interactival

Código creativo con p5.js

Aplicaciones interactivas simplificadas

por Sebastián Acevedo Álvarez

interactival

Título del libro: Código creativo con p5.js
Subtítulo: Aplicaciones interactivas simplificadas
Autor: Sebastián Acevedo Álvarez

Este libro puede descargarlo en: https://www.interactival.cl/, el formato digital de este libro puede compartirse libremente, pero no está permitido realizar copias físicas, es decir, no puede imprimirlo. Los códigos que aquí se incluyen puede utilizarlos donde estime conveniente, pero se solicita dar los créditos que correspondan.

Existen dos opciones donde acceder a los códigos que se presentan en este libro:

- https://www.interactival.cl/librop5js
- https://github.com/sbsacev/cursop5js

Se ha puesto un gran esfuerzo en asegurar que los códigos que aquí se presentan son funcionales y se ejecutan correctamente. De todas maneras, el autor de este libro niega cualquier responsabilidad por los daños del uso que pudiera eventualmente causar el contenido que se presenta en este libro. El cómo se utiliza la información presente en este libro y las consecuencias que pudiera causar se realiza bajo el riesgo del lector.

La biblioteca p5.js fue creada por Lauren Lee McCarthy y este proyecto está actualmente dirigido por Qianqian Ye. La biblioteca p5.js se considera que es software libre, se puede redistribuir y/o modificar según los términos de la GNU Lesser General Public License publicada por la Free Software Foundation, versión 2.1.

Registro de propiedad intelectual: 2024-A-253

ISBN PDF: 978-956-416-822-7

Gracias a Lorena Díaz por sus comentarios y sugerencias que siempre son bien recibidas, a Allison Michea por sus valiosos comentarios y las preguntas realizadas y a Fausto Espinoza por su buena disposición y comentarios realizados.

Si usted encontró algún error en este libro, o si quiere realizar algún comentario o sugerencia puede comunicarse con nosotros en:

➔ **Correo electrónico:** info@interactival.cl
➔ **Instagram:** @interactival.cl

Valdivia, Chile, 2024

Este libro está dirigido y pensado para que sea beneficioso para una gran cantidad de personas y se enfocará en una biblioteca de JavaScript (llamada **p5.js**). Con la intención de ser concreto con la información que se entrega, este libro no abarca todas las posibilidades que se puedan crear y programar con la biblioteca, sino que incluye lo que se considera más relevante para comenzar a programar con esta herramienta, de otra forma sería un libro extremadamente largo y podría desincentivar el aprendizaje. En conjunto con este libro, considere que **p5.js** tiene la ventaja de contar con una comunidad en línea a nivel mundial que comparte libremente, y muchas veces de forma desinteresada, el trabajo que desarrollan.

Este libro puede resultar útil para entusiastas y principiantes en la programación que estén dispuestos a adquirir una habilidad nueva. Será útil para estudiantes secundarios que deseen profundizar sus conocimientos del área científica y que quieran modelar o simular distintas áreas del conocimiento o que deseen realizar un proyecto de investigación dentro de sus actividades curriculares. También se espera que sea provechoso para estudiantes de educación superior que deseen generar resultados visuales y realizar aplicaciones interactivas reforzando los conocimientos que están adquiriendo en su proceso de formación profesional o también para realizar algún proyecto que estén desarrollando.

Se considera que este libro será importante para profesores de ciencias que requieran mostrar algún fenómeno a través de una simulación, o de un programa, que sea de utilidad para sus estudiantes y que puedan crear ellos mismos, así como para profesores de arte que deseen crear recursos visuales a partir de colores, geometrías, imágenes y sonidos que permitan a sus estudiantes indagar formas de expresión artística digital.

Finalmente, se cree que podría resultar de gran utilidad para artistas que quieran experimentar con formas distintas de expresión, para la manipulación de colores y geometrías, o que deseen indagar en el arte generativo o en otras formas de arte digital.

Este es un libro directo e introductorio, para llegar y hacer o para llegar y experimentar, donde la información se entregará de forma concreta, y sin un lenguaje demasiado técnico. Cuando esté escribiendo código, para crear una aplicación, podrá visualizar sus resultados de forma inmediata, sin necesidad de software de pago. Podrá probar su código, muchas veces no resultará lo que desea lograr, obtendrá un resultado inesperado o el programa que está ejecutando puede dejar de funcionar en algún momento, pero no hay problema, puede corregir y volver a intentarlo y cuando logre un resultado esperado, sentirá la satisfacción de haberlo creado y podrá decir que ha superado las dificultades a las que se enfrentó.

De forma general, es recomendable que se tengan conocimientos de geometría y álgebra básica para que el proceso de aprendizaje con este libro sea más provechoso. Sí es necesario contar con conocimientos de aritmética básica.

Al final de cada capítulo habrá una serie de propuestas de código que puede generar, por lo que se invita a tratar de desarrollarlas utilizando las herramientas vistas hasta el capítulo respectivo. Si ninguna de esas tareas propuestas le llama la atención, puede crear las que estime conveniente de acuerdo a sus necesidades y preferencias. Es probable que en algunas de esas “tareas” necesite información adicional relacionada a la geometría, matemáticas, física u otras disciplinas. Por lo que es posible que necesite tomarse algún tiempo para estudiar, en alguna medida, esos tópicos para llevar a cabo los objetivos planteados.

Para comenzar solo necesitará un computador con conexión a internet y el deseo de aprender una forma distinta de hacer las cosas. Las posibilidades son infinitas, el límite es su imaginación.

Sebastián Acevedo Álvarez

Contenidos

Contenidos....... 4
Capítulo 0. Introducción....... 6
Resumen....... 12
Capítulo 1. Variables, geometrías y funciones principales....... 13
1.1. Variables....... 13
1.2. Dibujo de geometrías sencillas....... 15
1.3. Geometrías más complejas....... 24
1.4. Mostrando texto en pantalla....... 25
1.5. Condición if....... 29
1.6. La función map....... 32
1.7. Ciclo for....... 34
1.8. Interactividad con el mouse....... 37
1.9. Interactividad con el teclado....... 42
1.10. Tabla de funciones matemáticas....... 47
1.11. Otra forma de ciclo....... 51
1.12. Geometrías en 3D....... 51
1.13. El tiempo en p5.js....... 54
1.14. Animaciones....... 57
Resumen....... 60
Capítulo 2. Vectores....... 62
2.1. Operaciones con vectores....... 63
- Suma:....... 63
- Resta:....... 64
- Multiplicación:....... 65
- División:....... 66
- Producto punto:....... 68
- Producto cruz:....... 69
- Magnitud:....... 70
- Normalizar:....... 70
- Ángulo entre vectores:....... 71
Resumen....... 72
Capítulo 3. Arreglos, imágenes, audio y video....... 74
3.1. Arreglos....... 74
3.2. Imágenes....... 78
3.3. Archivos txt....... 85

3.4. Sonidos.......... 89
3.5. Análisis de frecuencia.......... 91
3.6. Accediendo a la cámara.......... 95
3.7. DOM.......... 96
Resumen.......... 102
Capítulo 4. Funciones y objetos.......... 104
4.1. Funciones.......... 104
4.2. Objetos.......... 107
Resumen.......... 114
Capítulo 5. Manipulando píxeles.......... 115
Resumen.......... 121
Capítulo 6: Aleatoriedad y Perlin noise.......... 122
Resumen.......... 128
Capítulo 7. Ejemplos de aplicaciones.......... 130
7.1. Movimiento Parabólico.......... 130
7.2. Funciones y derivadas.......... 132
7.3. Onda viajera.......... 136
7.4. Coordenadas polares.......... 139
7.5. Código de colores para resistencias eléctricas.......... 142
7.6. Campo eléctrico.......... 145
7.7. Tabla periódica (JSON).......... 148
¿Cómo continuar?.......... 154
Glosario.......... 156
Índice de funciones.......... 158
Índice de figuras.......... 160

Capítulo 0. Introducción

Este documento muestra las bases de programación en JavaScript utilizando una biblioteca llamada **p5.js. ¿Qué es p5.js?,** desde la página oficial de esta biblioteca se describe a sí misma como: *"¡p5.js es una biblioteca de JavaScript para la programación creativa, que busca hacer que programar sea accesible e inclusivo para artistas, diseñadores, educadores, principiantes y cualquier otra persona! p5.js es gratuito y de código abierto porque creemos que el software y las herramientas para aprenderlo deben ser accesibles para todos…p5.js tiene un conjunto completo de funcionalidades para dibujar. Sin embargo, no estás limitado solo a dibujar en tu lienzo. Puedes tomar toda la página del navegador como tu bosquejo, incluyendo los objetos HTML5 para texto, video, cámara web y sonido."*[1]

Considere que programar es algo que se aprende haciendo, por lo que es importante que en conjunto con este manual, se tenga disponible un computador de escritorio o notebook (laptop) para ir probando los códigos que se explicitan acá. Para probar las características de interactividad que posee esta biblioteca, asegúrese de tener un teclado y un mouse disponibles, además de las ganas para adquirir nuevas habilidades. Una vez que tenga un código escrito y funcional, por simple que sea, tendrá toda la libertad para hacer los cambios que desee, y es bastante seguro que va a comenzar a tener ideas nuevas y querer desarrollarlas e implementarlas en el programa que está escribiendo. Tendrá la ventaja de observar inmediatamente los cambios que está incluyendo.

Información oficial de la biblioteca **p5.js** y adicional a la mostrada en este documento, puede encontrarse en su sitio web, en ese mismo sitio puede encontrar las referencias y ejemplos de códigos que puede utilizar[2].

[1] https://p5js.org/es/ sitio web oficial de la biblioteca p5.js.
[2] https://p5js.org/es/reference/ referencias de la biblioteca p5.js.

Todos los códigos se pueden generar y probar en el sitio https://editor.p5js.org, si bien hay otros editores disponibles (online y de escritorio[3]), se recomienda por su simplicidad comenzar a escribir código en el sitio señalado donde puede crear una cuenta para ir respaldando sus proyectos.

En el editor de código mencionado, por defecto, se pueden editar tres archivos, `index.html`, `sketch.js` y `style.css`, como se observa en la figura 0.1. Con el que trabajaremos principalmente será el llamado `sketch.js` (la extensión `js` es de JavaScript), y es ahí donde se escribirá el código. No se pondrá demasiado énfasis en la edición del archivo `index.html`, aunque será necesario hacerlo en algunas oportunidades, por ahora solo se considerará que es en este archivo donde se invoca a la biblioteca **p5.js** y al archivo de edición del código (`sketch.js`). En este archivo `html` también se establece el nombre del archivo que se utilizará como edición del estilo (`style.css`), archivo que no será trabajado en ninguna oportunidad en este manual, debido a que el mismo solo modifica la estética del archivo HTML, mas no el contenido ni los resultados del código.

El archivo `index.html` debe tener la siguiente estructura básica para comenzar a escribir código con la biblioteca **p5.js** en el archivo `sketch.js`. Si es necesario, este archivo `html` tendrá que ser modificado para incluir otras opciones en los programas que se están desarrollando, pero esto será indicado en el momento que se requiera. En el editor de código, se accede a este archivo, haciendo clic en la parte superior del código en el símbolo “**>**”, que al presionarlo se transforma en “**<**” y se muestran los tres archivos antes mencionados (ver Figura 0.1). Debería ver algo similar al código siguiente:

```
<!DOCTYPE html>
<html lang="en">
  <head>
```

[3] Openprocessing y VSCode por nombrar algunos.

```
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.7.0/p5.js"></s
cript>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.7.0/addons/p5.
sound.min.js"></script>
    <link rel="stylesheet" type="text/css" href="style.css">
    <meta charset="utf-8"/>

  </head>
  <body>
    <main>
    </main>
    <script src="sketch.js"></script>
  </body>
</html>
```

Note que la biblioteca se invoca en los tags `<script>`, y el archivo `sketch.js`, que es donde se escribirá el código del programa, es llamado en el último tag `<script>`. El editor, por defecto, incluye en el archivo `index.html` la biblioteca de **p5.js** asociada al sonido, para crear aplicaciones que requieran de esta característica.

Considere que los tres archivos señalados, son los necesarios para generar una página web que sea visible en un navegador (con `index.html`), que dicha página tenga alguna funcionalidad (con `sketch.js`) y que se vea adecuadamente en un formato agradable de ver (con `style.css`). Claramente esta no es la única forma de generar páginas web, pero si es una forma simple de hacerlo.

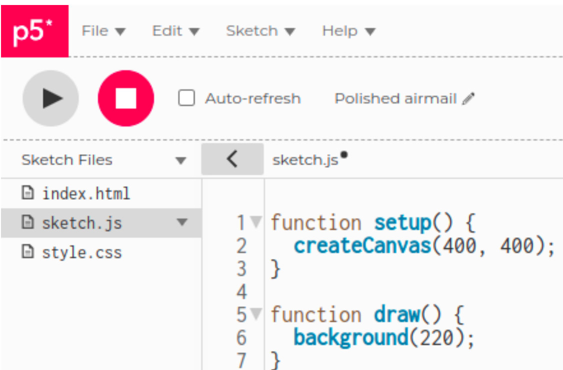


Figura 0.1. Archivos utilizados en la creación de un programa. Para ejecutar el programa se presiona el triángulo que apunta hacia la derecha.

En algunas oportunidades será necesario incluir otros archivos, para realizar esto puede cargar un archivo al editor o bien crear un nuevo archivo con la extensión correspondiente. Cuando sea necesario realizar esta tarea se indicará adecuadamente.

Todos los códigos, que se editarán en el archivo `sketch.js`, cumplen con la siguiente estructura:

```
function setup() {
  //esta parte se ejecuta una sola vez
  createCanvas(400,400);
}

function draw() {
  //esta parte se ejecuta una y otra vez por toda la eternidad
  background(220);
}
```

Donde la función **`setup()`**, y todo lo que se encuentra entre las llaves **`{}`**, se ejecuta una sola vez. La función **`draw()`**, y todo lo que se encuentra entre las llaves **`{}`**, se ejecuta una y otra vez, es un ciclo, se ejecuta hasta que se detiene el código o hasta que se produce un error. El texto escrito después de **`//`** son los comentarios que se pueden incluir en un código, y sirven para ordenar las ideas, para recordar que se quiso hacer o para

indicar a otras personas cuál fue el procedimiento seguido. Los comentarios no afectan el comportamiento del programa.

Al escribir código con esta librería se utilizan tres tipos diferentes de paréntesis, los paréntesis "`{}`" se utilizan principalmente para agrupar distintas secciones de código, por ejemplo, en alguna función (con **`function`**). Los paréntesis "`()`" se utilizan cuando una función necesite, o no, incluir parámetros para su uso. Y los paréntesis "`[]`" se utilizan específicamente para estructuras de programación llamadas arreglos.

De forma general, considere que el programa se ejecuta leyendo cada una de las líneas de código, desde arriba y hacia abajo, por lo que si quiere utilizar una variable, necesariamente esta tendrá que haber sido creada de forma previa en una línea superior.

Con la función **`createCanvas(600,400)`** se establece el ancho y el alto del canvas (en píxeles), siendo el canvas, el lienzo o el área donde se observan los resultados del código. Las variables **`width`** y **`height`** son palabras reservadas para el ancho y alto del canvas, establecidas en **`createCanvas()`**, y se pueden utilizar en cualquier parte del código, es decir, si se desea utilizar el valor **`600`** en el código, se puede escribir **`width`**, y se puede escribir **`height`** para utilizar el valor **`400`** (observe en el ejemplo de la página 14, de la sección "Variables", un modo de utilizarlo). Con la función **`background(220)`** se establece el color del fondo del canvas (en escala de grises si se utiliza un argumento, o en formato RGB si se utilizan tres argumentos). Para la función **`createCanvas()`**, así como para todas las funciones, sus argumentos (que en el caso anterior son 600 y 400) siempre van dentro de paréntesis redondos **`()`**, y si son más de un argumento van separados por una coma (**`,`**).

El canvas (o lienzo), que es donde se observarán los resultados del programa ejecutándose, se distribuye de tal manera, que si se imagina un sistema de referencia **`xy`**, el eje **`x`** apunta hacia la derecha y el eje **`y`** hacia

abajo. Es decir, la posición 0,0 (cero coma cero), se encuentra en la esquina superior izquierda del canvas (ver figura 0.2). En ciertas situaciones se puede cambiar el origen del sistema de referencia, con la función **`translate(x,y)`**, donde **`x`** e **`y`** son las coordenadas del nuevo punto que se considerará como origen del sistema de referencia.

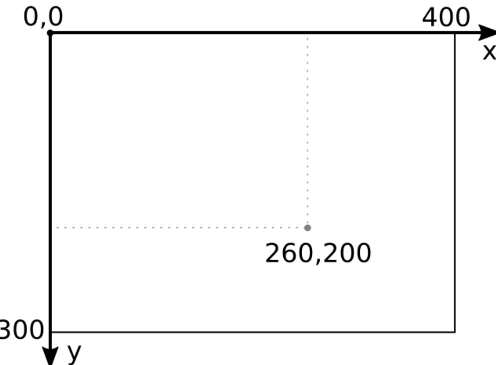


Figura 0.2. Sistema de referencia y su distribución en el canvas. Se indica la posición del punto (260,200) en un canvas de 400 píxeles de ancho y 300 de alto.

Como operadores matemáticos básicos se utilizan **`+`**, **`-`**, **`*`** y **`/`**, para suma, resta, multiplicación y división, respectivamente. En la sección **1.10.** de este documento se encuentran otras funciones matemáticas que serán de utilidad.

Cuando se escriba un programa será recomendable que al final de cada línea de código, que no sea un comentario, se incluya el carácter "**`;`**". No es necesario este carácter cuando en las líneas de código se incluyen la palabra clave **`function`**, o cuando las líneas incluyen un ciclo (ya sea **`for`** o **`while`**) o cuando se cierre una sección de código con "**`}`**". Revise los ejemplos de código en cada caso para verificar su uso más conveniente.

En programación, generalmente se pueden hacer las cosas de múltiples formas, los códigos que se muestran en este documento muestran solo

una forma de lograr cada uno de los objetivos planteados en cada situación.

Todos los códigos presentes en este libro, desde el más simple al más complejo, están disponibles en Github de manera gratuita[4]. En este lugar, los códigos se encuentran organizados de acuerdo al orden presente en este libro en sus capítulos y secciones.

Resumen

Se ha introducido la biblioteca **p5.js**, su utilidad y se ha indicado donde es posible comenzar a escribir código a través de un editor online. Se ha indicado la distribución del canvas y el sistema de referencia que se utiliza por defecto. Se ha indicado también la estructura que tendrán los códigos que se escribirán y la forma en la que se incluyen comentarios que no afectan el desarrollo de un programa.

Se ha indicado además donde revisar información oficial de la biblioteca a utilizar y se ha señalado dónde encontrar las referencias de ayuda en caso de ser necesario. También se ha señalado dónde encontrar los códigos que aparecen en este libro.

Su principal tarea, en este momento, es ingresar al editor online oficial de **p5.js** y crear una cuenta para probar los códigos que se compartirán a lo largo de este libro o para generar sus propias aplicaciones, de esta forma podrá guardar sus códigos y volver a ellos cuando lo estime conveniente.

[4] https://github.com/sbsacev/cursop5js

Capítulo 1. Variables, geometrías y funciones principales

1.1. Variables

Puede entender una variable como un elemento que puede ir cambiando su valor dentro de la ejecución de un programa y que debe declararse con algún nombre en algún lugar del código, no pueden existir variables que no hayan sido declaradas. Se recomienda que los nombres comiencen con minúsculas y que el nombre sea lo más descriptivo posible. Una vez definidas, deben respetarse las minúsculas y mayúsculas que puede tener cada nombre de variable para el resto del código. Es distinto una variable llamada **`valor`** a otra variable llamada **`Valor`** (con **`V`** mayúscula).

Con respecto a las variables se establecen principalmente dos grupos: **variables globales** y **variables locales**. Las variables globales se declaran, generalmente, antes de la función **`setup()`** y son variables a las que se puede acceder en cualquier parte del código. Las variables locales se utilizan en sectores específicos del código, no se puede acceder a ellas en sectores distintos a donde fueron declaradas. Todas las variables, globales y locales, se declaran utilizando la palabra clave **`let`** antes del nombre de la variable.

```
let a=300;
let b;
let c;

function setup() {
  createCanvas(400, 400);
  b=width/2;
}

function draw() {
  background(220);
  c=a+b;
  print(c);
}
```

En el ejemplo anterior se declaran **variables globales** **`a`**, **`b`** y **`c`**. La variable **`a`** se declara y a la vez se le asigna el valor 300, para asignar un valor a una variable se utiliza un signo **`=`**. La variable **`b`** se declara y se le asigna un valor dentro de **`setup()`** (igual a **`width/2`**, la mitad del ancho). La variable **`c`** se declara y se le asigna su valor (**`a+b`**) dentro de la función **`draw()`**. La función **`print()`** se utiliza para mostrar resultados en la consola del editor (se utiliza principalmente para verificar y controlar los resultados que se están obteniendo).

Con respecto a **variables locales** se puede ejemplificar de la siguiente manera:

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  for(let i=0; i<4; i++){
    print(i);
  }
}
```

La variable **`i`** se declara dentro del ciclo **`for()`** (tema que revisaremos posteriormente), como la función **`print()`** se encuentra entre los

paréntesis **{}**, siguientes a la función **for()**, se mostrará en consola los valores 0, 1, 2, 3, una y otra vez.

Pero si se utiliza la función **print(i)** fuera del ciclo **for()** y de sus paréntesis **{}**:

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);

  for(let i=0; i<4; i++){

  }
print(i);
}
```

Se mostrará en consola el siguiente mensaje:

```
ReferenceError: i is not defined
🌸 p5.js dice: [sketch.js, line 11] ...
```

Esto sucede porque, en el segundo caso, la variable local **i** fue definida dentro del **for()**, pero se está utilizando tal variable **i**, con **print(i)**, fuera del **for()**, es decir, en este caso solo se puede acceder a la variable **i** si se utiliza dentro del **for()**.

1.2. Dibujo de geometrías sencillas

La biblioteca **p5.js** permite mostrar en el canvas algunas figuras geométricas, entre ellas: líneas (rectas), círculos, elipses, rectángulos, arcos, triángulos y curvas (bezier). Todas las funciones y los ejemplos que se señalan a continuación debe incluirlas dentro de la función **draw()** descrita anteriormente, para permitir interacción con estas geometrías. Si utiliza estas funciones en la función **setup()**, estas se mostrarán

brevemente y serán ocultadas si se utiliza la función **`background()`** dentro de la función **`draw()`**.

Para dibujar una línea recta se utiliza la función **`line(x1,y1,x2,y2)`** con cuatro argumentos, donde **`x1,y1`** son las coordenadas del primer punto de la recta y **`x2,y2`** son las coordenadas del segundo punto de la recta. Por defecto, el color de la línea será negro, a no ser que se indique lo contrario.

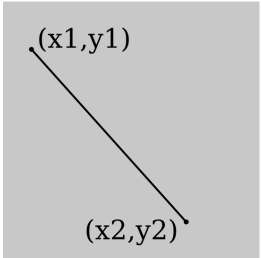


Figura 1.1. Línea recta con `x1 = 20`, `y1 = 40`, `x2 = 160`, `y2 = 180`.

Para dibujar un círculo, se utiliza la función **`circle(x,y,diam)`** con tres argumentos, donde **`x,y`** es la posición del centro del círculo y **`diam`** es el diámetro. Por defecto, un círculo se dibuja de color blanco y con un borde negro.

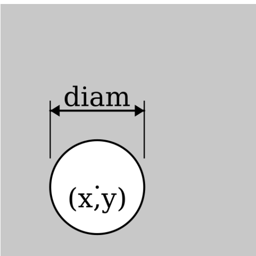


Figura 1.2. Círculo: `x = 80`, `y = 140`, `diam = 60`.

Para dibujar una elipse, se utiliza la función **`ellipse(x,y,ancho,alto)`** con cuatro argumentos, donde **`x,y`** es la posición del centro de la elipse, **`ancho`** es el ancho de la elipse y **`alto`** es

el alto. Si **ancho** y **alto** son iguales, se dibuja un círculo. Por defecto, una elipse se dibuja de color blanco y con un borde negro.

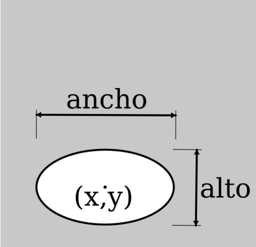


Figura 1.3. Elipse: `x = 80, y = 140, ancho = 80, alto = 40.`

Para dibujar un rectángulo, se utiliza la función **rect(x,y,ancho,alto)** con cuatro argumentos como mínimo, donde **x,y** es la posición del punto superior izquierdo del rectángulo, **ancho** es el ancho del rectángulo y **alto** es el alto. Si se quiere dibujar un rectángulo con respecto a su centro geométrico, se debe utilizar la función **rectMode(CENTER)** antes de dibujar el rectángulo. Existen otras opciones para redondear las esquinas del rectángulo, esto se realiza añadiendo más parámetros a la función **rect()** que los señalados en su descripción inicial. Por defecto, un rectángulo se dibuja de color blanco y con un borde negro.

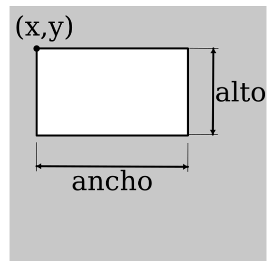


Figura 1.4. Rectángulo: `x = 30, y = 40, ancho = 120, alto = 70.`

Para dibujar un arco, se utiliza la función:
arc(x1,y1,ancho,alto,angulo1,angulo2), donde **x1,y1** es la posición del centro estimado del arco, **ancho** es el ancho estimado del arco y **alto** es el alto estimado del arco. Los valores **angulo1** y

angulo2 son los ángulos inicial y final, desde donde hasta donde se dibuja el arco. Es importante señalar que por defecto estos ángulos se encuentran en radianes, pero si se utiliza la función **angleMode(DEGREES)**, se podrán utilizar los ángulos en grados. El ángulo crece desde el eje **x** en sentido horario, de acuerdo al sistema de referencia indicado en la introducción de este documento.

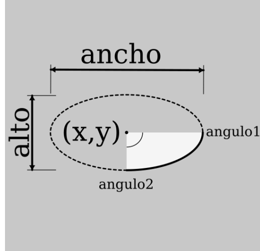


Figura 1.5. Arco: `x1 = 100, y1 = 100, ancho = 120, alto = 80, angulo1 = 0, angulo2 = PI/2` (en radianes).

Para dibujar un triángulo se utiliza la función **triangle(x1,y1,x2,y2,x3,y3)**, donde **x1,y1** son las coordenadas del primer vértice del triángulo, **x2,y2** son las coordenadas del segundo vértice y **x3,y3** son las coordenadas del tercer vértice.

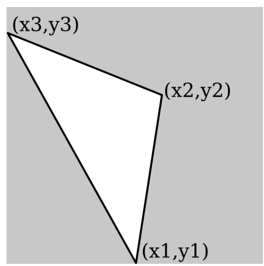


Figura 1.6. Triángulo: `x1 = 100, y1 = 200, x2 = 120, y2 = 80, x3 = 0, y3 = 10.`

Para dibujar una curva sencilla se utiliza la función:
bezier(x1,y1,x1c,y1c,x2c,y2c,x2,y2) donde los puntos **x1,y1** son las coordenadas del punto inicial de la curva y **x2,y2** son las coordenadas del punto final de la curva. Los parámetros **x1c,y1c** y

x2c,y2c son los valores de control para el punto inicial y final respectivamente. En la figura, aparte de la curva bezier, se muestran como referencias las líneas **line(x1,y1,x1c,y1c)** y **line(x2,y2,x2c,y2c)**.

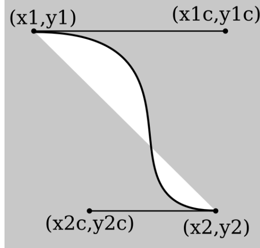


Figura 1.7. Bezier: `x1 = 10`, `y1 = 10`, `x1c = 190`, `y1c = 10`, `x2c = 60`, `y2c = 180`, `x2 = 180`, `y2 = 180`.

El siguiente código muestra un ejemplo de cómo utilizar las funciones señaladas anteriormente:

```
function setup() {
  createCanvas(400, 400);
  rectMode(CENTER);
  angleMode(DEGREES);
}

function draw() {
  background(220);

  line(0,0,300,300);
  circle(mouseX,mouseY,40);
  ellipse(300,100,200,50);
  rect(300,200,50,80);
  arc(300,300,100,100,0,90);
  triangle(30,350,30,300,100,350);
  bezier(150,200,180,230,180,150,200,300);
}
```

El resultado del código anterior se puede observar en la figura 1.8.

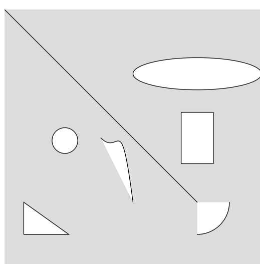


Figura 1.8. Muestra de figuras geométricas en el canvas.

Las palabras clave **`mouseX`** y **`mouseY`** entregan la posición **`x`** e **`y`** del mouse, por lo tanto, el círculo dibujado que incluye esas variables en su argumento sigue el movimiento del mouse.

Para todas las geometrías revisadas, existe la opción de alterar sus características, como por ejemplo cambiar el color de la línea (con **`stroke()`**), o cambiar el grosor de la línea (con **`strokeWeight(grosor)`**, **`grosor`** será un número que indica los píxeles de ancho que tiene la línea), también se puede rellenar la geometría dibujada con un color (**`fill()`**). Si no se requiere el uso de una línea se utiliza la función **`noStroke()`** y si no se requiere rellenar con un color existe la función **`noFill()`**.

Estas opciones en el código, deben escribirse antes de dibujar la geometría deseada, y si se incluyen varias geometrías despues de utilizar las opciones de color y grosor de línea, todas esas geometrías se veran afectadas por las mismas características de color y grosor de línea. Si se desea que cada geometría tenga un color de relleno o de línea distintas, deben usarse estas opciones distintas, antes de cada una de las geometrías.

Todas las opciones de entregar color (**`stroke()`** y **`fill()`**, incluida la función **`background()`**) funcionan con **un** argumento para escala de grises (entre 0 y 255) y con **tres** argumentos en formato RGB, Red, Green y Blue, respectivamente, donde cada argumento estará entre 0 y 255. Si

se utilizan **dos** argumentos, el primero corresponde a escala de grises y el segundo es para la transparencia. Si se utilizan **cuatro** argumentos, los primeros tres son los valores RGB y el cuarto es para la transparencia. Por ejemplo: **`fill(255,0,0)`** rellenará una geometría de color rojo, **`fill(255,255,0)`** rellenará una geometría de color amarillo y **`fill(0,0,255,40)`** rellenará una geometría de colo azul y con cierta transparencia.

Si desea conocer los valores RGB de un color específico, puede utilizar su buscador web de preferencia y buscar “selector de color” o “color picker”, probablemente el primer resultado de la búsqueda será una aplicación donde podrá recorrer todos los colores del espectro y obtener los valores RGB de un color individual (figura 1.9). Eventualmente, con **p5.js** se puede crear una aplicación propia de selector de color.

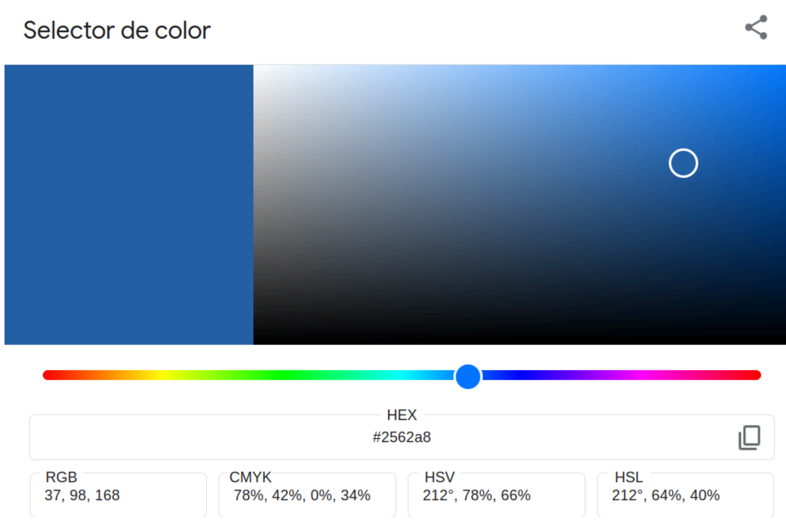


Figura 1.9. Herramienta de selector de color. En el ejemplo: R=37, G=98 y B=168.

Esta herramienta entrega, además, los valores asociados a otros perfiles de color (HEX, CMYK, HSV y HSL).

Un ejemplo de aplicación de color a figuras geométricas es el siguiente:

```
function setup() {
  createCanvas(400, 400);
  rectMode(CENTER);
  angleMode(DEGREES);
}

function draw() {
  background(220);

  stroke(0);
  line(0,0,300,300);
  fill(255,0,0);
  circle(50,200,40);
  fill(255,255,0,50);
  ellipse(300,100,200,50);
  rect(300,200,50,80);
  noFill();
  arc(300,300,100,100,0,90);
  strokeWeight(4);
  stroke(0,0,255,50);
  triangle(30,350,30,300,100,350);
  noFill();
  bezier(150,200,180,230,180,150,200,300);
}
```

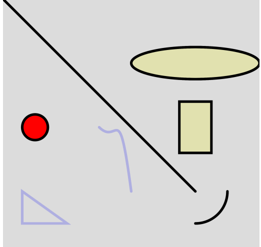


Figura 1.10. Muestra de geometrías con aplicación de color.

Si se desea que las opciones de color de linea, grosor de linea y color de llenado no afecten a otras geometrías, deben utilizarse estas opciones entre las funciones **`push()`** y **`pop()`**.

```
function setup() {
  createCanvas(400, 400);
```

```
  rectMode(CENTER);
  angleMode(DEGREES);
}

function draw() {
  background(220);
  push();
  fill(255,0,0);
  strokeWeight(8);
  stroke(0,255,0);
  circle(200,200,100);
  pop();
  rect(200,350,100,50);
}
```

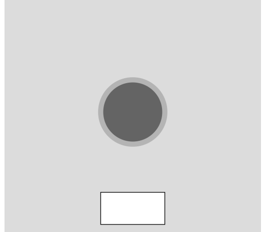


Figura 1.11. Cambio de color a figuras específicas.

Si dos figuras geométricas se superponen, queda encima (visible) la última que fue descrita en el código.

Finalmente, para rotar cualquiera de las geometrías dibujadas en el canvas, se puede anteponer la función **`rotate()`**, que requiere como argumento un ángulo (que por defecto es en radianes). Si se desea que el argumento esté en grados debe añadirse la opción **`angleMode(DEGREES)`**. La rotación se generará en sentido horario con respecto al origen del sistema de referencia, esquina superior izquierda, a no ser que se declare un origen del sistema de referencia distinto con la función **`translate()`**. Si solo se requiere rotar una sola geometría deberán usarse las funciones **`push()`** y **`pop()`**. En el ejemplo de la figura 1.12. se dibujan dos rectángulos, pero solo uno está girado un ángulo de

`PI/4` radianes (45°) después de cambiar el origen del sistema de referencia al centro del canvas.

```
function setup(){
  createCanvas(400,400);
}

function draw(){
  background(200,50,0);
  rect(100,40,50,20);

  push();
  translate(width/2,height/2);
  rotate(PI/4);
  rect(100,40,50,20);
  pop();
}
```

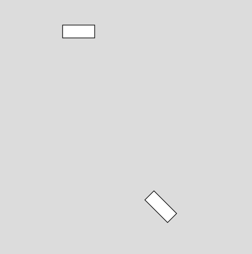


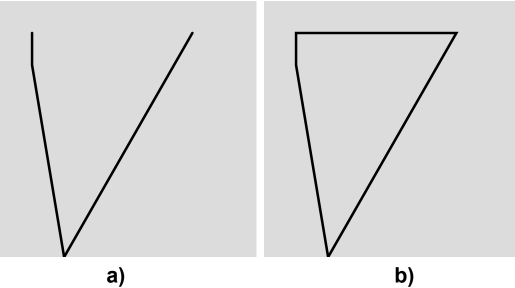
Figura 1.12. Rotación de una figura geométrica.

1.3. Geometrías más complejas

Se pueden dibujar geometrías distintas a las establecidas anteriormente entregando los diferentes puntos (o vértices) por donde se dibujarán las líneas.

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
```

```
  noFill();
  strokeWeight(4);
  beginShape();
  vertex(50,50);
  vertex(50,100);
  vertex(100,400);
  vertex(300,50);
  endShape();
}
```



a) b)

Figura 1.13. Aplicación de la función vertex() para mostrar geometrías en el canvas.

Note que es necesario utilizar las funciones **`beginShape()`** y **`endShape()`** y entre ellos se pondrán los puntos (vértices) que se representarán en el canvas con la función **`vertex(x,y)`**. Las mismas características de línea y color de llenado se pueden utilizar con estas figuras geométricas. Si para terminar la figura se utiliza **`endShape(CLOSE)`** (en vez de simplemente **`endShape()`**) la figura se dibuja con un borde completamente cerrado.

Esta forma de dibujar geometrías tomará más sentido cuando se revise el funcionamiento del ciclo **`for()`**.

1.4. Mostrando texto en pantalla

La función que permite mostrar texto en el canvas es **`text(texto,x,y)`**, donde **`texto`** es el texto que se quiere mostrar y

x,y son las coordenadas donde comenzará a escribirse el texto hacia la derecha. Si el texto se desea centrar con respecto al punto **x,y** entregado, puede utilizarse la opción **textAlign(CENTER,CENTER)**, el "primer" **CENTER** alinea al centro horizontalmente y el "segundo" **CENTER** alinea al centro verticalmente. Algunos ejemplos de uso de la función **text()** pueden revisarse a continuación:

```
let texto="Esto es un texto";
let a=400;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  text(texto, 200,30);
  text("El valor de a es: "+a, 50,100);

  push();
  textAlign(CENTER,CENTER);
  textSize(20);
  text("Esto también es un texto.",200,200);
  pop();

  text("Esto es un texto un poco más largo ubicado en una caja de
texto de ciertas dimensiones",200,300,100,150);
}
```

Figura 1.14. Múltiples formas de mostrar texto en el canvas.

Al usar la función **`text()`**, el texto que se puede mostrar puede ser una combinación de texto (entre comillas) más una o varias variables, en el ejemplo se realiza con **`"El valor de a es: "+a`**, donde **`a`** es la variable a mostrar como texto.

Note que usando las opciones **`push()`** y **`pop()`** se puede alinear un solo texto en el centro de acuerdo a las coordenadas entregadas en ese texto. Note también que el último **`text()`** utilizado, incluye dos valores más (**`100 y 150`**), que determinan el tamaño de la caja que incluirá el texto, como puede verse, estos dos valores adicionales son opcionales. Se puede cambiar el tamaño del texto con la función **`textSize()`** antes del texto que se requiera modificar y las opciones de grosor de línea y color vistas anteriormente, también las puede utilizar en textos.

Para efectos de mostrar texto en el canvas se puede utilizar comillas dobles (**`“texto”`**), o bien comillas simples (**`‘texto’`**) de forma indistinguible. Por lo que es lo mismo utilizar **`text(“Esto es un texto”, 20, 20)`** que **`text(‘Esto es un texto’,20,20)`**. El uso de comillas dobles o de comillas simples de forma indistinta, no es

exclusiva para el uso de la función **`text()`**, sino que siempre pueden utilizarse en cualquier sección del código que requiera el uso de comillas.

Cuando se muestra texto en pantalla, el texto aparece con una tipografía (una fuente) por defecto, pero existen ocasiones donde se preferirá utilizar una fuente distinta, para ello podemos subir una tipografía (un archivo en formato “**`ttf`**”, por ejemplo) y cargarla en una función **`preload()`** con la función **`loadFont()`** (otras opciones de subir archivos se revisarán en el capítulo 2), luego, y antes de utilizar la función **`text()`** en el código, debe utilizarse la función **`textFont()`**, que debe incluir como parámetro el nombre de la variable que contiene el archivo de la fuente.

```
let fuente;
let a=2;
let b=5;

function preload(){
    fuente = loadFont("times.ttf");
}

function setup(){
    createCanvas(512,512);
}

function draw(){
    background(0,150,250);
    textSize(20);
    text("Hola", width/2, 40);

    push();
    textFont(fuente);
    textSize(30);
    textAlign(CENTER);
    text("Hola, esto es p5.js", width/2, 80);
    pop();

    let suma=a+b;
    text("2 + 5 = "+suma, width/2, 120);
}
```

Hola

Hola, esto es p5.js

2+5 = 7

Figura 1.15. Uso de tipografías distintas y muestra de valores numéricos dentro de un texto.

En el ejemplo anterior se muestra un texto en el canvas con una fuente, y otro texto con otra fuente (“Hola, esto es p5.js” utiliza la fuente Times new roman), utilizando las opciones **`push()`** y **`pop()`**.

1.5. Condición `if`

La condición **`if()`** permite ejecutar una o varias acciones dependiendo si se cumplen, o no, una o varias condiciones. Para establecer estas condiciones se utilizan comparadores, por ejemplo:

- a<b, que significa a es menor que b.
- a>b, que significa a es mayor que b.
- a<=b, que significa a es menor o igual que b.
- a>=b, que significa a es mayor o igual que b.
- a!=b, que significa a es distinto que b.
- a==b (con dos signos igual), que significa a es igual a b.

La estructura de utilización de la función **`if()`**, es decir, su sintaxis, es la siguiente:

```
if(acá va la condición){
   acá van las acciones a realizar si se cumple la condición
}
```

La función **`if()`** permite realizar acciones en el caso que la condición presente en la función no se cumpla, su sintaxis sería así:

```
if(acá va la condición){
```

```
   si se cumple la condición, se realizan estas acciones
}else{
  si no se cumple la condición, se realizan estas acciones
}
```

Existe, además, la opción de establecer una nueva condición luego de la palabra **`else`**, y su sintaxis sería la siguiente:

```
if(acá va la condición){
   si se cumple la condición se realizan estas acciones
}else if(otra condición){
  si no se cumple la primera condición y si se cumple la otra
condición, se realizan estas acciones
}
```

Un ejemplo que incluye la estructura completa de un código funcional, con variables declaradas y resultados visibles, puede revisarse a continuación:

```
let a=1;
let b=2;
let c=3;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  if(a==b){
    text("a y b son iguales",200,200);
  }else{
    text("a y b son distintos",200,200);
  }

  if(b<c){
    text("b es menor que c",200,300);
  }else{
    text("b es mayor o igual que c",200,300);
  }
}
```

a y b son distintos

b es menor que c

Figura 1.16. Uso de condiciones.

Si quisiera que se realicen acciones cuando se cumplan dos condiciones simultáneamente, se puede incluir **`&&`** (a modo de `and`, 'y') en el **`if()`** como separador de las condiciones. Si se desea que se ejecuten acciones cuando se cumpla al menos una de dos o más condiciones, se utiliza **`||`** (a modo de `or`, 'o') en el **`if()`** como separador de las condiciones. Observe los ejemplos que siguen:

```
let a=1;
let b=2;
let c=3;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  if(a<b && b<c){
    text("Se cumplen ambas condiciones",100,200);
  }

  if(a>b || b<c){
    text("Se cumple una de las dos condiciones",100,300);
  }
}
```

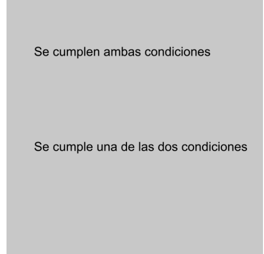


Figura 1.17. Uso de 'and' y 'or' en condiciones.

En el ejemplo anterior, en el primer **`if()`** deben cumplirse ambas condiciones (que **`a`** sea menor que **`b`** y que **`b`** sea menor que **`c`**) para ejecutar el código que sigue. En el segundo **`if()`** se ejecuta la acción si se cumple al menos una de las dos condiciones (cuando **`a`** es mayor que **`b`** ó cuando **`b`** es menor que **`c`**).

1.6. La función `map`

La función **`map()`** permite calcular cualquier valor de acuerdo a un rango de valores mínimo y máximo elegido, conociendo el rango de valores de entrada que tiene una variable.

Esta función requiere cinco argumentos y la forma de utilizarse es la siguiente: **`map(dato, vmin, vmax, smin, smax)`**, donde **`dato`** es la variable de entrada, **`vmin`** será el valor mínimo conocido que tiene la variable dato, **`vmax`** será el valor máximo conocido que tiene la variable **`dato`**, **`smin`** será el valor de salida mínima que se asocia con el valor **`vmin`** y **`smax`** será el valor de salida máxima que se asocia con el valor **`vmax`**. Es decir, si **`dato`** toma el valor **`vmin`**, la función **`map()`**, utilizada de esta forma, entregará el valor **`smin`**. En otra situación, si **`dato`** toma el valor **`vmax`**, la función map, utilizada de esta forma, entregará el valor **`smax`**. Y si **`dato`** toma un valor entre **`vmin`** y **`vmax`**, la función **`map()`** entregará un valor entre **`smin`** y **`smax`**. Esta función extrapola a valores

fuera del rango entre **smin** y **smax** si es que el valor de entrada **dato** está fuera del rango de entrada entre **vmin** y **vmax**.

La mejor forma de entender esto es con un ejemplo. Si una prueba en la universidad tiene un rango de puntaje entre 0 puntos como mínimo y 50 como máximo, y quisiera calcular la calificación final de acuerdo al puntaje obtenido, que tiene un rango de salida entre 1 y 7, se podría utilizar la función **nota=map(puntaje,0,50,1,7).** Esta forma es válida si la prueba tiene una exigencia de 50% (con la mitad del puntaje, tiene la mitad de la calificación, un 4.0). Pero si la prueba tiene una exigencia de 60% (es decir, con el 60% del puntaje máximo se obtiene una calificación 4.0), la nota 4.0 se obtiene con 30 puntos cuando el puntaje máximo es 50. En este caso, para obtener la calificación, la función **map()** podría utilizarse de la siguiente forma:

```
let puntaje=32;
let nota;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220,100,0);
  if(puntaje<=30){
     nota=map(puntaje,0,30,1,4);
     }else{
     nota=map(puntaje,30,50,4,7);
  }
  textSize(40);
  text(nota,width/2,height/2);
}
```

El ejemplo anterior mostrará como texto la nota obtenida de acuerdo al puntaje ingresado en la variable **puntaje**. Si se requiere entregar la calificación con un decimal podría utilizarse **text(round(10*nota)/10,width/2,height/2)**, donde la función **round()** redondea el valor al entero más próximo. Por ejemplo, si nota

entrega un valor `5.67`, al multiplicar por `10` queda `56.7`, al redondear con **`round()`**, queda en `57` y dividiendo por `10` da un `5.7`.

En conjunto con la función **`map()`**, suele usarse la función **`constrain()`**, que acota los valores de una variable entre un valor mínimo y un valor máximo designado. La función **`constrain()`** no extrapola. Por ejemplo, si **`valor=constrain(variable,10,50)`**, en el caso que **`variable`** sea igual a 5, a **`valor`** se le asignará el número 10; si **`variable`** es igual a 65, a **`valor`** se le asignará el número 50; y si **`variable`** es igual a 45, a **`valor`** se le asignará el número 45, porque está dentro del rango designado, entre 10 y 50.

1.7. Ciclo `for`

El ciclo **`for`** se utiliza para realizar una tarea de forma repetida las veces que sea necesaria. La sintaxis del ciclo **`for()`** es la siguiente:

```
for(variable que variará; hasta cuando variará; cómo variará){
  acciones que se realizarán de forma cíclica
}
```

La “`variable que variará`” puede declararse como una variable local y debe tener un valor inicial asignado, luego y separado por “`;`” se escribirá la condición hasta cuándo se repetirá el ciclo, similar a como se establece la condición en la función **`if()`**. Finalmente, y también separado por “`;`”, se indicará cómo irá variando la variable local “que variará”, esta podrá ir creciendo en una cantidad determinada, o ir decreciendo en una cantidad determinada, dependiendo del objetivo.

Por ejemplo, si se requiere dibujar 4 círculos alineados horizontalmente con un número distinto dentro de cada círculo, se puede seguir el siguiente ejemplo:

```
function setup() {
  createCanvas(400, 400);
}
```

```
function draw() {
  background(20,230,0);
  for(let i=0; i<4; i++){
    circle(i*80+50,200,40);
    textSize(20);
    textAlign(CENTER,CENTER);
    text(i+1,i*80+50,200);
  }
}
```

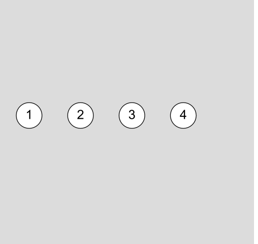


Figura 1.18. Uso de ciclo `for()`.

La variable local **`i`** se declara con **`let`** dentro del ciclo **`for()`**, comienza con el valor 0 y va creciendo de 1 en 1 (esto se realiza utilizando **`i++`**) mientras sea menor que 4 (**`i<4`**, o sea, desde 0 hasta 3). Dentro de los paréntesis **`{}`** se detallan las acciones que se ejecutarán de forma repetida. En este caso, por cada pasada del ciclo, la posición **`x`** del círculo va cambiando de acuerdo a la expresión **`i*80+50`**. Lo que se muestra con **`text(i+1,i*80+50,200)`** también va cambiando en cada pasada del ciclo, para **`i=0`** el texto que se escribe es 1 (0+1) en la posición **`x`** igual a 50, para **`i=1`** el texto que se escribe es 2 (1+1) en la posición **`x`** igual a 130, y así sucesivamente hasta que se cumple el ciclo.

Se puede usar un ciclo **`for()`** para generar gráficas o geometrías más complejas, por ejemplo se podría dibujar la función sen(x), o alguna variante, utilizando las opciones de **`beginShape()`** y **`endShape()`**, además de **`vertex()`** dentro de un ciclo **`for()`**. Ver ejemplo a continuación:

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  noFill();
  beginShape();
  for(let i=0; i<width; i++){
    let y=sin(0.1*i);
    let yreal=map(y,-1,1,300,100);
    vertex(i,yreal);
  }
  endShape();
}
```

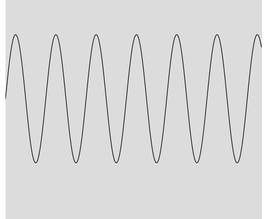


Figura 1.19. Uso de ciclo for() y vertex() para representar funciones matemáticas.

Note que en el ejemplo las variables **`i`**, **`y`** e **`yreal`** son variables locales. En el código que genera la gráfica de la figura 1.20. podría intentar que el valor 0.1, que se encuentra en el argumento de la función **`sin()`**, sea controlado con el movimiento del mouse.

En algunas oportunidades será necesario distribuir alguna geometría en todo el espacio del canvas, de izquierda a derecha y de arriba hacia abajo. Lo anterior es posible hacerlo con dos ciclos **`for()`**, uno dentro de otro. El ciclo externo establece una posición **`y`** para la ubicación de un objeto, y luego se recorre la posición **`x`** para cada uno de estos valores **`y`** con el ciclo interno. Una vez recorrido todos los valores de **`x`**, el valor **`y`** toma un nuevo valor y se repite el proceso.

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  for(let y=0; y<height; y+=20){
    for(let x=0; x<width; x+=20){
      circle(x+10,y+10,10);
    }
  }
}
```

El ejemplo anterior dibuja círculos en todo el área del canvas con la utilización de dos ciclos **`for()`**. El ciclo externo recorre y establece los valores de la posición **`y`**, y el ciclo interno recorre y establece los valores de la posición **`x`**. Cada círculo se ubica a una distancia de 20 píxeles de los demás círculos, y cada círculo tiene un diámetro de 10.

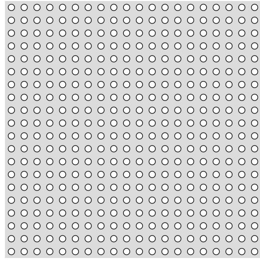


Figura 1.20. Distribución de la misma geometría en todo el plano del canvas.

1.8. Interactividad con el mouse

La biblioteca **p5.js** incluye varias formas de interactuar con el mouse, se revisarán algunas de ellas, considerando que se mostró el uso de **`mouseX`** y **`mouseY`** en una sección anterior.

Otra forma de interactuar con el movimiento del mouse son las opciones **`pmouseX`** y **`pmouseY`**, que entregan, respectivamente, las componentes **`x`**

e **y** previas a la posición actual del mouse. Un ejemplo del uso a continuación:

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  line(mouseX,mouseY,pmouseX,pmouseY);
}
```

Dentro de la sección **draw()** puede incluirse la opción **mouseIsPressed**, que entrega un valor verdadero (**true**), cuando se hace clic con el mouse y falso (**false**) cuando se suelta (true y false son valores booleanos, es decir, que puede tomar dos estados, 1 o 0, verdadero o falso, true o false), el siguiente ejemplo dibuja un círculo cuando se hace clic con el mouse:

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  if(mouseIsPressed){
     circle(200,200,100);
  }
}
```

Las opciones de interactividad con el mouse que vienen a continuación son funciones que se recomienda invocarlas fuera y después de terminada la función **draw()**. La primera a revisar será **mousePressed()** que se activa en el acto mismo de hacer clic con cualquier botón del mouse.

```
let control=0;

function setup() {
  createCanvas(400, 400);
}
```

```
function draw() {
  background(220);
  print(control);
}

function mousePressed(){
  control=1;
}

function mouseReleased(){
  control=0;
}
```

En el ejemplo anterior se utiliza también la función **mouseReleased()**, que se activa cuando se deja de presionar alguno de los botones del mouse. En el ejemplo, al presionar un botón del mouse la variable **control** toma el valor 1, pero cuando se deja de presionar vuelve a tomar el valor 0.

Otra forma de interacción con el mouse sucede cuando se hace clic con el mouse y se arrastra, esta función se llama **mouseDragged()**, y puede observar un ejemplo a continuación:

```
let x=0;
let y=0;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  circle(x,y,40);
}

function mouseDragged(){
  x=mouseX;
  y=mouseY;
}
```

En el ejemplo anterior el círculo que se dibuja sigue al mouse solo cuando se hace clic con el mouse y se arrastra a otro lugar, si solo se hace clic no sucede nada.

Con lo revisado anteriormente podría implementarse un botón que al hacer clic con el mouse realice una acción predefinida, para ello, la función **`mousePressed()`** (o la función **`mouseReleased()`**) deberá incluir una condición que reconozca que el puntero del mouse se encuentra dentro de la zona asignada al botón. Si este botón es circular, entonces basta que la distancia de la posición del puntero del mouse al centro de la circunferencia sea menor que el radio de la circunferencia. Esta distancia puede calcularse utilizando la función **`dist(mouseX,mouseY,x,y)`**, donde **`mouseX`** y **`mouseY`** son la posición del puntero del mouse y **`x`** e **`y`** son las coordenadas del centro de la circunferencia.

```
let x=200;
let y=300;
let r=30;
let R=255;
let G=0;
let B=0;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  fill(R,G,B);
  circle(x,y,2*r);
}

function mousePressed(){
  if(dist(mouseX,mouseY,x,y)<=r){
    R=0;
    B=255;
  }
}
```

En el ejemplo anterior se cambia el color del botón (de rojo a azul), cuando se hace clic dentro del mismo botón. Si se hubiera utilizado function **mouseReleased()**, en vez de **mousePressed()**, habría funcionado de manera bastante similar. ¿Cómo implementaría un botón rectangular?

La última forma de interactividad con el mouse que se revisará será con la rueda de desplazamiento del mouse con la función **mouseWheel().** Para llevar a cabo esta implementación, dibujaremos un rectángulo vertical en el canvas y sobre este rectángulo dibujaremos otro rectángulo, más pequeño, que se moverá sobre el primero a modo de un control deslizante. Este cambio de posición, hacia arriba y hacia abajo, se generará con el movimiento de la rueda del mouse.

```
let y;

function setup() {
  createCanvas(400, 400);
  rectMode(CENTER);
  y=height/2;
}

function draw() {
  background(220);
  fill(255);
  rect(width/2,height/2,40,300);
  fill(0,50,155);
  rect(width/2,y,40,20);
}

function mouseWheel(evento){
  y+=evento.delta/10;
  y=constrain(y,50,350);
  return false;
}
```

Como se puede revisar en el código anterior, se incorpora **function mouseWheel(evento)** después de la función **draw()**, donde la función **mouseWheel()** recibe un argumento (**evento** en el ejemplo) que reconoce cómo se genera el movimiento de la rueda con **evento.delta**,

este valor puede tomar valores positivos o negativos, de tal manera que el rectángulo pequeño cambia su posición vertical en una cierta cantidad **`evento.delta/10`**, el número 10 puede ser distinto, pero con este número se obtienen resultados aceptables. Luego se acota la posición **`y`** del rectángulo con **`y=constrain(y,50,350)`**. La línea de código **`return false`** es opcional, pero se recomienda utilizar cuando no se desea que el navegador completo cambie su posición vertical y afecte solo a lo que se requiere mover en el canvas.

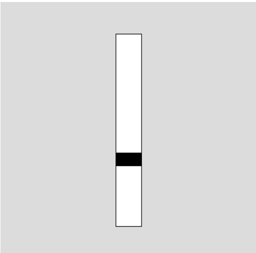


Figura 1.21. Representación de un deslizador con movimiento de la rueda del mouse.

1.9. Interactividad con el teclado

Interactuar con el teclado del computador también es posible con la biblioteca **p5.js**, y existen varias formas de hacerlo, primero se verán las opciones que se pueden utilizar dentro de la función **`draw()`**.

La opción **`keyIsPressed`** entrega un valor verdadero (**`true`**) cuando se presiona cualquier tecla del teclado, y entrega un valor falso (**`false`**) cuando no se presiona ninguna. La opción **`key`** entrega la tecla que se presionó (si presiono la “a”, **`key`** reconoce que se presionó la “a”). Veamos un ejemplo donde se ocupan estas dos opciones:

```
function setup() {
  createCanvas(400, 400);
  textAlign(CENTER,CENTER);
}
```

```
function draw() {
  background(220);
  textSize(40);
  text(key,200,100);
  if(keyIsPressed){
    circle(200,300,50);
  }
}
```

Cualquier aplicación creada que incluya la interacción con el teclado requiere que, al menos una vez, se haga clic con el mouse dentro de la zona del canvas para que funcione correctamente. En el ejemplo anterior se muestra la letra que se está presionando y se dibuja un círculo cada vez que se presiona una tecla.

Cada tecla del teclado tiene asignado un número o código, que la biblioteca **p5.js** llama **`keyCode`**, los números asociados a cada una de las teclas del abecedario puede verse a continuación:

a	65	**n**	78
b	66	**o**	79
c	67	**p**	80
d	68	**q**	81
e	69	**r**	82
f	70	**s**	83
g	71	**t**	84
h	72	**u**	85
i	73	**v**	86
j	74	**w**	87

k	75	x	88
l	76	y	89
m	77	z	90

El siguiente ejemplo hace uso del **`keyCode`** mostrando los valores asignados a cada tecla cuando se presiona cada una de ellas:

```
function setup() {
  createCanvas(400, 400);
  textAlign(CENTER,CENTER);
}

function draw() {
  background(220);
  textSize(40);
  text(keyCode,200,100);
  if(keyCode==66){
    circle(200,300,50);
  }
}
```

En el ejemplo anterior, si se presiona la tecla asociada al **`keyCode = 66`** (es decir, la tecla "b") se dibuja un círculo.

Existen otras formas de interactuar con el teclado a través de funciones específicas para ello, similares a las funciones de interactividad con el mouse. Entre ellas podemos nombrar dos: **`keyPressed()`**, que se activa cuando se presiona una tecla del teclado, y **`keyReleased()`**, que se activa cuando se deja de presionar la tecla. Un ejemplo que utiliza ambas funciones puede revisarse a continuación:

```
let control = 0;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  textSize(40);
  text(control, 200,200);
}

function keyPressed(){
  control = 1;
}

function keyReleased(){
  control = 0;
}
```

En el ejemplo anterior, al presionar una tecla, la variable **`control`** (que inicialmente tiene el valor asignado 0) toma el valor 1, y cuando se suelta vuelve a tomar el valor 0. Con la función **`text()`** se muestra el valor que toma la variable **`control`** en cada momento.

Las teclas de dirección (flechas) tienen asignados un nombre para cada una de ellas, que con uso de **`keyCode`** y de la función **`keyPressed()`** pueden utilizarse de la siguiente forma:

```
let texto = "";

function setup(){
  createCanvas(400,400);
}
function draw() {
 background(200);
 textSize(30);
 text(texto,200,200);
}

function keyPressed() {
  if (keyCode === UP_ARROW) {
```

```
    texto = "arriba";
  }
  if (keyCode === DOWN_ARROW) {
    texto = "abajo";
  }
  if (keyCode === LEFT_ARROW) {
    texto = "izquierda";
  }
  if (keyCode === RIGHT_ARROW) {
    texto = "derecha";
  }
}
```

Una forma que tiene **p5.js** para comprobar que una tecla está presionada es haciendo uso de la función **`keyIsDown()`**, como argumento de esta función se pueden incluir las palabras claves descritas en el ejemplo anterior (**`UP_ARROW`**, **`DOWN_ARROW`**, **`LEFT_ARROW`**, **`RIGHT_ARROW`**) o también el código asociado a cada una de las teclas (el **`keyCode`**).

```
let x=200;
let y=200;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220,20);
  if(keyIsDown(65)){
    fill(255,0,0);
  }else{
    fill(0);
  }
  circle(x,y,40);
  if(keyIsDown(LEFT_ARROW)){
    x-=5;
  }
}
```

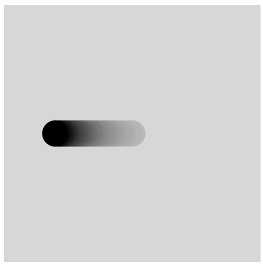


Figura 1.22. Control de movimiento con uso de teclado.

En el ejemplo anterior, si se presiona la tecla asociada al **`keyCode`** número 65 (es decir la tecla “**`a`**”), el círculo se pinta de color rojo, y cuando se suelta tal tecla el círculo vuelve a ser negro. Además, si se presiona la tecla de dirección izquierda, la figura se mueve a la izquierda mientras se mantiene presionada esa tecla y se detiene cuando se suelta.

Existen otras formas de interactuar con el teclado que dejamos al lector revisar en la sección de referencias de la biblioteca **p5.js**. Referencias indicadas en la introducción de este documento.

1.10. Tabla de funciones matemáticas

Función	Descripción	Ejemplo de uso
i++	Los dos signos + indican un incremento de 1 en la variable i	i++ es equivalente a i = i+1;
i--	Los dos signos - indican una disminución de 1 en la variable i	i-- es equivalente a i = i - 1;
i+=x	El valor de i se incrementa una cantidad x	si i=1; y i+=2; el nuevo valor de i es 3
i-=x	El valor de i disminuye una cantidad x	si i=1; y i-=2; el nuevo valor de i es -1
i*=x	El valor de i se multiplica una cantidad x	si i=2; y i*=3; el nuevo valor de i es 6

i/=x	El valor de i se divide una cantidad x	si i=8; y i/=2; el nuevo valor de i es 4
abs()	Devuelve el valor absoluto de un número	abs(-45) devuelve 45
ceil()	Aproxima el número a su entero superior	ceil(4.3) devuelve 5
dist(x1,y1,x2,y2)	Calcula la distancia entre dos puntos	dist(50,30,100,200) = 177.20045146669352
exp()	Calcula la función exponencial de un número	exp(10); devuelve 22026.465794806718
floor()	Aproxima el número a su entero inferior	floor(4.6) devuelve 4
log()	Calcula el logaritmo en base **e** de un número (logaritmo natural)	log(17); devuelve 2.833213344056216
%	Calcula el resto de una división	4 % 2 devuelve 0 4 % 3 devuelve 1
pow(a,n)	Calcula la potencia n de un número a	pow(4,3) equivale a 4^3: devuelve 64
round()	Redondea al entero más próximo	round(10.2) devuelve 10 round(10.6) devuelve 11
sq()	Calcula el cuadrado de un número	sq(3) equivale a 3^2: devuelve 9
sqrt()	Calcula la raíz cuadrada de un número	sqrt(121) equivale a $\sqrt{121}$: devuelve 11
random()	Entrega un valor aleatorio entre 0 y 1	random() p. ej: 0.2018147182620491
random(X)	Entrega un valor aleatorio entre 0 y X	random(200) p. ej: 32.01658902514245
random(a,b)	Entrega un valor aleatorio entre a y b	random(100,200) p. ej: 165.46520964388367
sin()	Devuelve el valor del seno de un	sin(PI/6) devuelve 0.5

	ángulo	
cos()	Devuelve el valor del coseno de un ángulo	cos(PI/6) devuelve 0.8660254037844387
tan()	Devuelve el valor de la tangente de un ángulo	tan(PI/6) devuelve 0.5773502691896257
asin()	Devuelve el valor, en radianes, del arcoseno de un valor	asin(0.5) devuelve 0.5235987755982988
acos()	Devuelve el valor, en radianes, del arcocoseno de un valor	acos(0.5) devuelve 1.0471975511965979
atan()	Devuelve el valor, en radianes, del arcotangente de un valor	atan(0.5) devuelve 0.4636476090008061
atan2(y,x)	Devuelve el valor, en radianes, del arcotangente, sería igual a atan(y/x)	atan(1,2) devuelve 0.4636476090008061

En la tabla anterior, todas las funciones asociadas a funciones trigonométricas asumen que los argumentos (y resultados, dado el caso) están, por defecto, en radianes. Si se desea trabajar en grados, se debe especificar con **`angleMode(DEGREES)`** dentro de la función **`setup()`**. Como se pudo observar en algunos códigos de ejemplos mostrados, el número π (pi) se puede utilizar con la palabra reservada **`PI`** (ambas letras en mayúscula) y tiene un valor igual a 3.14159…. Para obtener el valor de la mitad de π se puede utilizar **`HALF_PI`** (igual a π/2 = 1.57079…). Un cuarto del valor de π será **`QUARTER_PI`** (igual a 0.78539…) y dos veces el valor de π será **`TWO_PI`** (igual a 2π = 6.28318…).

Independiente de si se utiliza o no la opción **`angleMode(DEGREES)`** y sabiendo que 180° corresponden a π radianes, se puede pasar de una unidad de medida de ángulos a otra (de grados a radianes o de radianes a grados) utilizando la siguiente relación:

$$\frac{180}{\alpha} = \frac{\pi}{\beta}$$

donde α es el ángulo en grados y β es el ángulo en radianes.

La función '`%`', que calcula el resto en una división, permite utilizarse de una manera conveniente. Imagine que quiere realizar un conteo repetitivo, 0, 1, 2, 0, 1, 2,...y así sucesivamente. Podría tener una variable que vaya tomando los valores 0, 1 y 2, y que en vez de tomar el valor 3, vuelva a tomar el valor 0. Una forma de lograr lo anterior es tener una variable que crezca de uno en uno, desde 0 hasta un número grande y que en conjunto con la función '`%`' vaya entregando los valores 0, 1 y 2 esperados. Llamemos **`control`** a una variable que crece indefinidamente de 1 en 1 cuando se hace clic en el mouse, si utilizamos **`control%3`**, ésta expresión devolverá como resultado solamente los valores 0, 1 y 2, según vaya creciendo la variable **`control`**.

```
let control=0;

function setup() {
  createCanvas(400, 400);
  textAlign(CENTER,CENTER);
}

function draw() {
  background(220);
  textSize(20);
  text("control = "+control,width/2,40);
  text("control%3 = "+control%3,width/2,70);
  circle(control%3*100+100,300,50);
}

function mousePressed(){
  control++;
}
```

En el ejemplo anterior se muestra el valor de la variable control y el valor de **`control%3`**, además se ubica un círculo en el canvas, según los valores que toma la expresión **`control%3`**, en tres posiciones distintas. Si se necesita realizar un conteo de dos valores, 0, 1, 0, 1, … debe utilizarse la expresión **`control%2`**. Si se utiliza **`control%4`** se espera que se obtengan los valores 0, 1, 2, 3 de forma sucesiva.

1.11. Otra forma de ciclo

Otra estructura de programación que se puede utilizar es a través del ciclo **while.** Estos ciclos permiten ejecutar una, o varias acciones, mientras se cumpla una condición, por ejemplo:

```
let n=10;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  while(n>0){
    n=n-1;
    print(n);
  }
}
```

En el ejemplo anterior, lo que se encuentra dentro del **while()** se ejecutará mientras el valor de **n** sea mayor que 0. Dentro del ciclo, el valor **n** disminuye su valor de 1 en 1, comenzando en 10, y se va mostrando en consola el valor de **n**. Cuando **n** toma el valor 1, su valor cambiará a 0 dentro del **while()**, y como **n** en este momento ya no es mayor que 0, el ciclo dejará de ejecutarse y el programa continuará su ejecución después del **while()**.

1.12. Geometrías en 3D

Las formas geométricas revisadas con anterioridad permiten plasmar en el canvas figuras en dos dimensiones (2D) en un plano **xy**. Pero también existe la posibilidad de incorporar una tercera dimensión al canvas, el eje **z**. Para incluir esta opción es necesario incluir a la función **createCanvas()** un tercer parámetro (**WEBGL**), por ejemplo: **createCanvas(512,512,WEBGL)**. Una vez que se incluye esta opción, el origen del sistema de referencia cambia al centro del canvas, el eje **x**

sigue apuntando hacia la derecha, el eje **y** hacia abajo y el nuevo eje **z** apunta hacia afuera, es decir, hacia quien mira la pantalla.

Algunas geometrías predefinidas pueden utilizarse con esta opción, entre ellas esfera (**sphere**), caja (**box**), cilindro (**cylinder**) y cono (**cone**). Cuando se ubican cualquiera de estas figuras en el canvas, estas se muestran en el origen del sistema de referencia (en el centro del canvas por defecto), por lo que si se requiere ubicar estas geometrías en lugares distintos, deben utilizarse en compañía de la función **translate()**, que en esta oportunidad requerirá de tres parámetros para el origen del eje **x**, origen del eje **y** y origen del eje **z**, respectivamente. Además, es posible que sea necesario ubicar la función **translate()** y el objeto a mostrar entre las opciones **push()** y **pop()**, para representar las distintas figuras. Un ejemplo del uso de la opción **WEBGL** se puede revisar en el siguiente código:

```
function setup(){
    createCanvas(512,512,WEBGL);
}

function draw(){
    background(220,0,40);
    rotateY(map(mouseX,0,width,0,4*PI));
    rotateZ(map(mouseY,0,height,0,4*PI));
    directionalLight(250,250,250,-1,-1,-1);

    push();
    translate(-100,-300,-300);
    fill(0,0,300);
    box(40,60);
    pop();

    push();
    translate(100,-300,-300);
    fill(255,255,255,60);
    box(40,60);
    pop();

    push();
```

```
    translate(100,0,-300);
    sphere(40);
    pop();

    push();
    translate(-100,0,-300);
    fill(0,255,0);
    cylinder(20,50);
    pop();

    push();
    translate(0,-50,100);
    cone(40,50);
    pop();
}
```

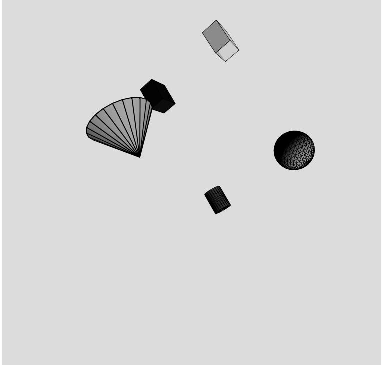


Figura 1.23. Representación de geometrías en 3D.

Cuando en el canvas no se utiliza la opción **WEBGL**, la función **rotate()** gira el objeto en torno al origen del sistema de referencia. Pero cuando se utiliza **WEBGL**, lo que se muestra en el canvas se puede rotar con respecto a cada uno de los ejes, con las funciones **rotateX()**, **rotateY()** y **rotateZ()**, que reciben como parámetro un ángulo en radianes por defecto.

Además, con la inclusión de un tercer eje **z**, se pueden incluir opciones de iluminación. En el ejemplo, se utiliza la función

`directionalLight(250,250,250,-1,-1,-1)`, cuyos tres primeros parámetros son los valores RGB del color de la iluminación y los últimos tres parámetros corresponden a las componentes **`x`**, **`y`** y **`z`** de la dirección en la que apuntará la luz. Esta función debe utilizarse antes de los objetos a iluminar.

Al utilizar la opción **`WEBGL`** podrá seguir generando las geometrías que se revisaron anteriormente, considerando que la función **`line()`**, debe recibir las componentes **`x`**, **`y`** y **`z`** de la posición inicial y final (**`line(x1,y1,z1,x2,y2,z2)`**), las demás geometrías (**`circle()`**, **`rect()`**, **`ellipse()`**) se pueden seguir utilizando como se revisó con anterioridad.

1.13. El tiempo en p5.js

La biblioteca **p5.js** posee algunas funciones que permiten obtener información del tiempo, **`hour()`**, **`minute()`**, **`second()`**, para la hora, minuto y segundos actuales, respectivamente.

Además se puede conocer la cantidad de milisegundos que lleva el programa ejecutándose utilizando la función **`millis()`**.

```
let h;
let m;
let s;

function setup() {
  createCanvas(400, 400);
  textAlign(CENTER,CENTER);
}

function draw() {
  background(220,255,50);
  h=hour();
  m=minute();
  s=second();
  textSize(60);
  text(nf(h,2)+":"+nf(m,2)+":"+nf(s,2), width/2,200);
  textSize(20);
  text(millis(),width/2,300);
}
```

El ejemplo anterior obtiene los valores de hora, minuto y segundos actuales y los muestra en el centro del canvas (la función **`nf()`** se explica en la sección 3.2.). Además, debajo de la hora actual, se muestra la cantidad de milisegundos que han transcurrido desde que se inicia la ejecución del programa utilizando la función **`millis()`**. Si se requiriera ver el detalle de la cantidad de segundos que han transcurrido podría utilizarse **`millis()/1000`**, ya que 1000 milisegundos corresponden a 1 segundo.

Las opciones anteriores de tiempo se pueden aplicar para la creación de un programa que funcione como un cronómetro, con control del teclado y/o con el control del mouse:

```
let ta = 0;
let tb = 0;
let tiempo;
let control = 0;
let texto='Iniciar cronómetro';

function setup() {
  createCanvas(512, 512);
  textAlign(CENTER);
}

function draw() {
  background(147, 252, 61);
  textSize(20);
  text(texto,width/2,80);
  tiempo=(tb-ta)/1000;
  if(tiempo>0){
  push();
  textSize(40);
  text(round(1000*tiempo)/1000+ " [s] ", width/2,120);
  pop();
  }
  text("Son las "+hour()+" horas, con "+minute()+" minutos y
"+second()+" segundos.",width/2,240);
}

function keyReleased() {
  control++;
}

function mouseReleased() {
  control++;
}

function keyPressed() {
   if (control % 2 == 0) {
    ta = millis();
    texto='Detener';
  }
  if (control % 2 == 1) {
    tb = millis();
    texto='Iniciar cronómetro';
  }
```

```
  return false;
}

function mousePressed() {
   if (control % 2 == 0) {
    ta = millis();
    texto='Detener';
  }
  if (control % 2 == 1) {
    tb = millis();
    texto='Iniciar cronómetro';
  }
}
```

Iniciar cronómetro
0.312 [s]

Son las 18 horas, con 49 minutos y 49 segundos.

Figura 1.24. Creación de un cronómetro con p5.js.

1.14. Animaciones

Considerando que todo lo que se encuentra dentro de la función **`draw()`** se desarrolla de forma cíclica, es posible generar animaciones para mover las figuras que se muestran en el canvas. Por ejemplo, si se desea que un círculo oscile en el canvas de un lugar a otro, podría incluirse una variable que vaya cambiando cada vez que se ejecute la función **`draw()`** y esta variable hará cambiar la posición del objeto. Esta variable podría denominarse **`t`** (para asimilar con la t de tiempo) y la

posición **`x`** del objeto que se mueve se le asigna, por ejemplo, la expresión:

`x = 100 cos(t) + 200`

La función *cos(t)* (coseno de t) es necesaria, porque la función *cos()* es una función oscilante entre valores -1 y 1, por lo tanto, al multiplicar por 100, variará entre -100 y 100 (y entre 100 y 300 al sumarle 200), también podría utilizarse la función *sen()* (**`sin()`**) que también toma valores entre -1 y 1. Entonces para que el objeto se mueva, el valor **`t`** deberá ir cambiando con una expresión como la siguiente: **`t=t+0.1`**, esto quiere decir que **`t`** tomará el valor inicial asignado y, luego, cambiará en una cantidad 0.1, si inicialmente **`t`** se declara con un valor igual a 0, en la primera pasada del ciclo **`draw()`** tomará un valor igual a 0.1, en la segunda pasada tomará el valor 0.2, en la tercera pasada el valor 0.3 y así sucesivamente.

Es importante destacar que la variable **`t`**, si bien está asociada al tiempo, no significa que el desarrollo de la animación se realice en tiempo real, es decir, si **`t=1`**, no significa un segundo, solo significa una unidad de tiempo determinada y generalmente una unidad de tiempo desconocida. Si desea realizar animaciones en tiempo real puede utilizar la función **`millis()`**, y asignar este valor a una variable global **`t`**.

Considere que escribir **`t=t+0.1`** se puede escribir de forma un poco más abreviada como: **`t+=0.1`**, por lo que ambas formas son equivalentes. Si, por ejemplo, **`t`** se declarara de la siguiente forma: **`t=t+0.4`**, en este caso **`t`** irá cambiando de 0.4 en 0.4 por cada pasada de la función **`draw()`** y la oscilación, en el ejemplo anterior, será más rápida.

El siguiente ejemplo moverá un círculo de forma oscilante y horizontal con respecto al centro del canvas:

```
let t=0;
let x;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(10,255,70,20);
  x=100*cos(t)+200;
  circle(x,200,20);
  t=t+0.1;
}
```

Siempre tendrá la opción de que un objeto se mueva de acuerdo a las funciones que se consideren necesarias, puede hacer que un objeto se mueva de forma parabólica, elíptica o cualquier otra curva que le parezca bien representar en un movimiento. Puede intentar cambiando la expresión **`x = 100*cos(t)+200`** por otra función matemática, e incluso puede declarar una variable **`y`** y asignarle una función matemática a esta variable para realizar un animación en dos dimensiones en el plano **`xy`**. Cualquier movimiento que se estudia en cinemática puede representarse.

Un movimiento circular podría animarse rotando un objeto con respecto al origen del sistema de referencia. Sería conveniente primero, trasladar el origen del sistema de referencia al centro del canvas con **`translate(width/2,height/2)`**, y luego dibujar el objeto en cierta posición, pero justo antes de dibujar este objeto se utiliza la función **`rotate()`** cuyo argumento va a ir cambiando por cada pasada de la función **`draw()`**.

El siguiente ejemplo mostrará el movimiento circular de un círculo con respecto al centro del canvas:

```
let t=0;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(0,55,170,20);
  translate(width/2,height/2);
  rotate(t);
  circle(100,0,20);
  t=t+0.1;
}
```

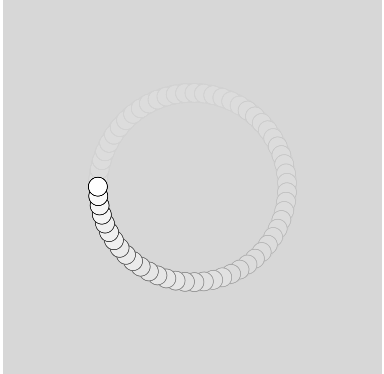


Figura 1.25. Animación de un movimiento circular con respecto al centro del canvas..

En el ejemplo anterior el círculo realiza giros en sentido horario, si se quisiera que girara en sentido antihorario debería cambiarse la línea **`t=t+0.1`** por **`t=t-0.1`** (o **`t-=0.1`**).

Resumen

En este capítulo, hemos abordado varios temas importantes. Comenzamos discutiendo la declaración de variables tanto a nivel global como local. Además, exploramos diversas opciones para crear figuras geométricas en el canvas. También revisamos funciones proporcionadas

por la biblioteca, como condicionales y ciclos. Al combinar estas herramientas con la capacidad de interactuar con el mouse y el teclado, ahora tendrá la capacidad de desarrollar aplicaciones interactivas que pueden ser extremadamente útiles para sus objetivos personales, académicos y posiblemente para sus estudiantes.

A partir de este punto, puede dar vida a sus aplicaciones incorporando la variable tiempo, lo que le permite crear animaciones con figuras en movimiento, incluso en tiempo real. Además, aprovechando las funciones matemáticas disponibles, tendrá la capacidad de generar gráficos matemáticos puros o relacionados con situaciones físicas.

Algunas sugerencias de programas a realizar son:

- Crear un programa que anime un problema de física de encuentro de dos cuerpos.
- Crear un programa que mueva una figura, cuya velocidad se controle con el movimiento del mouse o con las teclas de dirección del teclado.
- Crear un programa que permita graficar distintas funciones matemáticas, alterando sus características con el movimiento del mouse y teclado.
- Crear un programa que controle los vértices de un triángulo, y que se calculen los ángulos interiores y algunos de los lados faltantes del mismo.
- Crear un programa que plasme en el canvas un dibujo en el que se permita alterar el color y dimensiones con el movimiento del mouse.
- Crear un programa que haga modelos en 3D de moléculas de algunos compuestos químicos.
- Crear un programa en el que se visualice una simulación de un reloj analógico con segundero, minutero y horario.

Capítulo 2. Vectores

En el ámbito de la física, un vector se define como una magnitud física que posee tanto magnitud, dirección y sentido. Estos vectores son ampliamente empleados en diversas aplicaciones físicas, desempeñando un papel fundamental en el análisis y descripción de numerosos fenómenos. Entre las magnitudes que suelen expresarse mediante vectores se encuentran la velocidad, la aceleración, la fuerza, y muchas otras, cubriendo un amplio espectro de aplicaciones.

La utilidad que tienen los vectores en programación es permitir almacenar las distintas componentes del mismo, bajo un mismo nombre. Estas componentes, en un plano cartesiano, son llamadas componente **`x`** y componente **`y`** (y componente **`z`** si se trabaja en tres dimensiones). Todos los ejemplos que revisaremos a continuación se realizan en un plano **`xy`**, en dos dimensiones, pero se puede extender a un tercer eje **`z`** si se añade esta tercera componente al declarar cada vector.

Para declarar un vector en dos dimensiones es necesario utilizar el objeto **`createVector(valorx, valory)`**, donde **`valorx`** y **`valory`** son las componentes del vector a definir, un ejemplo concreto de definición de un vector sería:

```
let v;

function setup() {
  createCanvas(400, 400);
  v=createVector(3,5);
}

function draw() {
  background(220);
  text("componente x = "+v.x,40,40);
  text("componente y = "+v.y,40,60);
}
```

Se puede observar en el ejemplo, que para acceder a cada una de las componentes del vector **v**, se debe utilizar **v.x** para la componente **x** y **v.y** para la componente **y**.

2.1. Operaciones con vectores

Vamos a realizar distintas operaciones con vectores, para ello consideraremos dos vectores **v1** y **v2**. Las componentes de estos vectores serán **v1.x** y **v1.y** para el vector **v1** y **v2.x** y **v2.y** para el vector **v2**. Todos los ejemplos que siguen en esta sección se desarrollan, por simplicidad, dentro de la función **setup()**, pero nada impide que pueda incluir estas operaciones dentro de la función **draw()**.

- **Suma:**

Si se desea sumar ambos vectores, la componente **x** de la suma será **v1.x+v2.x** y la componente **y** será **v1.y+v2.y**. Existen dos formas de implementar una suma de dos vectores, con una diferencia entre ambas opciones, la primera de ellas (**v1.add(v2)**) se puede revisar en el siguiente ejemplo:

```
let v1;
let v2;

function setup() {
  createCanvas(400, 400);
  background(220);
  v1=createVector(3,5);
  v2=createVector(4,8);

  v1.add(v2);
}
```

Esta forma de sumar vectores asigna al vector **v1** el resultado de la suma de los vectores **v1+v2**. Es decir, guarda el resultado (sobreescribe) en el vector **v1**. Podría decirse que equivale a escribir “**v1=v1+v2**”, aunque esta forma no está aceptada para operaciones con vectores.

Si se desea que la suma de **`v1`** con **`v2`** no sobreescriba ninguna de los dos vectores que se están sumando, debe utilizarse **`p5.Vector.add(v1,v2)`**, y asignarle a un nuevo vector (**`v3`** por ejemplo) el resultado de esta suma:

```
let v1;
let v2;
let v3;

function setup() {
  createCanvas(400, 400);
  background(220);
  v1=createVector(3,5);
  v2=createVector(4,8);

  v3=p5.Vector.add(v1,v2);
}
```

- **Resta:**

Si se desea restar dos vectores (**`v1-v2`**), la componente **`x`** de la resta será **`v1.x-v2.x`** y la componente **`y`** será **`v1.y-v2.y`**. Al igual que en la suma, existen dos formas de implementar una resta de dos vectores, la primera de ellas (**`v1.sub(v2)`**) se puede revisar en el siguiente ejemplo:

```
let v1;
let v2;

function setup() {
  createCanvas(400, 400);
  background(220);
  v1=createVector(3,5);
  v2=createVector(4,8);

  v1.sub(v2);
}
```

Esta forma de restar vectores asigna al vector **`v1`** el resultado de la resta de los vectores **`v1-v2`**. Es decir, guarda el resultado (sobreescribe) en el

vector **`v1`**. Si se utiliza **`v2.sub(v1)`** se obtiene la resta **`v2-v1`** y se sobreescribe el vector **`v2`**.

Si se desea que la resta de **`v1`** con **`v2`** no sobreescriba ninguno de los dos vectores que se están restando, debe utilizarse **`p5.Vector.sub(v1,v2)`**, y asignarle a un nuevo vector el resultado de esta resta:

```
let v1;
let v2;
let v3;

function setup() {
  createCanvas(400, 400);
  background(220);
  v1=createVector(3,5);
  v2=createVector(4,8);

  v3=p5.Vector.sub(v1,v2);
}
```

- **Multiplicación:**

Esta primera forma que revisaremos de multiplicar un vector, será multiplicándolo por un escalar (un número), si **`v1`** es un vector, al multiplicar este por un escalar **`n`** (**`v1.mult(n)`**), cada componente del vector será multiplicada por este factor **`n`**, es decir **`n`** por **`v1`** entregará componentes **`n*v1.x`** y **`n*v1.y`** para las componentes **`x`** e **`y`**, respectivamente.

```
let v1;

function setup() {
  createCanvas(400, 400);
  background(220);
  v1=createVector(3,5);

  v1.mult(4);
}
```

De la misma forma que para las operaciones anteriores, esta forma de calcular le asigna el resultado al vector **v1**, que en el ejemplo tendrá las componentes 12 y 20, respectivamente. Para asignar el resultado a otro vector, sin alterar **v1**, puede utilizarse **v3=p5.Vector.mult(v1,4)**.

La misma función que permite multiplicar un vector por un escalar, permite multiplicar dos vectores, y cada componente de este tipo de multiplicación será la multiplicación de las componentes de los vectores que se multiplican. Es decir, si multiplica los vectores **v1** por **v2**, las componentes **x** e **y** de la multiplicación serán **v1.x*v2.x** y **v1.y*v2.y**, respectivamente. La sintaxis a utilizar es la misma que la explicada anteriormente para multiplicar un vector por un escalar. En el ejemplo que sigue, se le asigna a un vector **v3** el resultado de multiplicar **v1** por **v2**.

```
let v1;
let v2;
let v3;

function setup() {
  createCanvas(400, 400);
  background(220);
  v1=createVector(3,5);
  v2=createVector(4,8);

  v3=p5.Vector.mult(v1,v2);
}
```

- **División:**

La división funciona de forma análoga a como se utiliza la función para la multiplicación. Para comenzar se dividirá por un escalar (un número), si **v1** es un vector, al dividir este por un escalar **n**, cada componente del vector será dividida por este valor **n**, es decir, **v1** dividido en **n** entregará componentes **v1.x/n** y **v1.y/n** para las componentes **x** e **y**, respectivamente.

```
let v1;

function setup() {
  createCanvas(400, 400);
  background(220);
  v1=createVector(3,5);

  v1.div(4);
}
```

De la misma forma que para las operaciones anteriores, esta forma de calcular le asigna el resultado al vector **`v1`**, que en el ejemplo tendrá las componentes 0.75 y 1.25, respectivamente. Para asignar el resultado a otro vector, sin alterar **`v1`**, puede utilizar **`v3=p5.Vector.div(v1,4)`**.

La misma función que permite dividir un vector por un escalar, permite dividir dos vectores, y cada componente de esta división será la división de las componentes de los vectores que se dividen entre sí. Es decir, si divide los vectores **`v1`** en **`v2`**, las componentes **`x`** e **`y`** de la división serán **`v1.x/v2.x`** y **`v1.y/v2.y`**, respectivamente. La sintaxis a utilizar es la misma que la explicada anteriormente para dividir un vector por un escalar. En el ejemplo que sigue, se le asigna a un vector **`v3`** el resultado de dividir **`v1`** por **`v2`**.

```
let v1;
let v2;
let v3;

function setup() {
  createCanvas(400, 400);
  background(220);
  v1=createVector(3,5);
  v2=createVector(4,8);

  v3=p5.Vector.div(v1,v2);
}
```

- **Producto punto:**

Las formas revisadas anteriormente, que multiplican dos vectores entre sí y que dividen dos vectores entres sí, son formas que la biblioteca **p5.js** las acepta, pero que matemáticamente no están definidas. Sí está definida matemáticamente la siguiente forma de multiplicar vectores, a través de un producto punto, y que además tiene aplicación en la física.

Realizar un producto punto entre dos vectores **`v1`** y **`v2`** entregará un número (un escalar) que será calculado de la siguiente forma: **`v1.x*v2.x+v1.y*v2.y`**. La forma más común de implementar esto sería asignándole el resultado de esta operación (**`v1.dot(v2)`**) a una variable distinta:

```
let v1;
let v2;
let p;

function setup() {
  createCanvas(400, 400);
  background(220);
  v1=createVector(3,5);
  v2=createVector(4,8);

  p=v1.dot(v2);
}
```

En el ejemplo, el resultado del producto punto que se le asignó a la variable **`p`** toma el valor 52.

Como los vectores son magnitudes con dirección, se puede utilizar este tipo de producto para verificar si son o no perpendiculares entre sí. Se puede demostrar matemáticamente que dos vectores que son perpendiculares, al multiplicarse a través de un producto punto, da como resultado cero (0).

En física, el producto punto se utiliza para calcular muchas cantidades físicas, por ejemplo el flujo eléctrico, el flujo magnético, entre otras.

- **Producto cruz:**

El producto cruz es otra forma de multiplicar vectores entre sí, que sí está definida matemáticamente. Pero esta vez el resultado será un nuevo vector que será perpendicular a cada uno de los vectores que se están multiplicando, por lo tanto, si **`v1`** y **`v2`** solo tienen componentes en el plano **`xy`**, su producto cruz deberá tener necesariamente una componente en el eje **`z`**. Las componentes **`x`**, **`y`** y **`z`** del producto cruz entre dos vectores **`v1`** y **`v2`** son, respectivamente **`v1.y*v2.z-v1.z*v2.y`**, **`v1.z*v2.x-v1.x*v2.z`** y **`v1.x*v2.y-v1.y*v2.x`**. Si **`v1`** y **`v2`** no se definen con componente **`z`**, entonces **`v1.z`** y **`v2.z`** serán igual a cero.

La forma de implementar este tipo de producto es **`v1.cross(v2)`**, se puede revisar en el siguiente ejemplo:

```
let v1;
let v2;
let v3;

function setup() {
  createCanvas(400, 400);
  background(220);
  v1=createVector(3,5);
  v2=createVector(4,8);

  v3=v1.cross(v2);
}
```

En el ejemplo anterior, el vector **`v3`** tendrá componentes nulas (cero) en los ejes **`x`** e **`y`** (**`v3.x=0`** y **`v3.y=0`**), y tendrá un valor 4 en la componente **`z`** (**`v3.z=4`**).

Algunas de las magnitudes físicas que se calculan a partir de un producto cruz son la aceleración tangencial, la aceleración centrípeta, el torque, el campo magnético, entre otras.

- **Magnitud:**

La magnitud de un vector indica el tamaño de un vector y es un escalar, si **v1** es un vector, la forma de calcular su magnitud es la siguiente:

$$\sqrt{v1x^2 + v1y^2}$$

La expresión anterior se podría implementar fácilmente en el código, pero la biblioteca nos da una opción de calcularla a partir de una función definida para ello (**v1.mag()**):

```
let v1;
let m;

function setup() {
  createCanvas(400, 400);
  background(220);
  v1=createVector(3,5);

  m=v1.mag();
}
```

- **Normalizar:**

Como se pudo observar en el cálculo de la magnitud de un vector, la magnitud depende de las componentes **x** e **y** del vector. Normalizar significa obtener un nuevo vector que tenga la misma dirección que el vector a normalizar, pero con magnitud uno (1), llamado vector unitario. Esto se puede lograr dividiendo cada componente del vector por la magnitud del vector, y podría implementarse con las operaciones descritas anteriormente, pero existe una función especial para realizar esta acción (**v1.normalize()**).

```
let v1;

function setup() {
  createCanvas(400, 400);
  background(220);
  v1=createVector(3,5);

  v1.normalize();
}
```

En el ejemplo anterior, se actualizan las componentes de **v1** a 0.5144... para la componente **x** y 0.8574… para la componente **y**. Si se desea asignar la normalización a un nuevo vector, sin alterar las componentes de **v1**, puede realizarse la siguiente operación:

```
let v1;
let v3;

function setup() {
  createCanvas(400, 400);
  background(220);
  v1=createVector(3,5);

  v3=p5.Vector.normalize(v1);
}
```

En este caso, **v1** mantiene sus componentes 3 y 5, y **v3** adquiere las componentes luego de realizar la normalización de **v1**.

- **Ángulo entre vectores:**

Dados dos vectores **v1** y **v2**, que tienen cada uno una dirección determinada por sus componentes, se puede calcular el ángulo que existe entre ellos, este ángulo tendrá un valor entre -180° y 180° (o su equivalente en radianes). Para calcular el ángulo entre vectores se utiliza la función **v1.angleBetween(v2)**, que calculará el menor ángulo que existe entre ambos vectores. Si se gira el vector **v1** hacia el vector **v2** por el menor ángulo en sentido horario, entonces el ángulo será positivo. Si se gira el vector **v1** hacia el vector **v2** por el menor ángulo en sentido antihorario, entonces el ángulo será negativo.

```
let v1;
let v2;
let a;

function setup() {
  createCanvas(400, 400);
  angleMode(DEGREES);
  background(220);
  v1=createVector(5,0);
  v2=createVector(-5,3);
  a=v1.angleBetween(v2);
}
```

En el ejemplo, el valor **`a`**, del ángulo entre los vectores, será aproximadamente igual a 149°. Si el ángulo se hubiera calculado de la siguiente manera: **`v2.angleBetween(v1)`**, el ángulo que se hubiera obtenido sería negativo y cercano a -149°. Note que se utiliza la opción **`angleMode(DEGREES)`** para entregar el valor del ángulo en grados, si no se utiliza esta opción, el ángulo se entregará en radianes.

Resumen

En este capítulo, exploramos algunas de las operaciones con vectores que la biblioteca **p5.js** ofrece. Esto nos brinda la capacidad de realizar cálculos numéricos que, de otro modo, serían más complicados y requerirían una implementación más extensa. El trabajo con vectores se destaca como especialmente beneficioso, especialmente cuando se necesita utilizar las componentes **`x`** e **`y`** del sistema de referencia del canvas para el funcionamiento de la aplicación.

Algunos sugerencias de programas que se pueden implementar con vectores son:

- Crear un programa en el que se puedan dibujar dos o más vectores y calcule y muestre el valor de la suma de esos vectores (la resultante).
- Crear un programa que indique que dos vectores son perpendiculares o no.

- Crear un programa en el que se ingrese una variedad de fuerzas y calcular con esto la aceleración que adquiere una masa utilizando la segunda ley de Newton.
- Crear un programa que represente visualmente la velocidad y la aceleración de un objeto mientras se mueve en el canvas.
- Crear un programa que calcule el volumen de un paralelepípedo definido por tres vectores.
- Crear un programa que calcule el área de un triángulo dados los vértices del triángulo.
- Crear un programa que permita calcular el torque dada la fuerza y la posición donde se aplica esta fuerza.

Capítulo 3. Arreglos, imágenes, audio y video

3.1. Arreglos

Imagínese una caja con compartimentos vacíos con la idea de ir ubicando elementos, uno a uno, en cada uno de sus espacios y de forma ordenada. Esta caja con sus elementos en sus distintos compartimentos es lo que se asemeja a un arreglo en programación. Por lo tanto, un arreglo podría definirse como una estructura que permite guardar varias variables bajo un mismo nombre.

Es importante señalar que los elementos que se pueden incluir dentro de un arreglo son números, textos, imágenes, sonidos, objetos, por nombrar algunos, y pueden estar mezclados dentro de un mismo arreglo.

El siguiente es un ejemplo de una caja que representa un arreglo, en cada posición se le asigna un elemento distinto. El primer elemento de un arreglo siempre se comienza a contar con el índice 0.

Índice	0	1	2	3	4
Elemento	"manzana"	"pera"	"naranja"	"sandía"	"plátano"

El arreglo anterior tiene 5 elementos, a pesar de que el último elemento posee un índice igual a 4. La forma de declarar un arreglo con los elementos indicados en la tabla anterior es:

```
let frutas=["manzana","pera","naranja","sandía","plátano"];
```

Si se quisiera acceder a los elementos de este arreglo de forma individual, se utiliza el nombre del arreglo acompañado de paréntesis cuadrados (**[]**) y al interior de estos paréntesis se ingresa el índice que corresponde dentro del arreglo, por ejemplo, para acceder a la variable **"sandía"** se debe utilizar **frutas[3]**. Para acceder a la cantidad de elementos que tiene un arreglo se puede utilizar la función **frutas.length** (nombre del

arreglo acompañado de **`.length`**), considerando esto, se puede acceder al último elemento del arreglo utilizando **`frutas[frutas.length-1]`**.

El ejemplo anterior es un arreglo, el cual se declara su nombre e inmediatamente se le asignan algunos elementos dentro de él. En ciertos casos es necesario crear un arreglo vacío (como si fuera una caja vacía) para ir llenándola de a poco, la forma de declarar un arreglo vacío es la siguiente:

```
let numeros=[];
```

Una vez que tenemos el arreglo definido podemos comenzar a introducir valores dentro del arreglo (en este caso el arreglo tiene el nombre "**`numeros`**"). Como el primer elemento del arreglo tiene un índice 0, puedo asignar el primer valor del arreglo de la siguiente forma **`frutas[0]=random(100)`**. De esta forma el primer elemento del arreglo "**`numeros`**" tomará un valor aleatorio entre 0 y 100. Si quiero añadir otro elemento al arreglo, se podría escribir **`frutas[1]=random(50)`**, es decir, el segundo valor del arreglo tomará un valor aleatorio entre 0 y 50. Ir ubicando los elementos en el arreglo de forma manual, como se ha especificado, si bien es posible hacerlo, es más eficiente hacerlo a partir de un ciclo, por ejemplo:

```
let numeros=[];

function setup(){
   createCanvas(400, 400);
   background(200);
   for(let i=0; i<100; i++){
      numeros[i]=floor(random(500));
   }
}
```

De esta forma el arreglo "**`numeros`**" se llenará a través de un ciclo **`for()`** con 100 elementos, y en cada posición habrá un número aleatorio entero entre 0 y 500. En este momento tenemos el arreglo "**`numeros`**" creado y sería útil alterar estos elementos, como, por ejemplo, añadir y quitar elementos.

Para añadir elementos al arreglo ya creado se utiliza la función **`.push()`**, esta función añade un elemento al final del arreglo, en la última posición, independiente de la cantidad de elementos que tenga el arreglo, la forma de utilizar la función **`.push()`** es **`numeros.push(floor(random(200)))`**. De esta forma se añade un número entero aleatorio entre 0 y 200 al final del arreglo llamado "**`numeros`**". Esta forma de añadir elementos es útil cuando no se sabe la cantidad de elementos que tiene el arreglo.

Si se desea borrar el último elemento del arreglo se utiliza la función **`.pop()`** de la siguiente forma: **`numeros.pop()`**. Esta forma de quitar el último elemento es útil cuando no se sabe la cantidad de elementos que tiene el arreglo.

Si se opta por borrar **`n`** elementos del arreglo desde la posición indicada con el índice **`i`**, se utiliza la función **`.splice(i,n)`**. Por ejemplo, si se desea eliminar el primer elemento del arreglo se utilizaría **`numeros.splice(0,1)`**, 0 porque quiero borrar el primer elemento y 1 porque quiero borrar solo un elemento. Al utilizar esta opción, todos los elementos que no fueron eliminados, y con un índice superior al mayor índice eliminado, se mueven a una nueva ubicación (a otro índice más bajo), ubicándose en las posiciones que ocupaban los elementos eliminados.

Un código que involucra las opciones revisadas se muestra a continuación. En el ejemplo, presionando el botón del mouse en 4 diferentes ubicaciones del canvas, se realizan 4 diferentes acciones:

- **a.** En la esquina superior izquierda: se irán añadiendo elementos al arreglo.
- **b.** En la esquina superior derecha: se eliminará el último elemento del arreglo.
- **c.** En la esquina inferior izquierda: se eliminará el primer elemento del arreglo.
- **d.** En la esquina inferior derecha: se eliminarán 5 elementos del arreglo desde el índice 1**.**

```
let numeros=[];

function setup(){
   createCanvas(400, 400);
   background(200);
   for(let i=0; i<100; i++){
      numeros[i]=floor(random(500));
   }
}

function draw() {
  background(220,250,0);
  line(width/2,0,width/2,height);
  line(0,height/2,width,height/2);
  textSize(10);
  text("añade un elemento al final",20,20);
  text("borra el último elemento",220,20);
  text("borro el primer elemento",20,220);
  text("borro varios elementos",220,220);
}

function mousePressed(){
  if(mouseX<width/2 && mouseY<height/2){
    numeros.push(floor(random(200)));
}
  if(mouseX>width/2 && mouseY<height/2){
    numeros.pop();
  }

  if(mouseX<width/2 && mouseY>height/2){
    numeros.splice(0,1);
  }

  if(mouseX>width/2 && mouseY>height/2){
    numeros.splice(1,5);
  }
}
```

añade un elemento al final
borra el último elemento
borro el primer elemento
borro varios elementos

Figura 3.1. Añadir y eliminar elementos de un arreglo.

Estas funciones para generar y alterar los elementos de un arreglo no son exclusivas de la biblioteca **p5.js**, sino que pertenecen a JavaScript. Otras funciones que pueden utilizarse con los arreglos se pueden acceder a través de un navegador web[5].

3.2. Imágenes

La biblioteca **p5.js** permite cargar y mostrar imágenes en el canvas, para lograr esto, lo primero que es necesario realizar, es subir un archivo que contenga una imagen (que puede ser en formato **`"jpg"`** o **`"png"`** por ejemplo), en el editor del sitio oficial de **p5.js**[6] esta acción se puede realizar haciendo clic en el lugar que muestra la figura 3.2.:

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array Referencias de arreglos en JavaScript.
[6] https://editor.p5js.org/

Figura 3.2. ¿Cómo subir un archivo al editor de p5.js?

Una vez subida la imagen, quedará disponible para hacer uso de ella en el lugar del código que sea necesario. Esta forma de subir un archivo es la misma para diferentes tipos de archivo, por ejemplo, para subir un archivo de texto (“**`*.txt`**”), un archivo de audio (“**`*.mp3`**” o “**`*.wav`**”), un archivo de datos o para cualquier otro tipo de archivo. Si ud. está programando con otros editores de código, como VScode o a través de un servidor, tendrá que ubicar el archivo en la misma carpeta donde está hospedado su archivo “**`*.js`**”.

De igual manera, en el código del programa que se está escribiendo es recomendable utilizar una función que permite que el archivo sea cargado antes de seguir con el funcionamiento del programa, sobre todo si se trata de archivos grandes (que ocupan mucha memoria), esta función es **`preload()`** y es importante porque asegura que todos los archivos estén disponibles para cuando se necesiten en el código. Si no se utiliza esta opción, el programa podría presentar algunos problemas al acceder a los distintos archivos.

La forma más sencilla de mostrar una imagen se muestra a continuación:

```
let img;

function preload(){
    img=loadImage('archivo.png');
```

```
}

function setup(){
    createCanvas(400,400);
}

function draw(){
    background(0);
    image(img,0,0);
}
```

Figura 3.3. Mostrar una imagen en el canvas.

En el ejemplo anterior, se utilizó la función **`preload()`** para cargar la imagen con la función **`loadImage()`**, que recibe como parámetro el nombre del archivo subido entre comillas, respetando mayúsculas y minúsculas en el nombre. Dentro de la función **`draw()`** se utiliza la función **`image()`**, que es la que muestra la imagen en el canvas. Esta función **`image()`** requiere, al menos, tres parámetros, el primero es el nombre de la variable que contiene la imagen y el segundo y tercer parámetro determinan la posición de la esquina superior izquierda de la imagen. Si se requiere que la imagen se muestre con respecto a su centro puede utilizarse la opción **`imageMode(CENTER)`** dentro de **`setup().`** Para acceder al ancho y alto de la imagen cargada, en píxeles, se utilizan las funciones **`imagen.width`** y **`imagen.height`**, respectivamente.

A la función **`image()`** pueden añadirse un cuarto y quinto parámetro, estos parámetros indican la posición de la esquina inferior derecha de la

imagen, por ejemplo, si se desea que esta posición quede controlada con el movimiento del mouse se puede utilizar:

```
image(img,0,0,mouseX,mouseY);
```

Existen varias maneras de alterar la forma en que se muestra una imagen en el canvas. Utilizando la función **`scale()`**, antes de la función **`image()`**, cambiará la escala a la que se muestra la imagen, por ejemplo, si se le asigna un valor 0.5, la imagen tendrá un ancho igual a la mitad del ancho original de la imagen y un alto igual a la mitad del alto original de la imagen. Utilizando dos parámetros separados por una coma en la función **`scale(sx,sy)`** permite alterar la escala de forma independiente para el ancho y para el alto, respectivamente.

Podemos también alterar la imagen con uso de filtros (función **`filter()`**), estos filtros deben utilizarse después de la función **`image()`**. Un ejemplo del uso de escalas y filtros puede revisarse a continuación:

```
let img;

function preload(){
    img=loadImage('archivo.png');
}

function setup(){
    createCanvas(400,400);
}

function draw(){
    background(100);
    scale(0.5);
    image(img,0,0);
    filter(INVERT); //THRESHOLD, GRAY, OPAQUE
}
```

Figura 3.4. Alterar una imagen en el canvas con el movimiento del mouse.

En el ejemplo se encuentran comentados otros ejemplos de filtros que pueden utilizarse (**THRESHOLD**, **GRAY**, **OPAQUE**. Otros filtros pueden encontrarse en las referencias de la biblioteca).

Como fue indicado en la sección correspondiente a los "Arreglos", un arreglo también puede contener imágenes en cada posición. Si son varios los archivos de imagen que se desean utilizar y los nombres de estos archivos están enumerados de forma ordenada (por ejemplo: "**ani00.png**", "**ani01.png**", "**ani02.png**", etcétera), estos pueden cargarse a partir de un ciclo. En el ejemplo que sigue, el argumento para la función **loadImage()** presenta otra función llamada **nf()**, esta función requiere dos o tres argumentos, el primer argumento es el número que se desea incluir, el segundo argumento es la cantidad de cifras que van a incluir al número asignado al primer argumento (rellenando con ceros si es necesario) y el tercer argumento es la cantidad de decimales que se incluirán (el tercer argumento no es necesario si no se requieren decimales). Por ejemplo, si se utiliza **nf(1,2)**, devolverá el número **01**, si se utiliza **nf(23,3,2)**, devolverá **023.00**. Finalmente, en el ejemplo, **"ani"+nf(i,2)+".png"** se cargará el archivo "**ani00.png**" (**ani + 00 + .png**) cuando el valor de **i** tome el valor 0, el archivo "**ani01.png**" cuando **i** tome el valor 1, y así sucesivamente hasta que **i** tome el valor 9.

```
let x=0;
let img = [];

function preload(){
  for(let i=0; i<10; i++){
   img[i]=loadImage("ani"+nf(i,2)+".png");
  }
}

function setup() {
  createCanvas(500, 500);
  frameRate(20);
}

function draw() {
  background(220);
  let index=x%10;
  image(img[index],0,0);
  x++;
}
```

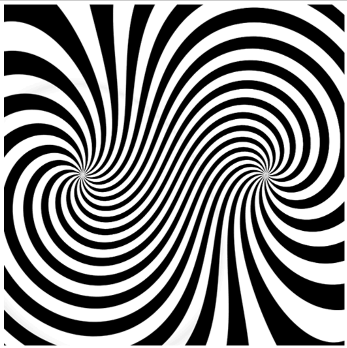


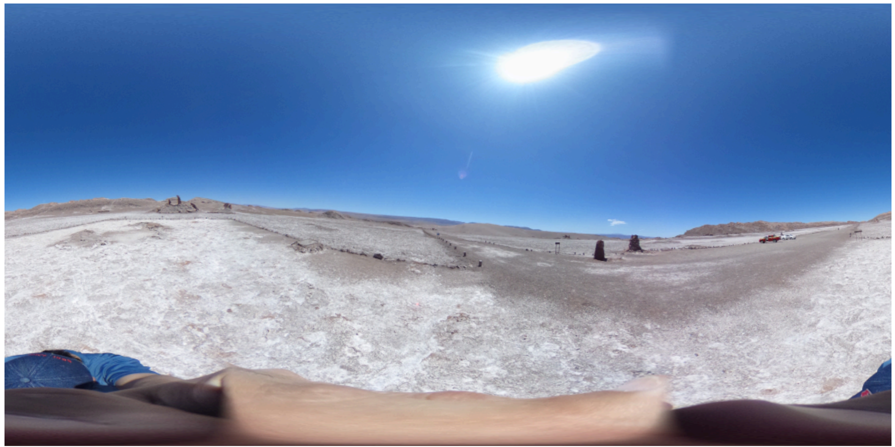
Figura 3.5. Imagen de un cuadro de una animación.

El ejemplo anterior muestra una animación cíclica cuadro a cuadro, cambiando el índice del arreglo de imágenes en cada pasada de la función **`draw()`**.

Una aplicación interesante que se puede realizar con la carga de una imagen es utilizar la opción **`WEBGL`** para geometrías en 3D y plasmar la imagen como textura de una esfera. Esta imagen debe obtenerse con una

cámara especial capaz de sacar fotos panorámicas o descargarse alguna desde la web. Estas imágenes tienen un ancho que es el doble del alto. La función que permite plasmar esta imagen en una esfera es **`texture(img)`**, donde **`img`** es la variable que incluye la imagen panorámica, y debe utilizarse antes de describir la esfera en el código. Añadiendo un poco de interacción con el mouse se puede girar la imagen panorámica a gusto:

```
let img;

function preload(){
    img = loadImage('foto360.png');
}

function setup(){
    createCanvas(512,512,WEBGL);
}

function draw(){
    background(0);
    noStroke();
    let b=map(mouseX,0,width,2*PI,0);
    let c=map(mouseY,0,height, -PI/2, PI/2);
    rotateX(c);
    rotateY(b);
    texture(img);
    scale(3);
    sphere(200);
}
```



a)

b)

Figura 3.6. a) Imagen panorámica extendida, posee el doble de ancho que de alto. **b)** Imagen panorámica proyectada en una esfera.

3.3. Archivos txt

Para trabajar con archivos de texto con extensión `*.txt` es necesario crear tal archivo en el editor o subir un archivo desde el disco de

memoria del dispositivo. La manera de subir este tipo de archivos se asemeja a la manera de subir imágenes.

Por las mismas razones expuestas en la sección “Imágenes” se recomienda cargar el archivo en el código que se está escribiendo dentro de una función **`preload()`**, sobre todo si el archivo es muy extenso. Para cargar el archivo de texto utilizaremos la función **`loadStrings(“archivo.txt”)`**. Una vez cargado el archivo de texto, se genera un arreglo que incluye en cada posición (o sea, en cada índice) cada una de las líneas que se incluyen en el archivo de texto. Por ejemplo, si el archivo de texto es una canción y en cada línea de ese archivo hay un verso de la canción, entonces el arreglo incluirá en cada posición una línea de verso. ¡Cuidado! porque pueden haber líneas que no contengan texto, y que de igual manera serán incluidas en el arreglo. Si desea conocer cuántas líneas tiene el archivo de texto, se puede utilizar la función **`archivo.length`**.

A la vez que cada línea del archivo de texto corresponde a un índice del arreglo del archivo, cada una de estas líneas en un arreglo en sí mismo, y los elementos de este arreglo son los caracteres incluidos en cada línea. Es decir, el archivo de texto original se considera un arreglo de arreglos.

Veamos un ejemplo para comprender lo que estamos describiendo. Imagine un archivo de texto llamado **`“archivo.txt”`** y que contiene la siguiente información:

```
Hola
¿cómo estás?
espero que bien
```

Al cargar el archivo se generará el siguiente arreglo, que llamaremos **`texto`**:

Índice	0	1	2
Elemento	**`“Hola”`**	**`“¿cómo estás?”`**	**`“espero que bien”`**

Es decir **texto[0]="Hola"**, **texto[1]="¿cómo estás?"** y **texto[2]="espero que bien"**. Pero a la vez **texto[0]** (y **texto[1]** y **texto[2]**) es un arreglo, y cada carácter de cada línea pertenece a ese arreglo, tomando como ejemplo **texto[0]**, este arreglo es el siguiente:

Índice	0	1	2	3
Elemento	**"H"**	**"o"**	**"l"**	**"a"**

Si queremos acceder a la letra **"H"** se debe utilizar esta forma: **texto[0][0]**, y, por lo tanto, **texto[0][1]** será igual a **"o"**, **texto[0][2]** será igual a **"l"** y **texto[0][3]** será igual a **"a"**. Si una línea de texto tiene algún espacio dentro de él, ese espacio formará parte como un elemento más del arreglo. En el ejemplo anterior, **texto[2][6]** será un espacio.

```
let texto;

function preload(){
  texto=loadStrings("archivo.txt");
}

function setup(){
  createCanvas(400,400);
}

function draw(){
  background(200);
  textSize(16);
  for(let i=0; i<texto.length; i++){
     text(texto[i],20,30+20*i);
  }
}
```

Hola
¿Cómo estás?
Espero que bien

Figura 3.7. Muestra de un archivo de texto en el canvas.

El ejemplo anterior muestra todo el contenido del archivo "**archivo.txt**" en el canvas.

Consideremos ahora un archivo de texto que tiene valores numéricos en su interior, llamemos "**valores.txt**" a este archivo que contiene la siguiente información:

```
0.1;0.3
0.03;2.8
4.5;7
```

En cada línea hay 2 valores numéricos separados por ";". Para acceder a estos valores de manera individual debemos cargar el archivo tal como lo vimos anteriormente, esto creará un arreglo con 3 elementos en el ejemplo (porque hay 3 líneas). Como cada línea tiene dos valores tendremos que hacer que se genere un arreglo con dos elementos para cada línea, esto se puede hacer utilizando la función **split()**.

```
let datos;
let x=[];
let y=[];

function preload(){
  datos=loadStrings("valores.txt");
}

function setup(){
  createCanvas(400,400);
}

function draw(){
```

```
  background(200);

  for(let i=0; i<datos.length; i++){
     let dato=split(datos[i],';');
     x[i]=dato[0];
     y[i]=dato[1];
  }
}
```

La función **`split()`** recibe dos parámetros, el primero es una línea de texto y el segundo es el separador (en este caso "`;`", pero puede ser otro). Si en la línea de texto hay un solo carácter igual al separador, se creará un arreglo con dos elementos (como en el ejemplo señalado). Si en la línea de texto hay dos caracteres igual al separador se creará un arreglo con tres elementos, y así sucesivamente. El código anterior genera la variable **`datos`** que contiene la información del archivo de texto. Se declara también un arreglo vacío para los valores que están a la izquierda del símbolo "`;`" (llamado **`x`**) y un arreglo vacío para los valores que estén a la derecha del símbolo "`;`" (llamado **`y`**). Dentro del ciclo **`for()`** se recorren todas las líneas del archivo de texto y dentro de cada vuelta del ciclo se genera una variable local que va guardando temporalmente los datos de cada línea en un arreglo (**`dato`**). Luego se designa el primer índice de este arreglo **`dato`** en el arreglo **`x`**, y se hace lo mismo con el segundo elemento del arreglo **`dato`** para ir completando el arreglo **`y`**. Una vez terminado el ciclo se generarán los arreglos: **`x=[0.1,0.03,4.5]`** y **`y=[0.3,2.8,7]`**. Estos valores pueden ser utilizados para graficar o para hacer cálculos numéricos con ellos.

3.4. Sonidos

En **p5.js** podemos cargar y reproducir archivos de audio (**`mp3`** y **`wav`**), para ello, al igual que con los tipos de archivo previamente explicados, debemos subir el archivo previamente y luego cargar el archivo en el código con la función **`preload()`**.

```
let cancion;
```

```
function preload(){
  cancion=loadSound("audio.mp3");
}

function setup() {
  createCanvas(400, 400);
  cancion.play();
}

function draw() {
  background(220);
}
```

El ejemplo anterior es el código más sencillo que permite reproducir un archivo de audio, donde el audio se reproduce utilizando la función `.play()`. Existen varias funciones que pueden utilizarse en un archivo de audio, algunas de ellas las veremos a continuación:

```
let cancion;

function preload(){
  cancion=loadSound("audio.mp3");
}

function setup() {
  createCanvas(400, 400);

}

function draw() {
  background(220);
}

function mousePressed(){
  if(cancion.isPlaying()){
    cancion.pause();
  }else{
    cancion.setVolume(0.1);
    cancion.play();
  }
}
```

En el ejemplo anterior la canción se reproducirá cuando se haga clic con el mouse con la función **`mousePressed()`** (**`cancion`** es la variable a la que se le asignó el archivo de audio). Dentro de esta función se encuentra una condición **`if(cancion.isPlaying())`**, cuya condición será verdadera si el audio se está reproduciendo y será falsa si el audio no se está reproduciendo. Si la canción se está reproduciendo, al presionar con el mouse, el audio se pausará, y si no se está reproduciendo, al presionar con el mouse el audio comenzará a sonar. Al pausar un audio, se guarda el momento en el cual se pausó el audio, y si se reproduce nuevamente, comenzará a sonar desde el momento en que se pausó, a diferencia de la función **`cancion.stop()`** que detiene la reproducción y vuelve el audio a su inicio. La función **`cancion.setVolume()`** determina el volumen al cual se reproducirá el audio y admite un número como parámetro.

Con la función **`cancion.duration()`** se obtiene la duración del archivo del audio en segundos, y con la función **`cancion.currentTime()`** se obtiene el instante actual que se está reproduciendo del audio, en segundos. Existe también la opción de ir a una ubicación específica del audio con la función **`cancion.jump()`**, que recibe como argumento el instante de tiempo al que se requiere ir del audio, por ejemplo, con **`cancion.jump(0.75)`** salta al tiempo **`0.75`** segundos del audio.

3.5. Análisis de frecuencia

Realizar un análisis de frecuencia de un sonido significa conocer la amplitud de cada una de las frecuencias que componen tal sonido, esta información es lo que llamaremos espectro de frecuencia. El análisis de frecuencia se basa en una herramienta llamada transformada de Fourier y dicho análisis se realiza digitalmente con una técnica llamada FFT (de transformada rápida de Fourier, por sus siglas en inglés) que se puede implementar a través de **p5.js**.

Primero, se obtendrá el espectro de un archivo de audio, para llevar a cabo este análisis crearemos una variable que tendrá el archivo de audio

que se analizará (**cancion**), también, es necesario crear una variable que incluirá el análisis FFT (**fft**) y otra variable que contendrá el espectro de frecuencia (**espectro**). A la variable **fft** se le asignará el objeto **new p5.FFT(0.9,512)**, el primer valor (0,9) es un valor que puede ser mayor o igual a 0 y estrictamente menor a 1. El segundo valor es importante que sea una potencia entera de base 2 (2^n, con n entero), esto por las características con que se realiza la FFT. Para obtener la información espectral en cada instante, dentro del **draw()** se debe asignar la función **fft.analyze()** a la variable **espectro**, con esto ya tenemos los datos que pueden graficarse y mostrarse en el canvas.

```
let cancion;
let boton;
let fft;

function preload(){
  cancion=loadSound("audio.mp3");
}

function setup() {
  createCanvas(512, 512);
  boton = createButton('play/pause');
  boton.mousePressed(toggle);
  fft = new p5.FFT(0.9,512);
  colorMode(HSB);
}

function draw() {
  background(0);
  let espectro = fft.analyze();
  for(let i = 0; i < espectro.length; i++){
  strokeWeight(4);
  stroke(map(espectro[i],0,300,0,255),100,100);
  point(i,map(espectro[i],0,300,height,0));
  }
}

function toggle(){
  if(cancion.isPlaying()){
     cancion.pause();
     }else{
```

```
    cancion.play();
  }
}
```

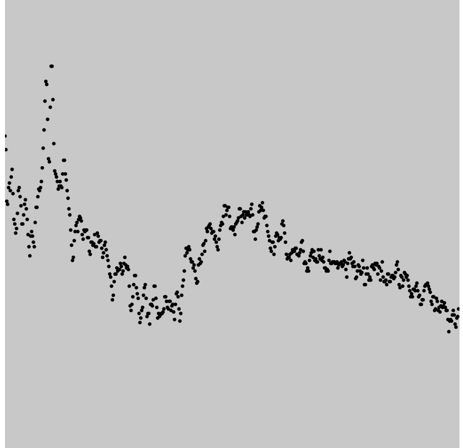


Figura 3.8. Espectro de frecuencia de una señal de audio.

En el ejemplo anterior se añade un botón para reproducir y pausar una canción. El espectro de frecuencia se grafica dentro de un ciclo **`for()`** y note que **`espectro`** es un arreglo con la información de amplitud de cada una de las frecuencias. Adicionalmente, se incluye dentro de la función **`setup()`** otro perfil de color HSB, para Hue, Saturation y Brightness (tono, saturación y brillo en español) con **`colorMode(HSB)`** para representar la amplitud en cada una de las frecuencias con diferentes colores.

El mismo tipo de análisis puede realizarse con el audio proveniente desde el micrófono del computador, para ello debemos definir una variable que acceda al micrófono con **`new p5.AudioIn()`**, y para la variable **`fft`** (igual a como fue declarada en el ejemplo anterior) debe detallarse la entrada que aceptará, en este caso la entrada de micrófono con **`.setInput()`**.

```
let mic;
let fft;
```

```
let espectro;

function setup() {
  createCanvas(512, 512);
  mic = new p5.AudioIn();
  mic.start();
  fft = new p5.FFT(0.9,512);
  fft.setInput(mic);
}

function draw() {
  background(0);
  espectro = fft.analyze();
  noFill();
  stroke(255,0,0);
  strokeWeight(4);
  beginShape();
  for(let i = 0; i < espectro.length; i++){
    vertex(i,map(espectro[i],0,300,height,0));
  }
  endShape();
}
```

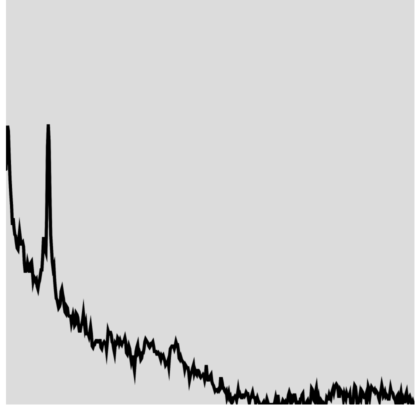


Figura 3.9. Espectro de frecuencia de la señal del micrófono.

En los espectros anteriores, las figuras pueden entenderse como un gráfico, donde el eje horizontal corresponde a las frecuencias que van creciendo de izquierda a derecha, y el eje vertical es la magnitud

correspondiente a cada una de las frecuencias, mientras más arriba mayor amplitud.

3.6. Accediendo a la cámara

Para acceder a la cámara web del dispositivo donde se esté programando es necesario generar una variable que es la que incluirá el video, en el ejemplo siguiente esta variable se llamará "**`video`**", y dentro de la función **`setup()`** se le asigna la captura del video con **`createCapture(VIDEO)`**.

```
let video;

function setup() {
  createCanvas(640, 480);
  video=createCapture(VIDEO);
  video.hide();
}

function draw() {
  background(220);
  image(video,0,0);
}
```

Figura 3.10. Captura de video desde la cámara del dispositivo.

Si queremos mostrar la imagen que captura la cámara se utilizará la misma función que se utiliza para mostrar imágenes estáticas (**`image()`**). Para que no se vea repetida la captura del video en pantalla y para dejar

solo imagen del video en el canvas es necesario utilizar la opción **`video.hide()`** dentro de **`setup()`**. Note que en **`createCanvas()`** se estableció el ancho y alto con la resolución que entrega la cámara, para que muestre la imagen completa.

Como la forma en la que se muestra el video de la cámara en el canvas es la misma en la que se muestran imágenes, todas las opciones revisadas en la sección "**2.2. Imágenes**" de este documento son válidas para el video.

3.7. DOM

DOM se refiere a Document Object Model y son una herramienta que permite realizar aplicaciones de forma más dinámica. A continuación se mostrará la opción de incluir algunas de estas herramientas en un programa, se revisará la inclusión de botones, deslizadores, entradas de texto, entre otras.

Para crear un botón debemos crear una variable al que se le asigne la función **`createButton()`**, que recibe como argumento el texto que se mostrará en el botón. Como opción se puede ubicar el botón en algún lugar del canvas con **`.position()`**, que recibe como argumentos las coordenadas **`x`** e **`y`** de su posición. Como deseamos que al presionar el botón se realice una acción, como argumento de la función **`.mousePressed()`** se puede indicar el nombre de la función que ejecutará estas acciones. Luego de esto, es necesario crear esta función y las acciones que realizará.

Un deslizador permite crear una herramienta que al deslizarse va tomando valores numéricos distintos desde un valor mínimo a un valor máximo, cada vez que se desliza esta herramienta ocurrirá algo en el programa que se está desarrollando. Para crear un deslizador necesitamos declarar una variable y a esta variable asignarle la función **`createSlider()`**, que requiere 3 valores como argumentos, el valor mínimo que tomará, el valor

máximo y el valor inicial que tomará al ejecutar el programa, respectivamente. Podría añadirse un cuarto argumento que indicará el valor del paso entre un valor y otro al deslizar, por defecto es 1. Esta herramienta también puede ubicarse en algún lugar del canvas con **`.position()`**.

Puede cambiarse el tamaño del botón y del deslizador utilizando la función **`.size()`**, que recibe dos valores como parámetros, para el ancho y alto respectivamente.

```
let desli;
let boton;
let val;
let control=0;

function setup(){
  createCanvas(400, 400);
  desli = createSlider(50, 200, 70);
  desli.position(10,20);

  boton = createButton("Botón");
  boton.position(10, 60);
  boton.mousePressed(presionado);
}

function draw(){
  val = desli.value();
  background(val,50,50);
  if(control % 2 == 1){
    circle(width/2, 150, 180);
  }
}

function presionado(){
  control++;
}
```

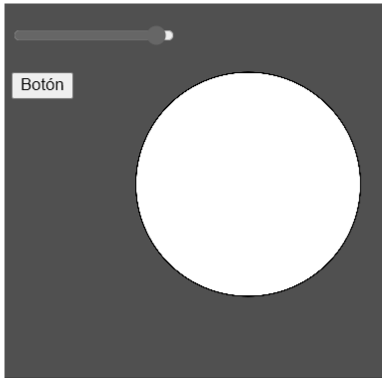


Figura 3.11. Uso de botón y deslizador en una aplicación.

El ejemplo anterior crea un deslizador que tiene como valor mínimo 50 y valor máximo 200, partiendo con un valor inicial de 70. Los valores que se van obteniendo al mover el deslizador cambian el canal rojo del fondo del canvas. Como no se indica un cuarto parámetro, al deslizar, los valores irán cambiando de 1 en 1. Este deslizador está ubicado en la posición **`x`** igual a 10 e **`y`** igual a 20. Es necesario generar una variable que contenga el valor del deslizador (**`val`** en el ejemplo), que accede al valor que va tomando el deslizador (**`desli.value()`** en el ejemplo).

También se creó un botón con la palabra "Botón" dentro, en la posición **`x`** igual a 10 e **`y`** igual a 60, y al presionar este botón se ejecuta la función "**`presionado`**", definida más abajo en el código, que va sumando el valor 1 a la variable control, que inicia con el valor 0.

Para crear una caja que permita ingresar texto al usuario del programa, crearemos una variable y le asignaremos la función **`createInput()`**, que requiere dos argumentos, el primero es el texto que se mostrará por defecto y el segundo es el tipo de valor que se ingresará, que puede ser un número (**`"number"`**) o un texto (**`"text"`**). Las opciones de posición (**`.position()`**) y tamaño (**`.size()`**) también son permitidas aquí.

```
let entrada;
```

```
let texto;

function setup(){
  createCanvas(400, 400);
  entrada = createInput('','number');
  entrada.position(20,20);
  entrada.size(100,40);

  texto = createInput('','text');
  texto.position(100,120);
  texto.size(200);
}

function draw(){
  background(237,205,116);
  let a = entrada.value();
  textSize(16);
  text("El doble del número ingresado es "+2*a,20,85);
  text("Has escrito "+texto.value(),100,160);
}
```

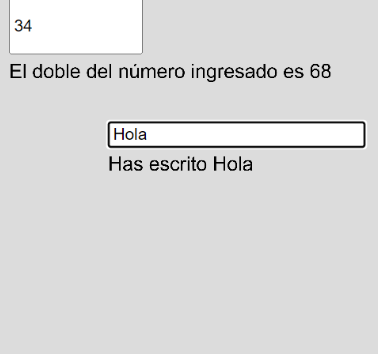


Figura 3.12. Cajas con posibilidad de ingresar texto.

En el ejemplo anterior se crea una entrada de texto que permite solo valores numéricos (**`entrada`**) y otra entrada de texto que permite palabras, letras, incluso números (**`texto`**). Para acceder al texto (o número) que se ingresa a una caja de texto se utiliza la función **`.value()`** (en el ejemplo, **`entrada.value()`** para el valor numérico y **`texto.value()`** para el texto que no es un número).

Otras opciones que acepta el DOM son: **createSelect()**, que permite crear menús para elegir un elemento de una lista desplegable, **createRadio()**, que permite crear casillas para elección de elementos y **createCheckbox()**, que permite crear una casilla de verificación.

La forma de generar un menú dentro de la función **setup()**, con sus opciones y sus valores asignados a cada opción se determina a partir de:

```
menu=createSelect();
menu.option("Opción 1","100");
menu.option("Opción 2","200");
menu.option("Opción 3","300");
menu.selected("200");
menu.position(20,20);
```

Donde **menu=createSelect()** genera el objeto como un menú desplegable, las distintas opciones se generan con **menu.option()**, en el que el primer argumento es el texto que mostrará en el menú y el segundo argumento es el valor asociado al texto mostrado, se pueden crear todas las opciones que se necesiten. El segundo argumento debe indicarse entre comillas, indicando que es un texto, por lo que es posible que deba convertirse a un valor numérico con **int()** (si se trata de un entero. **int("100")** será igual a 100) o con **float()** (si se trata de un número decimal. **float("10.13")** será igual a 10.13). Con **menu.selected()** se determina, si se opta por esto, cuál será la opción por defecto elegida en el menú, y con **menu.position(x,y)** se ubica el menú en el lugar del canvas en la posición **(x,y)** determinada. Para acceder al valor seleccionado se debe utilizar, dentro de la función **draw()**, la expresión **menu.value()**.

Utilizando **createRadio()** permite crear un ítem de selección similar a **createSelect()**, pero esta vez mostrando casillas para cada opción listada. La forma de generar un listado de opciones con **createRadio()** es:

```
  radio=createRadio();
  radio.option("100", "Opción A");
  radio.option("200", "Opción B");
  radio.option("300", "Opción C");
  radio.selected("200");
  radio.position(20,100);
```

Esta vez, el valor asignado a cada opción se indica en el primer argumento de **radio.option()** y en el segundo argumento se indica el texto mostrado. También se puede indicar si se desea cuál será la opción elegida por defecto con **radio.selected()**. Para acceder al valor seleccionado se debe utilizar, dentro de la función **draw()**, la expresión **radio.value()**.

Una casilla de verificación simple permite dos opciones, `true` o `false` (verdadero o falso), y se implementa con **createCheckbox()** como se muestra a continuación:

```
  check=createCheckbox("Cambio de color",false);
  check.position(20,60);
```

El objeto **createCheckbox()** requiere dos argumentos, el primero será el texto que se mostrará en el canvas y el segundo parámetro es el valor por defecto que tendrá la casilla (`true` o `false`). Para conocer cuál es la opción marcada se utiliza dentro de la función **draw()** la expresión **check.checked()**, que si la casilla está marcada será `true` (verdadero) y si no está marcada será `false` (falso).

Un código funcional, que implementa las tres últimas herramientas de interacción con el usuario, se indica a continuación:

```
let x=0;
let y=0;

function setup() {
  createCanvas(400, 400);
  rectMode(CENTER);
  menu=createSelect();
  menu.option("Opción 1","100");
```

```
  menu.option("Opción 2","200");
  menu.option("Opción 3","300");
  menu.selected("200");
  menu.position(20,20);

  radio=createRadio();
  radio.option("100", "Opción A");
  radio.option("200", "Opción B");
  radio.option("300", "Opción C");
  radio.selected("200");
  radio.position(20,100);

  check=createCheckbox("Cambio de color",false);
  check.position(20,60);
}

function draw() {
  background(200);
  x=radio.value();
  y=menu.value();

  if(check.checked()){
    fill(255,0,0);
  }else{
    fill(0,0,255);
  }
  circle(x,y,60);
}
```

En el ejemplo, la casilla de verificación simple pinta de rojo o azul un círculo que se muestra en el canvas, según la casilla esté presionada o no. El menú desplegable cambia la posición vertical del círculo de acuerdo a tres posiciones predefinidas (100, 200 o 300), y a su vez las opciones con casillas de verificación cambia la posición horizontal del círculo de acuerdo a tres posiciones predefinidas (100, 200 o 300).

Resumen

En este capítulo, hemos explorado la creación de arreglos y su versatilidad. Estos arreglos pueden contener una variedad de elementos, como números, imágenes, sonidos y más. Utilizando las funciones

proporcionadas, tendrá la capacidad de modificar y manipular los elementos dentro de cada arreglo. Además, a partir de ahora, puede desarrollar aplicaciones multimedia que incluyan imágenes, audio y texto. También podrá acceder a la cámara web y al micrófono para realizar análisis de frecuencia. Finalmente, hemos examinado cómo agregar interactividad a sus aplicaciones mediante la incorporación de botones, deslizadores, la creación de menús y casillas de verificación.

Algunas aplicaciones que podría crear con las herramientas vistas hasta este capítulo son:

- Crear un programa que permita visualizar una variedad de fotografías previamente cargadas y elegir con el mouse cuál de las fotografías se despliega.
- Crear un programa que funcione como un reproductor de audio de canciones en formato mp3 previamente cargadas.
- Crear un programa que despliegue el texto de una canción (o un poema o un párrafo de un libro) y que con el uso del teclado o del mouse pueda elegir qué sección del texto mostrar.
- Crear un programa que permita realizar un gráfico (de líneas, de barras, de torta, u otro gráfico) con datos incluidos en un archivo de texto.
- Crear un programa que permita controlar la posición de un objeto con la voz a través de un micrófono.
- Crear un programa que tome una fotografía a partir del uso de la cámara de video.
- Crear un programa que reproduzca una canción, que muestre el texto de la misma y una imagen relacionada con lo que se está escuchando. Puede incluir varias canciones.
- Crear un programa que permita mostrar el espectro de frecuencia de audios pregrabados de distintas aves y mostrar una foto de cada ave.

Capítulo 4. Funciones y objetos

4.1. Funciones

Una de las características de **p5.js** es que puede modularizar algunas secciones del código y crear funciones, esto se realiza cuando algún procedimiento o estructura, dentro del código, necesita realizarse en varias oportunidades. En la sección que viene se revisarán dos formas distintas de crear funciones, una de ellas será para representar información en pantalla (hacer un dibujo), y la segunda forma servirá para realizar una operación matemática.

Para experimentar con las funciones, se creará inicialmente un dibujo muy sencillo con cinco círculos, uno central y los otros cuatro representarán los pétalos de una flor. Primero se realizará sin el uso de una función.

```
let x=100;
let y=200;
let r=20;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  push();
  fill(255,255,0);
  circle(x+r,y,2*r);
  circle(x,y+r,2*r);
  circle(x-r,y,2*r);
  circle(x,y-r,2*r);
  fill(255,50,50);
  circle(x,y,2*r);
  pop();
}
```

Para crear este dibujo se define la posición **x** e **y** donde se ubicará la flor, además del radio **r** de cada círculo. De esta forma **x**, **y** y **r** serán variables globales que se pueden alterar para cambiar la posición y el tamaño de cada flor dibujada. En el ejemplo que se muestra el color de cada pétalo será siempre el mismo.

Lo anterior permite dibujar lo que asemeja a una flor en una ubicación específica del canvas, si se requiere repetir este dibujo, en distintas posiciones y con tamaños diferentes, podemos crear una función llamada **flor()**, donde sus variables de entrada, es decir sus argumentos, sean la posición **x**, la posición **y**, y el radio **r**, el radio es el que determinará el tamaño de cada flor. Los argumentos se indican entre paréntesis junto al nombre de la función y separados por "**,**". Dentro de la función **flor()** se puede copiar y pegar el código utilizado previamente, antes de generar la función.

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  flor(150,200,40);
  flor(200,300,10);
  flor(mouseX,mouseY,15);
}

function flor(x,y,r){
  push();
  fill(255,255,0);
  circle(x+r,y,2*r);
  circle(x,y+r,2*r);
  circle(x-r,y,2*r);
  circle(x,y-r,2*r);
  fill(255,50,50);
  circle(x,y,2*r);
  pop();
}
```

En el ejemplo anterior, en la función **draw()** se llama a la función **flor(x,y,r)** para representar tres flores en el canvas, una de esas flores se desplaza junto con el movimiento del mouse.

Una función que devuelve un valor numérico, dado un argumento, se puede generar como en el ejemplo que sigue. Crearemos una función que reciba un argumento numérico en radianes y la función entregará el valor en grados, la función recibirá el nombre **rag** (de “radianes a grados”).

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  textSize(20);
  text(rag(HALF_PI),200,200);
}

function rag(x){
  let g=180*x/PI;
  return g;
}
```

La función **rag** del ejemplo, devuelve el valor de un ángulo en grados, tomando el argumento **x** en radianes, es decir, multiplica **x** por 180 y lo divide por π. En una función de este tipo es necesario incluir **return** cuando se requiere que la función devuelva un valor numérico cuando ésta sea llamada desde la función **draw()**, así, al utilizar por ejemplo **rag(HALF_PI)**, se sabrá que el código lo interpretará como 90, y es el texto que se mostrará en el canvas.

Cuando se requiera crear una función que reciba más de un argumento, estos deben incluirse entre los paréntesis a continuación del nombre de la función y separados por “**,**”.

```
function setup() {
  createCanvas(400, 400);
}
```

```
function draw() {
  background(220);
  textSize(20);
  text(raiz(1,0,1),200,200);
}

function raiz(x, y, z){
  return sqrt(x*x+y*y+z*z);
}
```

La función **`raiz()`** del ejemplo anterior recibe 3 argumentos y devuelve el resultado de la raíz cuadrada de $x^2+y^2+z^2$.

Para todos los tipos de funciones creadas está permitido crear funciones que no contengan ningún argumento, para ello debe dejarse en blanco el interior de los paréntesis que acompañan al nombre de la función.

4.2. Objetos

Para revisar lo que es un objeto revisaremos un ejemplo concreto, inicialmente indicaremos simplemente, que un objeto es una forma de almacenar variables y sus funcionalidades.

La forma de definir un objeto será a partir de un archivo distinto del de donde estaremos escribiendo nuestro programa, y este objeto se definirá a partir de una clase (**`class`**), que también tendrá la extensión `*.js`. Este archivo que contiene el formato de cada objeto deberá invocarse en el archivo `index.html` para su correcto funcionamiento. La estructura básica de un objeto es la siguiente:

```
class Bola {   // Se define el nombre de una clase, por
convención se inicia con mayúscula
  constructor(x, y) { //Constructor permite recibir parámetros e
incluirlos en el objeto
    this.x = x;      //this. define las variables dentro del
objeto
    this.y = y;
  }

  mostrar() { //función que permite mostrar el objeto, por
ejemplo
 //Acá se añaden funciones para mostrar objeto en el canvas
  circle(this.x, this.y, 40);
  }

  mover() {  //función que permite mover el objeto, por ejemplo
  //Acá se añaden funciones para mover el objeto en el canvas
  this.x += 1;
  }
}
```

Comenzaremos con un ejemplo sencillo, una pelota (representada con un círculo) que se moverá en el canvas y que rebotará en las paredes para siempre estar a la vista. El código que permite realizar lo anterior, sin el uso de objetos de programación, será:

```
let x=200; //posición inicial x de la pelota
let y=200; //posición inicial x de la pelota
let vx;    //velocidad x de la pelota
let vy;    //velocidad y de la pelota

function setup() {
  createCanvas(400, 400);
  vx=random(4,6); //se le asigna valor a velocidad vx
  vy=random(5,8); //se le asigna valor a velocidad vy
}

function draw() {
  background(220,100,60,30);
  //si la pelota choca con los bordes, cambia de dirección
  if(x>width || x<0){
     vx*=-1; //si vx es positivo cambia a negativo (y viceversa)
     }
    if(y>height || y<0){
```

```
      vy*=-1; //si vy es positivo cambia a negativo (y viceversa)
      }

   circle(x,y,20);  //se dibuja el círculo
   x+=vx; //x cambia en una cantidad vx
   y+=vy; //y cambia en una cantidad vy
}
```

Se puede observar que la función **`background()`** tiene un cuarto parámetro, que permitirá que el objeto, al moverse, muestre una estela de su movimiento.

Si se necesitara que hubiera más de una pelota rebotando en el canvas podría hacerse básicamente repitiendo el código anterior, con nombres de variables de posición **`x`** e **`y`**, y nombres de variables de velocidad **`vx`** y **`vy`** específicas para cada una de las pelotas (**`x1`**, **`y1`**, **`x2`**, **`y2`**, **`x3`**, **`y3`**, etcétera para las posiciones y **`vx1`**, **`vy1`**, **`vx2`**, **`vy2`**, **`vx3`**, **`vy3`**, etcétera para las velocidades). Esto sería engorroso, por lo que en este caso se justifica la creación de un objeto, donde este objeto guarde en un formato las variables de posición y velocidad para cada una de las pelotas presentes.

Escribiremos este objeto, para ello inicialmente se creará un archivo nuevo llamado `pelota.js`, y dentro de este archivo se define una clase.

```
class Pelota{
  constructor(x,y){
    this.x=x;
    this.y=y;
    this.vx=random(4,8);
    this.vy=random(-6,6)
  }

  mostrar(){
    circle(this.x,this.y,20);
  }

  mover(){
    if(this.x>width || this.x<0){
       this.vx*=-1;
       }
    if(this.y>height || this.y<0){
       this.vy*=-1;
       }
    this.x+=this.vx;
    this.y+=this.vy;
  }
}
```

Este objeto se define con **`class`** y el nombre de esta clase (**`Pelota`** en el ejemplo), que por convención utiliza la primera letra en mayúscula. Posteriormente se escribe la función **`constructor()`**, donde se indicarán las variables iniciales que caracterizan al objeto creado, en este caso la posición inicial y velocidad para cada uno de los ejes. Es estrictamente necesario que estas variables iniciales incluyan el prefijo **`this.`**, que tendrá que ser incluido cada vez que se invoque a estas variables.

Dentro de esta clase se crean las funciones (**`mostrar()`** y **`mover()`** en el ejemplo). La primera función simplemente dibuja un círculo en el canvas y la segunda función cambia la posición de la pelota y detecta cuando se encuentra con alguno de los bordes.

Este archivo por sí solo no mostrará una pelota en el canvas, por lo que es necesario cambiar el código en el programa principal en el archivo `sketch.js`.

El archivo `sketch.js` quedará de la siguiente forma:

```
let pelota;

function setup() {
  createCanvas(400, 400);
  pelota= new Pelota(200,200);
}

function draw() {
  background(220,100,60,80);
  pelota.mostrar();
  pelota.mover();
}
```

Se declara la variable **`pelota`** que será el objeto, y, luego en la función **`setup()`** se crea este objeto con la palabra clave **`new`** y luego el nombre de la clase (**`Pelota`**), se indican como argumento dos valores que serán recibidos en la función **`constructor()`** como valores **`x`** e **`y`** en la clase creada anteriormente.

Dentro de la función **`draw()`** se llama a las funciones creadas en la clase del objeto, incluyendo previamente el nombre de la variable del objeto creado (**`pelota.mostrar()`** y **`pelota.mover()`** en el ejemplo).

Si ejecutamos el programa con estos dos archivos creados (`sketch.js` y `pelota.js`), de igual manera existirá un error, y es porque este segundo archivo no ha sido invocado en el archivo `index.html`, para evitar este problema, el archivo `html` deberá incluir cada archivo `js` creado:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.7.0/p5.js"></s
cript>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.7.0/addons/p5.
sound.min.js"></script>
    <link rel="stylesheet" type="text/css" href="style.css">
    <meta charset="utf-8" />

  </head>
  <body>
    <main>
    </main>
    <script src="sketch.js"></script>
    <script src="pelota.js"></script>
  </body>
</html>
```

Al ejecutarse el programa, este funcionará, pero todavía habrá una única pelota en el canvas rebotando en sus bordes. Si se quisiera incluir más pelotas, podría editarse solamente el archivo `sketch.js` para incluir nuevos elementos. El siguiente ejemplo mostrará 10 pelotas que se comenzarán a mover desde lugares aleatorios del canvas.

```
let pelotas=[];

function setup() {
  createCanvas(400, 400);
  for(let i=0; i<10; i++){
    pelotas[i]= new Pelota(random(100,200),random(100,200));
  }
}

function draw() {
  background(220,100,60,80);
  for(let i=0; i<pelotas.length; i++){
    pelotas[i].mostrar();
    pelotas[i].mover();
  }
}
```

Como se puede observar, fue necesario crear un arreglo, donde cada elemento de este arreglo es una pelota, es decir, un objeto. Cambiando dentro del ciclo **for()**, **i<10**, de 10 a otro valor, se creará la cantidad de pelotas que uno requiera sin la necesidad de declarar una gran cantidad de variables distintas.

El ciclo **for()** que se utiliza dentro de la función **draw()** permite mostrar cada uno de los elementos del arreglo. Otra forma de hacer lo mismo sería reemplazar tales líneas de código por las siguientes:

```
  for(let pelotitas of pelotas){
    pelotitas.mostrar();
    pelotitas.mover();
  }
```

Como **pelotas** es un arreglo, entonces todas las opciones utilizadas para los arreglos se pueden utilizar aquí, por ejemplo, si se añade al final y fuera de la función **draw()** el siguiente segmento de código, se irán añadiendo elementos al final del arreglo:

```
function mousePressed(){
  pelotas.push(new Pelota(mouseX,mouseY));
}
```

Al incluir las líneas anteriores de código, se incluirá una pelota al canvas desde la posición del mouse donde se haga clic y luego comenzará a moverse y comportarse como las demás pelotas.

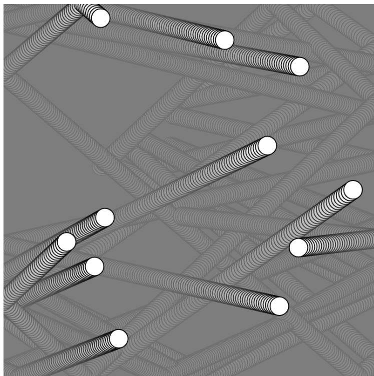


Figura 4.1. Varias pelotas rebotando en el canvas.

Resumen

En este capítulo, hemos explorado cómo crear objetos de programación utilizando **p5.js**. Los objetos son una poderosa herramienta que nos permite organizar de manera sistemática tanto datos como funciones dentro de una estructura coherente y reutilizable. En esencia, un objeto actúa como una plantilla que encapsula propiedades y comportamientos relacionados en una única entidad.

Un beneficio clave de trabajar con objetos es su capacidad para simplificar la creación de programas. Al definir una estructura de objeto clara y lógica, podemos desarrollar aplicaciones de manera más eficiente y efectiva, lo que a menudo resulta en un diseño más claro y fácil de entender.

En resumen, este capítulo se centró en cómo crear y utilizar objetos en p5.js, destacando su importancia en la organización, mantenimiento y desarrollo de programas más efectivos y estructurados.

Algunas de las aplicaciones que se sugieren y que podría crear con el uso de objetos son:

- Crear un programa que dibuje un círculo de diámetro aleatorio cada vez que se haga clic con el mouse, añadiendo cada círculo a un arreglo de objetos.
- Crear un programa que dibuje un círculo de diámetro aleatorio cada vez que se haga clic con el mouse, con la restricción que ningún círculo tape en ninguna medida a otro.
- Crear una aplicación que permita a los usuarios dibujar en la pantalla utilizando diferentes colores y herramientas de dibujo.
- Crear un programa que sea un juego, controlado con el mouse y/o teclado, donde se vayan eliminando distintos “enemigos” según vayan apareciendo en pantalla.
- Crear un programa que simule una gran cantidad de partículas de humo, que estas se vayan disipando y desapareciendo de un arreglo de partículas mientras ascienden en el canvas.

Capítulo 5. Manipulando píxeles

Cuando se establece el tamaño del canvas con la función **`createCanvas(ancho,alto)`**, como ya se señaló anteriormente, se crea un canvas con las dimensiones indicadas como argumento. Si se piensa en términos de la cantidad de píxeles presentes en el canvas, existirán **`ancho*alto`** píxeles, por ejemplo **`createCanvas(600,400)`** genera un canvas de 240000 píxeles. La biblioteca **p5.js** permite alterar el color de los píxeles del canvas uno por uno.

Lo primero que hay que entender es que los píxeles en el canvas representan un arreglo, y por cada píxel, este arreglo tiene 4 elementos, ¿por qué 4?, porque cada píxel tiene valores RGB, es decir, un valor para el canal rojo (R), un valor para el color verde (G) y un valor para el color azul (B), estos tres valores asignan el color de cada píxel. ¿Y el cuarto valor?, el cuarto valor asigna la transparencia de cada píxel, el llamado canal alfa. Se puede concluir entonces que el canvas es un arreglo que tiene **`ancho*alto*4`** elementos. Para el ejemplo del párrafo anterior, el arreglo tendría **`600*400*4 = 960000`** elementos. Cada uno de estos elementos del arreglo tendrá un valor entre 0 y 255.

Una forma de entender el arreglo es observar la siguiente tabla, que muestra la organización de los primeros 3 píxeles.

<table>
<tr><th>Índice del arreglo</th><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>etc.</td></tr>
<tr><th>Píxel</th><td colspan="4">0</td><td colspan="4">1</td><td colspan="4">2</td><td>etc.</td></tr>
</table>

Por lo anterior, si quisiera alterar el color del píxel número 2, hay que preocuparse de alterar los índices 8, 9, 10 y 11 del arreglo, o algunos de esos 4 índices.

Recuerde que los píxeles de igual manera forman una cuadrícula, para que se pueda ver el canvas de forma rectangular. Si considera, por ejemplo, un canvas de 12 píxeles de ancho y 5 de alto, que tendrá 60 píxeles, se verá esquemáticamente como se muestra a continuación:

y\x	0	1	2	3	4	5	6	7	8	9	10	12
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	**41**	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59

Para el caso anterior, el arreglo tendrá `12*5*4=240` elementos, si se quisiera saber qué índices del arreglo le corresponden al píxel 41, que de acuerdo al sistema de referencia le corresponde un **`x = 5`** y un **`y = 3`**, debe considerar la siguiente ecuación para conocer el primer índice asociado a esa posición del canvas (recuerde que son 4 índices por píxel):

```
índice = (y*ancho+x)*4
```

Entonces, para el caso específico indicado, la expresión anterior es igual a `(3*12+5)*4 = 164`. Por lo tanto, si se desea alterar el color del píxel 41, hay que preocuparse de cambiar los índices del arreglo correspondientes a los índices 164, 165, 166 y 167, o algunos de ellos. Es decir, para conocer los índices que le corresponden a un píxel específico, hay que conocer la coordenada **`x`** e **`y`** del píxel en el canvas.

Las funciones y comandos que permiten comenzar a alterar la información presente en cada uno de los píxeles son la función **`loadPixels()`**, que es el que genera el arreglo que contiene la información de cada uno de los píxeles. El comando **`pixels[]`**, que como es un arreglo, se utiliza con

paréntesis cuadrados **`[]`** para elegir el índice que se requiere alterar. Y la función **`updatePixels()`**, que permite actualizar y mostrar los cambios realizados en el canvas. El ejemplo más simple que muestra el cambio de un único píxel en el canvas es el siguiente:

```
let x=200
let y=100;
let indice;

function setup(){
  createCanvas(400,400);
  pixelDensity(1);
}

function draw(){
    background(255);
    indice=(y*width+x)*4;
    loadPixels();
    pixels[indice]=255; //valor canal R
    pixels[indice+1]=0; //valor canal G
    pixels[indice+2]=255; //valor canal B
    pixels[indice+3]=255; //valor canal alfa (transparencia)
    updatePixels();
}
```

Lo anterior cambiará el color de un único y solitario píxel a rosado, ubicado en la posición **`x = 200`**, **`y = 100`**. La función **`pixelDensity(1)`** tiene que ver con la densidad de píxeles que tiene la pantalla donde está visualizando y programando su código, por lo que es posible que no sea necesaria para todos quienes quieran probar este código, y quizás sea preferible comentar o eliminar dicha línea.

Evidentemente, sería preferible alterar una gran cantidad de píxeles para observar los resultados en el canvas de forma clara. Para llevar a cabo esta tarea, haremos uso de ciclos **`for()`**. Considere que se dibujará un rectángulo centrado en la posición del mouse, donde cada píxel de ese rectángulo tendrá un valor aleatorio de color, el valor del índice asociado al canal de transparencia alfa será siempre igual a 255:

```
let indice;

function setup(){
  createCanvas(400,400);
  pixelDensity(1);
}

function draw(){
  background(180);
  loadPixels();
  for(let x=mouseX-50; x<mouseX+50; x++){
    for(let y=mouseY-50; y<mouseY+50; y++){
      indice = (y*width+x)*4;
      pixels[indice]=random(255);
      pixels[indice+1]=random(255);
      pixels[indice+2]=random(255);
      pixels[indice+3]=255;
    }
  }
  updatePixels();
}
```

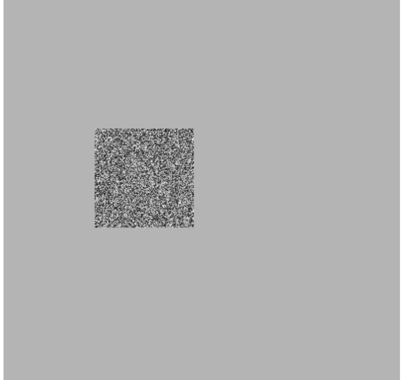


Figura 5.1. Cada píxel del rectángulo tiene un color aleatorio mientras se desplaza el puntero del mouse.

Con respecto al color de cada píxel, cuando se desplaza el puntero del mouse sobre el canvas existe la posibilidad de conocer el color del píxel donde se encuentra el puntero. La función **`get(x,y)`** entrega un arreglo

de 4 elementos (RGB y alfa) con valores entre 0 y 255 del píxel ubicado en la posición **(x,y)** del canvas. Para permitir que la función obtenga el color donde se encuentra el puntero del mouse debe utilizarse **col=get(mouseX,mouseY)**, donde **col** es el arreglo de 4 elementos, que debe declararse previamente como una variable global o local. Por lo tanto, **col[0]** es el valor asociado al canal R, **col[1]** el valor asociado al canal G, **col[2]** el valor asociado al canal B y **col[3]** es valor asociado al canal alfa.

Imagine que desea obtener la información de color de cada píxel que conforma una imagen, previamente se indicó cómo cargar una imagen y mostrarla en el canvas. Si la variable que contiene la imagen es llamada **img**, entonces se puede hacer uso de la función **get()** y obtener la información de color del píxel donde se encuentra el puntero del mouse utilizando **img.get(mouseX,mouseY).** Lo anterior se puede realizar sin necesariamente mostrar la imagen en el canvas con la función **image().**

Con la información entregada en este capítulo se puede, por ejemplo, generar un programa que simule el uso de una linterna alumbrando una imagen que está oscura. Para realizar esto, se obtienen los colores de los píxeles de la imagen cargada con **preload()**, y aquellos píxeles que se encuentren a una distancia a la posición del puntero del mouse menor a cierto valor, replicarán su color con la opción **pixels[]**, para finalmente actualizar los píxeles y representar en el canvas el resultado.

En el siguiente ejemplo se muestra el código que permite realizar lo anterior, mostrando la imagen en el canvas de manera oscurecida:

```
let imagen;
let indice;
let col;

function preload(){
  imagen=loadImage("fish.png");
}
```

```
function setup(){
  createCanvas(512,512);
  pixelDensity(1);
}

function draw(){
  background(0,20,255);

  image(imagen,0,0);
  fill(0,220);
  rect(0,0,width,height);
  loadPixels();
  for(let x=mouseX-70; x<mouseX+70; x++){
    for(let y=mouseY-70; y<mouseY+70; y++){
      let indice = (y*width+x)*4;
      col=imagen.get(x,y);
      if(dist(mouseX,mouseY,x,y)<70){
        pixels[indice]=col[0];
        pixels[indice+1]=col[1];
        pixels[indice+2]=col[2];
        pixels[indice+3]=255;
      }
    }
  }
  updatePixels();
}
```

Figura 5.2. Simulación de una linterna alumbrando una imagen oscura.

Resumen

El trabajo con píxeles en p5.js es una técnica versátil que permite acceder, modificar y manipular la información visual a nivel de píxel en tus proyectos creativos y de programación.

En este capítulo se revisó la forma en la que se representan los colores en **p5.js** y la manipulación de cada píxel con el modelo de color RGB (Rojo, Verde, Azul). Podrá acceder y modificar estos valores individualmente, lo que abre un amplio abanico de posibilidades en el mundo de la programación artística y el procesamiento de imágenes.

Con la información recibida se le sugiere implementar algunos de los programas que se listan a continuación:

- Crear una aplicación que aplique filtros de imagen como blanco y negro, sepia, etcétera, a una foto, cambiando los valores de cada píxel de la imagen.
- Crear una aplicación que simule técnicas de pintura como puntillismo o impresionismo de una imagen previamente cargada.
- Crear una aplicación que manipule la información de video que capta una cámara de video, para crear efectos de video en tiempo real.
- Crear una aplicación que permita realizar seguimiento de objetos en tiempo real a partir de la captura a través de una cámara de video.

Capítulo 6: Aleatoriedad y Perlin noise

Cuando se trabaja con variables, generalmente se espera que estas tomen un valor controlado por el mismo programa que se está desarrollando, o que tomen un valor calculado a partir de una expresión matemática bien definida. Otras veces, para añadir comportamiento imprevisible en los programas, se utilizan funciones que entregan valores aleatorios, por ejemplo se puede realizar una aplicación en el que una figura geométrica cambie de color de forma aleatoria o una aplicación en el que la posición de un objeto en el canvas cambie aleatoriamente (por ejemplo si se desea crear un juego donde los "enemigos" aparezcan en lugares distintos cada vez). Algunas expresiones de arte digital utilizan la aleatoriedad como herramienta para crear geometrías autónomas que van alterando su forma de una manera en la que no se puede predecir.

Se ha revisado anteriormente el uso de la función **`random()`**, que entrega un valor aleatorio entre 0 y 1, **`random(X)`**, que entrega un valor aleatorio entre 0 y **`X`** y **`random(X1,X2)`**, que entrega un valor aleatorio entre **`X1`** y **`X2`**. Como ejemplo de lo anterior puede realizarse una línea cuya altura en cada posición **`x`** tiene una altura aleatoria:

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  noFill();
  beginShape();
  for(let x=0; x<width; x++){
    vertex(x,random(100,300));
  }
  endShape();
  noLoop();
}
```

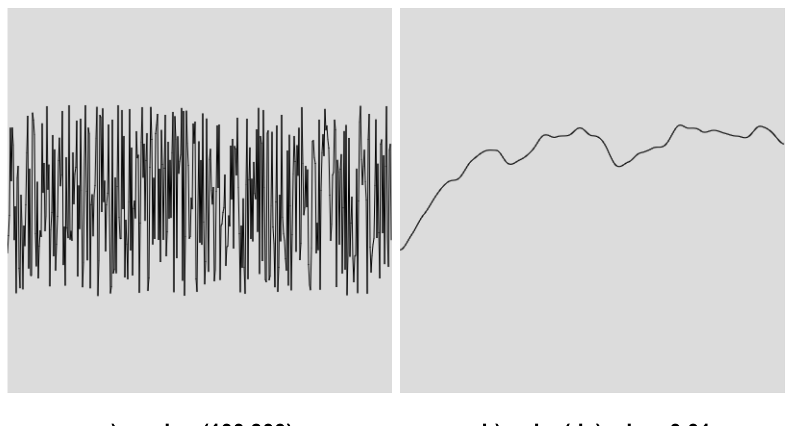
En el ejemplo anterior, la posición **x** utilizada en **vertex()** está perfectamente definida, pero su altura (con **random(100,300)**), tiene un valor aleatorio entre 100 y 300. Este programa dibuja una línea distinta cada vez que se ejecuta el programa. Cuando se incluye la función **noLoop()** el ciclo, que considera la función **draw()**, se detiene.

Otra forma de incluir aleatoriedad a sus programas es utilizando el llamado Perlin noise (ruido Perlin, creado por Ken Perlin para la película TRON del año 1983), que es una técnica que permite obtener valores aleatorios, donde cada valor aleatorio depende de los valores pasados y futuros. Esto permite generar alteraciones más naturales y de alguna manera un poco más controlada comparado con los valores que entrega la función **random()**. La forma de implementar el Perlin noise en **p5.js** es utilizando la función **noise(dx)**, que entrega valores entre 0 y 1, donde el argumento **dx**, deberá ir cambiando su valor continuamente para obtener los resultados esperados. El cómo cambia este valor **dx** será un indicador de la "suavidad" del cambio.

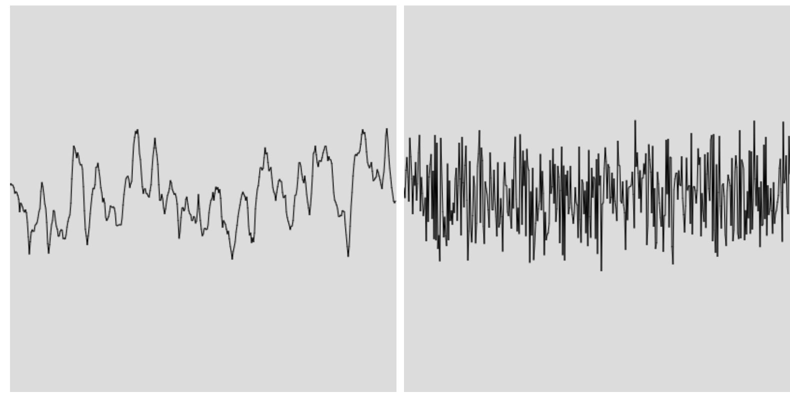
El ejemplo siguiente creará una línea aleatoria con el uso de Perlin noise, donde se observará que los cambios en altura son aleatorios, pero esta vez sus variaciones no serán tan bruscas como al utilizar la función **random()**. Esta "suavidad" se puede controlar entregando un valor (**inc=0.01**) con la que se incrementa el argumento **dx** dentro de la función **noise()**.

```
let dx=0;
let inc=0.01;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  noFill();
  beginShape();
  for(let x=0; x<width; x++){
    vertex(x,noise(dx)*200+100);
    dx+=inc;
  }
  endShape();
  noLoop();
}
```

En el ejemplo anterior, la altura de cada punto de la línea se calcula con **`noise(dx)*200+100`**, como la función **`noise()`** entrega valores entre 0 y 1, entonces al multiplicarse por 200 se obtendrán valores entre 0 y 200, pero considerando que finalmente se suma 100 a esta expresión, entonces quiere decir que la altura de cada punto tendrá una posición entre 100 y 300. En el ejemplo, la variable **`dx`** cambia su valor de 0.01 en 0.01.



a) random(100,300) **b) noise(dx) y inc=0.01**



c) noise(dx) y inc=0.1 **d) noise(dx) y inc=1**

Figura 6.1. Diferencia entre los usos de random() y noise() para generar una línea.

La figura 6.1. **a)** muestra una línea en el que para cada posición x, la altura tiene un valor aleatorio entre 100 y 300. Las figuras 6.1. **b)**, 6.1. **c)** y 6.1. **d)** muestran líneas creadas con Perlin noise, diferenciándose solamente en la forma en la que se incrementa el argumento de la función **`noise(dx)`** del ejemplo. Observe que incrementando el valor de la variable **`dx`** en una cantidad igual a 1, se obtiene un resultado similar a utilizar la función **`random()`**.

Extendamos las situaciones revisadas anteriormente a una situación en dos dimensiones, un plano. Para ello, imagine una cuadrícula en el que cada casilla de esta cuadrícula tiene un tono aleatorio en escala de grises. Esta situación puede implementarse utilizando un ciclo doble, uno de estos ciclos recorre los valores de **`x`** y el otro ciclo recorre los valores de **`y`**, donde **`x`** e **`y`** serán las posiciones de las casillas que tendrán un color aleatorio, cada casilla será un cuadrado. El código que permite realizar lo anterior puede revisarse a continuación:

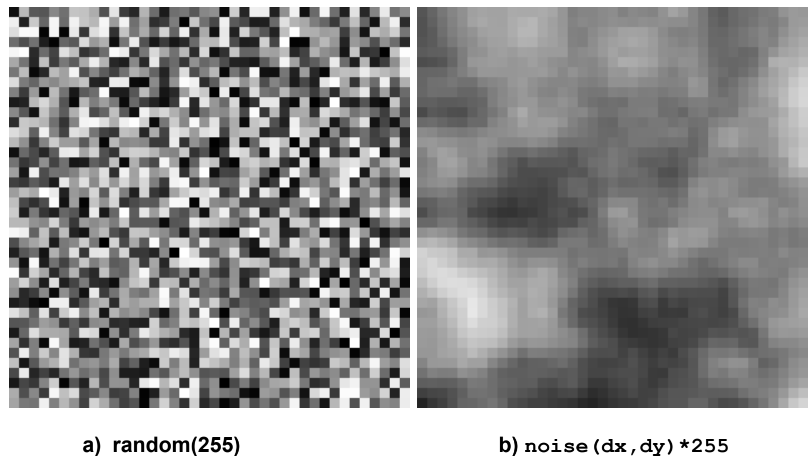
```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  for(let y=0; y<height; y+=height/40){
    for(let x=0; x<width; x+=width/40){
      noStroke();
      fill(random(255));
      rect(x,y,10,10);
    }
  }
  noLoop();
}
```

Podemos intentar algo similar con Perlin noise y dar color a cada casilla de la cuadrícula, pero esperando que esta vez cada casilla tenga un tono similar a los colores de las casillas adyacentes. Esto se puede realizar con la función **`noise(dx,dy)`** que es capaz de recibir dos argumentos para situaciones como estas en dos dimensiones. El siguiente ejemplo realiza lo descrito:

```
let dx;
let dy=0;
let inc=0.1;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  for(let y=0; y<height; y+=height/40){
     dx=0;
    for(let x=0; x<width; x+=width/40){
      noStroke();
      fill(noise(dx,dy)*255);
      rect(x,y,10,10);
      dx+=inc;
    }
   dy+=inc;
  }
```

```
  noLoop();
}
```

En el ejemplo anterior, la variable **dx** incrementa su valor en el ciclo **for()** interior. A su vez, en el ciclo **for()** externo la variable **dy** incrementa su valor en cierta cantidad. El valor de la variable **dx** se reinicia a 0 (cero) justo antes y cada vez que se comienza a recorrer el ciclo interno.



a) random(255) **b) noise(dx,dy)*255**

Figura 6.2. Diferencias entre random() y noise() en una distribución superficial.

Con la distribución aleatoria generada mediante el uso de Perlin noise en las celdas del plano representado en el canvas, es factible crear una animación en la que cada celda experimente cambios de tono suavemente. Esto resulta en una animación donde los colores de las celdas varían continuamente, manteniendo su carácter aleatorio. Esta tarea es viable debido a que la función **noise()** también opera en tres dimensiones, permitiendo la inclusión de tres argumentos, como por ejemplo: **noise(dx, dy, dz)**.

En el ejemplo que sigue, se añade un incremento para la variable **dz**, que es la que logrará el efecto de que el tono de cada casilla cambie mínimamente por cada pasada del ciclo generado por la función **draw()**.

Esta vez se debe reiniciar el valor de la variable **dy** antes del ciclo externo y se debe reiniciar la variable **dx**, justo antes del ciclo interno.

```
let dx;
let dy;
let dz=0;
let inc=0.1;
let cuan=40;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  dy=0;
  for(let y=0; y<height; y+=height/cuan){
     dx=0;
    for(let x=0; x<width; x+=width/cuan){
      noStroke();
      fill(noise(dx,dy,dz)*155+100,noise(dx,dy,dz)*155+100,255);
      rect(x,y,width/cuan,height/cuan);
      dx+=inc;
    }
   dy+=inc;
  }
  dz+=0.01;
}
```

Si bien se utilizó el Perlin noise para alterar el tono de las casillas presentes en el canvas, este puede utilizarse para alterar aleatoriamente su posición, su tamaño, su rotación, entre otras características gráficas.

Resumen

Se revisó el concepto de aleatoriedad en **p5.js**, que se utiliza principalmente para introducir variabilidad a las aplicaciones. Específicamente, el Perlin noise se utiliza para crear patrones suaves y coherentes en proyectos creativos y visuales. Estos conceptos son fundamentales para dar vida a elementos gráficos y animaciones con un aspecto más orgánico y natural.

Lo revisado en este capítulo es útil para crear variaciones en la posición, tamaño, color y otros atributos de elementos gráficos, lo que agrega un aspecto dinámico a las creaciones visuales.

Algunas de las aplicaciones que puede crear son:

- Crear una aplicación que simule el movimiento de nubes o de fluidos.
- Crear una aplicación que utilice Perlin noise para generar un terreno donde las variaciones de altura a lo largo de la superficie no sean tan pronunciadas.
- Crear una aplicación que genere patrones artísticos y abstractos y que cambie aleatoriamente con el tiempo.
- Crear una aplicación que utilice Perlin noise que permita mover una o varias partículas de un sistema.

Capítulo 7. Ejemplos de aplicaciones

Los siguientes ejemplos hacen uso de las mayorías de las herramientas y funciones revisadas en los capítulos anteriores. Se recomienda probar los códigos que siguen y efectuar los cambios que considere necesario, si se produce algún error, trate de corregirlo y/o vuelva a empezar. Respalde, comente y revise a detalle cada uno de los cambios que está realizando o también si es que está escribiendo un programa desde cero.

7.1. Movimiento Parabólico

De la teoría de un lanzamiento parabólico se obtiene que las ecuaciones para el eje x e y en función del tiempo son:

$$x(t) = x_0 + v_{0x}\, t$$

$$y(t) = y_0 + v_{0y}\, t - \frac{1}{2} g\, t^2$$

Si se desea representar en pantalla este tipo de movimiento, necesitamos que nuestro objeto dibujado en el canvas vaya cambiando su posición de acuerdo a estas ecuaciones. La forma más simple posible de lograr este objetivo es escribir un programa que cumpla con lo siguiente: "*Un programa que simule un lanzamiento parabólico de un objeto circular (que sería el proyectil) dentro de un entorno visual cuadrado, donde se pueda elegir la posición inicial* x_0 *e* y_0*, el ángulo de salida del proyectil, la rapidez con la que es lanzado y la aceleración de gravedad. Además, que cada vez que se ejecute el código, el fondo de la simulación cambie de color aleatoriamente, y si el proyectil sale del entorno visual de la pantalla, se reinicie el lanzamiento.*"

Un código que cumple con esta descripción es el que se muestra a continuación.

```
// Esto es un comentario y no tiene incidencia en el código
let x0, y0, angulo0, t, v0, g, x, y; //Se declaran las variables
a utilizar
let R, G, B; //Se declaran las variables para el color de fondo

// La función setup se ejecuta una sola vez
function setup() {
  createCanvas(400, 400); //Se genera el entorno visual de
400x400 pixeles
  x0 = 50; //Posición inicial eje x
  y0 = 150; //Posición inicial eje y
  angulo0 = 60; //Ángulo inicial de lanzamiento
  t = 0; //Se inicia el tiempo a t = 0
  v0 = 60; //Rapidez inicial
  g = 9.8; //Aceleración de gravedad

  R = random(255); //Valor aleatorio entre 0 y 255 (Rojo)
  G = random(255); //Valor aleatorio entre 0 y 255 (Verde)
  B = random(255); //Valor aleatorio entre 0 y 255 (Azul)
}

//La función draw se repite una y otra vez
function draw() {
  background(R, G, B,30); //Se establece el color de fondo
  x = x0 + v0*cos(angulo0*PI/180)*t; //Se calcula posición x del
proyectil en cada instante
  y = y0 + v0*sin(angulo0*PI/180)*t-0.5*g*pow(t,2); //Se calcula
posición y del proyectil en cada instante
  stroke(0); //El proyectil se dibuja con una borde negro
  fill(255, 0, 0); //El proyectil se rellena de color rojo
  ellipse(x, height - y, 16, 16); //El proyectil es un círculo de
diámetro 16 pixeles

  t += 0.1; //La simulación avanza cada 0.1 segundos, para más
lento disminuir este número

  //La siguiente condición establece que si el proyectil sale de
la pantalla, el tiempo se reinicia
  if (height - y > height || x > width || x < 0) {
    t = 0;
  }
}
```

Una vez con el código escrito (o copiado) en el editor, se puede probar cambiando cualquiera de los valores iniciales que están ingresados. Por ejemplo, se puede cambiar el ángulo de salida del proyectil, la rapidez o la aceleración de gravedad y se observará cuál es el efecto que se produce en el movimiento. Se recomienda al lector alterar el código presentado y lograr una interactividad mayor con la aplicación, añadiendo botones, deslizadores o hacer que el programa interactúe con el mouse o con el teclado.

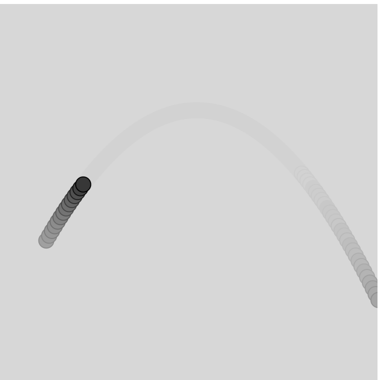


Figura 7.1. Lanzamiento parabólico.

7.2. Funciones y derivadas

La idea es poder crear una gráfica de un polinomio de grado 4, de una función $x(t)$, donde x y t pueden ser dos variables cualquiera, pero se considerarán como si fueran posición y tiempo, respectivamente, para darle un contexto físico. La función a graficar es de la forma:

$$x(t) = a\,t^4 + b\,t^3 + c\,t^2 + d\,t + e$$

Donde a, b, c, d y e son los factores del polinomio $x(t)$, y que pueden modificarse para observar cómo cambia la gráfica. Además, se tendrá la opción de representar la derivada de la función $x(t)$ en cada punto de la curva. Si x es posición y t el tiempo, entonces la derivada de $x(t)$ con

respecto al tiempo (que se denota $\frac{dx}{dt}$) corresponde a la velocidad en cada instante y se calcula de la siguiente manera:

$$\frac{dx}{dt} = 4a\,t^3 + 3b\,t^2 + 2c\,t + d$$

En matemáticas, calcular la derivada de una función significa calcular la pendiente de la función en cada uno de sus puntos.

Si x se mide en metros ([m]) y t en segundos ([s]), entonces la velocidad (o derivada de $x(t)$ en cada instante t) se mide en metros por segundo ([m/s]). Se incluirá la opción de reiniciar el programa y volver a cero todos los factores del polinomio.

```
let a, b, c, d, e; //declarar factores del polinomio

function setup() {
  createCanvas(800, 600);
  //acá se crean los deslizadores para las variables
  A = createSlider(-15, 15, 0, 30 / width);
  A.position(10, 10);
  A.size(80);
  B = createSlider(-15, 15, 0, 30 / width);
  B.position(10, 30);
  B.size(80);
  C = createSlider(-15, 15, 0, 30 / width);
  C.position(10, 50);
  C.size(80);
  D = createSlider(-15, 15, 0, 30 / width);
  D.position(10, 70);
  D.size(80);
  E = createSlider(-150, 150, 0, 30 / width);
  E.position(10, 90);
  E.size(80);

  // se crea el check para la Derivada
  checkbox = createCheckbox('', false);
  checkbox.position(10, 115);

  //se crea el botón de reinicio
  button = createButton('Inicio');
```

```
  button.position(10, height-30);
  button.size(50, 20);
  button.mousePressed(Resetear);
}

function draw() {
  background(0); //Se establece fondo negro
  textSize(14); //Tamaño de texto en 14
  noStroke();
  fill(0, 255, 0);

  //Se adquieren los datos de acuerdo a los deslizadores
  // y se muestran en pantalla.
  a = A.value();
  text("a = " + a, 100, 23);
  b = B.value();
  text("b = " + b, 100, 43);
  c = C.value();
  text("c = " + c, 100, 63);
  d = D.value();
  text("d = " + d, 100, 83);
  e = E.value();
  text("e = " + e, 100, 103);

  text("Derivada", 35, 129);
  text("x", width / 2 - 10, 10);
  text("t", width - 8, height / 2 + 12);

  push();
  translate(width / 2, height / 2); //traslada el origen (0,0) a
la mitad del canvas
  noFill();
  beginShape(); //inicio de gráfica de la curva
  for (let j = -width / 2; j < width / 2; j++) { //ciclo que
calcula la función
    let t1 = map(j, -width / 2, width / 2, -15, 15); //valores de
t
    let treal = map(t1, -15, 15, -width / 2, width / 2); //t en
pantalla
    let x1 = -(a * pow(t1, 4) + b * pow(t1, 3) + c * pow(t1, 2) +
d * t1 + e); //valores de x
    let xreal = map(x1, -200, 200, -height / 2, height / 2); //x
en pantalla
    stroke(255, 0, 0); //línea de color rojo
```

```
    vertex(treal, xreal); //dibuja cada uno de los puntos de la
curva
  } //fin del ciclo
  endShape(); //fin de gráfica de la curva
  pop();

  //Se calcula y muestra un punto sobre la curva de acuerdo a
posición del mouse
  let tt = map(mouseX - width / 2, -width / 2, width / 2, -15,
15);
  let mousexx = -(a * pow(tt, 4) + b * pow(tt, 3) + c * pow(tt, 2)
+ d * tt + e);
  let posX = map(mousexx, -200, 200, -height / 2, height / 2);
  ellipse(mouseX, posX + height / 2, 4, 4);

  noStroke();
  text("x = " + (-mousexx), width-200, 20); //muestra valor de x
en pantalla
  text("t = " + tt, width-200, 40); //muestra valor de t en
pantalla

  stroke(0, 255, 0); //ejes de color verde
  line(0, height / 2, width, height / 2); //eje t
  line(width / 2, 0, width / 2, height); //eje x

  if (checkbox.checked()) { //activa y calcula derivada
(tangente a la curva)
    let derivada = -(4 * a * pow(tt, 3) + 3 * b * pow(tt, 2) + 2 *
c * tt + d);
    noStroke();
    text("dx/dt = " + (-derivada), width-200, 60);

    push(); //dibuja la tangente a la curva
    translate(mouseX, posX + height / 2);
    rotate(atan(derivada/18)); //se inclina la línea de acuerdo a
valor de la derivada
    strokeWeight(2); //se ajusta grosor de la recta
    stroke(0, 0, 255); //línea de color azul
    line(-40, 0, 40, 0);
    pop(); //fin dibujo tangente
  }
}

function Resetear() { //función que reinicia las variables
```

```
  A.value(0);
  B.value(0);
  C.value(0);
  D.value(0);
  E.value(0);
  checkbox.checked(false);
}
```

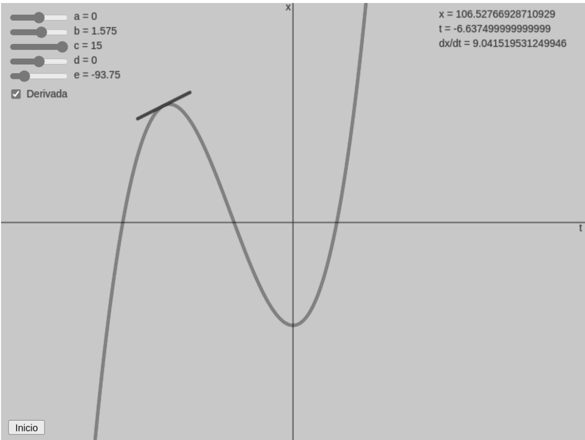


Figura 7.2. Polinomio de grado 4 y su derivada.

7.3. Onda viajera

Se considera una onda como una perturbación en el medio, este medio puede ser aire, agua, una cuerda, o el vacío si se considera una onda electromagnética. Una onda viajera será una onda que viaja de un lugar a otro y que transporta energía de un lugar a otro. Específicamente, una onda transversal es una onda en la cual la perturbación se genera de forma perpendicular con respecto a la dirección en la que avanza la onda.

En esta oportunidad, la idea es generar una animación que represente una onda viajera y transversal, que cumpla con las condiciones de una onda, que presente cierta velocidad de propagación (c), frecuencia (f), longitud de onda (λ) y amplitud (A). La altura "y" de cada punto "x" de la

onda en cada instante "*t*" se puede calcular utilizando la siguiente expresión, llamada función de onda:

$$y(x,t) = A \cos(2\pi \cdot x/\lambda - 2\pi \cdot f \cdot t)$$

Con el movimiento horizontal del puntero del mouse sobre la pantalla se controlará la frecuencia de oscilación de la onda y con el movimiento vertical del puntero se controlará la amplitud. La velocidad de propagación de la onda se establece como una constante, y a partir de esto y de la frecuencia se puede calcular la longitud de la onda:

$$\lambda = c/f$$

Esta longitud de onda se mostrará en pantalla como referencia, ¿qué sucede con λ si aumenta f?, ¿qué sucede con λ si disminuye f? Puede responder interactuando y observando la animación.

```
let x=[]; //arreglo para la posición en eje x
let y=[]; //arreglo para la posición en eje y
let largo=4; //distancia del ancho de la pantalla
let t=0; //tiempo inicial
let lambda; //longitud de onda
let c=10; //velocidad de la onda
let amplitud; //amplitud de la onda
let f; //variable para la frecuencia

function setup() {
 createCanvas(800, 400); //pantalla de 800 x 400 pixels
  for(let i=0; i<width; i++){ //se recorren todos los puntos del
ancho de la pantalla
  x[i]=i*largo/width; //se calcula la posición x de acuerdo al
ancho de la pantalla y se añade cada dato al arreglo x[i]
  }
 rectMode(CENTER);
 }

function draw() {
  background(0,0,255); //fondo de color azul claro
  noCursor(); //se oculta el cursor en pantalla
```

```
  f=map(mouseX,0,width,1,80); //la posición x del mouse asigna
la frecuencia
  amplitud=map(mouseY,0,height,150,0); //la posición y del mouse
asigna la amplitud
  amplitud=constrain(amplitud,0,150); //se limita la amplitud
entre 0 y 150
  lambda=c/f; //se calcula la longitud de onda λ

  //acá se comienza a dibujar la onda viajera
  stroke(255); //línea blanca
  strokeWeight(3); //grosor 3
  fill(0,0,122); //se rellena de color azul oscuro
  push();
  translate(0,height/2); //se traslada el origen al centro del
lado izquierdo
  beginShape(); //comienza a plantear la figura a dibujar
  vertex(0,height/2); //se añade un punto inicial abajo a la
izquierda
  for(let j=0; j<width; j++){ //se recorren todos los puntos del
ancho de la pantalla
  y[j]=amplitud*cos((2*PI/lambda)*x[j]-2*PI*f*t); //se calcula la
altura de cada punto de la onda de acuerdo a x[j], el tiempo y
la amplitud de la onda.
  vertex(j,y[j]); //se dibuja cada punto de la onda en pantalla
  }
  vertex(width,height/2); //se añade un punto final abajo a la
derecha
  endShape();
  //acá se termina de dibujar la onda

  strokeWeight(2); //línea de grosor 2
  fill(0); //color negro
  stroke(255,100,0); //línea rojiza
  ellipse(width/2,y[int(width/2)],8,8); //círculo de referencia
en la mitad de la onda
  pop();

  noStroke(); //el texto no tiene bordes
  fill(255); //el texto se rellena de color blanco
  textSize(16); //tamaño del texto
  textAlign(CENTER); //se alinea texto en el centro
  text("Longitud de Onda",width/2, height-30); //se añade texto
  rect(width/2, height-20,width/largo*lambda,2); //se dibuja
referencia para longitud de onda
```

```
  ellipse(mouseX,mouseY,4,4); //reemplazo el cursor por un
círculo
  t+=0.001;
}
```

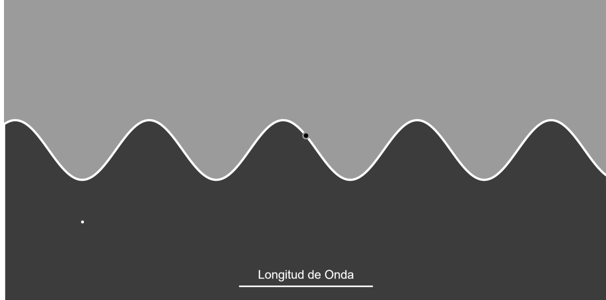


Figura 7.3. Onda transversal viajera.

7.4. Coordenadas polares

En coordenadas polares, cada punto del plano queda representado con la distancia al origen del sistema de referencia y con el ángulo que forma con respecto al eje x de un plano cartesiano. En la figura, el punto P de la gráfica se representa a partir de la distancia r y el ángulo θ que forma con el eje x:

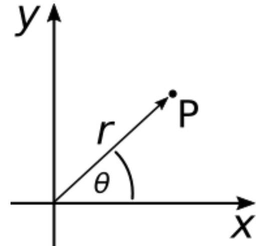


Figura 7.4. Sistema de ejes cartesiano y coordenadas polares.

Considerando conocimientos de trigonometría, se puede obtener que los valores de x e y de cada punto P son, respectivamente:

$$x = r \cdot cos(\theta)$$

$$y = r \cdot sin(\theta)$$

Estos puntos (x,y) son los que se representarán en pantalla.

En esta oportunidad crearemos un programa que sea capaz de graficar en coordenadas polares y que cumpla la siguiente relación:

$$r(\theta) = a + a \cdot cos\ (n \cdot \theta)$$

Donde a y n son parámetros que pueden ser alterados con el movimiento del puntero del mouse para visualizar diferentes formas. Los valores de a y n se mostrarán en pantalla para tener un mejor control sobre ellos. El color de la curva también se podrá cambiar con el movimiento del mouse sobre la pantalla. En la figura se muestran patrones con $\boldsymbol{n = 2.5}$ y $\boldsymbol{n = 8}$, respectivamente.

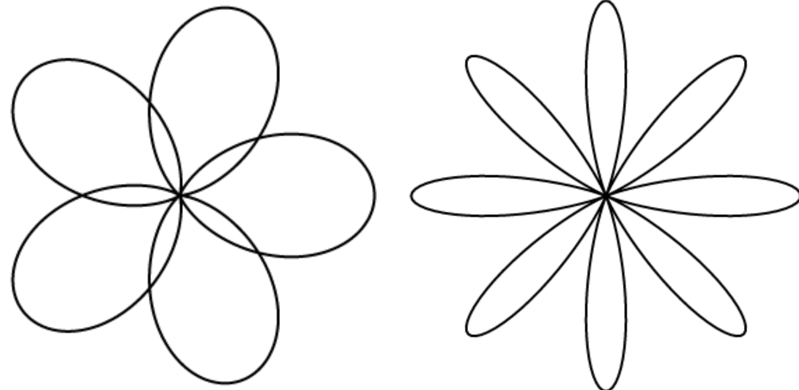


Figura 7.5. Representación de una función de coordenadas polares para dos valores de n.

```
function setup() {
  createCanvas(400, 400);
  angleMode(DEGREES); //el ángulo se cambia a grados
}

function draw() {
  background(0); //fondo negro
```

```
  let n = map(mouseX,0,width,0,10); //valor de n entre 0 y 10
  let a = map(mouseY,0,height,80,5); //valor de a entre 5 y 80

  translate(width/2,height/2); //origen al centro de la pantalla
  noFill(); //sin relleno

stroke(map(mouseY,0,height,200,255),map(mouseX,0,height,180,255),m
ap(mouseX,0,height,180,255)); //línea de color variable
  strokeWeight(2); //grosor de línea igual a 2
  beginShape(); //se comienza a graficar
  for (let ang = 0; ang <= 720; ang++) { //ciclo para el ángulo
hasta un valor 720
    let r = a + a * cos(n*ang); //relación de r con el ángulo
    let x = r * cos(ang); //cambio de coordenadas polar a eje x
    let y = r * sin(ang); //cambio de coordenadas polar a eje y
    vertex(x, y); //se añaden puntos a la curva
  }
  endShape(); //se termina de graficar

  noStroke(); //texto sin borde
  fill(255); //texto blanco
  textSize(16); //tamaño de texto igual a 16
  text("a = "+a,-180,170); //muestra el valor de a
  text("n = "+n,-180,190); //muestra el valor de n
}
```

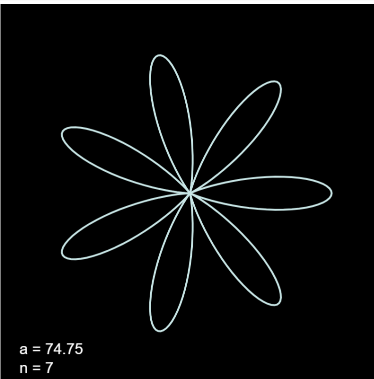


Figura 7.6. Coordenadas polares.

7.5. Código de colores para resistencias eléctricas

Una forma de determinar el valor de una resistencia eléctrica es a través de un código de colores. Para ello se utilizan tres bandas de colores y a cada color se le asigna un dígito entre 0 y 9. Luego con una operación matemática simple se obtiene el valor de la resistencia en Ohm [Ω].

El código de colores es el siguiente:

Color										
Dígito	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Por ejemplo, si el primer color es rojo, el segundo café y el tercero naranjo, que corresponden a los dígitos 2, 1 y 3, respectivamente, el cálculo se realiza de la siguiente forma:

$$R = (2 \cdot 10 + 1) \cdot 10^3 = 21000\ [\Omega]$$

De forma general, si c1, c2 y c3 son los dígitos asociados a tres colores, entonces el valor de la resistencia se puede calcular a partir de:

$$R = (c1 \cdot 10 + c2) \cdot 10^{c3}\ [\Omega]$$

El código que sigue a continuación permite calcular el valor de la resistencia eligiendo los colores que forman parte de una resistencia real.

```
let colores=[];
let col1=0;
let col2=0;
let col3=0;
let c1=0;
let c2=0;
let c3=0;

function setup() {
  createCanvas(600, 400);
```

```
  colores[0]=color(0,0,0);            //negro    0
  colores[1]=color(168,115,50);       //café     1
  colores[2]=color(255,0,0);          //rojo     2
  colores[3]=color(255,193,8);        //naranjo  3
  colores[4]=color(255,255,0);        //amarillo 4
  colores[5]=color(0,255,0);          //verde    5
  colores[6]=color(0,0,255);          //azul     6
  colores[7]=color(225,0,255);        //morado   7
  colores[8]=color(155,155,155);      //gris     8
  colores[9]=color(255,255,255);      //blanco   9
}

function draw() {
  background(220);

  push(); //dibujo la resistencia (fondo)
  strokeWeight(4);
  stroke(150);
  line(0,290,width,290);
  strokeWeight(2);
  stroke(255);
  line(0,294,width,294);
  noStroke();
  fill(245, 232, 162);
  rect(100,230,280,120,15);
  pop();

  push(); //defino los selectores de color
  for(let y=0; y<10; y++){
    for(let x=0; x<3; x++){
      fill(colores[y]);
      stroke(0);
      rect(40*x+150,20+y*20,30,20);
    }
  }
  pop();

  noStroke();
  fill(col1); //dibujo y pinto primera franja de color
  rect(150,230,30,120);

  fill(col2); //dibujo y pinto segunda franja de color
  rect(190,230,30,120);
```

```
  fill(col3); //dibujo y pinto tercera franja de color
  rect(230,230,30,120);
  push();
  //escribo los valores asociados a cada color y el valor de la
resistencia
  fill(0);
  textSize(30);
  text(c1,155,380);
  text(c2,195,380);
  text(c3,235,380);
  text("= "+(c1*10+c2)*pow(10,c3)+" [Ω]",260,380);
  textAlign(CENTER);
  text("Código de colores para resistencias eléctricas, expresadas
en Ω (ohm)",290,50,310,400);
  pop();
}

function mousePressed(){
  for(let y=0; y<10; y++){  //elijo la primera banda de color
  if(mouseX>150 && mouseX<180 && mouseY>20+y*20 &&
mouseY<40+y*20){
     c1=y;
    col1=colores[y];
  }
  }

  for(let y=0; y<10; y++){  //elijo la segunda banda de color
  if(mouseX>190 && mouseX<220 && mouseY>20+y*20 &&
mouseY<40+y*20){
     c2=y;
    col2=colores[y];
  }
  }

   for(let y=0; y<10; y++){  //elijo la tercera banda de color
  if(mouseX>230 && mouseX<260 && mouseY>20+y*20 &&
mouseY<40+y*20){
     c3=y;
    col3=colores[y];
  }
  }
}
```

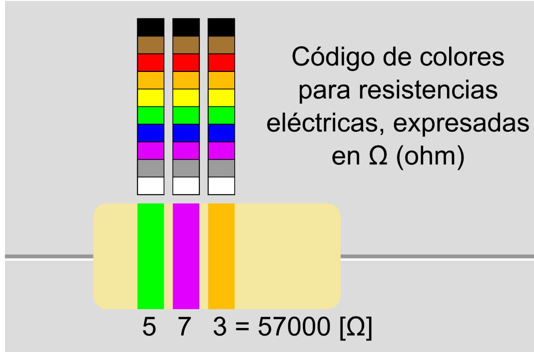


Figura 7.7. Cálculo de resistencias con el código de colores.

7.6. Campo eléctrico

El campo eléctrico es un concepto físico relacionado con la presencia de cargas eléctricas, se dice que cargas eléctricas positivas y negativas generan un campo eléctrico en todo el espacio que las rodea. El campo eléctrico es un vector, por lo que puede representarse con magnitud, dirección y sentido. Las cargas eléctricas positivas generan un campo eléctrico que apunta desde la carga y hacia afuera de ella, en cambio, las cargas eléctricas negativas generan un campo eléctrico que apunta hacia la carga.

Se representará en un programa, un código que muestra el campo eléctrico de una carga puntual positiva y de una carga eléctrica negativa al hacer clic con el mouse, en diferentes puntos del canvas. Se hará uso de un objeto que incluye la información de magnitud, ángulo y transparencia del color con la que es representado el campo eléctrico. Este objeto se utilizará varias veces, para representarlo en la totalidad del canvas con cierta espacialidad entre ellos. Para definir esto utilizaremos una clase en un archivo distinto, que es donde se definirá el objeto.

Por lo anterior, es necesario que el archivo `index.html` contenga no solo el llamado a la biblioteca **p5.js** y al código del programa `sketch.js`, sino

que también invoque al archivo donde fue generado el objeto, llamado en este caso `flechas.js`.

Cada uno de estos tres archivos está mostrado a continuación. Para el correcto funcionamiento del programa, asegúrese que la información que contienen estos tres archivos es la correcta.

Archivo `index.html`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.5.0/p5.js"></s
cript>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.5.0/addons/p5.
sound.min.js"></script>
    <link rel="stylesheet" type="text/css" href="style.css">
    <meta charset="utf-8" />

  </head>
  <body>
    <main>
    </main>
    <script src="sketch.js"></script>
    <script src="flechas.js"></script>
  </body>
</html>
```

Archivo `sketch.js`:

```
let flechas = [];
let cuantos = 15;
let signo = 1;

function setup() {
  createCanvas(600, 400);
  textAlign(CENTER, CENTER);
  for (let j = 0; j < cuantos; j++) { //arreglo de flechas
    for (let i = 0; i < cuantos; i++) {
      let posX = (i * width) / cuantos + width / (2 * cuantos);
```

```
      let posY = (j * height) / cuantos + height / (2 * cuantos);
      flechas.push(new Flechas(posX, posY));
    }
  }
}

function draw() {
  noCursor();
  background(200, 200, 0);
  for (let flecha of flechas) { //una forma de invocar a todos
los objetos
    flecha.show();  //función creada en la clase Flechas
  }
  fill(255, 125, 30);
  ellipse(mouseX, mouseY, 16, 16); //dibujo un círculo como carga
  textSize(20);
  if (signo == 1) {
    fill(0);
    text("+", mouseX, mouseY);
  } else {
    fill(0);
    text("-", mouseX, mouseY);
  }
}

function mouseReleased() { //al hacer clic se cambia el signo de
la carga
  signo*=-1;
}
```

Archivo `flechas.js`:

```
class Flechas {
  constructor(x, y) { //parámetros que recibirá cada flecha
    this.x = x;
    this.y = y;
  }

  show() { //función que dibuja cada flecha
    push();
    let d = dist(mouseX, mouseY, this.x, this.y);
    let alfa = map(d, 0, 500, 255, 0);
    let largo = map(d, 0, 500, 20, 2);
```

```
    stroke(0, 0, 0, alfa);
    strokeWeight(3);
    translate(this.x, this.y);
    rotate(atan2(signo * (-mouseY + this.y), signo * (-mouseX +
this.x))); //se gira el ángulo de acuerdo a la posición del mouse
    line(largo, 0, largo - 3, -3);
    line(largo, 0, largo - 3, 3);
    line(-largo, 0, largo, 0);
    pop();
  }
}
```

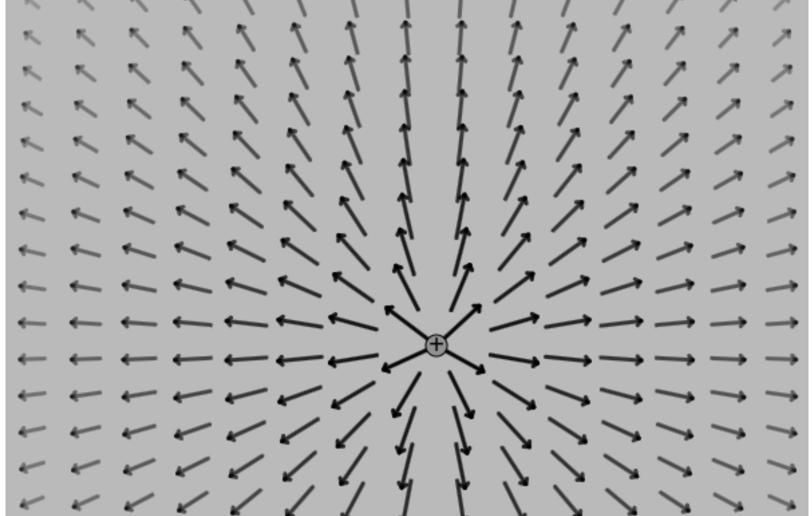


Figura 7.8. Representación de campo eléctrico generado por una carga positiva.

7.7. Tabla periódica (JSON)

El siguiente ejemplo de aplicación pretende crear una tabla periódica que muestre algunas características de los elementos químicos de la tabla periódica. Para obtener la información de cada elemento químico utilizaramos un tipo de archivo que permite guardar variada información de muchos elementos. Este tipo de archivo tiene una extensión **`*.json`**. Por lo tanto, a continuación se muestran los dos

archivos necesarios para crear esta aplicación, el primero será el archivo donde se editará el código del programa (`sketch.js`) y el segundo será un extracto del archivo `json`, que contiene toda la información de todos los elementos químicos.

```
let tabla;
let col = [];
let mx = 0;
let my = 0;
let control = 0;

function preload() {  //Se carga el archivo json
  tabla = loadJSON("tabla.json");
}

function setup() {
  createCanvas(800, 600);
  for (let i = 0; i < tabla.elements.length; i++) {
    col[i] = "#" + tabla.elements[i].cpk_hex; //color del
elemento
  }
}

function draw() {
  background(36, 95, 138);
  push();
  textAlign(CENTER);
  fill(255);
  textSize(30);
  text("Tabla periódica", width / 2, 40);
  pop();
  for (let i = 0; i < tabla.elements.length; i++) {
    let x = tabla.elements[i].xpos * 42 - 20; //posición x
    let y = tabla.elements[i].ypos * 50; //posición y
    push();
    stroke(0, 100);
    fill(color(col[i]));
    rect(x, y, 42, 50);
    textAlign(CENTER);
    textSize(20);
    fill(0);
    text(tabla.elements[i].symbol, x + 20, y + 30); //símbolo
    pop();
```

```
  }

  for (let i = 0; i < tabla.elements.length; i++) {
    let x = tabla.elements[i].xpos * 42 - 20;
    let y = tabla.elements[i].ypos * 50;
    if (mx >= x && mx < x + 42 && my >= y && my < y + 50) {
      push();
      strokeWeight(4);
      if (col[i] === "#null") {
        fill(color("#ffffffdd"));
      } else {
        fill(color(col[i] + "dd"));
      }
      rectMode(CENTER);
      rect(width / 2, height / 2, 420, 500, 10, 10, 10, 10);
      fill(0);
      textSize(180);
      textAlign(CENTER);
      text(tabla.elements[i].symbol, width / 2, 240);
      textSize(28);
      text(tabla.elements[i].name, width / 2, 300); //nombre
      textSize(18);
      fill(255);
      text(tabla.elements[i].electron_configuration, width / 2,
580);  //configuración electrónica
      pop();
      textSize(20);
      fill(0);
      text(tabla.elements[i].atomic_mass, 210, 450); //masa
atómica
      text(tabla.elements[i].number, 565, 80);
     //se dibujan las capas atómicas
      for (let j=0; j<tabla.elements[i].shells.length; j++) {
        fill(0, 0, 255);
        circle(515, 450, 10);
        noFill();
        circle(515, 450, j * 20 + 30);
        for (let k=0; k<tabla.elements[i].shells[j]; k++) {
          push();
          translate(515, 450);
          rotate((k * 2 * PI) / tabla.elements[i].shells[j]);
          if (j == tabla.elements[i].shells.length - 1) {
            strokeWeight(2);
            fill(255, 250, 0);
```

```
          } else {
            fill(255, 0, 0);
          }
          circle((j * 20 + 30) / 2, 0, 8);
          pop();
        }
      }
    }
  }
}

function mousePressed() { //Se detecta en qué elemento se hace
clic
  mx = mouseX;
  my = mouseY;
}
```

La variable que contiene el archivo `json` es llamado **`tabla`**, como dentro del archivo `json` los elementos están agrupados en un arreglo llamado **`elements`**, entonces se puede saber la cantidad de elementos químicos con la función **`tabla.elements.length`**. Además, se puede acceder a cada elemento utilizando la opción **`tabla.elements[i]`**, donde **`i`** es el índice de un elemento cualquiera. El archivo `json` incluye para cada elemento: su nombre, su masa atómica, su densidad, su símbolo, configuración electrónica, y otras características. Se puede acceder a cualquiera de estas características de cada elemento, por ejemplo, si se desea obtener el valor de la masa atómica del elemento **`i`** se puede utilizar **`tabla.elements[i].atomic_mass`**, y se hace de forma similar para acceder a otras características. La característica **`atomic_mass`**, así como todas las demás, se encuentran definidas en el archivo json.

A continuación se muestra un extracto de la información que se muestra en el archivo `json` para un solo elemento, este es el formato con el que está registrado cada unos de los elementos en el archivo completo. Para acceder al archivo `json` completo, puede visitar el repositorio en Github previamente señalado[7].

[7] https://github.com/sbsacev/cursop5js/blob/main/cap07/01_7/tabla.json

```
{
    "elements": [
        {
            "name": "Hidrógeno",
            "appearance": "colorless gas",
            "atomic_mass": 1.008,
            "boil": 20.271,
            "category": "diatomic nonmetal",
            "density": 0.08988,
            "discovered_by": "Henry Cavendish",
            "melt": 13.99,
            "molar_heat": 28.836,
            "named_by": "Antoine Lavoisier",
            "number": 1,
            "period": 1,
            "phase": "Gas",
            "summary": "Hydrogen is a chemical element with
chemical symbol H and atomic number 1. With an atomic weight of
1.00794 u, hydrogen is the lightest element on the periodic table.
Its monatomic form (H) is the most abundant chemical substance in
the Universe, constituting roughly 75% of all baryonic mass.",
            "symbol": "H",
            "xpos": 1,
            "ypos": 1,
            "shells": [
                1
            ],
            "electron_configuration": "1s1",
            "electron_configuration_semantic": "1s1",
            "electron_affinity": 72.769,
            "electronegativity_pauling": 2.2,
            "ionization_energies": [
                1312
            ],
            "cpk_hex": "ffffff"
        },
.
.
.
```


Este archivo continúa con la información de todos los elementos.

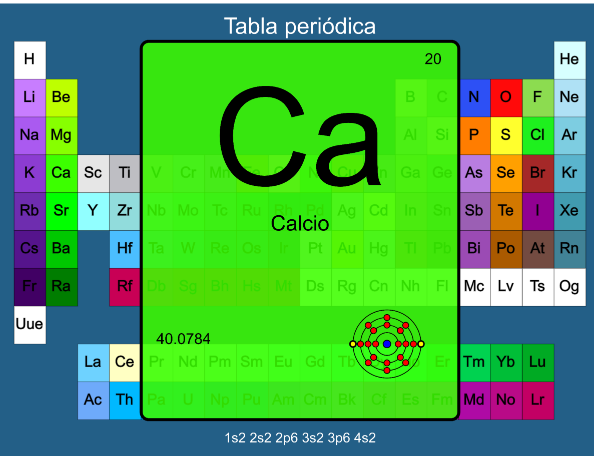


Figura 7.9. Tabla periódica utilizando la información de un archivo JSON.

¿Cómo continuar?

Si ha llegado hasta aquí y considera que los conocimientos que ha adquirido son suficientes para crear sus propias aplicaciones, o si ya ha creado sus propias aplicaciones con **p5.js**, es posible que se esté preguntando con qué otros recursos de programación puede ser útil continuar para crear aplicaciones distintas, con otras características y con mayores recursos que los revisados en este libro.

Inicialmente se recomienda continuar con Processing[8], que es mantenida por la Processing Foundation[9], la misma fundación que mantiene el proyecto de la biblioteca **p5.js**. Processing permite crear aplicaciones basadas en Java de forma local en el propio computador, a diferencia de **p5.js** que está basado en JavaScript para crear aplicaciones para sitios web.

Processing permite, entre otras cosas, hacer una conexión con placas de desarrollo para interactuar con sensores a través de las aplicaciones que se crean para obtener resultados visibles, por lo que se sugiere indagar en esta posibilidad.

Con **p5.js** también puede crear aplicaciones web que interactúen con placas de desarrollo, pero se requiere un software distinto que haga la "conexión" entre la placa y la biblioteca **p5.js**, es decir, si requiere utilizar una placa de desarrollo en sus proyectos se recomienda profundizar e intentarlo directamente a través de Processing.

Si su interés es el arte y las formas de expresión digital puede recurrir a bibliografía que existe para eso, en este sentido, se recomienda el libro "Generative art: a practical guide using processing" de Matt Pearson, donde encontrará un desarrollo de las herramientas que le pueden ser de

[8] https://processing.org/ sito oficial de Processing.
[9] https://processingfoundation.org/ sitio web de la Processing foundation.

utilidad. Si bien este libro está enfocado en Processing, es perfectamente aplicable a **p5.js** haciendo algunos ajustes en los códigos que se presentan.

Junto con la biblioteca **p5.js** pueden utilizarse una gran variedad de otras bibliotecas para extender las posibilidades de las aplicaciones que está creando, algunas de estas bibliotecas están agrupadas en el sitio en el sitio web de la biblioteca[10], donde algunas sirven para crear botones, otra para detectar colisiones de objetos, entre muchas más. Una de ellas es ml5[11] que permite experimentar y crear aplicaciones con inteligencia artificial de forma sencilla (ml viene de machine learning) bajo la misma filosofía de **p5.js**. Le permitirá crear aplicaciones que reconozca posturas corporales, que reconozca rostros, posiciones de la mano, entre otras posibilidades. Por lo que es una buena manera de continuar si desea aplicar más tecnología e interactividad en sus aplicaciones con recursos de la inteligencia artificial.

Puede continuar con sus proyectos creativos y académicos entrenando un sistema que le permita adquirir información de la cámara de video o a través de un micrófono. Para esto existe una herramienta de la compañía Google que facilita este entrenamiento[12], donde puede entrenar un sistema para detectar objetos, animales, rostros, posturas corporales, sonidos, estilos de música, por nombrar algunos ejemplos, para crear un modelo de aprendizaje automático que puede ser importado en **p5.js** para ser utilizado en sus aplicaciones web.

Mucha de la información presentada en este libro se encuentra de forma online, por lo que recursos de búsqueda online y plataformas de video son un buen lugar donde recurrir si busca la solución a algún problema en particular que le pueda surgir.

---

[10] https://p5js.org/es/libraries/ listado de bibliotecas que se pueden utilizar con p5.js.
[11] https://ml5js.org/ sitio web de la biblioteca de Machine learning para p5.js.
[12] https://teachablemachine.withgoogle.com/ herramienta de Google para entrenar modelos.

Glosario

Arreglo: Lista de variables que comparten el mismo nombre y que apuntan a múltiples ubicaciones de la memoria del computador.

Bezier: Una curva Bézier es una curva matemática utilizada en gráficos vectoriales, diseño gráfico y animación para representar curvas suaves. Estas curvas se definen mediante un conjunto de puntos de control que determinan la forma y el trayecto de la curva.

Canvas: Es el lienzo o área de dibujo donde se observa gráficamente los resultados del código que se está escribiendo.

Ciclo: Proceso que permite la iteración de una o varias reglas o pasos una y otra vez en pocas líneas de código.

Condición: Expresión booleana, es decir, una expresión que puede tener valor verdadero o falso. Usado como argumento para la función condicional `if()`.

CSS: Cascading Style Sheets (o Hoja de Estilo de Cascadas en español) es un lenguaje de estilo utilizado para definir la presentación y el diseño de documentos HTML. Se utiliza para controlar cómo se ve y se presenta el contenido en una página web.

DOM: Document Object Model (Modelo de Objetos del Documento en español). El DOM se utiliza para interactuar y manipular los elementos de una página web a través de scripts o código JavaScript, lo que permite cambiar dinámicamente el contenido y el comportamiento de una página.

Espectro: Representación gráfica de la amplitud o intensidad de cada frecuencia de una señal, onda o fenómeno.

FFT: Fast Fourier Transform (o Transformada Rápida de Fourier). Es una herramienta matemática que permite obtener el espectro de frecuencia de una señal, onda o fenómeno.

Función: En programación, corresponde a una estructura o bloque de código que encapsula una serie de instrucciones o acciones específicas. Estas funciones se utilizan para organizar y modular el código, lo que facilita la creación de programas y proyectos más grandes y complejos.

Github: Plataforma en línea que se utiliza para gestionar proyectos de desarrollo de software, control de versiones y colaboración entre programadores.

HTML: HyperText Markup Language (o Lenguaje de Marcado de Hipertexto en español). Es el lenguaje estándar utilizado para crear páginas web y documentos en la World Wide Web.

JavaScript: Lenguaje de programación versátil que se ejecuta en el navegador web del usuario y permite la creación de páginas web interactivas y dinámicas.

Objeto: Es una estructura de programación en el que se agrupan variables y sus funcionalidades.

Openprocessing: Plataforma en línea que se centra en la creación y el intercambio de proyectos creativos y experimentales basados en el procesamiento creativo y generativo.

Perlin noise: Técnica matemática utilizada en gráficos por computadora y generación de terreno para crear patrones aparentemente aleatorios y naturales.

Píxel: Unidad más pequeña de información visual en una imagen digital o pantalla.

Processing: Es un lenguaje de programación y entorno de desarrollo integrado (IDE) basado en Java, está diseñado para ser una herramienta accesible y amigable para programadores, artistas visuales y educadores, y proporciona funciones y bibliotecas para facilitar la creación de gráficos, animaciones y aplicaciones interactivas.

RGB: Iniciales de tres colores básicos en el modelo de color aditivo: Rojo (Red), Verde (Green) y Azul (Blue). El modelo de color RGB se utiliza para representar y crear una amplia gama de colores utilizando la mezcla de estos tres colores primarios.

Variable: Nombre que se le da a una ubicación en la memoria del computador donde se guardan datos. Este dato puede ser modificado durante la ejecución de un programa.

Vector: Tipo de objeto utilizado para representar y manipular valores numéricos bidimensionales o tridimensionales, generalmente utilizados para describir posiciones, direcciones y movimientos en un espacio 2D o 3D.

VSCode: Visual Studio Code, es un entorno de desarrollo integrado (IDE) gratuito y de código abierto desarrollado por Microsoft.

WEBGL: Web Graphics Library, es una tecnología web que permite a los navegadores web representar gráficos 3D de alto rendimiento en tiempo real en páginas web.

Índice de funciones

Función	Pág.
.add	64
.analyze	93
.angleBetween	72
.cross	70
.currentTime	92
.delta	41
.div	68
.dot	69
.duration	92
.get	120
.height	81
.hide	96
.isPlaying	92
.jump	92
.length	75
.mag	71
.mousePressed	97
.mult	66
.normalize	71
.option	101
.pause	91
.play	91
.pop	77
.position	97
.push	77
.selected	101
.setInput	94
.setVolume	92
.size	98
.splice	77
.stop	92
.sub	65
.value	99
.width	81
%	49
abs	48
acos	49
angleMode	18
arc	17
asin	49
atan	49
atan2	49
background	10
beginShape	25
bezier	18
box	52
ceil	48
circle	16
class	108
colorMode	94
cone	52
constrain	34
constructor	109
cos	49
createButton	97
createCanvas	10
createCapture	96
createCheckbox	101
createRadio	101
createSelect	101
createSlider	97
createVector	63
cylinder	52
DEGREES	18, 23, 50
directionalLight	54
dist	40
DOWN_ARROW	46
draw	9
ellipse	16
else	30
endShape	25
endShape(CLOSE)	25
exp	48
false	38
fill	20
filter	82
floor	48
for	34
get	119
HALF_PI	50
height	10
hour	55
if	29
image	81
imageMode	81
json	149
key	42
keyCode	43
keyIsDown	46
keyIsPressed	42
keyPressed	44
keyReleased	44
LEFT_ARROW	46
let	13
line	16
loadFont	28
loadImage	81
loadPixels	117
loadSound	91
loadStrings	87
log	48
map	32
millis	55
minute	55
mouseDragged	39
mouseIsPressed	38
mousePressed	38
mouseReleased	39
mouseWheel	41
mouseX	20, 37
mouseY	20, 37
nf	83
noFill	20
noise	124
noLoop	124
noStroke	20
p5.AudioIn	94
p5.FFT	93
p5.Vector.add	65
p5.Vector.div	68
p5.Vector.mult	67
p5.Vector.normalize	72
p5.Vector.sub	66
PI	50
pixelDensity	118
pixels	117
pmouseX	38
pmouseY	38
pop	22
pow	49
preload	28, 80
print	14
push	22
QUARTER_PI	50
random	49, 123
rect	17
rectMode	17
return	41, 107
RIGHT_ARROW	46
rotate	23
rotateX	54
rotateY	54
rotateZ	54
round	49
scale	82
second	55
setup	9
sin	49
sphere	52
split	89
sq	49
sqrt	49
stroke	20
strokeWeight	20
tan	49
text	26
textAlign	26
textFont	28

texture 85
this. 109
translate 11, 23
triangle 18
true 38
TWO_PI 50
UP_ARROW 46
updatePixels 118
vertex 25
WEBGL 52
while 51
width 10

Índice de figuras

Figura	Créditos	Página
Figura 0.1.	Captura de pantalla editor p5.js, https://editor.p5js.org	9
Figura 0.2.	Sistema de referencia. Elaboración propia.	11
Figura 1.1.	Línea. Elaboración propia.	16
Figura 1.2.	Círculo. Elaboración propia.	16
Figura 1.3.	Elipse. Elaboración propia.	17
Figura 1.4.	Rectángulo. Elaboración propia.	17
Figura 1.5.	Arco. Elaboración propia.	18
Figura 1.6.	Triángulo. Elaboración propia.	18
Figura 1.7.	Curva Bezier. Elaboración propia.	19
Figura 1.8.	Geometrías. Elaboración propia.	20
Figura 1.9.	Selector de color. Herramienta elaborada por google.	21
Figura 1.10.	Geometrías y color. Elaboración propia.	22
Figura 1.11.	Color en figura. Elaboración propia.	23
Figura 1.12.	Rotación. Elaboración propia.	24
Figura 1.13. a y b	Vértices. Elaboración propia.	25
Figura 1.14.	Texto. Elaboración propia.	27
Figura 1.15.	Tipografías. Elaboración propia.	29
Figura 1.16.	Condiciones. Elaboración propia.	31
Figura 1.17.	Condiciones ‘and’ y ‘or’. Elaboración propia.	32
Figura 1.18.	Ciclo for. Elaboración propia.	35
Figura 1.19.	Ciclo for y vertex. Elaboración propia.	36

Figura 1.20.	Ciclo for doble. Elaboración propia.	37
Figura 1.21.	Deslizador. Elaboración propia.	42
Figura 1.22.	Control con teclado. Elaboración propia.	47
Figura 1.23.	Geometrías 3D. Elaboración propia.	54
Figura 1.24.	Cronómetro. Elaboración propia.	58
Figura 1.25.	Animación. Elaboración propia.	61
Figura 3.1.	Arreglos. Elaboración propia.	79
Figura 3.2.	Archivos en editor de p5.js. Captura de pantalla.	80
Figura 3.3.	Imagen generada con Dall-E 2, OpenAI.com.	81
Figura 3.4.	Imagen generada con Dall-E 2, OpenAI.com.	83
Figura 3.5.	Animación. Elaboración propia.	84
Figura 3.6. a y b	Imagen panorámica, Valle de la Luna, San Pedro de Atacama, año 2019. Elaboración propia.	86
Figura 3.7.	Archivo de texto. Elaboración propia.	89
Figura 3.8.	Espectro de frecuencia. Elaboración propia.	94
Figura 3.9.	Espectro de frecuencia de micrófono. Elaboración propia.	95
Figura 3.10.	Captura desde webcam. Elaboración propia.	96
Figura 3.11.	Botón y deslizador. Elaboración propia.	99
Figura 3.12.	Entradas de texto. Elaboración propia.	100
Figura 4.1.	Objetos rebotando. Elaboración propia.	114
Figura 5.1.	Píxeles de color aleatorio. Elaboración propia.	119
Figura 5.2.	Imagen generada con Dall-E 2, OpenAI.com. Elaboración propia.	121
Figura 6.1. a, b, c y d.	Aleatoriedad de una dimensión. Elaboración propia.	125, 126
Figura 6.2. a y b	Aleatoriedad de dos dimensiones. Elaboración propia.	128
Figura 7.1.	Movimiento parabólico. Elaboración propia.	133

Figura 7.2.	Gráfica de funciones. Elaboración propia.	137
Figura 7.3.	Onda viajera. Elaboración propia.	140
Figura 7.4.	Ejes cartesianos. Elaboración propia.	140
Figura 7.5.	Patrones polares. Elaboración propia.	141
Figura 7.6.	Patrón polar. Elaboración propia.	142
Figura 7.7.	Resistencia eléctrica. Elaboración propia.	146
Figura 7.8.	Campo eléctrico. Elaboración propia.	149
Figura 7.9.	Tabla periódica. Elaboración propia.	154

interactival

Código creativo con p5.js

Aplicaciones interactivas simplificadas

"Código creativo con p5.js - Aplicaciones interactivas simplificadas" es un libro que le permitirá adquirir los conocimientos necesarios para comenzar a realizar sus propias aplicaciones, es un libro de introducción a la programación, pensado para estudiantes de educación secundaria o de educación superior, para docentes del área científica o del área artística, o para artistas que deseen indagar en expresiones de arte digital. Conocerá la forma de declarar variables, como plasmar figuras geométricas en pantalla, como crear condiciones y ciclos, así como añadir interactividad con el teclado y mouse. Al mismo tiempo podrá crear aplicaciones que incluyan imágenes, audio, texto y aplicaciones que necesiten acceso a una cámara de video. Se introducirá en la creación de objetos de programación y en la manipulación de los pixeles que se muestran en pantalla. Finalmente revisará la creación de aplicaciones en las que podrá incorporar aleatoriedad a sus programas, lo que le permitirá crear comportamientos naturales y orgánicos, y con esto, crear arte generativo. Se incluyen, al final de cada capítulo, propuestas de aplicaciones a crear, así como también se incluyen los códigos de algunas aplicaciones terminadas que puede alterar a su conveniencia. Esperamos que pueda aprovechar este libro, donde el límite es su imaginación.

ISBN: 978-956-416-822-7

www.interactival.cl
interactival.cl