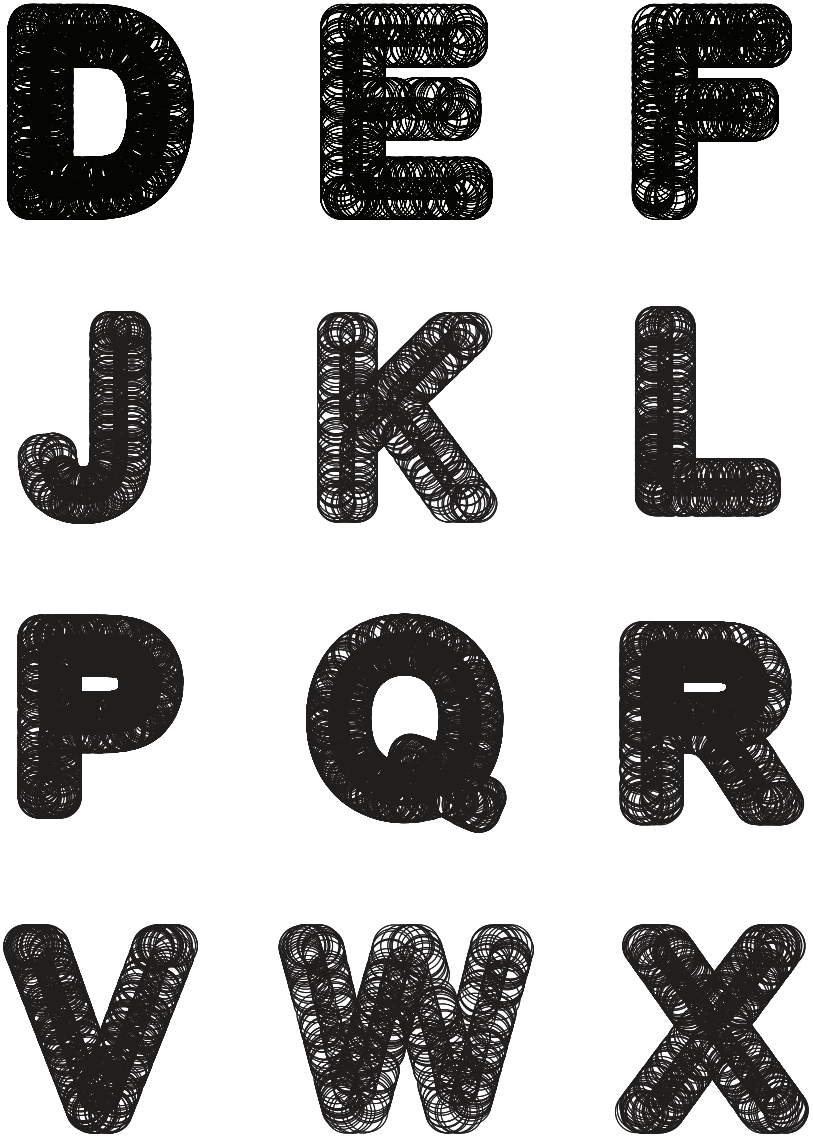
This design combines two separate Caligraft code structures that together create a striking contrast of line qualities. While Processing does not effectively support multiple samples, flattened frames can be combined in print applications. The word "Fear" belongs to sample code 1, while "M" uses sample code 2.



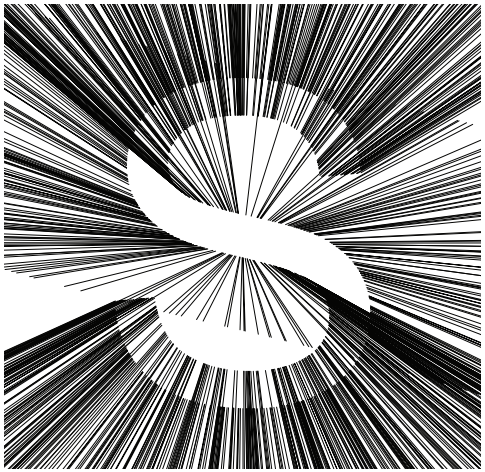
Yeohyun Ahn.
Viviana Cordova.

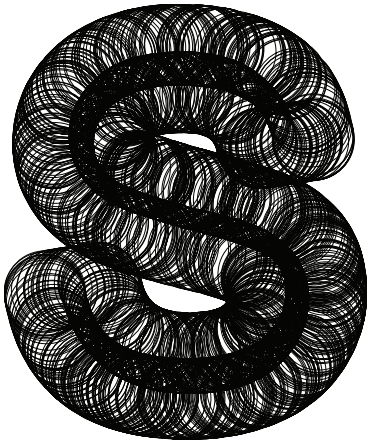


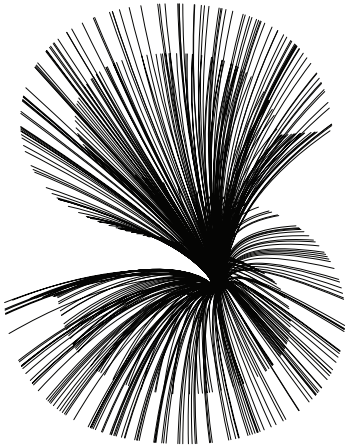
Slinky typeface *ƒ*.

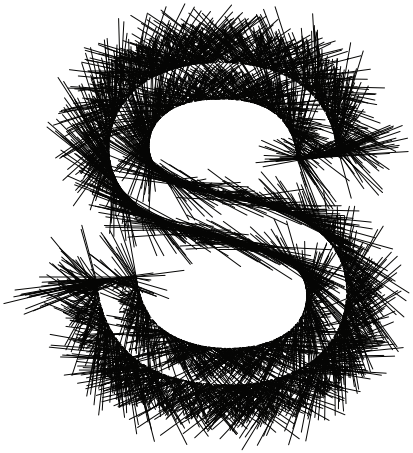

D E F
J K L
P Q R
V W X

Yeohyun Ahn.
Viviana Cordova.









To create the four variations here, alter the highlighted red code (on page 80) of sample code 2 with these four code examples (right).

```
line(p.x/1.2, p.y/1.2, p.x*5, p.y*5)
```

```
bezier(p.x, p.y, 10, 10, 90, 90, 15, 80)
```

```
ellipse(p.x/1.1, p.y/1.1, 27,27)
```

```
line(p.x/1.2, p.y/1.2, p.x +
len*cos(angle), p.y + len*sin(angle))
```

17.2

Caligraft
Sample Code 2
Slinky, Spike, Grass, Hole and Light Dark

To create the letters on the facing page using Processing, you need a TrueType (.ttf) seed font. Here, we use Arial.ttf, but any TrueType font will work. Arial.ttf should be placed under the same folder as the sample code 2. Download the Geomerative library (by Ricard Marxer Piñón) from www.caligraft.com. After it is downloaded, place it inside the Libraries folder, within your Processing application folder. Now you are ready to explore new forms of typography with Processing. After entering the sample code 2 in the Processing sketch window, click Run. The resulting output is shown bottom left on the facing page. You can then alter the highlighted red code to produce the other three letterforms shown—for example, if you change the existing `bezier(p.x, p.y, 10, 10, 90, 90, 15, 80)` to `line(p.x/1.2, p.y/1.2, p.x*5, p.y*5)`, you will get the top-left letter "S."

To use this code:
1. Arial.ttf is needed
2. Download Geomerative Library from www.caligraft.com.

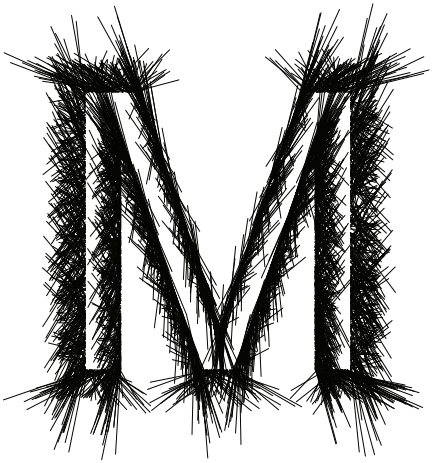
```
import geomerative.*;
RFont f;
RGroup grp;
RShape s;
float len;
float angle;
float pos;
//---------------------- Runtime properties ------------------------------
// Save each frame
boolean SAVEVIDEO = false;
boolean SAVEFRAME = false;
boolean APPLICATION = false;
String DEFAULTAPPLETRENDERER = JAVA2D;
int DEFAULTAPPLETWIDTH = 600;
int DEFAULTAPPLETHEIGHT = 600;
String DEFAULTAPPLICRENDERER = OPENGL;
int DEFAULTAPPLICWIDTH = 600;
int DEFAULTAPPLICHEIGHT =600;
//------------------------------------------------------------------------
// The error range for the tangent position and angle
float ANGLEERROR = 0.3;
float POINTERROR = 0;
// The length variation of the tangent
// -> 500: sketchy, blueprint
// -> 150: light blueprint
// -> 2000: mystic
float LENGTHTANGENT = 130;
// The initial text
String STRNG = "S";
String FONT = "arial.ttf";
// The alpha value of the lines
int ALPHAVALUE = 2;
// The velocity of the calligraphy
int VELOCITY = 10;
int MARGIN = 50;
String newString = "";
void setup(){
int w = DEFAULTAPPLICWIDTH, h = DEFAULTAPPLICHEIGHT;
String r = DEFAULTAPPLICRENDERER;
if(!APPLICATION){
// Specify the width and height at runtime
w = int(param("width"));
h = int(param("height"));
r = (String)param("renderer");
// (String) will return null if param("renderer") doesn't exist
if (r != OPENGL && r != P3D && r != JAVA2D && r != P2D) {
r = DEFAULTAPPLETRENDERER;
}
// int() will return 0 if param("width") doesn't exist
if (w <= 0) {
w = DEFAULTAPPLETWIDTH;
}
// int() will return 0 if param("height") doesn't exist
if (h <= 0) {
h = DEFAULTAPPLETHEIGHT;
}
}
size(w,h,r);
background(255);
framerate(25);
LENGTHTANGENT = LENGTHTANGENT * width/800F;
try{
smooth();
}catch(Exception e){}
```

continued on page 80

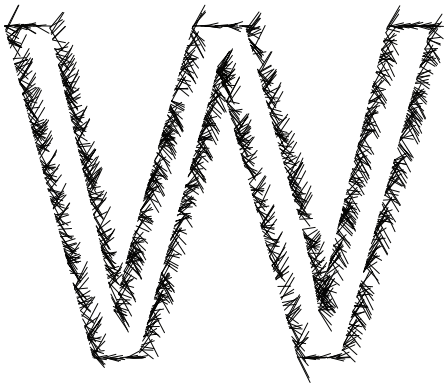
Yeohyun Ahn.

Viviana Cordova.

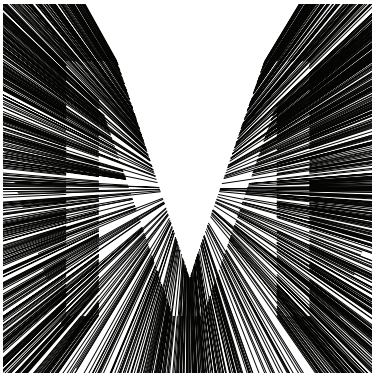
```
noFill();
stroke(0, 0, 0, 50);
f = new RFont(this, FONT, 372, RFont.CENTER);
initialize();
}
void draw(){
pushMatrix();
translate(width/2,height/2);
// Draw very low alpha and long tangents on random
   points of each letters
for(int i=0;i<grp.countElements();i++){
s = (RShape)(grp.elements[i]);
for(int j=0;j<s.countSubshapes();j++){
for(int k=0;k<VELOCITY;k++){
pos = random(0, 1);
RPoint tg = s.subshapes[j].getCurveTangent(pos);
RPoint p = s.subshapes[j].getCurvePoint(pos);
p.x = p.x + random(-POINTERROR,POINTERROR);
p.y = p.y + random(-POINTERROR,POINTERROR);
len = random(-LENGTHTANGENT, LENGTHTANGENT);
angle = atan2(tg.y, tg.x) + random(-ANGLEERROR,
   ANGLEERROR);
bezier(p.x, p.y, 10, 10, 90, 90, 15, 80);
}
}
}
popMatrix();
}
void initialize(){
grp = f.toGroup(STRNG);
grp.centerIn(g,MARGIN,1,1);
background(
```



Spike typeface *ƒ*.

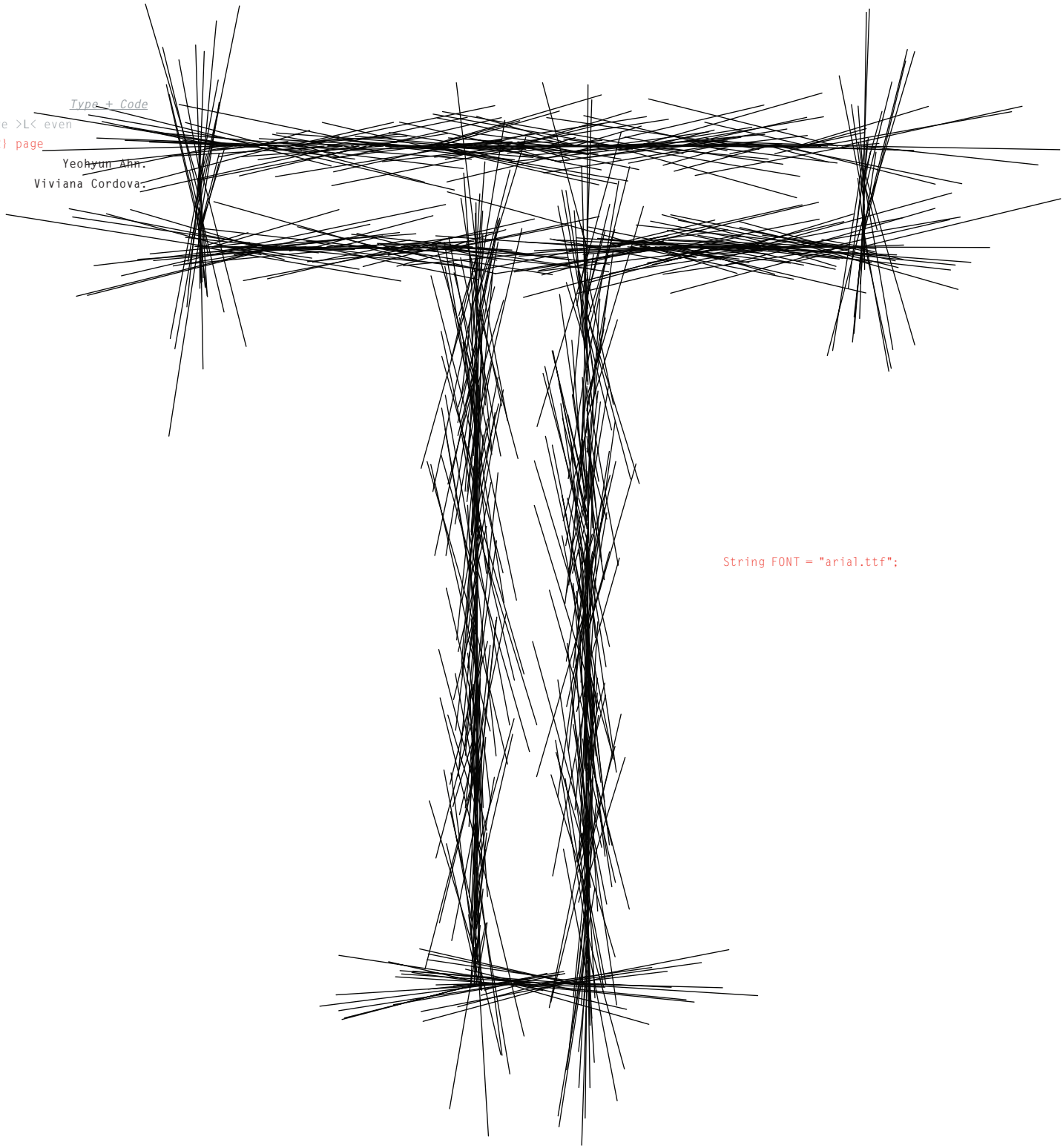


Grass typeface *ƒ*.



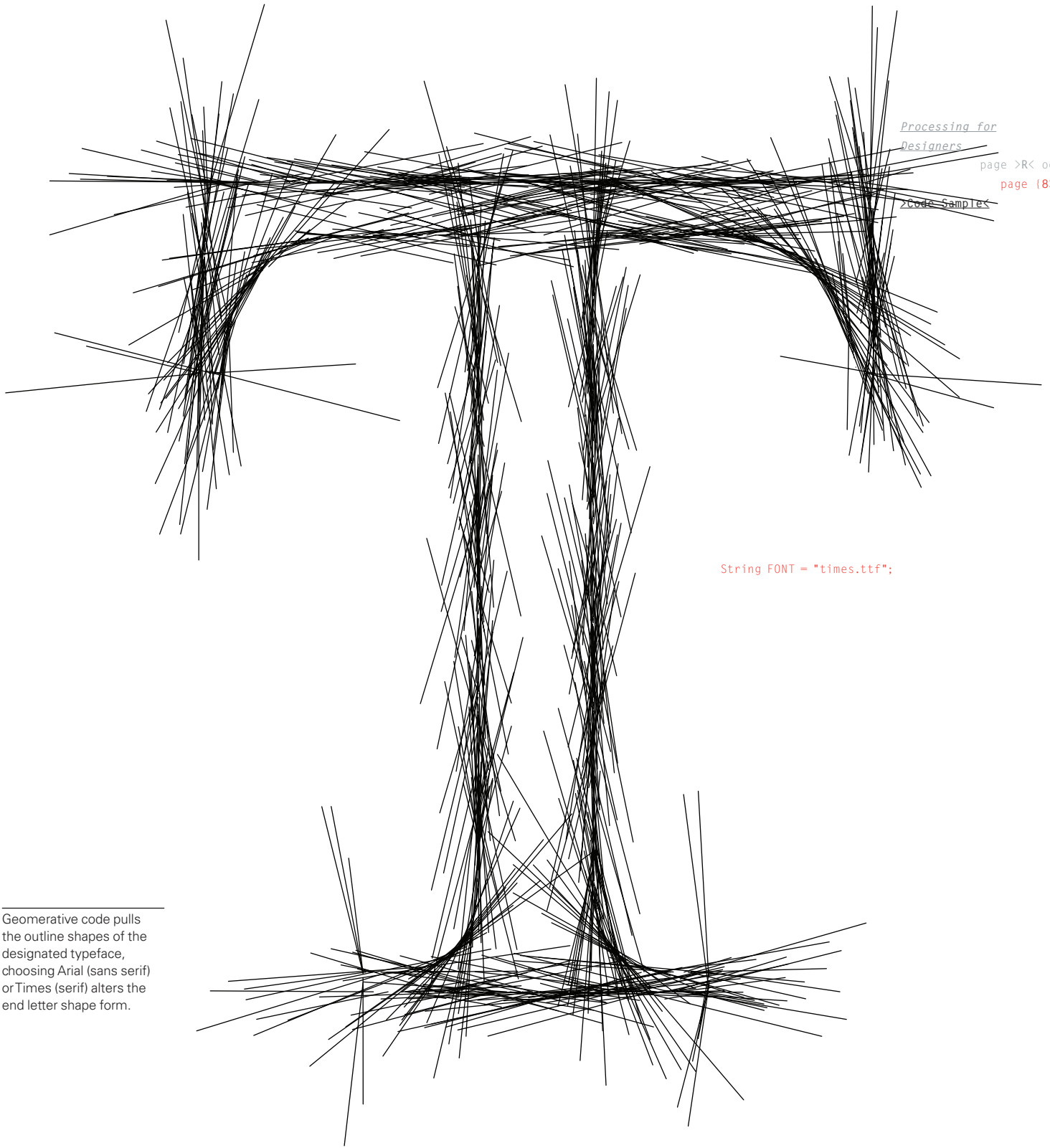
Light Dark typeface *ƒ*.

Yeohyun Ahn.
Viviana Cordova.



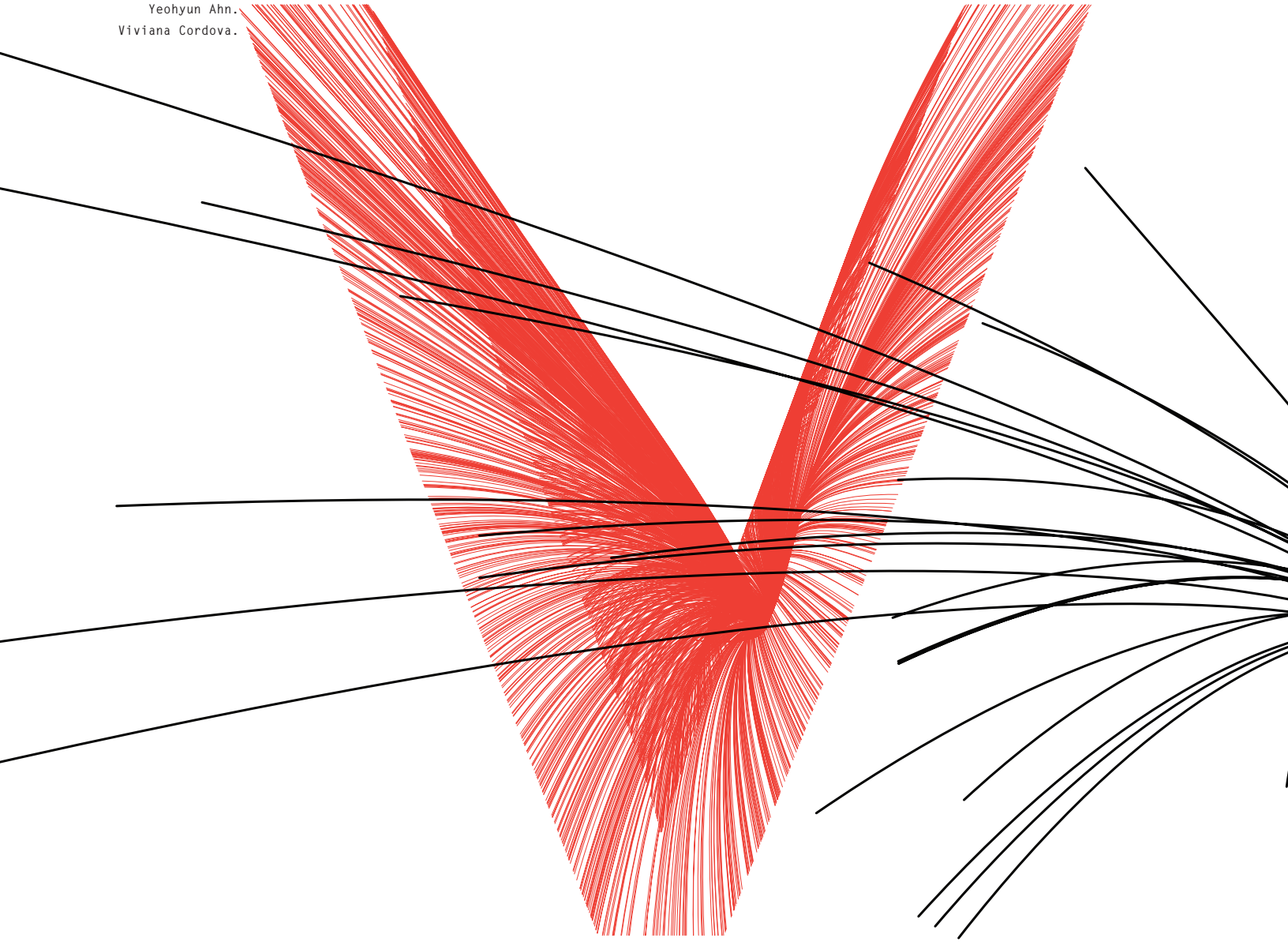
```
String FONT = "arial.ttf";
```

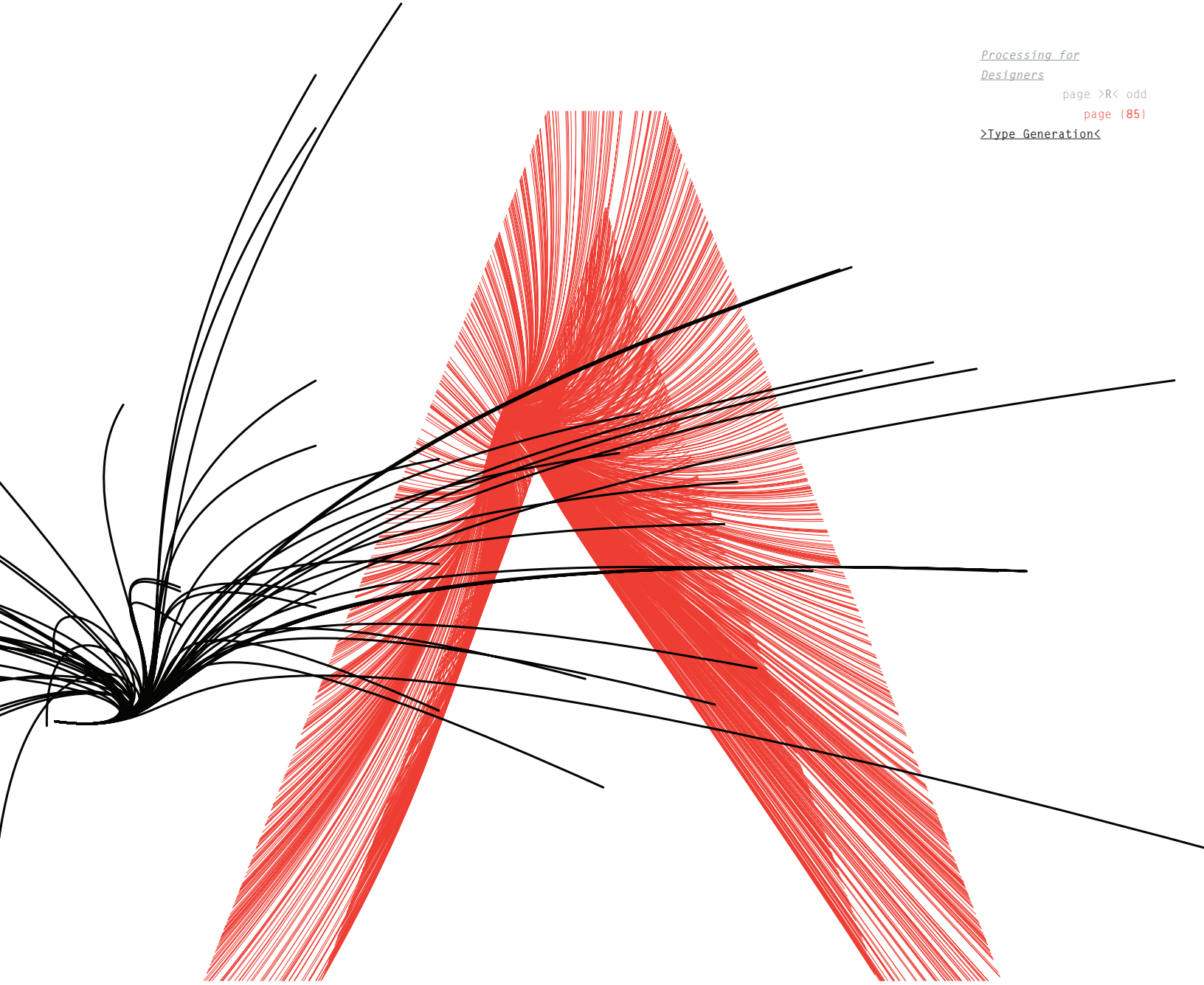
```
String FONT = "times.ttf";
```

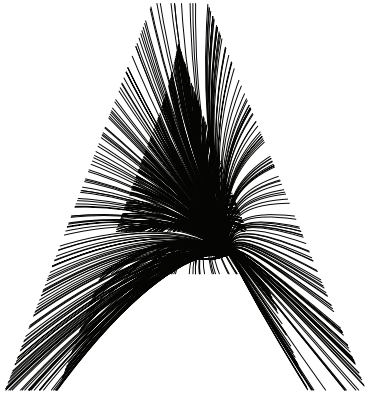


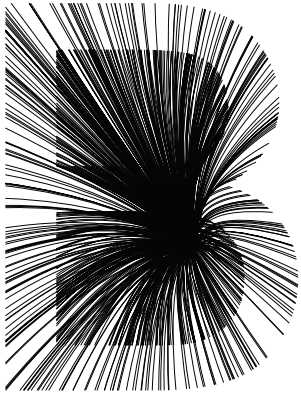
Geomerative code pulls the outline shapes of the designated typeface, choosing Arial (sans serif) or Times (serif) alters the end letter shape form.

Yeohyun Ahn.
Viviana Cordova.









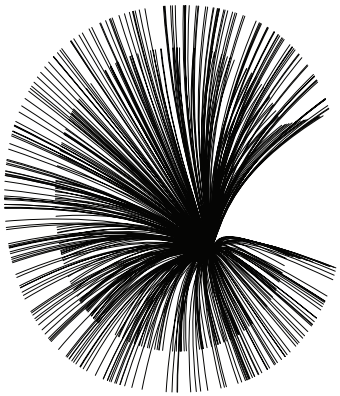
Type + Code

page >L< even

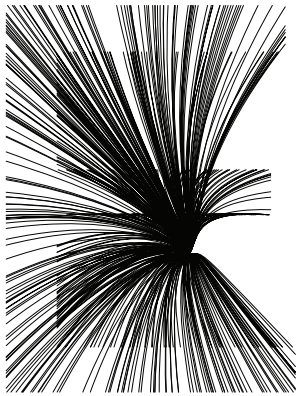
{86} page

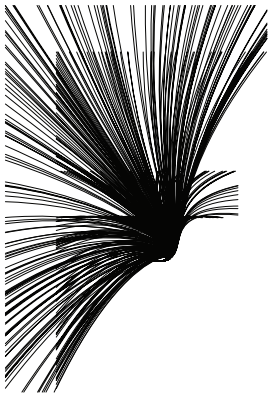
Yeohyun Ahn.

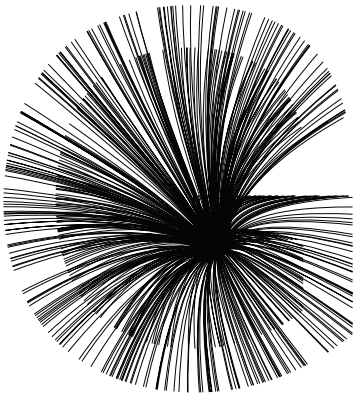
Viviana Cordova.

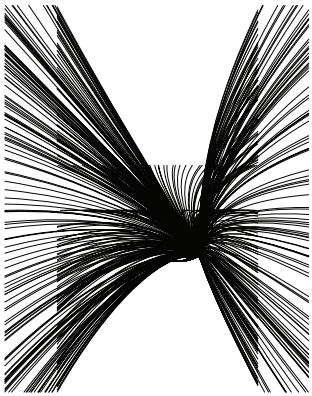


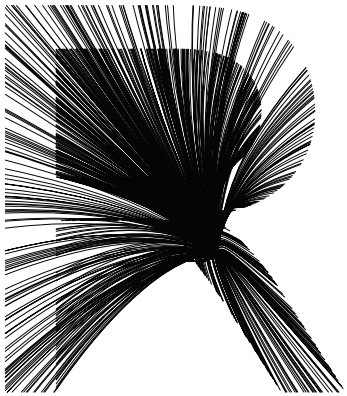
Hole typeface *f*.

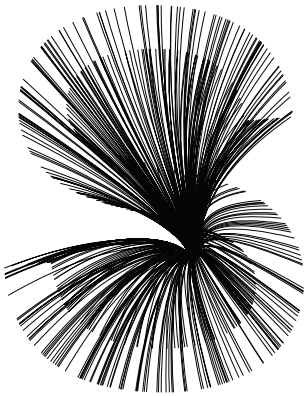


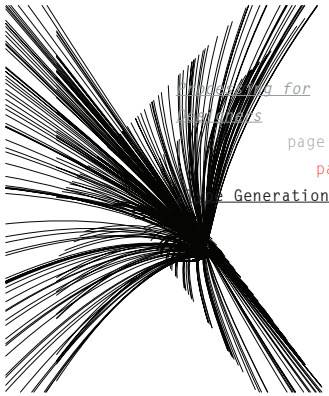


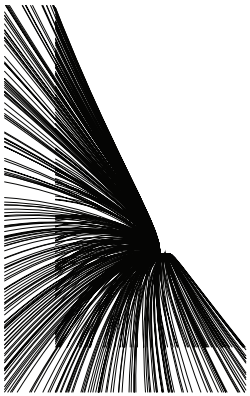




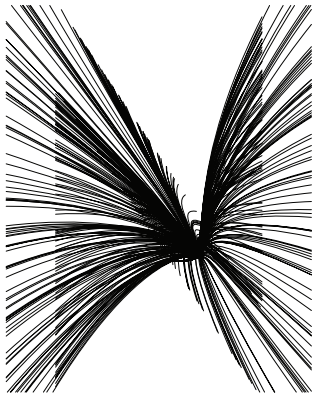


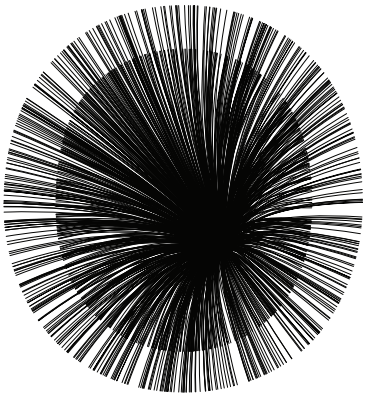


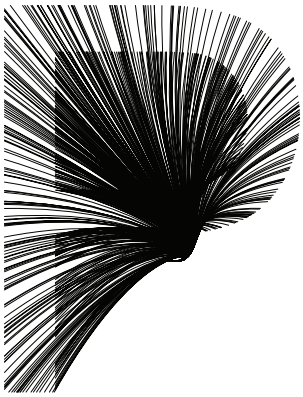


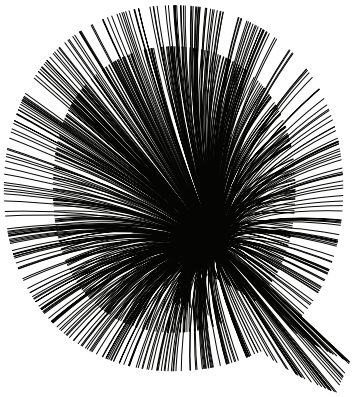


Processing for Designers
page >R< odd
page {87}
>Type Generation<

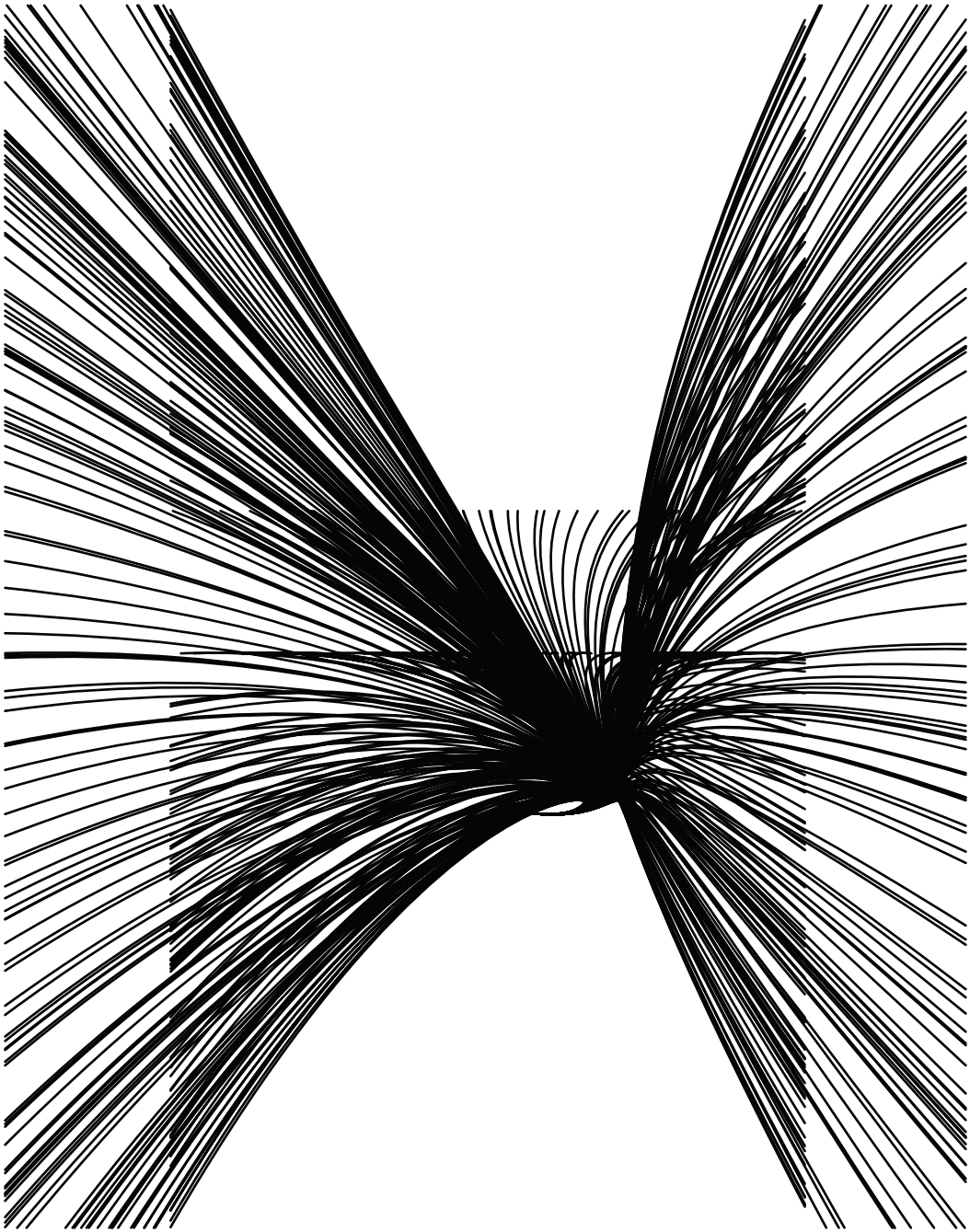


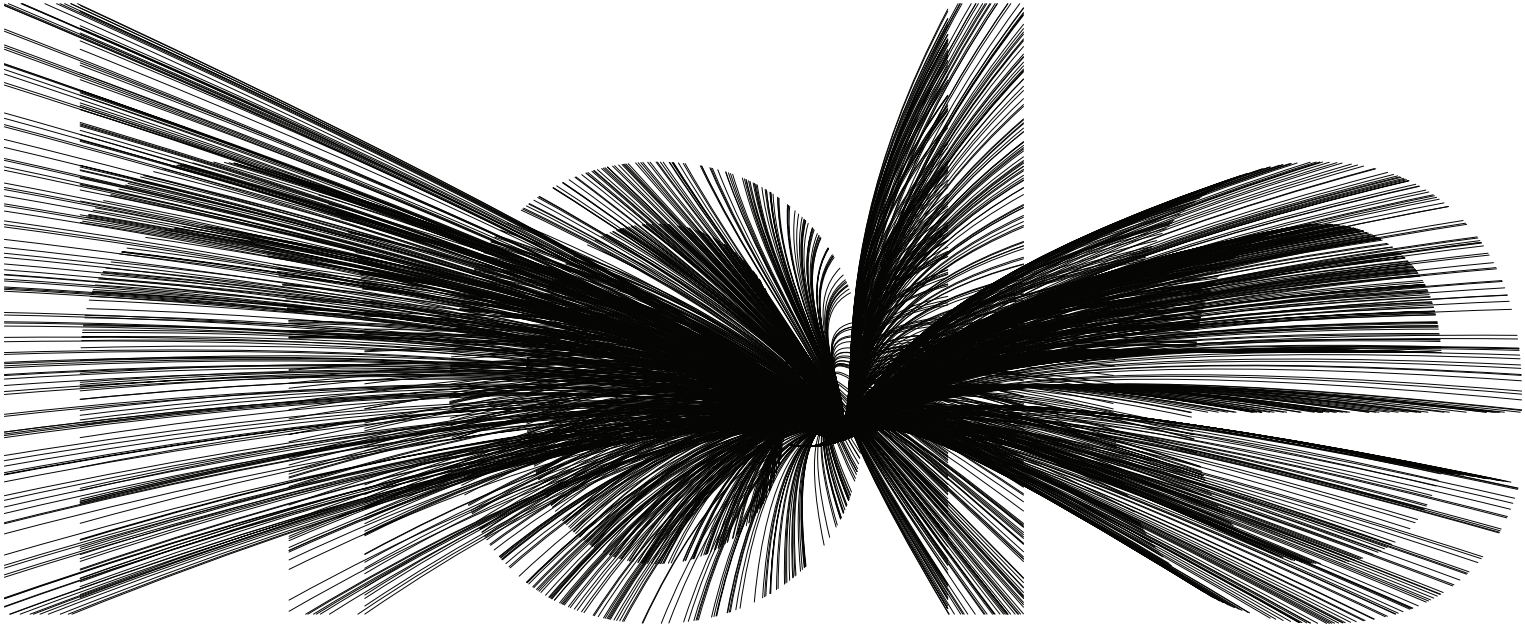






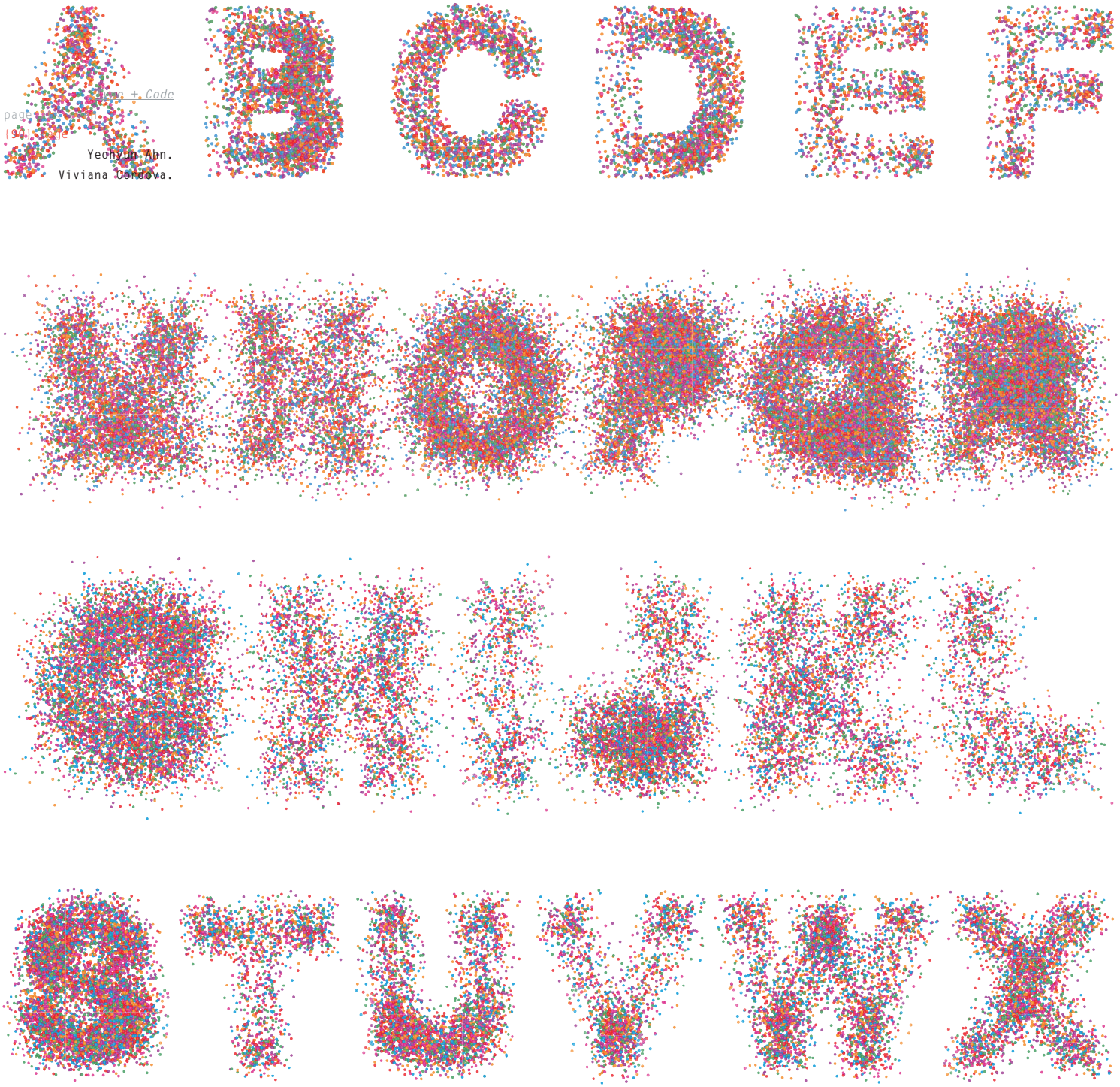
Yeohyun Ahn.
Viviana Cordova.

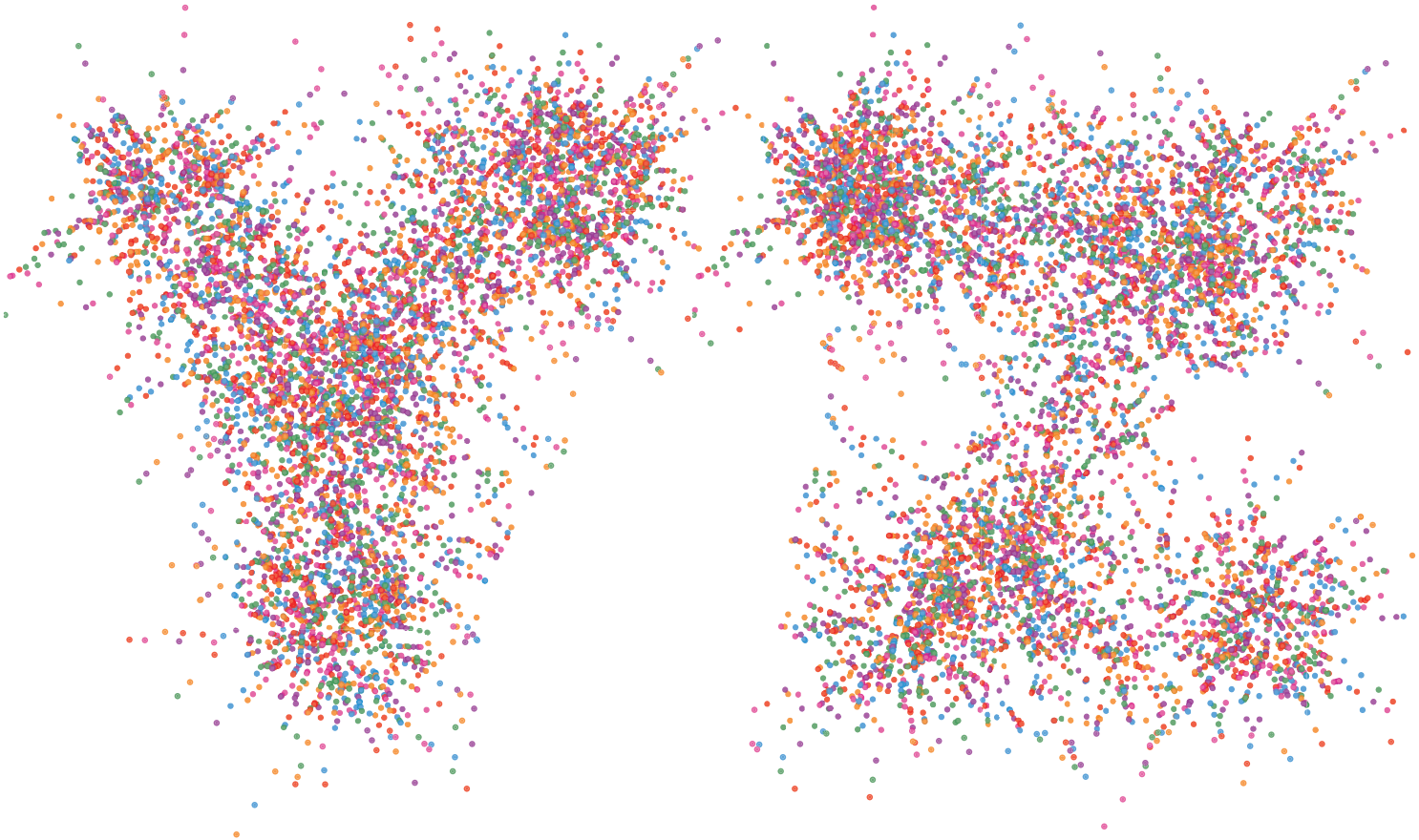




Geomerative codes can be used for single letter or full word renderings.

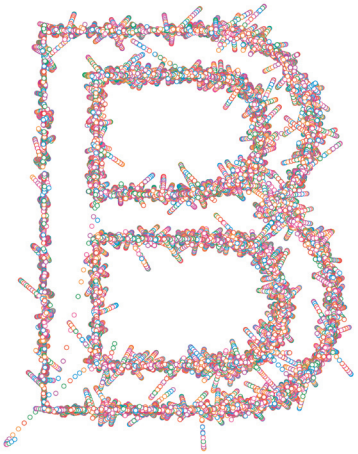
Yeonyun Ahn.
Viviana Cordova.



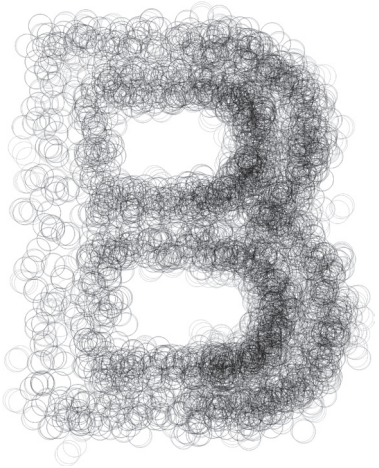


Points typeface *f*.

Yeohyun Ahn.
Viviana Cordova.



```
ellipse(lastpos.x/1.5,lastpos.y/1.5,8, 8);
```



```
ellipse(lastpos.x/2,lastpos.y/2,25, 25);
With stroke(120,100);
```

The red highlighted code determines the position and circumference of the ellipse. The numeric values for black have been added to effect the stroke (right example) and override the previously written color palette.

17.3

Caligraft
Sample Code 3
Points and Hair

Code 3: list.txt should be within the same folder as sample code 3.
The on/off settings inside list.txt:

On

On

On

On

On

On

The index file toggles between "on" and "off," and allows the user to indicate which color set is active or inactive without altering the base Processing file. Individual colors can be determined by the user and rendered by accessing the list.txt file.

Color Index
color Color_Index[]={#e02892, #9a3594, #0094d8, #3f9a5a, #f5851f, #ed1c24};

- e02892
- 9a3594
- 0094d8
- 3f9a5a
- f5851f
- ed1c24

To use this code:
1. Arial.ttf is needed
2. Download Geomerative Library from www.caligraft.com.

```
import processing.opengl.*;
import geomerative.*;
float toldist;
RFont f;
RGroup grupo;
boolean restart = false;
Particle[] psys;
int numPoints, numParticles;
float maxvel;
//----------------------- Runtime properties ---------------------------------
// Save each frame
boolean SAVEVIDEO = false;
boolean SAVEFRAME = false;
boolean APPLICATION = true;
String DEFAULTAPPLETRENDERER = P3D;
int DEFAULTAPPLETWIDTH = 800;
int DEFAULTAPPLETHEIGHT = 800;
String DEFAULTAPPLICRENDERER = OPENGL;
int DEFAULTAPPLICWIDTH = 800;
int DEFAULTAPPLICHEIGHT = 800;
//----------------------------------------------------------------------------
// Text to be written
String STRNG = "B";
// Font to be used
String FONT = "arial.ttf";
// Velocity of change
int VELOCITY = 1;
// Velocity of deformation
float TOLCHANGE = 0.0001;
// Coefficient that handles the variation of amount of ink for the drawing
float INKERRCOEFF = 0.001;
// Coefficient that handles the amount of ink for the drawing
float INKCOEFF = 0.1;
// Coefficient of precision: 0 for lowest precision
float PRECCOEFF = 4;
String newString = "";
int Start_Second;
int Gray1 = 150;
int Gray2 = 150;
int Gray3 = 150;

color Color_Index[]={#e02892, #9a3594, #0094d8, #3f9a5a, #f5851f, #ed1c24};
int Mood_Trigger[] = {0,0,0,0,0,0}; int Active_Color =0;
int Alpha = 200;
void setup(){
int w = DEFAULTAPPLICWIDTH, h = DEFAULTAPPLICHEIGHT;
String r = DEFAULTAPPLICRENDERER;
if(!APPLICATION){
// Specify the width and height at runtime
w = int(param("width"));
h = int(param("height"));
r = (String)param("renderer");
// (String) will return null if param("renderer") doesn't exist
if (r != OPENGL && r != P3D && r != JAVA2D && r != P2D) {
r = DEFAULTAPPLETRENDERER;
}
// int() will return 0 if param("width") doesn't exist
if (w <= 0) {
w = DEFAULTAPPLETWIDTH;
}
// int() will return 0 if param("height") doesn't exist
if (h <= 0) {
h = DEFAULTAPPLETHEIGHT;
}
}
```

continued on page 94

Yeohyun Ahn.

Viviana Cordova.

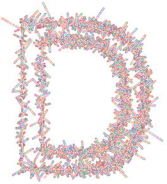
```
size(w,h,r);
framerate(35);
try{
smooth();
}
catch(Exception e){
}
background(255);
f = new RFont(this,FONT,72,RFont.CENTER);
initialize();
Start_Second = second();
}
void draw(){
pushMatrix();
translate(width/2, height/2);
noStroke();
for(int i=0;i<numParticles;i++){
for(int j=0;j<VELOCITY;j++){
psys[i].update(grupo);
psys[i].draw(g);
}
}
popMatrix();
toldist =1;
}
void initialize(){
toldist = ceil(width/200F) * (6F/(STRNG.
   length()+1));
maxvel = width/80F * INKERRCOEFF * (6F/(STRNG.
   length()+1));
grupo = f.toGroup(STRNG);
RCommand.setSegmentStep(1-
   constrain(PRECCOEFF,0,0.99));
RCommand.setSegmentator(RCommand.UNIFORMSTEP);
grupo = grupo.toPolygonGroup();
grupo.centerIn(g, 5, 1, 1);
background(255);
RPoint[] ps = grupo.getPoints();
numPoints = ps.length;
numParticles = numPoints;
psys = new Particle[numParticles];
for(int i=0;i<numParticles;i++){
psys[i] = new Particle(g,i,int(float(i)/
   float(numParticles)*125));
psys[i].pos = new RPoint(ps[i]);
psys[i].vel.add(new RPoint(random(-
   10,10),random(-10,10)));
}
toldist = 1;
}
void keyReleased(){
if(keyCode==ENTER){
STRNG = newString;
newString = "";
initialize();
}
else if(keyCode==BACKSPACE){
if(newString.length() !=0 ){
newString = newString.substring(0,newString.
   length()-1);
}
}
else if(keyCode!=SHIFT){
newString += key;
}
}
public class Particle{
// Velocity
RPoint vel;
// Position
RPoint pos;
RPoint lastpos;
// Characteristics
int col;
int hueval;
float sz;
// ID
int id;
// Constructor
public Particle(PGraphics gfx, int ident, int
   huevalue){
pos = new RPoint(random(-gfx.width/2,gfx.width/2),
   random(-gfx.height/2,gfx.height/2));
lastpos = new RPoint(pos);
vel = new RPoint(0, 0);
//colorMode(HSB);
colorMode(RGB);
sz = random(2,3);
id = ident;
hueval = huevalue;
}
// Updater of position, velocity and color
   depending on a RGroup
public void update(RGroup grp){
lastpos = new RPoint(pos);
pos.add(vel);
RPoint[] ps = grp.getPoints();
if(ps != null){
float distancia = dist(pos.x,pos.y,ps[id].x,ps[id
   ].y);
if(distancia <= toldist){
id = (id + 1) % ps.length;
}
RPoint distPoint = new RPoint(ps[id]);
distPoint.sub(pos);
distPoint.scale(random(0.028,0.029));
vel.scale(random(0.5,1.3));
vel.add(distPoint);
float sat = constrain((width-
   distancia)*0.25,0.001,255);
float velnorm = constrain(vel.norm(),0,maxvel);
sat = abs(maxvel-velnorm)/maxvel*INKCOEFF*255;
sat = constrain(sat,0,255);
col = color(hueval,150,255,sat*(toldist/80));
}
}
public void setPos(RPoint newpos){
lastpos = new RPoint(pos);
pos = newpos;
}
// Drawing the particle
public void draw(PGraphics gfx){
smooth();
//ReadListFile();
int Current_Second = second();
if(abs(Current_Second-Start_Second)>0.1)
{
```

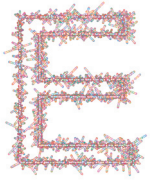
```
ReadListFile();
Start_Second = Current_Second;
}
if(Active_Color>0)
{
int index = (int) random(0, Active_Color);
int cindex = Mood_Trigger[index];
int r = (int)red(Color_Index[cindex]);
int g = (int)green(Color_Index[cindex]);
int b = (int)blue(Color_Index[cindex]);
stroke(r,g,b, Alpha);
}else
{
stroke(255,255,255, Alpha);
}
// Change the position and circumference of the
   ellipse, followed by the numeric color value for
   a black stroke
ellipse(lastpos.x/1.5,lastpos.y/1.5,6, 6);
}
}
void ReadListFile()
{
String lines[] = loadStrings("list.txt");
String off="Off";
String on="On";
Active_Color = 0;
for (int i=0; i < lines.length; i++)
{
if(lines[i].equals(on)==true)
{
Mood_Trigger[Active_Color]=i;
Active_Color++;
}else
{
//Mood_Trigger[i]=0;
}
}
}
```

Change the visual properties of the letter design (between the two previous examples on page 92) by altering the position and circumference values in the red highlighted code.

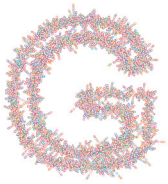
The authors would like to acknowledge Ricard Marxer Piñón for his free software (open source) application Caligraft written for the Processing Geomerative library. Code samples 1, 2, and 3 liberally uses this resource to create many of the dynamic typefaces in this publication. Each sample was recreated and written by Yeohyun Ahn to combine Processing code structures with the visual properties and language of graphic design and typography. For further information on Caligraft, please contact Ricard Marxer Piñón at www.ricardmarxer.com.

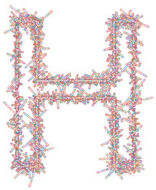
Yeohyun Ahn.
Viviana Cordova.





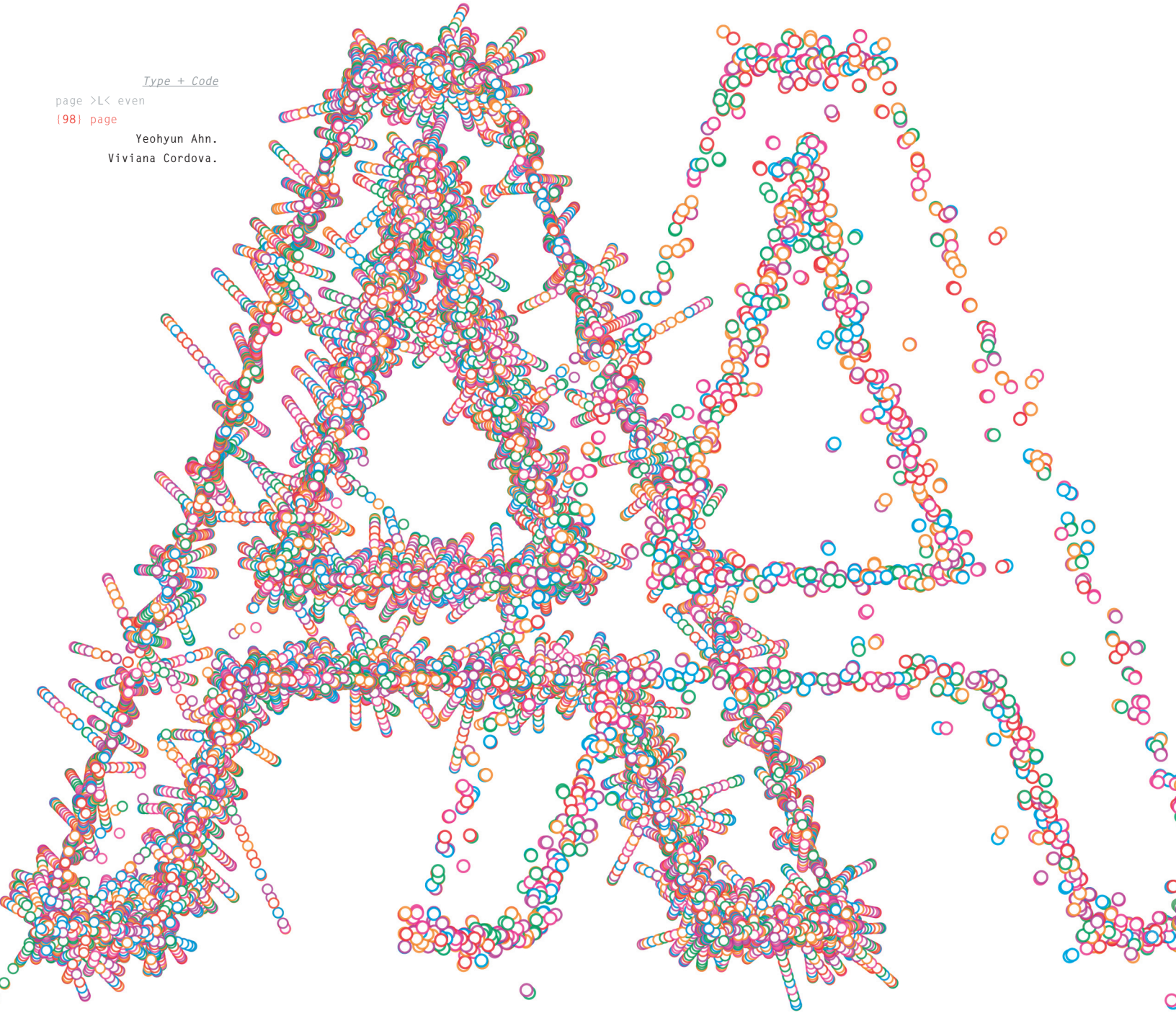
Points typeface *f*.

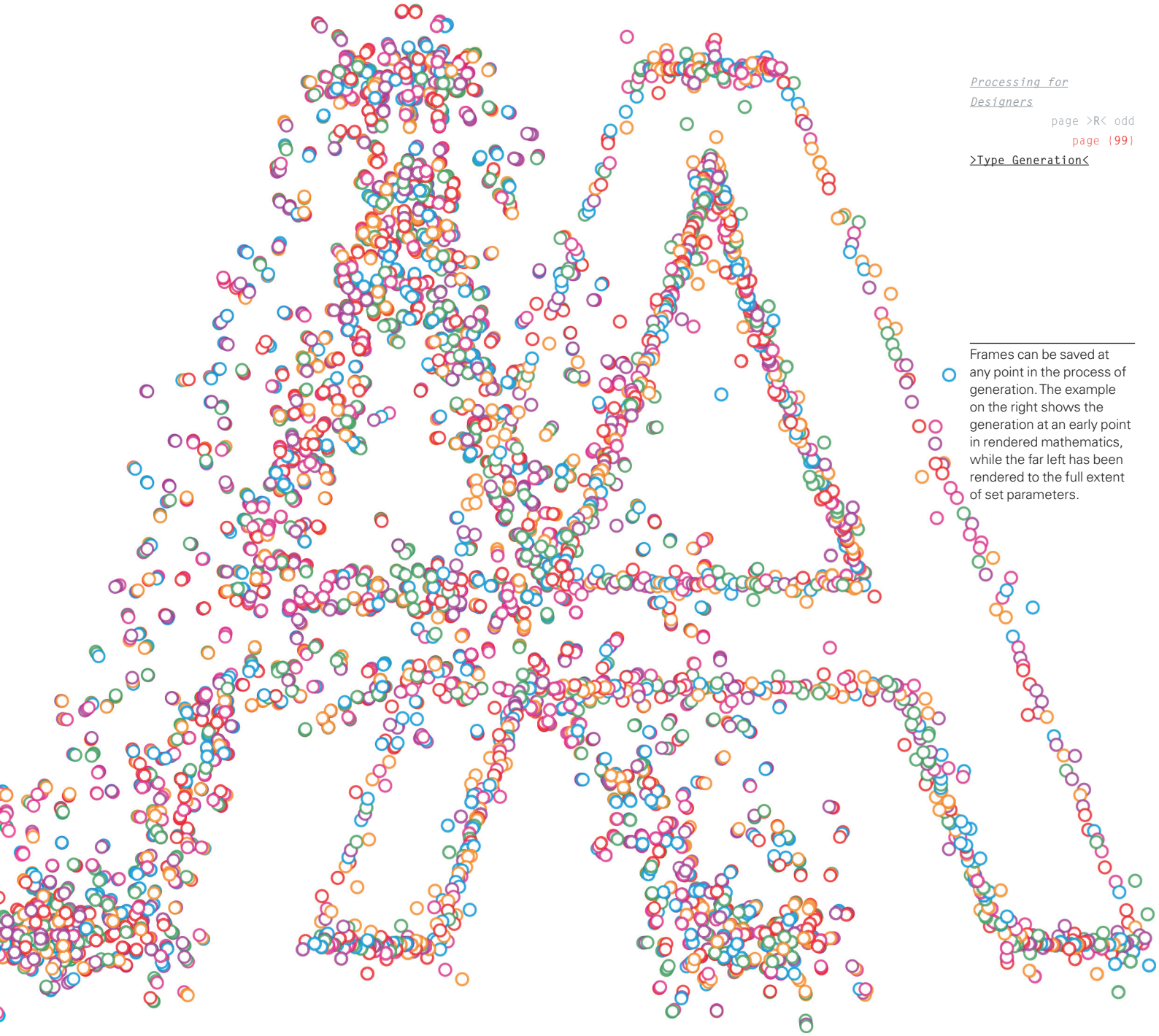





I
S
T
J
K
U
M
N
W
V
P
Y
X
R
O
Z

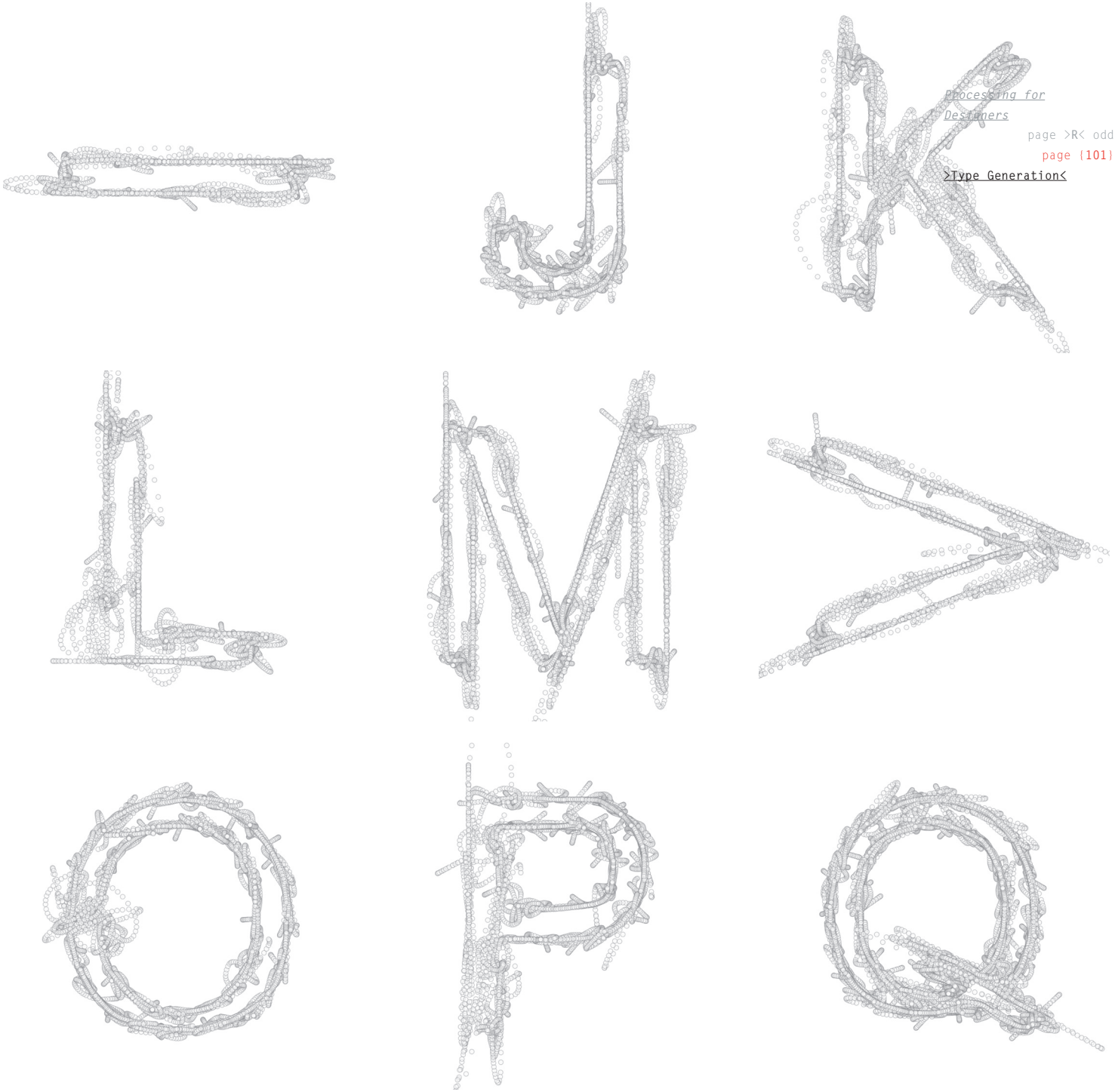
Yeohyun Ahn.
Viviana Cordova.





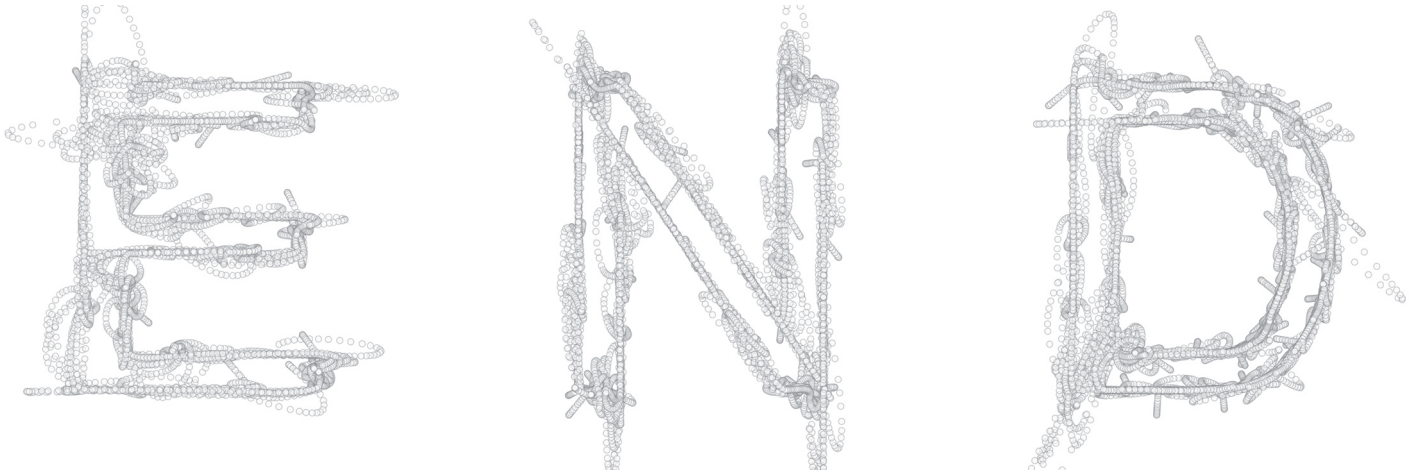
Frames can be saved at any point in the process of generation. The example on the right shows the generation at an early point in rendered mathematics, while the far left has been rendered to the full extent of set parameters.

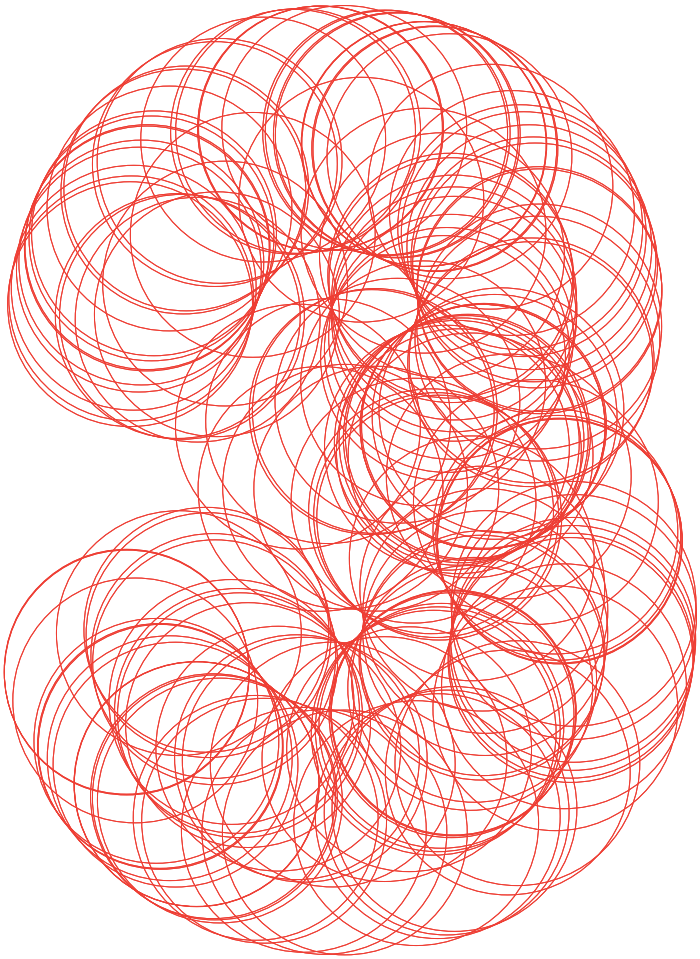
Yeohyun Ahn.
Viviana Cordova.

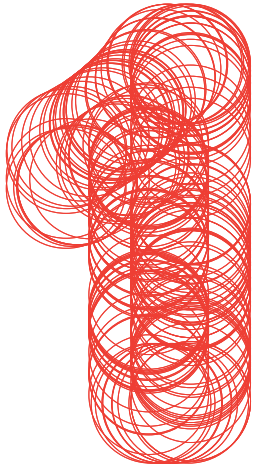


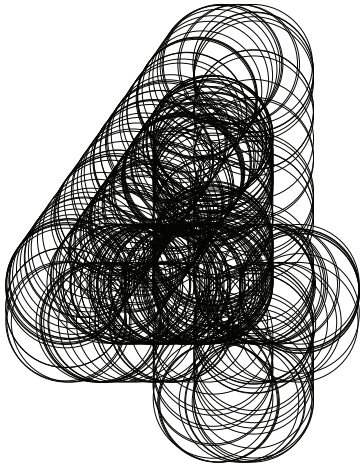
Yeohyun Ahn.
Viviana Cordova.

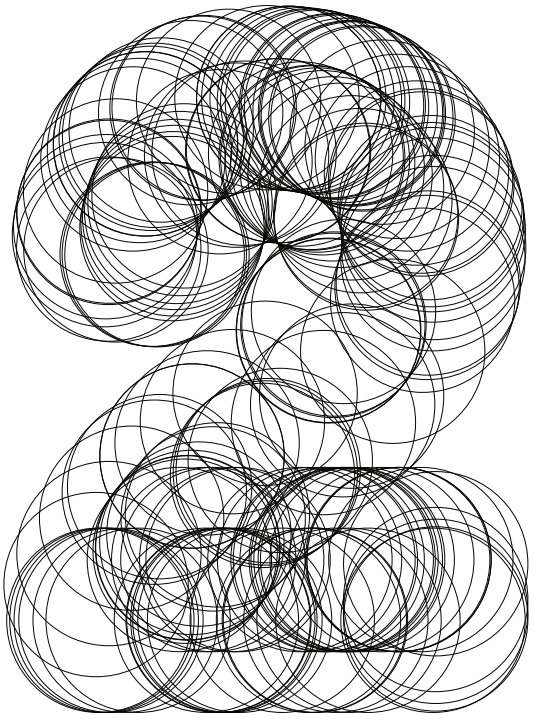
Chain typeface *f*.

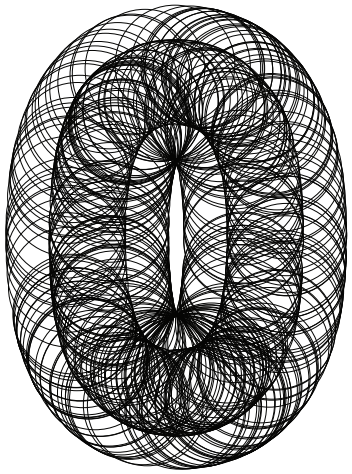

END











Slinky typeface *f*.