LA PRODUCCIÓN DE DISEÑO GRÁFICO A TRAVÉS DE LA PROGRAMACIÓN COMPUTACIONAL

GRAFIC DESIGN PRODUCTION THROUGH COMPUTER PROGRAMING

Hugo Solís García*

* Profesor-investigador de la Universidad Nacional Autónoma Metropolitana – Lerma, Departamento de Artes y Humanidades. Correo electrónico: h.solis@correo.ler.uam.mx.

En este ensayo se hace un ejercicio para acompañar al lector que no tenga experiencia en programación computacional en la creación de materiales visuales que le permitan entender la esencia de la programación y los posibles beneficios en aplicar el pensamiento computacional al diseño gráfico. El objetivo es explicar conceptos básicos de programación y aprovechar el entorno de programación de Processing para que el lector se sienta cómodo y pueda acceder de manera sencilla al campo de la programación. El ensayo presenta un contexto del porqué de este ejercicio, así como explicaciones graduales de conceptos que terminan en composiciones visuales que pueden ser utilizadas como punto de partida para la creación de materiales originales por parte del lector.

Palabras clave: Pensamiento computacional, diseño gráfico, programación.

On this paper, we develop an exercise for guiding the reader without previous computer programming training to create visual materials that could allow him or her to understand the basics of coding and also the benefits of applying a computational thinking the graphic design. The objective of the paper is to present basic concepts of computer science and take advantage of the Processing Framework in such a way that the reader could be comfortable and could access in an easy way to the field of coding. The essay presents a context about why the author wrote the document as well as gradual concepts that end up in a series of graphic compositions that could, eventually, be used as a starting point for creating original materials by the reader.

Keywords: Programing thinking, graphic design, coding

Introducción

Entre los años 2002 y 2004, realicé una maestría en Arte y Ciencia en el Media Laboratory del Massachusetts Institute of Technology (MIT). El Media Lab -como se le conoce coloquialmente- es un referente en el campo de la creatividad digital y está estructurado en grupos de trabajo con áreas de investigación específicas. Sin embargo, el paradigma de la interdisciplina permea el centro y existen espacios formales e informales para que los miembros de los grupos se relacionen y compartan inquietudes y proyectos. Uno de los espacios para interactuar entre grupos son los cursos que integran la maestría.

Yo formaba parte del grupo Opera of the Future, bajo la dirección de Tod Machover, cuya orientación era, en ese momento, la creación de interfaces musicales. Tenía yo una formación en el campo de la música experimental y algunas nociones de cómputo. Además, en esa época estaba interesado en las relaciones audiovisuales. Esta curiosidad me llevó a tomar el curso de Gráficos Computacionales con John Maeda, quien era en aquellos momentos el director del Grupo Aesthetics (mas) Computation Group. Recordemos que John estuvo muy relacionado con el grupo de Muriel Cooper Visible Language Workshop (VLW) y, posteriormente, desarrolló el programa Design by Numbers (Maeda, 2001). Este programa puede ser visto como un padre del muy difundido lenguaje Processing desarrollado por Ben Fry y Casey Reas en el grupo de Maeda (Reas y Fry, 2003).

Durante el curso trabajábamos con versiones de desarrollo de Processing, mientras Ben Fry mejoraba y terminaba la primera versión pública. Vivir de primera mano el proceso de creación de la herramienta de trabajo, mientras recibía la retroalimentación visual de John Maeda sin contar con una formación gráfica formal, pero con conocimientos básicos de programación, resultó ser una experiencia muy interesante para establecer relaciones entre el diseño gráfico y el código computacional. Por una parte, estaba interesado en hacer materiales visuales atractivos y, por otro lado, no

contaba con la técnica manual para lograrlo, pero tenía las bases computacionales para entender códigos generativos básicos.

Es dentro de este contexto que, como muchas personas de mi generación, aprendo a pensar computacionalmente con un imaginario visual. Si bien es cierto que la mayor parte de mi trabajo es sonoro, también es cierto que las metáforas y analogías visuales son de ayuda para entender las estructuras lógicas esenciales de la computación.

Este texto busca establecer un puente entre el mundo de la programación computacional y el pensamiento visual propio del diseño. En este sentido, la naturaleza de la programación con sus repeticiones y pensamiento formal genera una relación evidente con las estéticas minimalistas y formales derivadas de la Bauhaus. Con esto no pretendo especificar que el diseño computacional tenga una estética determinada pero sí, que el pensamiento formal del código comparte profundas similitudes con muchos de los preceptos que acompañaron dicha escuela y sus secuelas.

Contexto

Me gustaría detenerme a revisar el significado de la programación gráfica computacional creativa en el ámbito social y pedagógico. El uso de lenguajes computacionales para la gráfica artística tiene raíces en los experimentos de los años cuarenta. Sin embargo, dicho pensamiento de nicho estaba concentrado en centros especializados de creatividad computacional.

En México, por ejemplo, no hubo cursos formales de computación en las escuelas de ingeniería o de matemáticas hasta después de los años setenta. Estos cursos eran dados en el pizarrón sin siquiera contar con una computadora. En dicho contexto, la orientación pragmática del campo y la limitante de recursos hacían que la computación y, por tanto, el desarrollo de código computacional, estuviera orientado a la resolución de problemas de números puros en el campo de las matemáticas y a la creación de herramientas de automatización en el campo de las ingenierías. Contadas son las

excepciones en las que las computadoras fueron utilizadas para fines creativos o artísticos en México durante ese periodo.

Hasta hace no mucho tiempo (antes de los años ochenta), podíamos ver, por un lado a las escuelas de ingeniería y de matemáticas donde se enseñaba programación computacional con fines utilitarios o de investigación teórica y, por otro, a las escuelas de arte y diseño donde la mayor parte del proceso de aprendizaje se concentraba en la adquisición de una técnica manual.

Con la popularización de las computadoras y la creación de una industria de software, se desarrolló paquetería para el diseño. Cuando el presupuesto lo permitía, las instituciones adquirían paquetería que simplificaba y agilizaba el proceso de diseño. Sin embargo, el paradigma de pensamiento se mantuvo con pocos cambios, pues se sustituyó el restirador y los plumones por un monitor y un mouse, pero las nociones visuales y las técnicas de trabajo siguieron intactas. Una vez más la programación computacional se mantuvo fuera de la praxis del diseño. El diseñador digitalizó su herramienta, pero no modificó su paradigma de pensamiento. Se mantuvo la división entre el computólogo formado en la escuela de ingeniería o de matemáticas y el creativo visual formado en la escuela de arte que, ahora, utilizaba una computadora con paquetería especializada.

Tomó más tiempo para que, a finales de los noventa, se desdibujara, poco a poco, la frontera entre el programador y el diseñador gráfico. Por una parte, se consolidó la pedagogía en el campo de la computación y los estudiantes de dicha área encontraron salidas expresivas a sus conocimientos y, por otra parte, la adquisición de conocimientos en el campo de la programación se democratizó -tal vez gracias a internet- y, de esta manera, un estudiante de arte podía, si así lo quería, adquirir las bases para experimentar con sus materiales visuales. Finalmente, algunas instituciones se dieron cuenta del potencial creativo que tiene la programación computacional en el campo de la creación visual e iniciaron el proceso de introducir cursos relacionados con estas temáticas.

En este sentido, cabe mencionar que, probablemente al día de hoy, en México, la visión de muchas instituciones ha quedado corta. Muy pocas son las instituciones de enseñanza artística que contemplan la programación computacional como elemento integral de su plan de estudios y lo han manejado como un campo optativo o de especialización. Es probable que poco veamos un cambio en este paradigma y a medida que la programación computacional se vea como un elemento más dentro del proceso de enseñanza a nivel básico, apoyado por la creciente integración del modelo STEM (por sus siglas en inglés Science, technology, engineering, and mathematics) en diferentes niveles de enseñanza, tendremos una mayor integración del pensamiento computacional en el campo del diseño.

Pensamiento computacional

Será natural preguntarnos por qué es importante el pensamiento computacional en el campo del diseño y en el campo de las artes gráficas.

Por una parte, a nivel superficial es común y colectivamente aceptable asumir que el pensamiento computacional pertenece a la esfera del pensamiento duro y metodológico y que el diseño y las artes gráficas pertenecen al campo de la creatividad y, por tanto, se asocian con la libertad y la expresión intuitiva. Esta tajante división, producto de una visión histórica de las disciplinas, ha prevalecido por mucho tiempo en el imaginario colectivo y ha generado una división donde no existe. En las ciencias de la computación hay un pensamiento creativo constante y en el campo de las artes gráficas (y en las artes en general) hay más pensamiento riguroso del que se quiere, cotidianamente, aceptar.

Por otra parte, el pensamiento computacional es un lenguaje, una representación de la realidad similar al lenguaje natural o al pensamiento lógico-matemático. Compartiendo mucho de este último, el pensamiento computacional ayuda a clarificar ideas y a representar conceptos que serían difíciles de expresar de forma escrita. Cuantas

veces no nos ha quedado claro un concepto al ver un diagrama que era muy complicado entender únicamente con texto. De esta misma manera, hay manifestaciones del pensamiento que encajan naturalmente en el pensamiento estructurado que ofrece la computación. Pensar en código ayuda y nos obliga a observar las esencias, las similitudes, los detalles y las excepciones. Pensar en código nos ayuda a entender las estructuras subyacentes y a generar modelos. Como nos dice Olafur Eliasson, los modelos son en sí mismos realidades (Eliasson, 2007). Pero hay un potencial en los modelos, pues integran, decantan y condensan. Pensar en código ayuda a entender las generalidades de un sistema y entender un sistema nos ayuda a manipularlo y analizarlo. Tener mayor claridad de un sistema nos da como resultado mayor entendimiento y, con ello, se expanden nuestras posibilidades de juego y percepción.

Herramientas de lectura

Este documento hará constantes referencias a códigos computacionales. Se podría haber optado por presentarlo en un formato de seudocódigo para mantener neutralidad gramatical, sin embargo, los ejemplos serán códigos del lenguaje de Processing. Como ya se mencionó, es un lenguaje gráfico con una gramática concisa muy utilizado en las escuelas de arte digital y en las industrias creativas. Processing es gratuito y ha generado una comunidad de usuarios muy grande que comparte ejemplos y códigos (Reas y Fry, 2014).

El documento está dirigido, sobre todo, a lectores con nula o muy poca formación en el campo de la programación computacional. A los ya iniciados en dicha área, los ejemplos podrán parecerles simples. Existe también una comunidad que tiene conocimientos computacionales profundos, pero que no los ha relacionado con el pensamiento gráfico. Estos lectores también podrán sacar provecho de algunos de los ejemplos que aquí se presenten y pueden, si lo desean, saltarse las primeras partes básicas.

Se le invita al lector a realizar la lectura de este documento junto a una sesión abierta del lenguaje de Processing que le permita correr los materiales presentados. En este sentido, este texto es un experimento y un juego para vincular el ejercicio de escritura con el código computacional que resulta en materiales gráficos. Este texto no pretende enseñar a programar, pero tampoco asume que el lector ya sabe hacerlo. Sin embargo dialoga con el código computacional, lo que genera una problemática interesante. Tal vez lo mejor será ver este texto como una invitación a entender el pensamiento computacional desde una visión gráfica. Si el lector encuentra términos o definiciones que no comprende, es probable que sean conceptos o estructuras tradicionales de la programación y que se encuentren en cualquier libro de computación básica o en recursos digitales. Por lo anterior, se invita al lector a que realice la búsqueda de conceptos en fuentes secundarias a la par de la lectura.

Es probable que esta manera de integrar lenguaje natural en forma escrita, código computacional y representación gráfica, debiera plantearse desde un material audiovisual, tal vez, interactivo. Sin embargo, desarrollarlo de manera escrita implica sus propios retos y su propio paradigma de pensamiento.

De la hoja en blanco al primer primitivo

Processing es un lenguaje compilado, es decir, que antes de poder ejecutar el programa es necesario terminarlo todo. Una vez finalizado, el compilador lo revisa, y si todo está gramaticalmente correcto, lo ejecuta. Para simplificarle el inicio al programador Processing asume varios parámetros predeterminados que permiten ejecutar lo que se conoce como un Sketch. Si ejecutas el programa con la flecha superior izquierda, observarás que se abre una ventana de cien por cien pixeles con fondo gris. Este código en blanco que resulta en una ventana pequeña es tu temida hoja en blanco. A lo largo del texto la iremos complementando.

Como cualquier lenguaje de programación, Processing está construido con funciones que puedes utilizar para construir tu material. Una función actúa de manera similar al concepto del verbo en el lenguaje natural. Por ejemplo, si en español decimos corre, en código podríamos decir corre(), pero si en español decimos corre a 10 kilómetros por hora, en código podríamos decir corre(10). Lo que está entre paréntesis se conoce como parámetro y, en cierto sentido, puede ser pensado como un adverbio. Una función puede o no tener parámetros y puede tener uno o varios parámetros de entrada. Los parámetros de entrada deben ser del tipo requerido. En un momento veremos un poco más en relación con los tipos.

Todos los lenguajes de programación tienen funciones prefabricadas con las que podemos construir estructuras más complejas. De cierta forma, las funciones son similares a los bloques de lego que se van uniendo uno a otro para formar estructuras más complejas. Para ver la lista completa de funciones preexistentes en Processing, debemos revisar su referencia, a la cual se accede desde la pestaña help -> reference. Dicha lista se abre en un navegador y como podrás ver contiene una gran cantidad de elementos que constituyen el lenguaje. Una de ellas es size(). Y si revisamos la documentación veremos que requiere, necesariamente, de dos valores enteros (números sin decimales) que definen el tamaño de nuestro sketch o boceto. size(792, 612) definiría un boceto con las proporciones de una hoja carta en visión horizontal.

En la misma lista de referencia, nos vamos a encontrar con una sección de primitivos gráficos llamados *2d primitives* (que no debemos confundir con las variables primitivas tradicionales como *int*, *float*, *long*, *boolean*, etc.), en ella nos encontramos con las funciones line(), point() y rect(). Aunque en la página de referencia aparezcan con paréntesis en blanco, una vez que vemos ya su definición en su página individual de referencia, podemos ver que cada una de estas funciones recibe cierto número y tipo de parámetros. Por ejempo, point() espera ser utilizada con dos números flotantes (con decimales) de entrada quedando por ejemplo point(80.0,90.0).

Es importante enfatizar que antes de cualquier cosa los lenguajes de programación son, en cierta forma, calculadores, ya que tienen operadores matemáticos tradicionales como la suma, la resta, la división y la multiplicación. Por tanto, si nosotros quisiéramos dibujar un punto en la parte central de nuestra hoja en blanco, deberíamos o podríamos establecer la función point(792 / 2, 612 / 2).

Un último detalle a mencionar es que, al igual que las oraciones en el español, se marcan con un punto al final. En la mayoría de los lenguajes, las ideas individuales (digamos líneas) se deben terminar con un punto y coma. Siendo así tenemos nuestro punto de partida con el siguiente código completo que define un boceto y un primitivo gráfico en su centro:

```
size(792, 612);
point(792 / 2, 612 / 2);
```

Figura 1: Primer boceto con pixel al centro.

Presta mucha atención al mero centro de tu boceto porque encontrarás que hay un único pixel en color negro. Si el programa no abre el boceto y ves una lista de errores y una pestaña en rojo, probablemente, hay un error tan simple como una falta de una coma, un error de minúsculas o mayúsculas o la falta de un paréntesis o punto y coma. Hemos pasado de la hoja en blanco a un boceto con un primitivo gráfico.

De lo básico a lo simple

Otra función que se utiliza, comúnmente, es Background(). Es importante hacer hincapié en que los lenguajes de programación son sensibles a las mayúsculas y que, por tanto, no es lo mismo background() que Background(). La primera no funcionará. Esta función se encarga de establecer el color del fondo. Revisando nuevamente la documentación, podemos ver que esta función tiene varias versiones de uso, pudiendo recibir un, tres o cuatro parámetros. Ocupados solo de la versión de un parámetro, esta se debe interpretar como un tono de grises donde 0 es negro y 255 es blanco.

¿Por qué este rango? Porque contando el 0 esto nos da 256 valores y 256 valores son las posibilidades numéricas que nos da la manipulación de 8 dígitos binarios; 8 espacios binarios o bits constituyen un byte y el byte es la unidad tradicional de una palabra en los sistemas digitales. Esto quiere decir que tenemos una resolución de 1 byte de color para la gama de grises, por tanto, si incluimos Background(255) en nuestro código, cambiaremos el fondo predeterminado de gris a blanco.

Ahora bien, uno de los temas fundamentales en el pensamiento computacional es el beneficio que tienen los lenguajes para conceptualizar la repetición. Es aquí donde empezamos a ver el verdadero potencial de trabajar el diseño con un pensamiento computacional. Después de todo, pintar un punto en el centro de una hoja no es difícil en el mundo analógico. Pintar 200 puntos separados equidistantemente de manera perfecta requiere ya de cierta destreza. Todos los lenguajes computacionales tienen mecánicas de repetición.

El operador For() se utiliza para establecer las condiciones de las repeticiones. Es un operador ternario en el que cada elemento de la definición se divide por un punto y coma. El primer elemento establece la variable que será observada y, comúnmente, se define ahí mismo. El segundo elemento es una condición que, de ser cierta, permite una repetición del ciclo y el tercer elemento nos posibilita modificar el valor de la condición observada, por lo que, normalmente, se incrementa para obtener un contador.

En la sección anterior, estamos haciendo referencia a dos términos fundamentales de la computación de los cuales no hemos hablado: las variables y las condiciones. Una variable es un valor de cierto tipo que puede cambiar a lo largo de la vida de un programa. Para poder referirnos a ella es necesario que tenga un nombre único dentro de un contexto dado y en algunos lenguajes es necesario definirla para poder utilizarla. Además, en muchos lenguajes es necesario predeterminar los tipos de materiales con los que dicha variable trabajará pudiendo ser entre muchos otros, un número entero o un número con punto decimal, o un caracter de texto, etc. De esta manera int contador nos está diciendo que requerimos espacio en memoria para un número entero que estará etiquetado con el nombre contador. Una vez definido lo podemos utilizar y asignarle valores int contador = 0 etiqueta y da valor en un solo paso.

El otro punto importante tiene que ver con los operadores de comparación, los cuales devuelven o regresan —como se dice comúnmente— un resultado boleano de falso o verdadero; 5 > 6 devolverá falso, ya que 5 no es mayor que 6 y 5 > 3 devolverá verdadero, ya que 5 sí es mayor que 3. Los operadores de comparación (esto son > (mayor), >= (mayor o igual), < (menor), > (menor o igual), == (igual) y =! (diferente) se utilizan, frecuentemente, con contextos de operaciones lógicas que permiten diseñar bifurcaciones en los procedimientos. Esto quiere decir que se usan para que en función de la verdad o falsedad de la evaluación nuestros procedimientos tomen un camino u otro.

El hecho de poder bifurcar o modificar las rutinas de nuestros pensamientos es lo que permite que un lenguaje de programación modifique su trayectoria y, por tanto, la linealidad y la secuencia. En nuestro caso, se utiliza dentro del For para evaluar nuestro contador y, con ello, definir la cantidad de veces que se realizará el ciclo. De esta manera es que podemos construir el comando completo de la siguiente manera.

```
for(int contador = 0; contador < 100; contador++){
  //Realización de la rutina repetidamente
}
```

La línea anterior se debe interpretar o leer como: Para una variable de tipo entero etiquetada con el nombre contador e inicializada con el valor cero. Mientras que contador valga menos de 100, incrementarla uno a uno (para esto el operador ++). Luego se puede ver que abren y cierran unas llaves. Las llaves constituyen un contexto o cuerpo que es el que se realizará 100 veces. En cada una de estas repeticiones la variable tendrá un valor distinto empezando en cero e incrementándose hasta 99 (porque 100 no es menor 100). 0, 1, 2, 3, 4, ... hasta 99. Es importante saber que cualquier línea que empiece con // es un comentario que no es interpretado por el compilador y, por tanto, es solo un texto para ayudar a la comprensión humana.

Ahora bien, ya hemos visto que *point* es una función de gráficos primitivos que permite dibujar un pixel en nuestro boceto. Contamos con otros primitivos gráficos de 2 dimensiones como el rectángulo, el triángulo, la elipse (y, por tanto, el círculo) y la línea. De esta manera, si quisiéramos dibujar rectángulos horizontalmente el código completo podría ser:

```
size(792, 612);
background(255);
for(int contador = 0; contador < 79; contador++){
  rect(contador * 10,612 / 2 - 4,8,8);
}
```

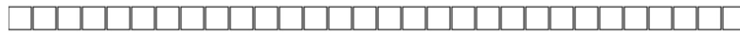

Figura 2: Línea de cuadros.

Nuestro código ha crecido un poco desde su primera versión, pero el resultado gráfico tiene ya un pequeño gesto visual minimalista. Su contenido se puede desglosar en: 1) establecimos el tamaño, 2) seleccionamos un fondo blanco, 3) establecemos un ciclo de 79 de elementos con el operador for() y 4) dentro del cuerpo de for() pintamos un rectángulo con la función rect(), la cual requiere de cuatro parámetros que se interpretan como coordenada x de la esquina superior izquierda, coordenada y de la esquina superior izquierda, ancho y alto. De esta manera es que pintamos 79 rectángulos de 8 * 8 pixeles cada uno, todos ellos en la horizontal central. Para ello tomamos el alto del boceto (612), lo dividimos entre dos y le restamos 4, que es la mitad del alto del rectángulo. Lo más interesante de este ejemplo es que utilizamos el valor de nuestra variable contador multiplicada por 10 para establecer la posición de cada uno de los rectángulos. Recordemos que la variable contador tendrá un valor distinto en cada ciclo. Este valor se incrementa en uno, pero lo multiplicamos por 10 para obtener posiciones de 0, 10, 20, 30 hasta 790. El ancho de nuestro boceto es de 792. Es importante

mencionar que estamos aprovechando de manera predeterminada las funciones gráficas primitivas, pintan un contorno negro de la figura con un fondo blanco.

Al igual que en el lenguaje natural, en el lenguaje computacional existen muchas maneras de decir una misma idea. Sin embargo, hay formas más concisas, menos redundantes y más claras para expresar una idea. En el pensamiento computacional hay un concepto de elegancia. El código anterior funciona y es entendible, pero no completamente elegante. Un código elegante, además de claro y conciso, nos permite modificarlo y aumentarlo con mayor sencillez. Aprovechando el uso de algunas variables predefinidas por *processing* (*width* y *height* adoptan los valores definidos en la función size() y con unas variables de apoyo podemos hacer una nueva versión más elegante.

```
size(792, 612);
background(255);
//valor para definir
int cantcuadros = 30, margenEntreCuadros = 2;
//variables intermedias
float brincoHorizontal = width / float(cantcuadros);
//variables intermedias
float tamCuadro = brincoHorizontal - margenEntreCuadros;
//ciclo
for(int contador = 0; contador < cantcuadros; contador++){
  rect(contador * brincoHorizontal, // posX
  height / 2 - (tamCuadro/2), //posY
  tamCuadro, tamCuadro); //size
}
```

Este código tiene la misma funcionalidad que el anterior, sin embargo, es más elegante, pues es más claro y ofrece posibilidad de modificación de manera sencilla. En este caso, el resultado visual puede cambiar definiendo únicamente los valores de cuántos cuadros se desean y cuál es el margen deseado entre ellos. El resto del código se adapta a la situación y no tenemos números coloquialmente llamados *duros* que tengamos que cambiar dentro del código. Incluimos comentarios dentro del código para clarificar y ordenamos los renglones para claridad visual.

En el código anterior, es importante entender que todos los valores son dependientes de la cantidad de cuadros deseados. Pudiera haber sido de otra manera y que el punto de partida fuera el tamaño de cada cuadro. Habría tantos cuadros como el tamaño del boceto lo permitiera. Decidimos la primera opción un tanto de manera arbitraria, pero nos permite jugar más con los resultados visuales.

De lo simple a lo minimalista

En este momento contamos con un código que permite la creación de una colección de rectángulos blancos. Nuestro código nos permite seleccionar la cantidad y margen entre ellos. Un siguiente experimento que podemos hacer es buscar resultados visuales más sofisticados aprovechando la simplicidad del código. Es decir, pensar en el diseño desde una perspectiva de economía de la gramática computacional. Una opción es, por ejemplo, el anidamiento de estructuras de ciclos. Si en un lenguaje computacional podemos pedir que algo se haga 100 veces, también podemos pedir que 100 veces se haga 100 veces una rutina sin mucho problema. Un ejemplo interesante del pensamiento computacional aplicado al diseño.

```
for(int contadorE = 0; contadorE < 10; contadorE++){
  println("*****");
  for(int contadorI = 0; contadorI < 10; contadorI++){
    println("contadorE = " + contadorE + "; contadorI = " +
contadorI);
  }
}
```

Por pura curiosidad, ejecuta el código anterior en otro archivo de Processing. En esta ocasión, no verás nada en el boceto que se abre, pero sí verás cómo van cambiando los valores en la consola inferior de la ventana de Processing. Esta es la consola y en ella es en la que el programa puede dar información importante. También nos sirve para ver valores de nuestro código con la función println() como en el ejemplo anterior. Si revisas con detenimiento, la línea con cinco asteriscos aparece 10 veces (pues está dentro del

primer ciclo y la línea con los nombres y valores de las variables contadorE y contadorI aparece un total de 100 veces. Esto aunque el ciclo interno se realiza 10 veces. Sin embargo, está dentro de otro ciclo que se realiza 10 veces; es decir, que 10 veces estamos pidiendo que 10 veces se haga la impresión de los valores. Lo anterior es un concepto poderoso con aplicaciones interesantes para el diseño. Veamos qué pasa si en lugar de únicamente imprimir valores en consola aprovechamos el recurso para dibujar primitivos geométricos. A continuación se presenta un código que servirá como punto de partida para varios experimentos visuales.

```
size(792, 612);
background(255);
int tamObjeto = 33, margenEntreObjetos = 8; //valor para definir
int margenMinX = 60, margenMinY = 160;
int paso = tamObjeto + margenEntreObjetos;
int canObjetosEnX = (width - margenMinX) / paso;
int canObjetosEnY = (height - margenMinY) / paso;
int margenFinalX = (width - (canObjetosEnX * paso)
      + margenEntreObjetos) / 2;
int margenFinalY = (height - (canObjetosEnY * paso)
      + margenEntreObjetos) / 2;
for(int contadorY = 0; contadorY < canObjetosEnY; contadorY++){
  for(int contadorX = 0; contadorX < canObjetosEnX;
contadorX++){
    rect(contadorX * paso + margenFinalX,
    contadorY * paso + margenFinalY,
    tamObjeto,tamObjeto);
  }
}
```

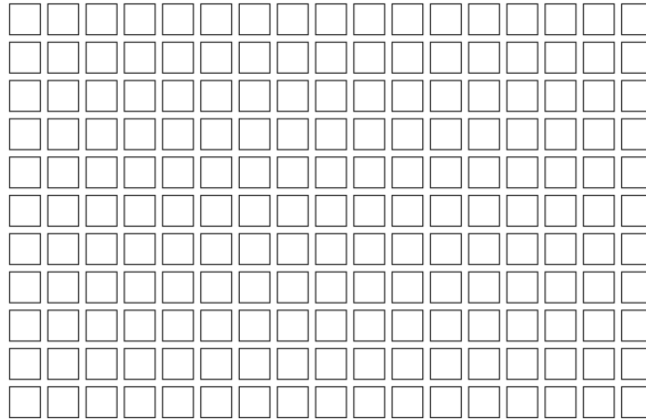

Figura 3: Cuadrícula.

Contar con nombres de variables que hagan referencia a su utilidad siempre es una guía de ayuda interna. En esta ocasión, además del tamaño de los objetos y el espacio entre ellos, estamos incluyendo dos variables para establecer los márgenes en la hoja. Por tanto, tenemos ahora variables intermedias para calcular la cantidad de objetos posibles en la horizontal y en la vertical. Tal vez la operación más complicada en este código es el cálculo de los márgenes finales denominados margenFinalX y margenFinalY. La dificultad radica en entender que el margen real es producto de restar el espacio que ocupan al total de las dimensiones del boceto tomando en cuenta que el tamaño de cada objeto debe tomar en cuenta su espacio con los otros objetos. De ahí que el tamaño para los cálculos sea con la variable paso. Dado que la diferencia entre la hoja y el espacio que ocupan los objetos se tiene que dividir entre el margen superior e inferior en el caso de la vertical y entre el izquierdo y derecho en el caso de la horizontal, este valor se tiene que dividir entre 2. Ya dentro del ciclo se agrega al espacio que se obtiene con la multiplicación de contadorX * paso, que recordaremos se incrementa en cada ciclo.

Como dijimos, esta matriz será el punto de partida para nuestros experimentos visuales, sin embargo, los rectángulos son únicamente una guía de poco interés visual.

Jugar con la gama de colores puede ser un atractivo punto de partida. Para ello, realizaremos algunos cambios mínimos. Será interesante comparar la versión de partida y esta nueva versión para comprender qué cambios mínimos pueden resultar en cambios significativos.

```
size(792, 612);
background(255);
noStroke();
int tamObjeto = 20, margenEntreObjetos = 5; //valor para definir
int margenMinX = 60, margenMinY = 160;
int paso = tamObjeto + margenEntreObjetos;
int canObjetosEnX = (width - margenMinX) / paso;
int canObjetosEnY = (height - margenMinY) / paso;
int margenFinalX = (width - (canObjetosEnX * paso)
    + margenEntreObjetos) / 2;
int margenFinalY = (height - (canObjetosEnY * paso)
    + margenEntreObjetos) / 2;
for(int contadorY = 0; contadorY < canObjetosEnY; contadorY++){
  for(int contadorX = 0; contadorX < canObjetosEnX;
contadorX++){
    fill(map(contadorX + contadorY, 0,
    canObjetosEnX + canObjetosEnY, 0, 255));
    rect(contadorX * paso + margenFinalX,
    contadorY * paso + margenFinalY,
    tamObjeto,tamObjeto);
  }
}
```

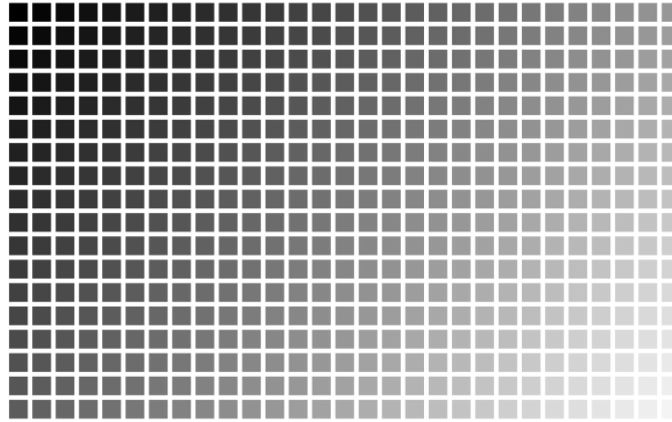

Figura 4: Gradiente.

La función noStroke() nos permite decir que desde que la función es llamada y hasta que no se solicite nuevamente un cambio en dicho estado, todas las funciones para dibujar gráficos 2D ya no tendrán el contorno negro predeterminado. El otro cambio está en agregar la función fill() que de recibir un único valor lo interpreta como un tono de grises entre 0 y 255. Nuevamente, este color se queda seleccionado hasta que se solicite un nuevo cambio de color (a esta mecánica se le conoce como *máquina de estados*). Nuestra función fill() tiene en su interior otra función:

```
map(contadorX + contadorY, 0, canObjetosEnX + canObjetosEnY, 0,
255)
```

Tener funciones dentro de funciones es muy común, se conoce como anidamiento. Podemos pensar el anidamiento de funciones como el uso de oraciones subordinadas en el lenguaje natural. En este caso, la función map() se utiliza para establecer el rango de la gama de colores. Esta función espera cinco parámetros. El primer parámetro es el valor a escalar entre dos rangos. El segundo y tercer parámetro establecen el mínimo y el máximo del rango posible de entrada, y el cuarto y quinto parámetro establecen el rango que se

desea de salida. En este caso, el valor de la suma de los dos contadores es la entrada y nuestro rango total es la cantidad total de objetos. El valor se debe escalar a nuestro rango de 0 a 255. El gesto resultante es pulcro y sencillo. Se observa una intención, pero el gradiente consecutivo en los tonos de gris da movimiento a la composición. La clave en este material está en relacionar el tono de cada rectángulo a su posición.

La variación como experimentación

Ya dijimos que pequeños cambios en el código puede tener resultados complejos. El lector con nula o muy poca experiencia se podría preguntar ¿cómo busco o logro cambios en un código? Aprender a programar es una labor de muy largo aliento. A final de cuenta, la programación está basada en un lenguaje y, como tal, tomó tiempo aprenderlo de la misma manera que toma tiempo aprender un idioma ajeno. No es suficiente con conocer la gramática. Tal vez es aquí donde radica la parte central del pensamiento computacional. El programador debe trasladar un deseo, en este caso visual, a una serie de estructuras definidas. Sin embargo, al igual que un niño pequeño aprende su idioma lo hace primero con palabras aisladas, frases cortas y, sobre todo, por medio de la repetición, la variación y la experimentación. La variación como técnica de aprendizaje tiene un valor, pues nos permite hacer comparaciones entre la variación y la copia y, con ello, encontrar las diferencias que permiten entender funcionalidades aisladas. El terreno digital nos permite explorar sin mucha dificultad, ya que no hay desperdicio analógico y el error no cuesta mucho. Por tanto, el terreno digital permite una constante experimentación. Modificar valores, agregar funciones, incorporar otras es una mecánica de aprendizaje en sí misma.

Hasta este punto de la lectura, los códigos que se han manejado ha crecido un poco. De una línea de código, tenemos ahora un puñado de líneas que producen un material visual. El lector podría estar tentado a ya no revisar el código y pensarlo como una imagen de libro. Un material secundario al cual regresará en otro momento. Nada

menos conveniente. Los códigos son la esencia del texto y el texto escrito es un reforzamiento para su entendimiento. Veamos, ahora, una serie de variaciones donde en cada una jugaremos con algún concepto pendiente.

Rompiendo la regla

En el siguiente código, usamos, por primera vez, una estructura de control y un operador lógico; if() es una palabra reservada que se utiliza para establecer condiciones. Por su naturaleza se utiliza, normalmente, en complemento con los operadores de comparación para establecer casos. También aparece el operador lógico && que se utiliza para concatenar unidades lógicas. En el caso de && (AND), solo cuando las dos unidades son verdaderas, el resultado es verdadero. En nuestro contexto hemos definido las variables ausenteX y ausenteY como una quinta parte del total de las cantidades en las filas y las columnas, respectivamente. Esto determinará un punto en la región superior izquierda de nuestro boceto sin importar la cantidad de objetos determinada.

```
size(792, 612);
background(255);
noStroke();
int tamObjeto = 20, margenEntreObjetos = 5; //valor para definir
int margenMinX = 60, margenMinY = 160;
int paso = tamObjeto + margenEntreObjetos;
int canObjetosEnX = (width - margenMinX) / paso;
int canObjetosEnY = (height - margenMinY) / paso;
int margenFinalX = (width - (canObjetosEnX * paso)
    + margenEntreObjetos) / 2;
int margenFinalY = (height - (canObjetosEnY * paso)
    + margenEntreObjetos) / 2;
int ausenteX = canObjetosEnX / 5, ausenteY = canObjetosEnY / 5;
for(int contadorY = 0; contadorY < canObjetosEnY; contadorY++){
  for(int contadorX = 0; contadorX < canObjetosEnX;
contadorX++){
    fill(map(contadorX + contadorY,0,
    canObjetosEnX + canObjetosEnY,0,255));
    if(contadorX == ausenteX && contadorY == ausenteY)
      fill(255,0,0);
    rect(contadorX * paso + margenFinalX,
```

```
    contadorY * paso + margenFinalY,
    tamObjeto,tamObjeto);
  }
}
```

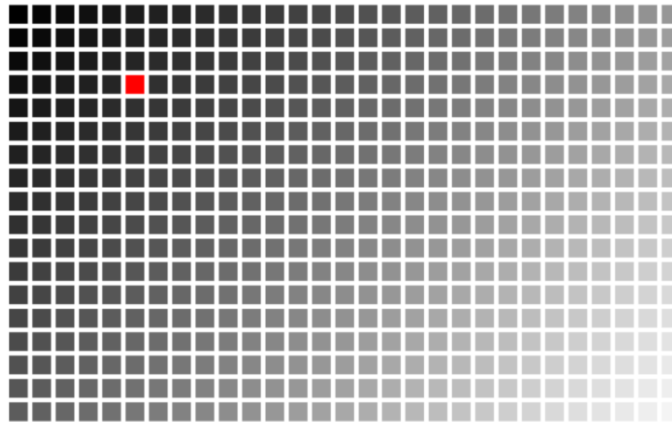

Figura 5: Rompiendo la regla.

Como vimos con anterioridad, en cada ciclo dentro del for anidado establecemos un tono de gris con la función fill(). Ahora insertamos una situación específica con la línea:

```
if(contadorX == ausenteX && contadorY == ausenteY)
fill(255,0,0);
```

Con esta línea establecemos que cuando el valor de la variable contadorX sea igual que el valor establecido en la variable ausenteX y (esto es muy importante el *Y*) cuando contadorY sea igual que el valor de la variable ausenteY, entonces utilizaremos el color rojo. Es interesante hacer notar que varias de las funciones de color en Processing se interpretan como tono de grises si ha{y un parámetro o como valores de rojo, verde y azul cuando tenemos tres parámetros. En este caso estamos asignado el rojo intenso con

fill(255,0,0). Absoluto rojo, cero verde y cero azul. Un punto muy importante a resaltar es que es muy común que funciones como for, if, entre muchas otras, tengan un *cuerpo* delimitado por llaves ({ }). Lo anterior es lo más común, pero dado que solo tenemos una línea de código que cambia si nuestro if es verdadero, entonces lo omitimos conscientemente.

Esta nueva versión tiene modificaciones muy simples que cambian completamente la composición visual. La integridad visual que se obtenía con el gradiente completo se convierte ahora en un marco y contorno para un elemento resaltado y, por lo tanto, sujeto de la visión. Nuestra mirada se centra en la notoria excepción en rojo y, por ello, el resto de la composición se convierte en textura secundaria, mas no por eso menos importante, pues sin ella el rectángulo rojo carecería de contexto.

Dinamismo controlado

Generar materiales minimalistas y precisos es muy ideosincrático dentro de los sistemas digitales, pues los ciclos y las repeticiones precisas son funciones básicas dentro de los lenguajes de programación. ¿Qué sucede si nos interesa agregar una mecánica de variación o dinamismo controlado? ¿Cómo podemos emular las imprecisiones propias del mundo analógico? Una mecánica es la aleatoriedad controlada, la cual nos permite agregar de manera azarosa cierto valor o cierto rango a una variable particular. La función random() genera un número seudoaleatorio con números decimales (flotante). Tiene dos versiones. Con un parámetro genera un número entre cero y el parámetro; con dos genera un número entre los dos. Utilizamos dicha función para *alimentar* varias funciones. strokeWeight() se utiliza para definir el grueso de los contornos de las figuras y espera un número flotante que define los pixeles del grueso. stroke() define el color y en nuestro caso lo alimentamos con un número aleatorio entre 10 (casi negro) y 200. No utilizamos el rango completo hasta 255, ya que no queremos que se hagan contornos muy blancos. A las variables desviacionX, desviacionY y desviacionSize se les asigna

también, a cada una de ellas, un número aleatorio entre el negativo de un cuarto del margen entre objetos y el positivo de un cuarto del margen entre objetos. Estos valores se suman en el momento de definir el tamaño de cada rectángulo y su posición.

```
size(792, 612);
background(255);
int tamObjeto = 20, margenEntreObjetos = 5; //valor para definir
int margenMinX = 60, margenMinY = 160;
int paso = tamObjeto + margenEntreObjetos;
int canObjetosEnX = (width - margenMinX) / paso;
int canObjetosEnY = (height - margenMinY) / paso;
int margenFinalX = (width - (canObjetosEnX * paso)
    + margenEntreObjetos) / 2;
int margenFinalY = (height - (canObjetosEnY * paso)
    + margenEntreObjetos) / 2;
for(int contadorY = 0; contadorY < canObjetosEnY; contadorY++){
  for(int contadorX = 0; contadorX < canObjetosEnX;
contadorX++){
    strokeWeight(random(1.0, 2.0));
    stroke(random(10, 200));
    float desviacionX = random(margenEntreObjetos * -0.25,
    margenEntreObjetos * 0.25);
    float desviacionY = random(margenEntreObjetos * -0.25,
    margenEntreObjetos * 0.25);
    float desviacionSize = random(margenEntreObjetos * -0.25,
    margenEntreObjetos * 0.25);
    rect(contadorX * paso + margenFinalX + desviacionX,
    contadorY * paso + margenFinalY + desviacionY,
    tamObjeto + desviacionSize,tamObjeto + desviacionSize);
  }
}
```

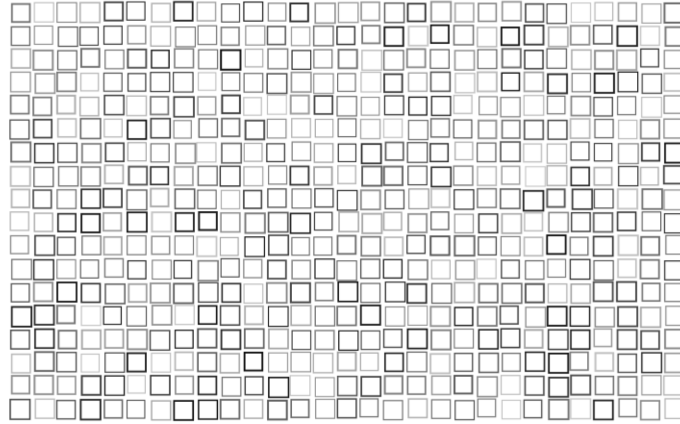

Figura 6: Dinamismo controlado.

El resultado es un conjunto de rectángulos a manera de pequeños azulejos. Se conserva el minimalismo de la matriz, pero se obtiene un toque orgánico y natural. La diferencia en los gruesos de las líneas y en el tono de negro de cada rectángulo da vida y movimiento a esta versión. En el siguiente código se agrega un poco de rotación aleatoria a cada cuadro y se cambia el tamaño y margen de los objetos. Debido a la manera en que Processing —y muchos programas gráficos— calculan internamente las operaciones gráficas, la rotación se logra mediante una agregación de transformaciones y, por ello, en esta ocasión, será necesario utilizar funciones como pushMatrix() y popMatrix() para aislar cada una de las rotaciones. Imaginemos que deseamos rotar una imagen en el mundo analógico. Para ello necesitamos un pivote de referencia. La función traslate() se utiliza para esto de forma tal que ahora la posición del rectángulo está supeditada a esta función. Una vez posicionados en el lugar adecuado, la función rotate() establece en radianes la rotación. Dado que los radianes son una escala poco familiar, utilizamos la función radians() para convertir de grados a radianes. Como se puede ver se escogió una posible rotación aleatoria entre −5 y 5.

Cabe la pena detenernos un momento en este último aspecto. ¿Por qué estos y no otros valores? Como en muchos de los valores presentes en el código, se definen bajo una aproximación. Se ejecuta el código y se evalúa el resultado. Si los valores son bajos o se exceden, según el criterio del diseñador-programador se ajustan de manera iterativa. Es aquí donde el entrenamiento visual es importante. La experiencia del diseñador y su formación tradicional son importantes. Uno de los problemas más notorios en el mundo digital es la falta de sutileza. Dado que el esfuerzo para poner una línea o diez mil líneas es similar, se pierde la dimensión de los gestos sutiles. En el mundo analógico dibujar diez mil líneas es diez mil veces más esfuerzo que una línea y, por ello, el diseñador se plantea cuidadosamente sus recursos y, con ello, sus rangos expresivos. Hacer conscientes los rangos y las capacidades humanas nos ayuda a trasladar el pensamiento analógico al mundo digital y, así, establecer estéticas coherentes. A continuación, el código de la versión con rotaciones:

```
size(792, 612);
background(255);
int tamObjeto = 40, margenEntreObjetos = 10; //valor para
definir
int margenMinX = 60, margenMinY = 160;
int paso = tamObjeto + margenEntreObjetos;
int canObjetosEnX = (width - margenMinX) / paso;
int canObjetosEnY = (height - margenMinY) / paso;
int margenFinalX = (width - (canObjetosEnX * paso)
    + margenEntreObjetos) / 2;
int margenFinalY = (height - (canObjetosEnY * paso)
    + margenEntreObjetos) / 2;
for(int contadorY = 0; contadorY < canObjetosEnY; contadorY++){
  for(int contadorX = 0; contadorX < canObjetosEnX;
contadorX++){
    strokeWeight(random(1.0, 2.0));
    stroke(random(10, 200));
    float desviacionX = random(margenEntreObjetos * -0.25,
    margenEntreObjetos * 0.25);
    float desviacionY = random(margenEntreObjetos * -0.25,
    margenEntreObjetos * 0.25);
    float desviacionSize = random(margenEntreObjetos * -0.25,
    margenEntreObjetos * 0.25);
```

```
    float rotation = random(radians(-5), radians(5));
    pushMatrix();
    translate(contadorX * paso + margenFinalX,
    contadorY * paso + margenFinalY);
    rotate(rotation);
    rect(desviacionX,desviacionY,
    tamObjeto + desviacionSize,tamObjeto + desviacionSize);
    popMatrix();
  }
}
```

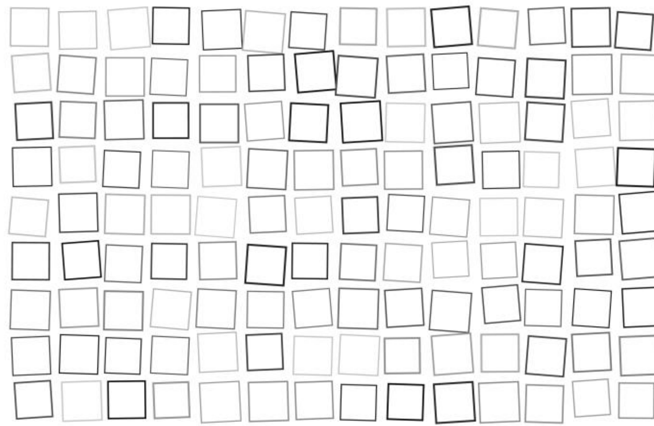

Figura 7: Rotaciones.

Homenaje a la Bauhaus

Como pequeño homenaje a la Bauhaus presentamos, a manera de cierre, la siguiente versión de nuestro boceto:

```
size(792, 612);
background(255);
noStroke();
ellipseMode(CORNER);
int tamObjeto = 40, margenEntreObjetos = 10; //valor para
definir
int margenMinX = 60, margenMinY = 160;
int paso = tamObjeto + margenEntreObjetos;
int canObjetosEnX = (width - margenMinX) / paso;
```

```
int canObjetosEnY = (height - margenMinY) / paso;
int margenFinalX = (width - (canObjetosEnX * paso)
    + margenEntreObjetos) / 2;
int margenFinalY = (height - (canObjetosEnY * paso)
    + margenEntreObjetos) / 2;
int contador = 0;
for(int contadorY = 0; contadorY < canObjetosEnY; contadorY++){
  for(int contadorX = 0; contadorX < canObjetosEnX;
contadorX++){
    int tresColores = int(random(3));
    if(tresColores == 0) fill(255,0,0,210);
    if(tresColores == 1) fill(0,0,255,180);
    if(tresColores == 2) fill(255,255,0,255);
    if(contador % 3 == 0)
      rect(contadorX * paso + margenFinalX,
      contadorY * paso + margenFinalY,
      tamObjeto,tamObjeto);
    if(contador % 3 == 1)
      ellipse(contadorX * paso + margenFinalX,
      contadorY * paso + margenFinalY,
      tamObjeto,tamObjeto);
    if(contador % 3 == 2)
      triangle((contadorX * paso + margenFinalX) + tamObjeto/2,
      contadorY * paso + margenFinalY,
      (contadorX * paso + margenFinalX) + tamObjeto,
      (contadorY * paso + margenFinalY) + tamObjeto,
      contadorX * paso + margenFinalX,
      (contadorY * paso + margenFinalY) + tamObjeto);
      contador++;
  }
}
```

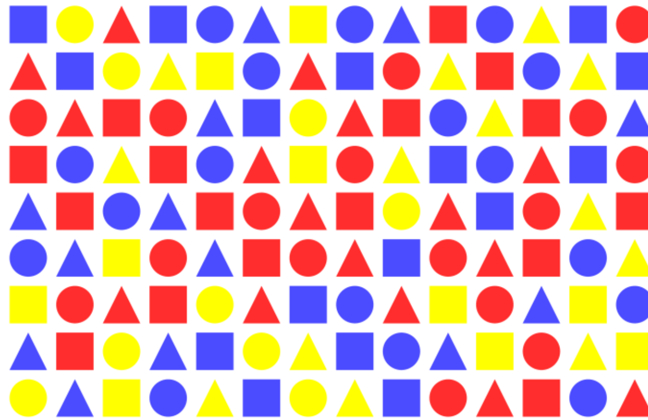

Figura 8: Homenaje a la Bauhaus.

En esta última versión aparecen algunas novedades interesantes. Por una parte, encontramos la función ellipseMode(CORNER), la cual establece las manera en la que serán interpretados los parámetros de posición cuando se llama la función ellipse(). Hacemos uso de dicha función para homologar el sistema de coordenadas con respecto a la ya utilizada función de rect(). Por su parte, las funciones ellipse() y triangle(), de manera similar a rect(), se encargan de dibujar primitivos de formas geométricas diferentes.

Lo interesante en esta versión es el dinamismo que se logra jugando, por un lado, con la secuencia ordenada de figuras, mientras que se determina el color aleatorio. Si observamos detenidamente, la secuencia de figuras es siempre la misma: cuadrado, círculo, triángulo, cuadrado, círculo, triángulo, etc. Sin embargo, con la cantidad de 14 figuras que se logra estableciendo el tamaño de 40 y el margen de 10, se genera un desfase natural en los inicios de figura de forma tal, que de manera vertical también se obtiene la secuencia: cuadrado, círculo, triángulo, cuadrado, círculo, triángulo, etc. Esto cambiará si se altera el tamaño de las figuras, pero aprovechamos la situación para

generar variación. La secuencia se obtiene a través de definir un contador general que se incrementa en uno en cada nuevo ciclo de figura. Se tienen tres if cada uno comparando el contador módulo 3 contra cero, uno y dos. Recordemos que la operación de módulo es el residuo de la división, por lo que el módulo 3 de un contador que se incrementa en uno nos dará la secuencia 0,1,2,0,1,2. etc. De esta manera, podemos generar la secuencia cuadrado, círculo, triángulo, cuadrado, círculo, triángulo, etc.

En otra parte del código definimos una variable llamada tresColores, la cual igualamos en cada ciclo un valor entero aleatorio 0, 1 o 2. Recordemos que la función random() nos regresa flotantes, pero truncamos el flotante con la función int(). Posteriormente, utilizamos una serie de 3 if para seleccionar uno de los posibles colores: rojo, verde o amarillo. Utilizamos la versión de cuatro parámetros en la función fill() para poder definir el *alpha* (transparencia) de los colores y, con ello, poder equilibrar el brillo de lo composición.

Como se puede apreciar, si revisamos los últimos códigos, las variaciones no son muchas entre ellos y la lógica de desarrollo es similar en todos ellos, aunque los resultados gráficos sean diversos. Esta parte de la potencialidad del pensamiento computacional aplicado al diseño. Aunque podríamos continuar haciendo variaciones y jugando con diferentes versiones, los ejemplos anteriores son suficientes para dar una idea general e introductoria del pensamiento computacional. Personas interesadas pueden acceder a libros especializados como lo es el libro Generative Design: Visualize, Program, and Create with Processing (Bohnacker, 2012).

Conclusiones

A lo largo del documento, hemos visto la creación, paso a paso, de una colección de bocetos gráficos utilizando el lenguaje de programación Processing relacionando el pensamiento computacional con conceptos básicos del diseño, en particular, del diseño minimalista inspirado en los conceptos de la Bauhaus. Aunque el objetivo es acercar e

invitar a los lectores que no tienen formación computacional, es claro que este no puede ser el único recurso y, que de ser de su interés, el lector deberá profundizar aprovechando los diversos medios que existen para el aprendizaje de la computación. Aun así, realizar un trabajo escrito que abra las puertas a los diseñadores tiene un valor introductorio particular.

Después de hacer un recorrido desde un boceto con un solo punto hasta un diseño interesante y atractivo, podemos ver que aprovechar el pensamiento computacional en el diseño gráfico tiene un valor y un potencial importante. La programación computacional no suplanta el entrenamiento visual y la sensibilidad que se desarrolla con la preparación constante, sin embargo, el pensamiento computacional puede potenciar y abrir caminos diferentes para los diseñadores, además de que en ciertas circunstancias pragmáticas puede hacer muy eficiente o preciso el trabajo.

Bibliografía

H. Bohnacker y col. Generative Design: Visualize, Program, and Create with Processing. Princeton Architectural Press, 2012. isbn: 9781616890773. url: https://books.google.com.mx/books?id=tSS9uAAACAAJ.

O. Eliasson y col. Take Your Time: Olafur Eliasson. San Francisco Museum of Modern Art, 2007. isbn: 9780500093405. url: https://books.google.com.mx/books?id=grdlUpw9i9cC.

J. Maeda. Design by Numbers. Mit Press. MIT Press, 2001. isbn: 9780262632447. url: https://books.google.com.mx/books?id=cptXSf5kS_IC.

Casey Reas y Ben Fry. Processing; A Programming Handbook for Visual Designers and Artists. 2nd. MIT Press, 2014.

Casey Reas y Benjamin Fry. "Processing: a learning environment for creating interactive Web graphics". En: In Proceedings of the SIGGRAPH 2003 conference on Web graphics. 2003.