



P

Processing:

Un lenguaje
al alcance
de todos



Ignacio Buioli
Jaime Pérez Marín

Processing, un lenguaje al alcance de todos

**Edición 2013
versión 02**

Índice

Índice.....	3
Introducción.....	4
Estructuras: Elementos del Código.....	6
Formas: Coordinadas y Primitivas.....	9
Datos: Variables.....	17
Matemáticas: Funciones Aritméticas.....	20
Control: Decisiones.....	25
Control: Repetición.....	31
Formas: Vértices.....	36
Matemáticas: Curvas.....	43
Color: Color por Números.....	48
Imagen: Visualización y Tinta.....	56
Datos: Texto.....	59
Datos: Conversión y Objetos.....	61
Tipografía: Visualización.....	65
Matemáticas: Trigonometría.....	69
Matemáticas: Aleatoriedad.....	77
Transformaciones: Matrices y Traslaciones.....	81
Transformaciones: Rotación y Escala.....	84
Estructuras: Continuidad.....	89
Estructuras: Funciones.....	95
Formas: Parámetros y Recursión.....	105
Valores de Entrada: Mouse.....	110
Dibujo: Formas Estáticas.....	117
Valores de Entrada: Teclado.....	120
Valores de Entrada: Eventos.....	123
Valores de Entrada: Mouse II.....	128
Valores de Entrada: Tiempo y Fechas.....	134
Movimiento: Líneas y Curvas.....	138
Movimiento: Mecánicos y Orgánicos.....	146
Datos: Arrays.....	151
Imagen: Animación.....	160
Imagen: Píxeles.....	164
Tipografía: Movimiento.....	168
Tipografía: Respuesta.....	172
Color: Componentes.....	175
Imagen: Filtro, Mezcla, Copia, Máscara.....	181
Imagen: Procesamiento de Imagen.....	186
Datos de Salida: Imágenes.....	194
Datos de Salida: Exportar Archivos.....	197
Estructuras: Objetos.....	200
Dibujos: Formas Cinéticas.....	212
Extensión: Modos.....	216
Extensión: Figuras 3D.....	220
Extensión: XML.....	229
Apartados: Documentación.....	231

El presente libro, de tipo manual actualizable, corresponde al aprendizaje del software de Processing 2.0. El hecho de poseer una versión anterior o posterior no entorpece ni impide el aprendizaje con el presente manual. Sin embargo, se recomienda actualizarse a la última versión estable. (ante cualquier duda, consultar en: <http://processing.org/download/>).

Queda entendido que Processing es creado, en principio, por Casey Reas y Ben Fry, y se trata de un software de código abierto que se distribuye bajo una licencia GNU GLP (*General Public License*).

Introducción

Dado el avance de los medios de producción multimedial, nace un potente software dedicado a la producción de imágenes, animaciones e interactivos. El software denominado *Processing*. El proyecto da inicio en el 2001, realizado por Casey Reas y Ben Fry, a partir de terminologías realizadas en el MIT Lab, dirigido por John Maeda, e influenciado por el proyecto *Design by Numbers*. Es así que Processing se convierte en un poderoso entorno de producción basado en Java.

Nos pareció importante la creación de un libro tipo manual de introducción a Processing que se encuentre disponible en español, puesto que la mayoría de los manuales del mercado se encuentran únicamente en inglés y no hay ninguna versión al castellano. No es de poca monta la cantidad de hispanohablantes que existen en el mundo, especialmente interesados en aprender lenguajes de programación orientados a entornos Java. Por tal motivo, este manual comenzó, en principio, como una traducción de los manuales de Processing. No obstante, la distribución de los mismos no nos convencían en cuanto a ductilidad de lectura, por lo tanto decidimos tomarlos de base para crear nuestro propio manual que ayude a aquellos (que por diversos motivos no saben leer en inglés) interesados en la producción multimedial que el software ofrece. En este libro, se ofrece, además, apéndices correspondientes a cada sector y ejemplos que muestran el funcionamiento de las estructuras y de los programas. Se recomienda que, una vez conseguido el programa, los ejemplos sean vistos desde ahí y no solo desde la imagen que lo acompaña.

Como se menciona anteriormente, este libro, principalmente, comienza como una traducción del libro *Processing: A programming Handbook for Visual Designer and Artist* de Casey Reas y Ben Fry. No obstante, se incluyen conceptos y material no presente en dicho libro (así como también se quitan cosas del mismo que no nos han parecido pertinentes).

Software

-¿Qué es Processing?

Es un software de código abierto, y cualquier persona puede contribuir en su mejora. Desprovisto de interfaces innecesarias, se vale de un lenguaje de programación (de igual nombre) basado en Java para realizar composiciones de gran interés.

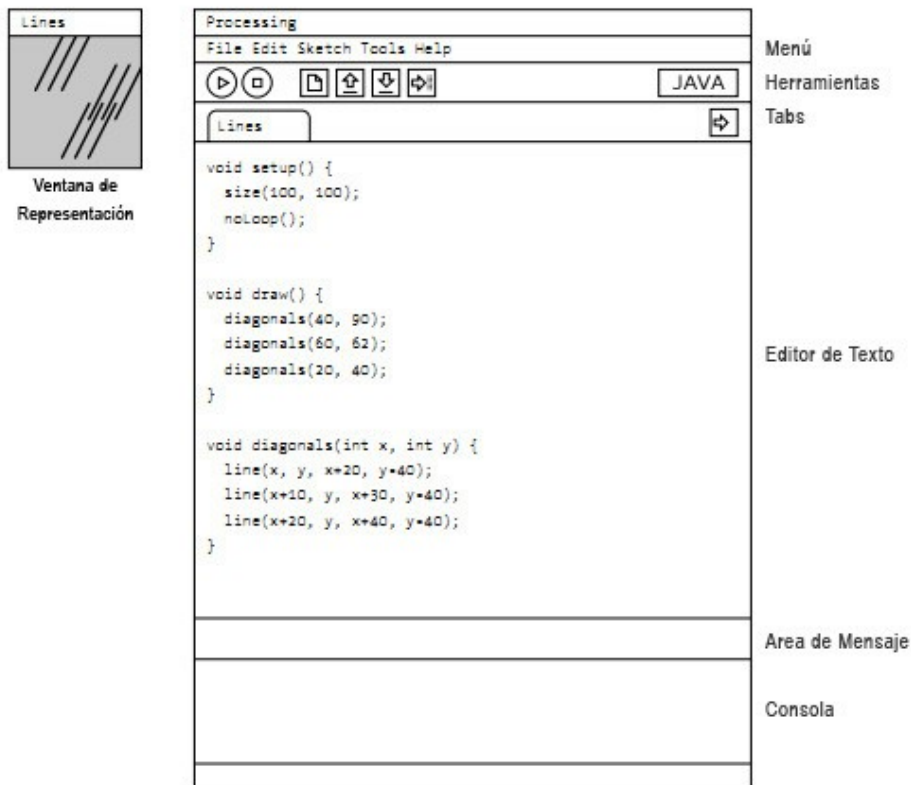
Como software, consiste básicamente en un simple editor de texto para escribir código, un área de mensaje, una consola de texto, un sistema de pestañas para manejar archivos, una barra de herramientas con botones de accionar común y una barra de menú. Cuando el programa se ejecuta, se abre una ventana de representación. Los archivos que se escriben en Processing se denominan *sketch*. Esos *sketch* se escriben en el editor de texto. Admite las funciones de copiar/pegar y buscar/reemplazar texto. La consola muestra errores producidos al ejecutar el programa. También, puede mostrar texto a través de las funciones *print()* y *println()*.

-Descarga

El software de Processing puede descargarse del sitio web homónimo del programa. Desde un navegador web, ingresar a www.processing.org y buscar la sección de descarga (*download*). Admite sistemas operativos

de Linux, Macintosh y Windows. En el sitio se encuentran las instrucciones mismas para la descarga.

-Contexto



Barra de Menú:

- File: */Archivo/*. Comandos para manejar y exportar archivos.
- Edit: */Editar/*. Controles para editar texto. (Copiar, pegar, cortar, encontrar, reemplazar, etc.).
- Sketch: */Sketch/*. Control para ejecutar/frenar el programa y para añadir librerías.
- Tools: */Herramientas/*. Herramientas de asistencia para Processing.
- Help: */Ayuda/*. Referencias a archivos y al lenguaje.

Barra de Herramientas:

- Run: */Ejecutar/*. Compila el código, abre una ventana de representación y muestra el programa.
- Stop: */Parar/*. Termina de correr el programa.
- New: */Nuevo/*. Crea un nuevo Sketch.
- Open: */Abrir/*. Provee de opciones para abrir un sketch del libro de sketch, abrir un ejemplo, o un sketch en cualquier sitio del ordenador.
- Save: */Guardar/*. Guarda el actual sketch en la actual ubicación. Para otra ubicación usar el opción "Save as".
- Export: */Exportar/*. Exporta el actual sketch como un applet de Java unido a una archivo HTML.

Unidad 1

Estructuras: Elementos del Código

Elementos que se introducen en esta Unidad:

```
// (comentario), /* */ (comentario multilinea), ";"(terminador de acción), "," (coma), print(), println()
```

El hecho de crear un programa implica recurrir a la escritura y aprender un lenguaje. Similar a cuando aprendemos un nuevo lenguaje oral o escrito, necesitamos aprender una sintaxis y una lógica. Escribir en un lenguaje humano es muy complicado. Posee ambigüedad en las palabras, y mucha flexibilidad para la construcción de párrafos. En ese sentido, el lenguaje de máquina es más sencillo, pero posee una dificultad clara: la lógica de la programación. Usualmente, un ser humano puede percatarse de un error de sintaxis y darlo por alto, restándole parcial o completa importancia. Sin embargo, en un lenguaje de máquina las cosas son de una forma o de otra: O está bien, o está mal.

-Comentarios

Los comentarios son ignorados por los ordenadores, pero son **muy importantes** para los humanos. Processing permite agregar notas en cualquier sector del código. Pueden ser de solo una línea o de muchas líneas. Ya que los programas usan signos muy arcaicos y de difícil identificación, muchas veces es complicado recordar que hacía cada sector individualmente. Por lo tanto se recurre a la utilidad del comentario, muy útil a la hora de revisar el código, ya sea por el propio programador como por otros. El siguiente programa explica, por sí solo, como se comenta:

```
// Dos barras laterales son usadas para comentar
// Todo el texto en la misma línea es parte de un comentario
// No debe haber espacio entre las barras, por ejemplo "// /", de lo contrario
// el comentario no funcionará.

// Si desea muchas líneas
// lo ideal es usar el método multilinea.

/*
Una barra lateral seguida por un asterisco
permite el comentario de multilinea.
*/
```

-Funciones

Las funciones permiten dibujar formas, colores, realizar cálculos matemáticos, entre otras variadas acciones. Por lo general **se escriben en minúsculas y seguidas por paréntesis**. Algunas funciones aceptan *parámetros*, los cuales se escriben entre los paréntesis. Si acepta más de uno, son separados por una coma (,). A continuación un programa que incluye dos funciones: `size()` y `background()`.

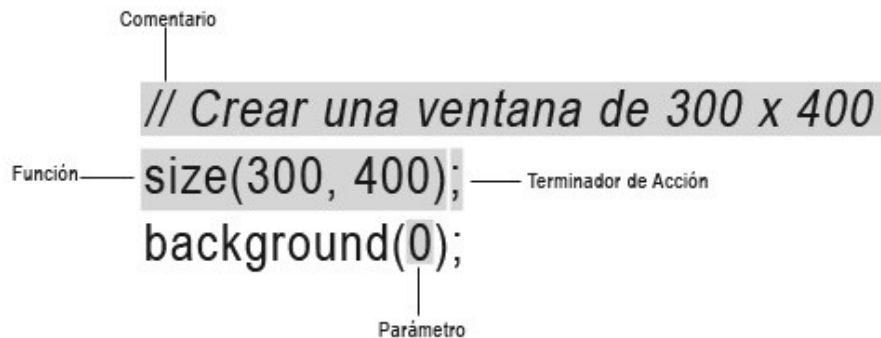
```
//Con esta función establecemos el tamaño de la ventana de presentación
//El primer parámetro corresponde al ancho de la ventana
//El segundo parámetro corresponde al alto.
size(400, 300);

//Con esta función establecemos el color de fondo de la ventana
//Acepta diversos parámetros, para una escala de grises bastará con valores de 0
(negro) a 255 (blanco).
background(0);
```

-Expresiones y Acciones

Si usáramos una analogía, la expresión de un software es como una frase. Las expresiones, por lo general,

van acompañadas de algún operador como +, -, * o /, ya sea a la izquierda o a la derecha del valor. Una expresión en programación puede ser básica, como un solo número, o una compleja cadena de elementos. De esta manera, una expresión siempre tiene un valor determinado.



Expresión	Valor
5	5
10 . 5 * 2	21 . 0
((3+2) * -10)+1	-49

Hay expresiones que también pueden usarse en comparación de un valor con otro. Los operadores de > /mayor a/ y < /menor a/ devuelven solo dos valores: true (verdadero) y false (falso).

Expresión	Valor
6 > 3	true
54 < 50	false

Un conjunto de expresiones pueden formar una *acción*, lo que en programación equivale a una *oración*. Se completa cuando se presenta el terminador de la acción. En Processing, el terminador de acción es el punto-y-coma (;). Al igual que hay diversos tipos de oraciones, hay diversos tipos de acciones. Una acción puede definir una variable, ejecutar una variable, asignar una variable, ejecutar una función, o construir un objeto. A continuación unos ejemplos:

```
size(200, 200); //ejecuta la función size y determina los valores 200 y 200
int x; //declara una nueva variable
x = 102 //asigna un valor a la variable
background(x); //ejecuta la función background
```

Si se eliminara el punto-y-coma, el programa daría un error.

-Sensibilidad

En nuestra lengua, hay casos en los que las palabras comienzan en mayúsculas y casos en los que no. Es a lo que se llama *letra capital*. Por ejemplo, nombres de lugares como Buenos Aires o Andalucía, o nombres propios como Pablo o Enrique, todos ellos comienzan con la letra capital (primer letra en mayúsculas). Hay diversos lenguajes de programación que son permisivos con esto y suelen dejarlo pasar. En el caso de Processing, se produce una **diferenciación** entre mayúsculas y minúsculas, siendo que la correcta forma de escritura es en **minúsculas**. Escribir size() como Size() produciría un error.

```
size(200, 200);
Background(102); //ERROR - La B en "background" está como letra capital.
```

-Espacios en Blanco

Existe una gran variedad de lenguajes de programación que son estrictos en cuanto a los espacios en blanco que se dejan entre cada estructura. Sin embargo, Processing se presta para esto y le **resta** importancia.

Podemos tener el siguiente código:

```
size(200, 200);
background(102);
```

Y escrito de la siguiente manera funcionará exactamente **igual**:

```
size
(200
    ,
    200
    );
background
(
    102
    )
;
```

-Consola

Cuando un programa es ejecutado, la computadora realiza acciones a tal velocidad que es imposible percibir las para el ojo humano. Por lo tanto, es importante mirar la consola, no solo para errores, sino también para entender que ocurre **detrás** del programa.

La consola en Processing, se encuentra como un espacio en negro debajo del editor de texto. Como es muy importante entender que ocurre dentro del programa, existen las funciones `print()` y `println()`. Estas funciones no envían páginas a imprimir, ni muestran nada en la ventana de representación. Simplemente muestran texto en la consola. La consola puede ser usada para mostrar una variable, confirmar un evento o chequear datos externos que están ingresando. Al igual que los comentarios, `print()` y `println()` pueden hacer más clara la lectura del código.

```
//Si se desea imprimir texto, este debe estar entre comillas
println("Processing...");          //Imprime "Processing..." en la consola

//Si se desea imprimir una variable
//no debe ponerse su nombre entre comillas
int x = 20;
println(x);          //Imprime "20" en la consola

//Mientras println() escribe cada cosa en una sola línea, print() escribe todo
en la misma línea
print(20);
println(30);        //Imprime "2030" en la consola
println(80);        //Imprime "80" en una nueva línea de la consola

//También pueden concatenarse múltiples textos con el operador "+" (no confundir
con su uso matemático)
int x = 20;
int y = 80;
println(x + " : " + y);          //Imprime "20 : 80" en la consola
```


Unidad 2

Formas: Coordinadas y Primitivas

Elementos que se introducen en esta Unidad:

`size()`, `point()`, `line()`, `triangle()`, `quad()`, `rect()`, `ellipse()`, `bezier()`, `background()`, `fill()`, `stroke()`, `noFill()`, `noStroke()`, `strokeWeight()`, `strokeCap()`, `strokeJoin()`, `smooth()`, `noSmooth()`, `ellipseMode()`, `rectMode()`

Dibujar una forma con un código puede ser difícil porque todos sus aspectos de su ubicación deben ser especificados con un número. Cuando uno está acostumbrado a dibujar con un lápiz o formas moviéndose en una pantalla con el ratón, puede tomar mucho tiempo empezar a pensar en la relación de la red con una pantalla de coordenadas estrictas. La diferencia fundamental entre ver una composición sobre papel o en su mente y su traducción en notación de código es muy amplia, pero muy sencilla de entender.

-Coordenadas

Antes de hacer un dibujo, es importante que pensemos acerca del tamaño y las características de la superficie sobre la que vamos a dibujar. Si vamos a hacer un dibujo en papel, podemos elegir multitud de utensilios y tipos de papel. Para un esbozo rápido, papel de periódico y carboncillo es lo más apropiado. Para un dibujo más refinado, puede ser preferible papel suave hecho a mano y lápices. Contrariamente, cuando estás dibujando en un ordenador, las opciones principales disponibles son el tamaño de la ventana y el color del fondo.

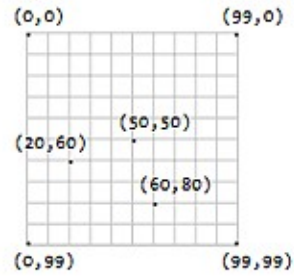
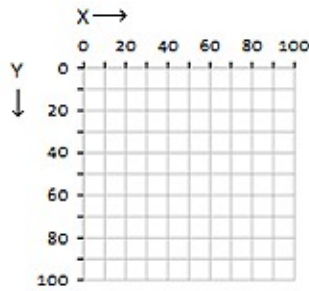
La pantalla de un ordenador es una **rejilla** de pequeños elementos luminosos llamados píxeles. Las pantallas vienen en muchos tamaños y resoluciones. Existen tres tipos diferentes de pantallas para nuestro estudio, y todas ellas tienen un número diferente de píxeles. Los portátiles tienen 1.764.000 píxeles (1680 de ancho por 1050 de alto), las pantallas planas tienen 1.310.720 píxeles (1280 de ancho por 1024 de alto) y los viejos monitores tienen 786.432 píxeles (1024 de ancho por 768 de alto). Millones de píxeles pueden sonar como una cantidad muy vasta, pero producen una pobre calidad visual comparado a un medio físico, como el papel. Las pantallas modernas tienen una resolución de aproximadamente cien puntos por pulgada, mientras que las impresoras modernas proveen más de mil puntos por pulgada. Por otra parte, las imágenes en papel son fijas, mientras que las pantallas tienen la ventaja de ser capaces de cambiar su imagen muchas veces por segundo. Los programas en Processing pueden controlar **todos** o un **subconjunto** de píxeles de la pantalla. Cuando pulsa el botón Run, una ventana de representación se abre y te permite leer y escribir dentro de los píxeles. Es posible crear imágenes más grandes que la pantalla, pero en la mayoría de los casos, haremos una ventana de representación igual o menor a la pantalla. El tamaño de la ventana de representación está controlada por la función `size()`:

```
size(ancho, alto)
```

La función `size()` tiene dos parámetros: el primero establece el ancho de la ventana y el segundo su alto.

```
//Dibuja la ventana de representación de 320 de ancho y 240 de alto (píxeles).  
size(320,240);
```

Una posición de la pantalla está comprendida por un eje de coordenadas x y un eje de coordenadas y. El eje de coordenadas x es la distancia horizontal desde el origen y el eje de coordenadas y es la distancia vertical. En Processing, el origen es la **esquina superior izquierda** de la ventana de representación y coordina los valores hacia abajo y hacia la derecha. La imagen de la izquierda muestra el sistema de coordenadas, y la imagen de la derecha muestra varias posiciones en la rejilla:



Una posición se escribe **con el valor del eje x seguido del valor del eje y**, separados por una coma. La notación para el origen es $(0, 0)$, la coordenada $(50, 50)$ tiene 50 de coordenada x y 50 de coordenada y, y la coordenada $(20, 60)$ tiene 20 de coordenada x y 60 de coordenada y. Si el tamaño de la ventana de representación es de 100 píxeles de ancho y 100 píxeles de alto, el píxel de la esquina superior izquierda es $(0, 0)$, el píxel de la esquina superior derecha es $(99, 0)$, el píxel de la esquina inferior izquierda es $(0, 99)$, y el píxel de la esquina inferior derecha es $(99, 99)$. Esto se ve más claro si usamos la función `point()`.

-Figuras primitivas

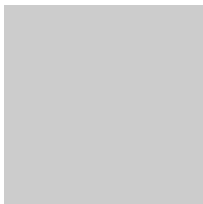
Un punto es el elemento visual más simple y se dibuja con la función `point()`:

`point(x, y)`

Esta función tiene dos parámetros: el primero es la coordenada x y el segundo es la coordenada y. A menos que se especifique otra cosa, un punto es del tamaño de un sólo píxel.



```
point(20, 20);
point(30, 30);
point(40, 40);
point(50, 50);
point(60, 60);
```

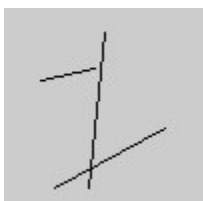


```
point(-500, 100); //Los parámetros negativos no provocan
point(400, -600); //error, pero no se verán en la ventana de
point(140, 2500); //representación.
point(2500, 100);
```

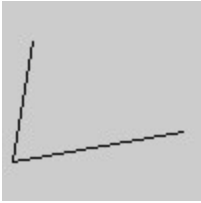
Es posible dibujar cualquier línea mediante una serie de puntos, pero son más simples de dibujar con la función `line()`. Esta función tiene cuatro parámetros, dos por cada extremo:

`line(x1, y1, x2, y2)`

Los primeros dos parámetros establecen la posición donde la línea empieza y los dos últimos establecen la posición donde la línea termina.



```
line(25, 90, 80, 60);
line(50, 12, 42, 90);
line(45, 30, 18, 36);
```



```
line(15, 20, 5, 80);  
line(90, 65, 5, 80);
```

La función `triangle()` dibuja un triángulo. Tiene seis parámetros, dos por cada punto:

```
triangle(x1, y1, x2, y2, x3, y3)
```

El primer par define el primer punto, el segundo par el segundo punto y el último par el tercer punto. Cualquier triángulo puede ser dibujado conectando tres líneas, pero la función `triangle()` hace posible dibujar una figura con relleno. Triángulos de todas formas y tamaños pueden ser creados cambiando los valores de los parámetros.

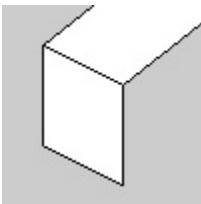


```
triangle(55, 9, 110, 100, 85, 100);  
triangle(55, 9, 85, 100, 75, 100);  
triangle(-1, 46, 16, 34, -7, 100);  
triangle(16, 34, -7, 100, 40, 100);
```

La función `quad()` dibuja un cuadrilátero, un polígono de cuatro lados. Esta función tiene ocho parámetros, dos por cada punto.

```
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```

Variando los valores de los parámetros se puede construir rectángulos, cuadrados, paralelogramos y cuadriláteros irregulares.



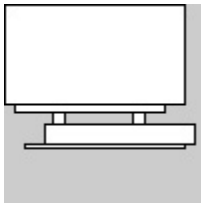
```
quad(20, 20, 20, 70, 60, 90, 60, 40);  
quad(20, 20, 70, -20, 110, 0, 60, 40);
```

Dibujar rectángulos y elipses funcionan de una manera diferente a las figuras vistas anteriormente. En lugar de definir cada punto, los cuatro parámetros establecen la posición y las dimensiones de la figura. La función `rect()` dibuja un rectángulo:

```
rect(x, y, ancho, alto)  
rect(x, y, ancho, alto, radio) //A partir de la versión 2.0  
rect(x, y, ancho, alto, si, sa, id, ii) //A partir de la versión 2.0
```

Los dos primeros parámetros establecen la localización de la esquina superior izquierda, el tercero establece el ancho, y el cuarto el alto. Use el mismo valor de `ancho` y `alto` para dibujar un cuadrado. La versión con el parámetro `radio` permite agregar borde redondeados al rectángulo. El parámetro `radio` corresponde al radio de redondeo. La versión con otros cuatro parámetros permite agregar un radio distinto de redondeo a

cada esquina. Siendo *si* (Superior Izquierdo), *sa* (Superior Derecho), *id* (Inferior Derecho) y *ii* (Inferior Izquierdo).

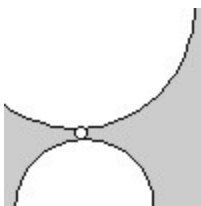


```
rect(0, 0, 90, 50);
rect(5, 50, 75, 4);
rect(24, 54, 6, 6);
rect(64, 54, 6, 6);
rect(20, 60, 75, 10);
rect(10, 70, 80, 2);
```

La función `ellipse()` dibuja una elipse en la ventana de representación:

`ellipse(x, y, ancho, alto)`

Los dos primeros parámetros establecen la localización del centro de la elipse, el tercero establece la anchura, y el cuarto la altura. Use el mismo valor de `ancho` y de `alto` para dibujar un círculo.

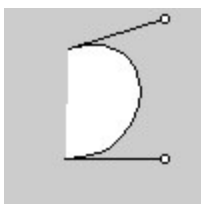


```
ellipse(35, 0, 120, 120);
ellipse(38, 62, 6, 6);
ellipse(40, 100, 70, 70);
```

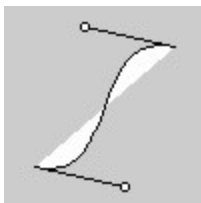
La función `bezier()` puede dibujar líneas que no son rectas. Una curva Bézier está definida por una serie de puntos de control y puntos de anclaje. Una curva es dibujada entre dos puntos de anclaje, y los puntos de control definen su forma:

`bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)`

Esta función requiere ocho parámetros para definir cuatro puntos. La curva se dibuja entre el primer punto y el cuarto, y los puntos de control están definidos por el segundo y tercer punto. En los programas que se utilizan curvas Bézier, tales como Adobe Illustrator o Inkscape, los puntos de control son representados por pequeños nodos que sobresalen de los bordes de la curva.



```
bezier(32, 20, 80, 5, 80, 75, 30, 75);
//Dibujamos los puntos de control.
line(32, 20, 80, 5);
ellipse(80, 5, 4, 4);
line(80, 75, 30, 75);
ellipse(80, 75, 4, 4);
```

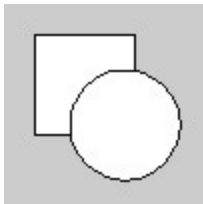


```
bezier(85, 20, 40, 10, 60, 90, 15, 80);
//Dibujamos los puntos de control.
line(85, 20, 40, 10);
ellipse(40, 10, 4, 4);
line(60, 90, 15, 80);
ellipse(60, 90, 4, 4);
```

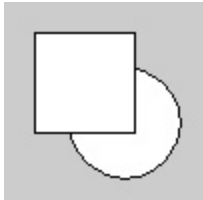
-Orden de dibujo

El orden en que dibujamos las figuras en el código, define qué figuras aparecerán sobre otras en la ventana de representación. Si dibujamos un rectángulo en la primera línea de un programa y una elipse en la segunda línea, el rectángulo aparecerá debajo de la elipse cuando ejecutemos el programa. Revirtiendo el orden, el

rectángulo se coloca arriba.



```
rect(15, 15, 50, 50); //Abajo
ellipse(60, 60, 55, 55); //Arriba
```



```
ellipse(60, 60, 55, 55); //Abajo
rect(15, 15, 50, 50); //Arriba
```

-Valores de grises

Los ejemplos vistos anteriormente han usado el fondo por defecto de color gris claro, líneas negras, y figuras blancas. Para cambiar estos valores, es necesario introducir sintaxis adicional. La función `background()` establece el color de la ventana de representación con un número entre 0 y 255. Este rango puede ser incómodo si no estás familiarizado con programas de dibujo en el ordenador. El valor 255 es blanco y el valor 0 es negro, con un rango de valores de grises en medio. Si no se define un valor para el fondo, se usa el valor por defecto 204 (gris claro).

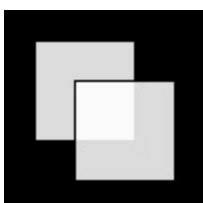
La función `fill()` define el valor del relleno de las figuras, y la función `stroke()` define el valor del contorno de las figuras dibujadas. Si no se define un valor de relleno, se usa el valor por defecto 255 (blanco). Si no se define un valor de contorno, se usa el valor por defecto 0 (negro).



```
rect(10, 10, 50, 50);
fill(204);
stroke(102);
rect(20, 20, 50, 50);
fill(153);
stroke(153);
rect(30, 30, 50, 50);
fill(102);
stroke(204);
rect(40, 40, 50, 50);
```

Cuando se ha definido un valor de relleno o contorno, se aplica a todas las figuras dibujadas después. Para cambiar el valor de relleno o contorno, usamos la función `fill()` o `stroke()` de nuevo.

Un parámetro opcional adicional para `fill()` o `stroke()` regula la transparencia. Definiendo el parámetro a 255 hace que la figura sea totalmente opaca, y a 0 totalmente transparente:

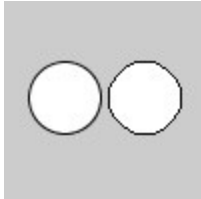


```
background(0);
fill(255, 220);
rect(15, 15, 50, 50);
rect(35, 35, 50, 50);
```

El relleno y el contorno de una figura se puede eliminar. La función `noFill()` detiene a Processing de rellenar figuras, y la función `noStroke()` detiene la creación de líneas y contornos de las figuras. Si usamos `noFill()` y `noStroke()` no dibujaremos nada en la pantalla.

-Atributos de dibujo

Además de cambiar los valores de relleno y contorno de las figuras, también es posible cambiar atributos de la geometría. Las funciones `smooth()` y `noSmooth()` activan y desactivan el suavizado (conocido como filtro *antialiasing*). Cuando usamos una de estas funciones, afectará a todas las funciones dibujadas después. Si usamos primero `smooth()`, usar `noSmooth()` cancelará el ajuste, y viceversa.



```
smooth();
ellipse(30, 48, 36, 36);
noSmooth();
ellipse(70, 48, 36, 36);
```

Los atributos de la línea, están controlados por las funciones `strokeWeight()`, `strokeCap()` y `strokeJoin()`. La función `strokeWeight()` tiene un parámetro numérico que define el grosor de todas las líneas dibujadas después de usar esta función. La función `strokeCap()` requiere un parámetro que puede ser `ROUND`, `SQUARE` o `PROJECT`. `ROUND` redondea los puntos finales, y `SQUARE` los cuadra. `PROJECT` es una mezcla de ambos: redondea las esquinas cuadradas suavemente. Esta función se usa en líneas. La función `strokeJoin()` tiene un parámetro que puede ser `BEVEL`, `MITTER` o `ROUND`. Estos parámetros determinan la forma del contorno de la figura. `BEVEL` corta en diagonal las esquinas cuadradas, `MITTER` cuadra las esquinas (es el valor por defecto) y `ROUND` redondea las esquinas.



```
smooth();
strokeWeight(12);
strokeCap(ROUND);
line(20, 30, 80, 30); //Línea superior
strokeCap(SQUARE);
line(20, 50, 80, 50); //Línea central
strokeCap(PROJECT);
line(20, 70, 80, 70); //Línea inferior
```



```
smooth();
strokeWeight(12);
strokeJoin(BEVEL);
rect(12, 33, 15, 33); //Figura izquierda
strokeJoin(MITTER);
rect(42, 33, 15, 33); //Figura central
strokeJoin(ROUND);
rect(72, 33, 15, 33); //Figura derecha
```

-Modos de dibujo

Por defecto, los parámetros para `ellipse()` definen la coordenada x del centro, la coordenada y del centro, el ancho y el alto. La función `ellipseMode()` cambia la forma en que se usan estos parámetros para dibujar elipses. La función `ellipseMode()` requiere un parámetro que puede ser `CENTER`, `RADIUS`, `CORNER` o `CORNERS`. El modo por defecto es `CENTER`. El modo `RADIUS` también usa el primer y segundo parámetro de `ellipse()` para establecer el centro, pero el tercer parámetro debe ser la mitad del ancho, y el cuarto parámetro debe ser la mitad del alto. El modo `CORNER` hace que `ellipse()` funcione de manera parecida a `rect()`. Causa que el primer y segundo parámetro se trasladen a la esquina superior izquierda del rectángulo que circunscribe la elipse y usa el tercer y cuarto parámetro para definir la anchura y la altura. El modo `CORNERS` tiene un efecto similar a `CORNER`, pero causa que el tercer y cuarto parámetro de `ellipse()` se definan en la esquina inferior derecha del rectángulo.



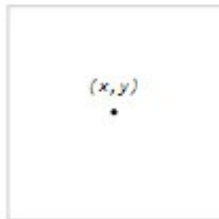
```
smooth();
noStroke();
ellipseMode(RADIUS);
fill(126);
ellipse(33, 33, 60, 60); //Elipse gris
fill(255);
ellipseMode(CORNER);
ellipse(33, 33, 60, 60); //Elipse blanca
fill(0);
ellipseMode(CORNERS);
ellipse(33, 33, 60, 60); //Elipse negra
```

Con un estilo similar, la función `rectMode()` afecta a cómo se dibujan los rectángulos. Requiere un parámetro que puede ser `CORNER`, `CORNERS` o `CENTER`. El modo por defecto es `CORNER`, y `CORNERS` causa que el tercer y cuarto parámetro de `rect()` se definan en la esquina contraria a la primera. El modo `CENTER` causa que el primer y segundo parámetro de `rect()` definan el centro del rectángulo y usa el tercer y cuarto parámetro para el ancho y el alto.

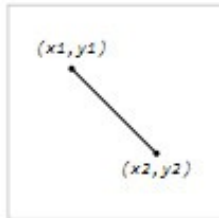


```
noStroke();
rectMode(CORNER);
fill(126);
rect(40, 40, 60, 60); //Rectángulo gris
rectMode(CENTER);
fill(255);
rect(40, 40, 60, 60); //Rectángulo blanco
rectMode(CORNERS);
fill(0);
rect(40, 40, 60, 60); //Rectángulo negro
```

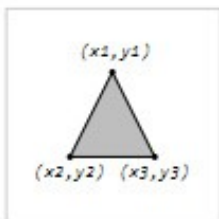
Apéndice de Primitivas



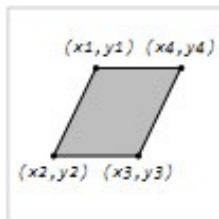
`point(x, y)`



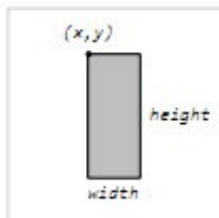
`line(x1, y1, x2, y2)`



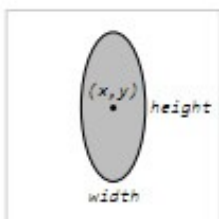
`triangle(x1, y1, x2, y2, x3, y3)`



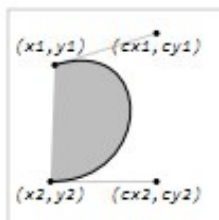
`quad(x1, y1, x2, y2, x3, y3, x4, y4)`



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



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



`bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)`

Unidad 3

Datos: Variables

Elementos que se introducen en esta Unidad:

`int, float, boolean, true, false, = (asignación), width, height`

¿Qué son los datos? Los datos, con frecuencia son características físicas asociadas a un algo. Por ejemplo, cuando decimos que una persona llamada Juan Pérez, en su registro de conducir figura una M (por género masculino) sabemos que ese valor M está asociado a la persona Juan Pérez.

En los sistemas informáticos, los datos son guardados como números y caracteres. Por ejemplo, los ordenadores están constantemente recibiendo datos del mouse y el teclado. Cuando se crea un programa, pueden guardarse datos de, por ejemplo, una forma, un color, o el cambio constante de la posición del mouse.

-Tipos de datos

Processing puede **almacenar y modificar** muchos tipos de datos, como números, letras, palabras, imágenes, colores, fuentes y valores booleanos (`true` y `false`). El hecho de guardar datos implica un mayor o menor uso de la memoria del ordenador donde estemos trabajando. No es lo mismo guardar la palabra "Andalucía" que guardar simplemente la "A". Cada dato es representado como una serie de bits (0 y 1). Por ejemplo, 010000100 puede ser interpretado como una letra.

Como seres humanos, no hará falta aprender el lenguaje binario para programar, Processing se presta para que el trabajo nos sea mucho más sencillo. Sin embargo, 01000001 puede ser interpretado como la letra "A" o como el número 65. Por lo tanto, es importante definir que tipo de dato estamos trabajando.

El primer tipo de datos que enunciaremos, serán los datos numéricos. Existen dos clases de datos numéricos: enteros y decimales (flotantes). Cuando hablamos de enteros, nos referimos a números completos, como 5, -120, 8 y 985. Processing representa variables de tipo entero con `int`. En cambio, los números decimales, también llamados de *punto-flotante*, crean fracciones de los números enteros como 12.8, -120.75, 8.333 y 985.8676543. Processing representa los datos tipo decimales con `float`.

La variable más simple en Processing es de tipo `boolean`. Solo admite dos valores `true` o `false` /*verdadero o falso*/. Suele utilizarse cuando es necesario que el programa tome una decisión ignorando el resto de las posibilidades. A continuación, una tabla con los tipos de variable y el tamaño que ocupa cada una:

<u>Nombre</u>	<u>Tamaño</u>	<u>Rango de Valores</u>
<code>boolean</code>	1 bit	<code>true / false</code>
<code>byte</code>	8 bits	-128 a 127
<code>char</code>	16 bits	0 a 65535
<code>int</code>	32 bits	-2147483648 a 2147483647
<code>float</code>	32 bits	3.40282347E+38 a - 3.40282347E+38
<code>color</code>	32 bits	16777216 colores

-Variables

Una variable es un **contenedor** para guardar datos. Las variables permiten que cada dato sea reutilizado **muchas veces** en un programa. Cada variable tiene dos partes un *nombre* y un *valor*. Si una variable almacena el valor 21 y es llamada *edad*, el nombre *edad* puede aparecer muchas veces en el programa. Cuando el programa se ejecute, la palabra *edad* cambiará por el valor de 21. En adición a este nombre y valor, hay que declarar que tipo de datos soporta esa variable.

Una variable debe ser, siempre, declarada antes de ser usada. Una declaración de variable consta del tipo de datos que aceptará esa variable, seguida de un nombre creado por nosotros. En el siguiente ejemplo se declaran varios tipos de variables y se les asigna un valor:

```
int x;           //Declaración de variable x del tipo entero
```

```
float y;           //Declaración de variable y del tipo flotante
boolean b;        //Declaración de variable b del tipo booleana

x = 50;           //Asignar el valor 50 a la variable x
y = 12.6;         //Asignar el valor 12.6 a la variable y
b = true;         //Asignar el valor true a la variable b
```

Hay una forma mas sintetizada de hacer lo mismo. Podemos, entonces, escribir lo mismo en una sola línea:

```
int x = 50;
float y = 12.6;
boolean b = true;
```

Más de una variable del mismo tipo pueden ser declaradas en la misma línea y luego asignarse un valor por separado a cada una:

```
float x, y, b;
x = -5.56;
y = 12.6;
b = 76.789;
```

Cuando una variable es declarada, es importante ver que clase de dato se le va a asignar para elegir correctamente el tipo de dato. Ni el nombre, ni el tipo de dato puede ser cambiado una vez declarado. Si es posible reasignar otro valor:

```
int x = 69;        //Declara variable x y le asigna el valor de 69
x = 70;           //Cambiar el valor de x por 70
int x = 71;       //ERROR - La variable x ya existe
```

El símbolo de igual (=) se utiliza para asignar valores, únicamente. Le asigna el valor que se encuentra en el lado derecho, a la variable del lado izquierdo. Por lo tanto, es importante que lo que se encuentre del lado izquierdo sea una variable:

```
5 = 25;          //ERROR - Lo que se encuentre del lado izquierdo debe ser una
                //variable
```

Hay que tener en cuenta el tipo de datos que estemos manejando. No es posible asignar un tipo de datos a una variable que solo acepte otra clase. Por ejemplo, no podemos asignar valores decimales a una variable tipo entero:

```
int x = 12.89;    //ERROR - La variable es tipo entero y se le está asignando un
                //valor decimal

float f = 12.89;
int y = f;        //ERROR - La variable es tipo entero y se le está asignando un
                //valor decimal
```

Las variables pueden tener nombres que describan su contenido. Eso simplifica mucho la tarea a la hora de programar. Además, esto podría ayudar a reducir la cantidad de comentarios. Aún así, queda en el programador elegir que clase de nombres utilizar. Por ejemplo, para variables de temperatura, se podrían utilizar los siguientes nombres:

```
t
temp
temperatura
```

```
tempCuarto
temperaturaCuarto
```

La primer letra tiene que ser un carácter en minúsculas, las siguientes pueden ser prácticamente cualquier cosa. Aún así, no se admiten acentos en ninguna parte del código.

-Variables de Programa

Processing, como lenguaje, ha construido variables muy útiles que ya vienen pre-programadas, a fin de facilitarle el trabajo al usuario. Se las denomina *variables del programa* o *variables del sistema*. El ancho y el alto de la ventana de representación están almacenadas como variables denominadas `width` /*ancho*/ y `height` /*alto*/, respectivamente. Si el programa no incluye `size()`, por defecto `width` y `height` adquieren el valor de 100.

```
println(width + " , "+ height);      //Imprime "100, 100" en la consola
size(300, 200);
println(width + " , "+ height);      //Imprime "300, 200" en la consola
```

Frecuentemente son usadas cuando se requiere mantener la escritura del programa en diferentes escalas y que estas sean proporcionales.

```
size(100, 100);
ellipse(width * 0.5, height * 0.5, width * 0.66, height * 0.66);
line(width * 0.5, 0, width * 0.5, height);
```

Unidad 4

Matemáticas: Funciones Aritméticas

Elementos que se introducen en esta Unidad:

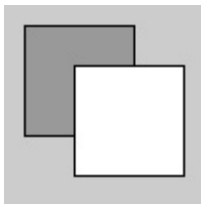
+ (suma), - (resta), * (multiplicación), / (división), % (módulo), () (paréntesis)
++ (incremento), -- (decremento), += (asignar de suma), -= (asignar resta),
*= (asignar multiplicación), /= (asignar división),
- (negativo)
ceil(), floor(), round(), min(), max()

Tener amplios conocimientos en matemáticas puede ser importante a la hora de programar. Sin embargo, **no es necesario** ser un basto conocedor de la misma para disfrutar de la programación. Hay muchas formas y estilos para programar, y cada quien elige individualmente a que aspectos otorgarles un mayor grado de minuciosidad y a cuales descuidarlos un poco.

Es muy común que aquellos que tuvieron una cantidad elevada de matemáticas en la escuela (por ejemplo, en orientaciones en ciencias naturales o en alguna técnica) se sientan atraídos al ver su matemática convertida en movimiento. No obstante, no es necesario ser un verdadero conocedor. Processing es un lenguaje que se presta a la matemática desde un aspecto más tranquilo. Si bien pueden realizarse complejas operaciones, también pueden lograrse resultados similares con una matemática más intuitiva. En esta clase de manuales, se explican funciones aritméticas sencillas para lograr grandes resultados, puesto funciones de mayor complejidad caen fuera de los objetivos de este libro.

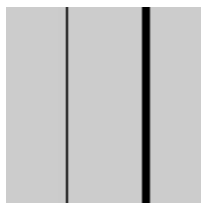
-Aritmética

En programación, las propiedades visuales de una imagen son asignadas por **números**. Esto quiere decir que podemos controlarlas de forma aritmética. El siguiente es un claro ejemplo de eso:



```
int grisVal = 153;
fill(grisVal);
rect(10, 10, 55, 55);           //Dibuja un rectángulo gris
grisVal = grisVal + 102;        //Asigna 255 a grisVal
fill(grisVal);
rect(35, 30, 55, 55);          //Dibuja un rectángulo blanco
```

Utilizando los operadores de suma, resta, multiplicación y división, podemos controlar, por ejemplo, la posición de los elementos. El signo + se utiliza para la suma, el signo - para la resta, el signo * para la multiplicación y el signo / para la división.



```
int a = 30;
line(a, 0, a, height);
a = a + 40;           //Asigna 70 a la variable a
strokeWeight(4);
line(a, 0, a, height);
```



```
int a = 30;
int b = 40;
line(a, 0, a, height);
line(b, 0, b, height);
strokeWeight(4);
line(b - a, 0, b - a, height); //Pueden usarse cálculos
                                //entre variables como
                                //valores
```



```
int a = 8;
int b = 10;
line(a, 0, a, height);
line(b, 0, b, height);
strokeWeight(4);
line(a * b, 0, a * b, height);
```



```
int a = 8;
int b = 10;
line(a, 0, a, height);
line(b, 0, b, height);
strokeWeight(4);
line(a / b, 0, a / b, height);
```



```
int y = 20;
line(0, y, width, y);
y = y + 6; //Asigna 26 a la variable y
line(0, y, width, y);
y = y + 6; //Asigna 32 a la variable y
line(0, y, width, y);
y = y + 6; //Asigna 38 a la variable y
line(0, y, width, y);
```

Los signos +, -, *, / y = posiblemente sean muy familiares, pero el signo % es mucho más exótico. El operador % es el de módulo. Determina cuando un número es dividido por otro, o sea, da como resultado el resto del mismo. El número a la derecha es dividido por el que pongamos a la izquierda. Devolverá como resultado el resto de la operación.

Expresión	Resultado	Explicación
9 % 3	0	El 3 solo entra tres veces en el 9, no sobra nada.
9 % 2	1	El 2 solo entra cuatro veces en el 9, sobra 1.
35 % 4	3	El 4 entra ocho veces en el 35, sobra 3.

Sin muchos problemas, puede explicarse más claramente con un cuento. Cuatro personas están hambrientas y van a comer panes a un restaurante. Si hay solo 35 panes, entonces esas 4 personas solo podrán comer 8 panes cada uno, y sobrarán 3 panes. Eso que sobra es el resultado del módulo. Hay muchos usos para el módulo, como por ejemplo saber cuando un número es par o impar. Sin embargo, en Processing es muy común utilizarlo para mantener a los números en rango. Por ejemplo, si una variable sigue incrementándose (1, 2, 3, 4, 5, 6, 7, 8...), el módulo puede convertirla en una secuencia. Un incremento continuo puede convertirse en un ciclo entre 0 y 3 por aplicar % 4:

x	0	1	2	3	4	5	6	7	8	9	10
x%4	0	1	2	3	0	1	2	3	0	1	2

Muchos ejemplos a lo largo de las explicaciones usarán el módulo de esta forma.

Al trabajar con operaciones matemáticas y variables, es importante tener en claro que tipo de datos estamos manejando. La combinación de dos enteros (int) nos dará como resultado otro entero. La combinación de dos decimales (float) siempre nos dará como resultado un decimal. Pero la combinación de un entero (int) con un decimal (float) siempre nos dará un número decimal, a pesar de la operación diera un resultado exacto. Es importante ser cuidadoso al momento de combinar tipos de datos para evitar errores:

```
int i = 4;
float f = 3.0;
int a = i / f;           //ERROR - Imposible asignar un número decimal a una
                        //variable tipo entero
float b = i / f;       //Asigna 1.3333334 a la variable b del tipo flotante
```

Es también importante prestar atención a algunos conceptos matemáticos, como por ejemplo el concepto de límite, donde un número dividido por cero tiende a *infinito*. En esos casos el programa producirá un error:

```
int a = 0;
int b = 12 / a;         //ERROR - Excepción Matemática: producida por el cero.
```

-Prioridad en el Operador. Agrupación

La prioridad del ordenador determina que operaciones se realizan **antes que otras**. Esto es de gran importancia ya que puede ser determinante para nuestro resultado. Por ejemplo, la expresión $3 + 4 * 5$, nos dará como resultado 23, puesto que tiene prioridad la multiplicación. Primero, $4 * 5$ resultará en 20 y a continuación se le sumará 3, dando por resultado 23. La prioridad puede ser cambiada agregando paréntesis. Si tuviéramos la expresión $(3 + 4) * 5$, entonces la prioridad estaría en la suma, dando como resultado final 35.

```
x = 3 + 4 * 5;         //Asigna 23 a la variable
y = (3 + 4) * 5       //Asigna 35 a la variable
```

La siguiente tabla, a modo explicativo, muestra que elementos tiene prioridad sobre otros. Siendo, pues, que los que se encuentran mas arriba, tiene mas prioridad que los que están debajo:

Multiplicativos	*	/	%
Aditivos	+	-	
Asignación	=		

-Atajos

Cuando programamos, recurrimos mucho a estructuras cíclicas. Muchas veces es necesario utilizar atajos de escritura para reducir el tamaño del código y que sea mas legible. El operador de incremento es ++, mientras que el de decremento es --. Ambos son usados como atajos de suma y resta.

```
int x = 1;
x++;           //Equivale a x = x + 1;
println(x);   //Imprime 2

int y = 1;
y--;          //Equivale a y = y - 1;
println(y);   //Imprime 0
```

Si deseáramos que se actualicé el valor antes de la expresión, solo cambiaremos de lugar el operador de incremento/decremento:

```
int x = 1;
println(++x); //Imprime 2
println(x);   //Imprime 2
```

Al asignar el operador de suma con el operador de asignación (+=) obtendremos el operador de suma

asignada. En el caso de combinar el de resta con el de asignación (-=) obtendremos el operador de resta asignada:

```
int x = 1;
println(x);          //Imprime 1
x += 5;             //Equivale a x = x + 5;
println(x);          //Imprime 6

int y = 1;
println(y);          //Imprime 1
y -= 5;             //Equivale a y = y - 5;
println(y);          //Imprime -4
```

Al asignar el operador de multiplicación con el operador de asignación (*=) obtendremos el operador de multiplicación asignada. En el caso de combinar el de división con el de asignación (/=) obtendremos el operador de división asignada:

```
int x = 4;
println(x);          //Imprime 1
x *= 2;             //Equivale a x = x * 2;
println(x);          //Imprime 8

int x = 4;
println(x);          //Imprime 1
x /= 2;             //Equivale a x = x / 2;
println(x);          //Imprime 2
```

El operador de negación (-) es utilizado cuando deseamos cambiar el signo (positivo/negativo) de algún número:

```
int x = 5;
x = -x;             //Equivale a x = x * -1
println(x);          //Imprime -5
```

-Limitando Números

Las funciones `ceil()`, `floor()`, `round()`, `min()` y `max()` son utilizadas para perfeccionar las operaciones matemáticas. La función `ceil()`, redondea un número decimal convirtiéndolo en un entero. Lo convierte en el valor igual o mayor. Redondea hacia arriba.

```
int x = ceil(2.0);   //Asigna 2 a x
int y = ceil(2.1);   //Asigna 3 a y
int w = ceil(2.5);   //Asigna 3 a w
int z = ceil(2.9);   //Asigna 3 a z
```

La función `floor()`, redondea un número decimal convirtiéndolo en un entero. Lo convierte en el valor igual o menor. Redondea hacia abajo.

```
int x = floor(2.0);  //Asigna 2 a x
int y = floor(2.1);  //Asigna 2 a y
int w = floor(2.5);  //Asigna 2 a w
int z = floor(2.9);  //Asigna 2 a z
```

La función `round()`, redondea un número decimal convirtiéndolo en un entero. Lo convierte en un entero mayor o menor, dependiendo del decimal. Si el decimal es menor a .5, entonces redondea hacia abajo. Si el decimal es mayor o igual a .5, redondea hacia arriba.

```
int x = round(2.0);    //Asigna 2 a x
int y = round(2.1);    //Asigna 2 a y
int w = round(2.5);    //Asigna 3 a w
int z = round(2.9);    //Asigna 3 a z
```

Además, las funciones `ceil()`, `floor()` y `round()` funcionan con variables tipo decimales. A pesar de no ser muy útil, puede utilizarse.

```
float y = round(2.1); //Asigna 2.0 a y
```

La función `min()` determina el número mas chico en una secuencia de números. La función `max()` determina el número más grande en una secuencia de números. Ideal para determinar el número mayo o menor de una secuencia. Ambas funciones pueden admitir dos o tres parámetros.

```
int u = min(5, 9);           //Asigna 5 a u
float t = min(12.6, 7.89)    //Asigna 7.89 a t
int v = min(4, 88, 65);     //Asigna 4 a v

int y = max(5, 9);          //Asigna 9 a y
float x = max(12.6, 7.89)   //Asigna 12.6 a x
int w = max(4, 88, 65);    //Asigna 88 a w
```


Unidad 5

Control: Decisiones

Elementos que se introducen en esta Unidad:

> (mayor a), < (menor a), >= (mayor o igual a), <= (menor o igual a), == (igual), != (distinto)
if, else, { } (corchetes), || (lógica O), && (lógica Y), ! (lógica NO)

Cuando un programa corre, lo hace en el orden en que ubicamos la líneas. Primero la primer línea, luego la segunda, luego la tercera, y así. El programa finaliza cuando lee la última línea. Muchas veces, y para agregar interés, es necesario que esta lectura se quiebre. A ese orden suele llamárselo *flujo*. Existen, entonces, estructuras de control para rompe el flujo del programa.

-Expresiones Relacionales

¿Qué es la verdad? Esta gran cuestión filosófica es muy sencilla de responder en la programación.

Sencillamente, si algo es `true` (verdadero) es una **verdad**, y si algo es `false` (falso) es todo lo **contrario**.

Se trata tan solo de una noción lógica, no es necesario que se trate realmente de una "verdad". Sin embargo, si en la lógica del programa, eso es una verdad, entonces el programa devolverá un valor `true`, de lo contrario devolverá un valor `false`. Tan sencillo como un 1 y un 0.

<u>Expresión</u>	<u>Evaluación</u>
<code>3 < 5</code>	<code>true</code>
<code>3 > 5</code>	<code>false</code>
<code>5 < 3</code>	<code>false</code>
<code>5 > 3</code>	<code>true</code>

Cualquiera de estas expresiones puede leerse en español. ¿El número tres es menor al número cinco? Si la respuesta es "sí" expresa el valor `true` (verdadero). Al compararse dos valores con una expresión relacional, solo pueden dar dos resultados posibles: `true` o `false`. A continuación, una tabla con los valores relacionales y su significado:

<u>Operador</u>	<u>Significado</u>
<code>></code>	Mayor que
<code><</code>	Menor que
<code>>=</code>	Mayor o igual a
<code><=</code>	Menor o igual a
<code>==</code>	Igual a
<code>!=</code>	Distinto de

Las siguientes líneas de código muestran el resultado de comprar dos valores con una expresión relacional:

```
println(3 > 5);           //Imprime false
println(5 > 3);           //Imprime true
println(5 > 5);           //imprime false

println(3 < 5);           //Imprime true
println(5 < 3);           //Imprime false
println(5 < 5);           //imprime false

println(3 >= 5);          //Imprime false
println(5 >= 3);          //Imprime true
println(5 >= 5);          //imprime true

println(3 <= 5);          //Imprime true
println(5 <= 3);          //Imprime false
println(5 <= 5);          //imprime true
```

El operador de igualdad (==) determina si los dos valores que se evalúan son equivalentes. Devuelve true si la igualdad se cumple. En cambio, el operador de diferenciación (!=) evalúa si dos valores no son equivalentes. Devuelve true si la desigualdad se cumple.

```
println(3 == 5);      //Imprime false
println(5 == 3);      //Imprime false
println(5 == 5);      //imprime true

println(3 != 5);      //Imprime true
println(5 != 3);      //Imprime true
println(5 != 5);      //imprime false
```

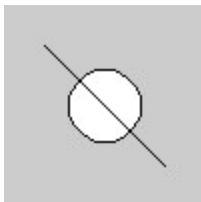
-Estructuras Condicionales

Las estructuras condicionales le permiten a un programa saber que línea de código ejecutar y cuales no. Las líneas de código solo serán "visibles" para el programa si se cumple una condición. Permiten al programa diferenciar acciones dependiendo el valor de variables. Por ejemplo, el programa dibuja una línea o una elipse dependiendo el valor de una variable. Las estructura IF es usada en Processing para tomar esas decisiones:

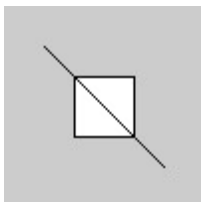
```
if(condición){
acciones;
}
```

La condición debe ser una expresión que se resuelve con true o false. Si la condición es true, el código que este desde la apertura del corchete ({) hasta el cierre del mismo (}), se ejecuta. En caso de ser false, el programa directamente no "lee" las acciones.

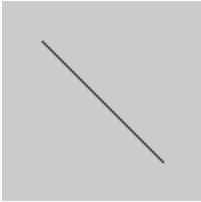
Los siguientes tres ejemplos pretenden mostrar el funcionamiento de la estructura IF. Se recurre al mismo código, solo que con diferentes valores la variable x:



```
//Las condiciones son "x > 100" y "x < 100"
//Como el valor de x es 150
//se ejecutará el primer bloque IF
//y se "eliminará" lo que ocurre en el segundo
int x = 150;
if (x > 100){                          //si es mayor que 100
    ellipse(50, 50, 36, 36);           //dibuja una elipse
}
if(x < 100){                            //si es menor que 100
    rect(35, 35, 30, 30);             //dibuja un rectángulo
}
line(20, 20, 80, 80);
```

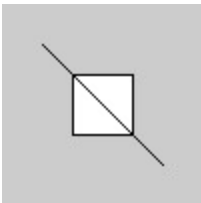


```
//Las condiciones son "x > 100" y "x < 100"
//Como el valor de x es 50
//se ejecutará el segundo bloque IF
//y se "eliminará" lo que ocurre en el primero
int x = 50;
if (x > 100){                          //si es mayor que 100
    ellipse(50, 50, 36, 36);           //dibuja una elipse
}
if(x < 100){                            //si es menor que 100
    rect(35, 35, 30, 30);             //dibuja un rectángulo
}
line(20, 20, 80, 80);
```

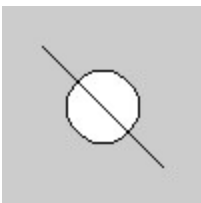


```
//Las condiciones son "x > 100" y "x < 100"  
//Como el valor de x es 100  
//no se ejecutará lo que ocurre en ninguna estructura IF  
int x = 100;  
if (x > 100){ //si es mayor que 100  
    ellipse(50, 50, 36, 36); //dibuja una elipse  
}  
if(x < 100){ //si es menor que 100  
    rect(35, 35, 30, 30); //dibuja un rectángulo  
}  
line(20, 20, 80, 80);
```

En el caso específico de que la condición de como resultado `false`, y estemos deseando que aún así, ocurra algún evento, se utilizará, como agregado a la estructura `IF`, la estructura `ELSE`. La estructura `ELSE` extiende a la estructura `IF`, permitiendo agregar acciones cuando la condición devuelve un resultado `false`.

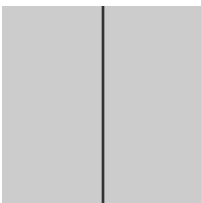


```
int x = 90; //Como x vale 90, dibujará un rectángulo  
if (x > 100){ //si es mayor que 100  
    ellipse(50, 50, 36, 36); //dibuja una elipse  
} else { //sino,  
    rect(35, 35, 30, 30); //dibuja un rectángulo  
}  
line(20, 20, 80, 80); //siempre dibuja una línea
```



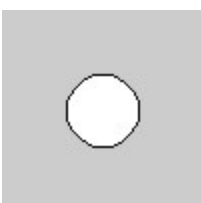
```
int x = 290; //Como x vale 290, dibujará una elipse  
if (x > 100){ //si es mayor que 100  
    ellipse(50, 50, 36, 36); //dibuja una elipse  
} else { //sino,  
    rect(35, 35, 30, 30); //dibuja un rectángulo  
}  
line(20, 20, 80, 80); //siempre dibuja una línea
```

Las condicionales pueden ser "encadenadas" una dentro de otra para tener completo control de las líneas de código. En el siguiente ejemplo se evalúa si el valor es mayor a 100, y luego de eso si es mayor a 300.



```
int x = 420;  
if (x > 100){ //Condición para dibujar elipse o línea  
    if (x < 300){ //Condición, lo determina  
        ellipse(50, 50, 36, 36);  
    } else {  
        line(50, 0, 50, 100);  
    }  
} else {  
    rect(33, 33, 34, 34);  
}
```

Podemos, también, ganar incluso un mayor control en las decisiones al combinar la estructura `IF` con la estructura `ELSE`, consiguiendo la estructura `ELSE IF`.



```
int x = 420;  
if (x < 100){ //Si es menor a 100  
    rect(33, 33, 34, 34); //dibuja un rectángulo  
} else if (x > 300){ //Sino, si es mayor a 300  
    ellipse(50, 50, 36, 36); //dibuja una elipse  
} else { //Sino,  
    line(50, 0, 50, 100); //dibuja una línea  
}
```

-Operadores Lógicos

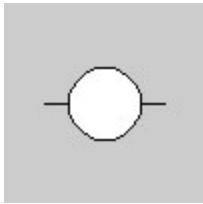
Los operadores lógicos se utilizan al combinar dos o mas expresiones relacionales y para invertir los valores lógicos. Los símbolos de los operadores lógicos corresponden a los conceptos de Y, O y NO.

<u>Operador</u>	<u>Significado</u>
&&	Y
	O
!	NO

La siguiente tabla muestra todas las operaciones posibles y los resultados:

<u>Expresión</u>	<u>Evaluación</u>
true && true	true
true && false	false
false && false	false
true true	true
true false	true
false false	false
!true	false
!false	true

El operador lógico O (| |), hace que el valor sea un true solo si una parte de la expresión es true (verdadera).

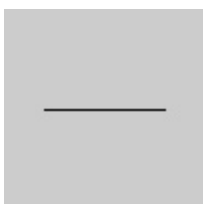


```
int a = 10;
int b = 20;
if((a > 5 ) || (b < 30)){ //Si a es mayor a 5 o b es menos
    //a 30 dibuja una
    line(20, 50, 80, 50); //línea. Como ambas condiciones
    //se cumplen, se
    //dibuja la línea
}
if((a > 15) || (b < 30)){ //Si a es mayor a 15 o b es menor
    //a 30, dibujar una
    ellipse(50, 50, 36, 36); //elipse. Solo una de las
    //condiciones se cumplen, se
    //dibuja la elipse
}
```

Los procesos de lógica se **descomponen en pasos**. Los paréntesis son utilizados para delimitar las componentes y así **simplificar** el trabajo. En las siguientes líneas se muestra un breve paso a paso de como debe interpretarse:

Paso 1	(a > 5) (b < 30)
Paso 2	(10 > 5) (20 < 30)
Paso 3	true true
Paso 4	true

El operador lógico Y (&&), hace que el valor sea true solo si ambas expresiones son true.



```
int a = 10;
int b = 20;
if((a > 5 ) && (b < 30)){ //Si a es mayor a 5 y b es menos
    //a 30 dibujar una
    line(20, 50, 80, 50); //línea. Como ambas condiciones
```

```

} //se cumple, se
//dibuja la línea
if((a > 15) && (b < 30)){ //Si a es mayor a 15 y b es menor
//a 30 dibujar una
    ellipse(50, 50, 36, 36); //elipse. Solo una de las
//condiciones se cumplen,
} //NO se dibuja la elipse

```

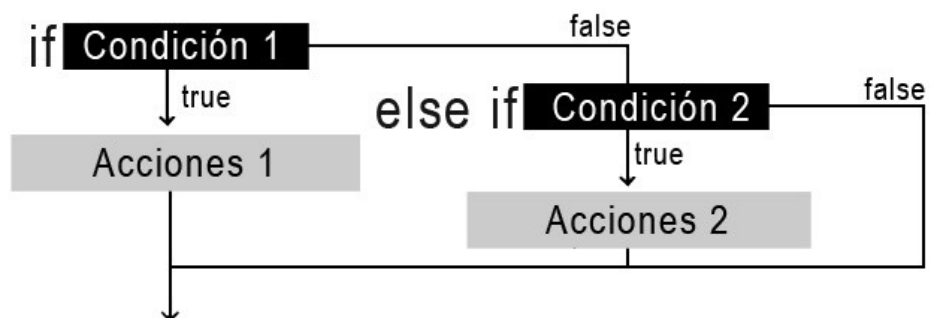
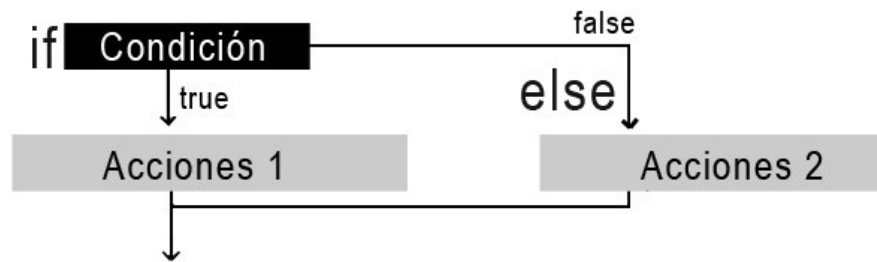
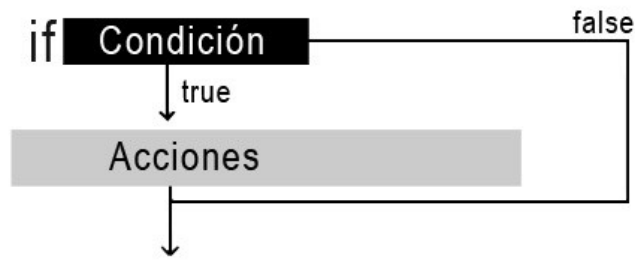
El operador lógico NO (!) es una marca. Simplemente **invierte** el valor lógico. Por ejemplo, si se trata de un valor true, al escribir !true, estaríamos convirtiendo su valor en false. Solo es posible aplicarlo a una variable del tipo boolean.

```

boolean b = true;
println(b); //Imprime true
b = !b;
println(b); //Imprime false
println(!b); //Imprime true
println(5 > 3); //Imprime true
println(!(5 > 3)); //Imprime false
int x = 20;
println(!x); //ERROR - Solo puede aplicarse a variables boolean

```

Apéndice de Estructura IF, ELSE y ELSE-IF



Unidad 6

Control: Repetición

Elementos que se introducen en esta Unidad:

for

La breve historia que poseen los ordenadores nos remonta a su antigua función que era realizar cálculos velozmente. Actualmente, las computadoras emergen como máquinas que pueden realizar ecuaciones matemáticas muy complejas a una increíble velocidad. Los antiguos ordenadores eran distribuidos como máquinas que podía realizar precisas y rápidas ecuaciones repetitivas.

-Iteración

Las estructuras repetitivas son encargadas de producir iteraciones. Es decir, a través de pocas líneas de código se puede **repetir**, prácticamente, infinitas veces una misma línea de código. Esto ayuda a un mejor rendimiento del programa, pero especialmente a un mejor rendimiento del programador, que con simples estructuras puede reducir notablemente el tiempo y, por ende, errores. Se utiliza una estructura denominada estructura FOR, la cual es encargada de repetir un ciclo las veces que queramos. En el siguiente ejemplo se muestra el original código, compuesto de unas 14 líneas, y la versión con el ciclo FOR compuesto tan solo de 4:

Código Original

```
size(200, 200);  
line(20, 20, 20, 180);  
line(30, 20, 30, 180);  
line(40, 20, 40, 180);  
line(50, 20, 50, 180);  
line(60, 20, 60, 180);  
line(70, 20, 70, 180);  
line(80, 20, 80, 180);  
line(90, 20, 90, 180);  
line(100, 20, 100, 180);  
line(110, 20, 110, 180);  
line(120, 20, 120, 180);  
line(130, 20, 130, 180);  
line(140, 20, 140, 180);
```

Código utilizando un FOR

```
size(200, 200);  
for (int i = 20; i < 150; i += 10) {  
    line(i, 20, i, 180);  
}
```

La estructura FOR perfecciona las repeticiones, pudiendo simplificarse su código básico en esta simple y funcional estructura:

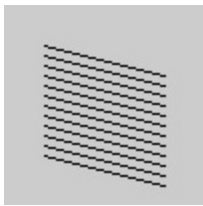
```
for(iniciador, condición, actualización){  
acciones;  
}
```

Los paréntesis asociados a esta estructura corresponden a tres acciones internas: *iniciador*, *condición* y *actualización*. Las acciones dentro del bloque del código se ejecutan constantemente, siempre y cuando la *condición* devuelva un *true* (verdadero). El *iniciador* asigna el valor inicial de la iteración, mientras que la *actualización* modifica el valor del *iniciador* con cada bucle. Los pasos en un FOR son los siguientes:

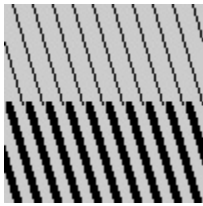
- 1- El *iniciador* comienza a ejecutarse
- 2- La *condición* evalúa si es *true* o *false*
- 3- Si la *condición* es *true*, continúa en el paso 4, si es *false*, pasa al paso 6
- 4- Ejecuta las acciones en el bloque
- 5- Ejecuta la *actualización* y pasa al paso 2

6- Sale de la estructura y continúa ejecutando el resto del programa

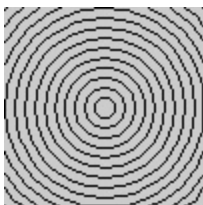
Los siguientes ejemplos muestran diversas formas de uso de la estructura FOR en un programa:



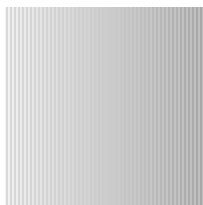
```
// El iniciador es "int i = 20", la condición es "i < 80",  
// y la actualización es "i += 5". Cabe notar que el  
// punto-y-coma termina los dos primeros elementos  
for (int i = 20; i < 80; i += 5) {  
    // Esta línea continuará ejecutándose hasta que "i"  
    // sea mayor a 80.  
    line(20, i, 80, i+15);  
}
```



```
for (int x = -16; x < 100; x += 10) {  
    line(x, 0, x+15, 50);  
}  
strokeWeight(4);  
for (int x = -8; x < 100; x += 10) {  
    line(x, 50, x+15, 100);  
}
```



```
noFill();  
for (int d = 150; d > 0; d -= 10) {  
    ellipse(50, 50, d, d);  
}
```



```
for (int i = 0; i < 100; i += 2) {  
    stroke(255-i);  
    line(i, 0, i, 200);  
}
```

-Iteración Anidada

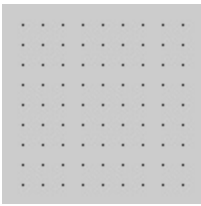
La estructura FOR produce repeticiones en **una** dimensión. Anidando iteraciones podemos crear efectos sumamente interesantes. Por ejemplo, teniendo tan solo dos coordenadas de puntos, si los anidamos en una estructura FOR, podemos cambiar una simple dimensión a una figura de dos dimensiones.



```
for (int y = 10; y < 100; y += 10) {  
    point(10, y);  
}
```

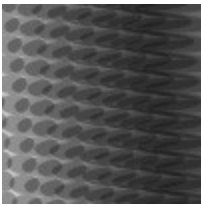


```
for (int x = 10; x < 100; x += 10) {  
    point(x, 10);  
}
```

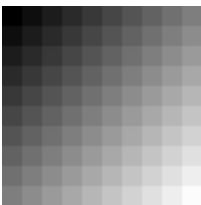



```
for (int y = 10; y < 100; y += 10) {
    for (int x = 10; x < 100; x += 10) {
        point(x, y);
    }
}
```

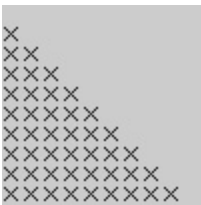
La técnica es muy útil para crear fondos, texturas y los conocidos *patterns*. Los números producidos por las variables de control de repeticiones pueden aplicarse a la posición, al color, al tamaño, a la transparencia o a cualquier otra cosa de atributo visual.



```
fill(0, 76);
noStroke();
smooth();
for (int y = -10; y <= 100; y += 10) {
    for (int x = -10; x <= 100; x += 10) {
        ellipse(x + y/8.0, y + x/8.0, 15 + x/2, 10);
    }
}
```



```
noStroke();
for (int y = 0; y < 100; y += 10) {
    for (int x = 0; x < 100; x += 10) {
        fill((x+y) * 1.4);
        rect(x, y, 10, 10);
    }
}
```



```
for (int y = 1; y < 100; y += 10) {
    for (int x = 1; x < y; x += 10) {
        line(x, y, x+6, y+6);
        line(x+6, y, x, y+6);
    }
}
```

-Formato de las Estructuras

Es importante el espacio entre las estructura para que sea legible y clara la lectura de las mismas. Las líneas de código dentro de las mismas suelen llevar tabulaciones. Cuando el programa se hace largo una clara lectura es ideal en esta clase de estructuras. En este libro utilizamos la siguiente convención de espacios en blanco:

```
int x = 50;
if (x > 100) {
    line(20, 20, 80, 80);
} else {
    line(80, 20, 20, 80);
}
```

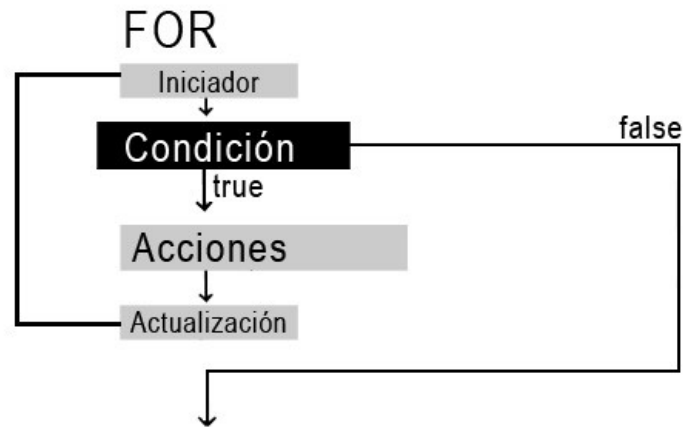
La siguiente es una alternativa de formato usada muy frecuentemente:

```
int x = 50;
if (x > 100) {
```

```
    line(20, 20, 80, 80);  
} else {  
    line(20, 80, 80, 20);  
}
```

Es esencial utilizar un formato limpio para mostrar la jerarquía del código. El entorno de Processing tratará de ajustar su código a medida de que escribe. Aún así, en la sección *Tools* siempre puede utilizarse la herramienta “*Auto Format*” para limpiar el código cuando se lo necesite.

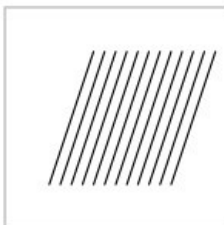
Apéndice de Estructura FOR



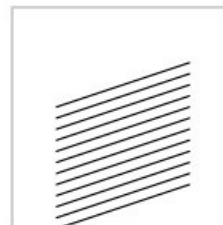
```
for (int x = 20; x <= 80; x += 5) {
    line(x, 20, x, 80);
}
```



```
for (int x = 20; x <= 80; x += 5) {
    line(20, x, 80, x);
}
```



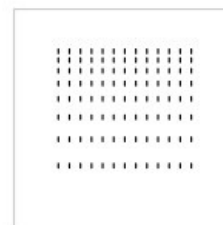
```
for (int x = 20; x < 80; x += 5) {
    line(x+20, 20, x, 80);
}
```



```
for (float x = 80; x > 20; x -= 5) {
    line(20, x+20, 80, x);
}
```



```
for (int y = 20; y <= 80; y += 10) {
    for (int x = 20; x <= y; x += 5) {
        line(x, y, x-3, y-3);
    }
}
```



```
for (float y = 20; y <= 80; y *= 1.2) {
    for (int x = 20; x <= 80; x += 5) {
        line(x, y, x, y-2);
    }
}
```

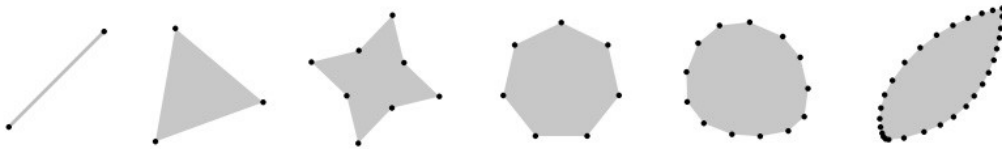
Unidad 7

Formas: Vértices

Elementos que se introducen en esta Unidad:

`beginShape()`, `endShape()`, `vertex()`, `curveVertex()`, `bezierVertex()`

Las figuras geométricas introducidas en la unidad *Formas: Coordenadas y Primitivas*, consiguen un efecto visual altamente potente. No obstante, un programador puede valerse de simples ecuaciones que le permiten crear figuras más complejas. Ya no estamos hablando de figuras primitivas, que solo reciben coordenadas en x y en y, sino de figuras de mayor complejidad construidas a base de vértices.



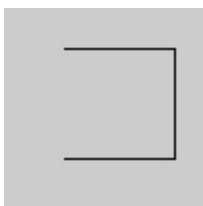
Estas figuras son realmente simples en comparación a las posibilidades que ofrece. En los video-juegos actuales, se utilizan aproximadamente una media de **15000** vértices, lo que convierte a esta herramienta como una función imprescindible en programación.

-Vértice

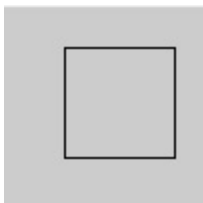
Para crear una figura hecha con vértices, utilizaremos, en principio, la función `beginShape()`, después definiremos cada punto (vértice) con la función `vertex()`, y, finalmente, completaremos la figura con `endShape()`. Las funciones de `beginShape()` y `endShape()` siempre deben usarse en pares. Los paréntesis de la función `vertex()` aceptan dos parámetros, una coordenada de x e y:

```
vertex(x, y)
```

Por defecto, la figura que se forme tendrá un relleno blanco con contorno en negro. Las funciones `fill()`, `stroke()`, `noFill()`, `noStroke()` y `strokeWeight()`, introducidas en la unidad *Formas: Coordenadas y Primitivas*, son útiles para controlar la visualización de la figura. Para cerrar la figura, puede usarse la constante `CLOSE` en la función `endShape()`.

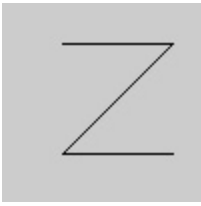


```
noFill();  
beginShape();  
vertex(30, 20);  
vertex(85, 20);  
vertex(85, 75);  
vertex(30, 75);  
endShape();
```



```
noFill();  
beginShape();  
vertex(30, 20);  
vertex(85, 20);  
vertex(85, 75);  
vertex(30, 75);  
endShape(CLOSE);
```

El orden en que se escriben los vértices **cambia** la forma en la que la figura es dibujada. El siguiente ejemplo es exactamente igual que el ejemplo anterior, solo que se han invertido el tercer vértice por el cuarto:



```
noFill();  
beginShape();  
vertex(30, 20);  
vertex(85, 20);  
vertex(30, 75);  
vertex(85, 75);  
endShape();
```

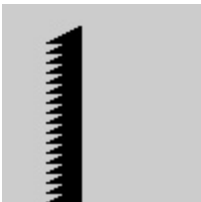
Añadiendo mas vértices y formatos se puede potenciar de manera increíble la potencia visual. También, como se vio antes, se puede utilizar la estructura FOR para mejorar el sistema.



```
fill(0);  
noStroke();  
smooth();  
beginShape();  
vertex(10, 0);  
vertex(100, 30);  
vertex(90, 70);  
vertex(100, 70);  
vertex(10, 90);  
vertex(50, 40);  
endShape();
```



```
noFill();  
smooth();  
strokeWeight(20);  
beginShape();  
vertex(52, 29);  
vertex(74, 35);  
vertex(60, 52);  
vertex(61, 75);  
vertex(40, 69);  
vertex(19, 75);  
endShape();
```



```
noStroke();  
fill(0);  
beginShape();  
vertex(40, 10);  
for (int i = 20; i <= 100; i += 5) {  
    vertex(20, i);  
    vertex(30, i);  
}  
vertex(40, 100);  
endShape();
```

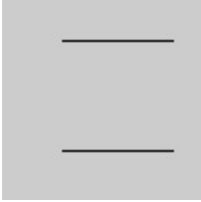
Una figura compleja puede ser compuesta por miles de vértices, pero hay que tener en cuenta que muchos vértices podrían hacer más lento al programa.

-Puntos y Líneas

La función `beginShape()` puede aceptar diversos parámetros que sirven para definir datos en los vértices. Los mismos vértices pueden ser usados para crear figuras sin relleno o líneas continuas. Las constantes `POINTS` y `LINES` son usadas para definir la configuración de puntos y líneas dentro de una figura.



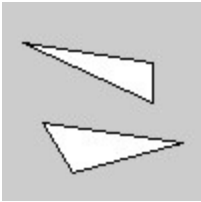
```
//Dibuja un punto en cada vértice
beginShape(POINTS);
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```



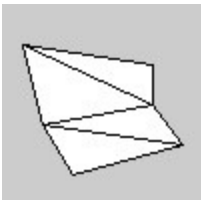
```
//Dibuja una línea ente cada par de vértices
beginShape(LINES);
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

-Formas

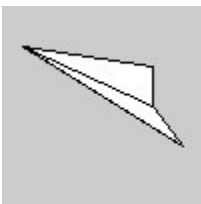
No solo podemos configurar puntos y líneas, sino que también **formas**. Utilizando las constantes TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, y QUAD_STRIP con la función beginShape(). En estos casos necesitaremos ser mucho más cuidadoso en el ámbito espacial de la figura y el orden en que ponemos los vértices.



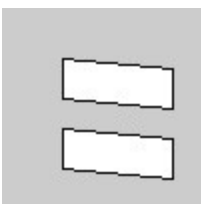
```
//Conecta cada grupo de tres vértices
beginShape(TRIANGLES);
vertex(75, 30);
vertex(10, 20);
vertex(75, 50);
vertex(20, 60);
vertex(90, 70);
vertex(35, 85);
endShape();
```



```
//Comienza con el tercer vértice, conecta todos los
//vértices subsecuentes a los dos primeros
beginShape(TRIANGLE_STRIP);
vertex(75, 30);
vertex(10, 20);
vertex(75, 50);
vertex(20, 60);
vertex(90, 70);
vertex(35, 85);
endShape();
```

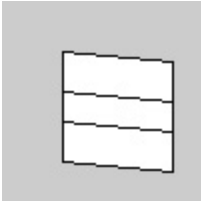


```
beginShape(TRIANGLE_FAN);
vertex(10, 20);
vertex(75, 30);
vertex(75, 50);
vertex(90, 70);
vertex(10, 20);
endShape();
```



```
beginShape(QUADS);
vertex(30, 25);
vertex(85, 30);
vertex(85, 50);
vertex(30, 45);
vertex(30, 60);
vertex(85, 65);
```

```
vertex(85, 85);
vertex(30, 80);
endShape();
```



```
beginShape(QUAD_STRIP);
vertex(30, 25);
vertex(85, 30);
vertex(30, 45);
vertex(85, 50);
vertex(30, 60);
vertex(85, 65);
vertex(30, 80);
vertex(85, 85);
endShape();
```

-Curvas

La función `vertex()` funciona muy bien para crear figuras a base de líneas, pero si queremos crear una curva, necesitamos valernos de las funciones `curveVertex()` y `bezierVertex()`. Estas funciones pueden usarse entre `beginShape()` y `endShape()`, siempre y cuando `beginShape()` no tenga ningún parámetro.

La función `curveVertex()` es usada para conectar dos puntos con una curva. Se le envían dos parámetros:

```
curveVertex(x, y)
```

El primero y el último `curveVertex()`, junto con `beginShape()` y `endShape()`, actúan como puntos de control. La curvatura de cada segmento es calculada por los `curveVertex()` que hay en medio de los puntos de control:



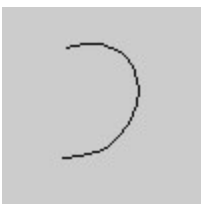
```
smooth();
noFill();
beginShape();
curveVertex(20, 80); //C1
curveVertex(20, 40); //V1
curveVertex(30, 30); //V2
curveVertex(40, 80); //V3
curveVertex(80, 80); //C2
endShape();
```

En cambio, cada `bezierVertex()` define la posición de dos puntos de control y de un punto de anclaje.

```
bezierVertex(cx1, cy1, cx2, cy2, x, y)
```

El primer `bezierVertex()` se ubica junto con el `beginShape()`. Anteriormente al `bezierVertex()`, debemos poner un `vertex()`, que definirá el primer punto ancla. La curva se forma entre punto `vertex()` y las coordenadas `x` e `y` que se le envían a `bezierVertex()`.

Los primeros cuatro parámetros restantes de `bezierVertex()` corresponden al control de la curvatura de la figura.



```
noFill();
beginShape();
vertex(32, 20); //V1
bezierVertex(80, 5, 80, 75, 30, 75); //C1, C2, V2
endShape();
```

Para curvas mas complejas, es necesario agregar una mayor cantidad de `bezierVertex()`, y parámetros de formato:



```
smooth();
noFill();
beginShape();
vertex(15, 30); //V1
bezierVertex(20, -5, 70, 5, 40, 35); //C1, C2, V2
bezierVertex(5, 70, 45, 105, 70, 70); //C3, C4, V3
endShape();
```



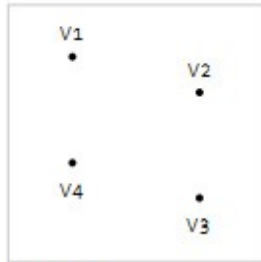
```
smooth();
noStroke();
beginShape();
vertex(90, 39); //V1
bezierVertex(90, 39, 54, 17, 26, 83); //C1, C2, V2
bezierVertex(26, 83, 90, 107, 90, 39); //C3, C4, V3
endShape();
```



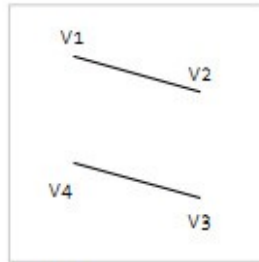
```
smooth();
noFill();
beginShape();
vertex(15, 40); //V1
bezierVertex(5, 0, 80, 0, 50, 55); //C1, C2, V2
vertex(30, 45); //V3
vertex(25, 75); //V4
bezierVertex(50, 70, 75, 90, 80, 70); //C3, C4, V5
endShape();
```

El trabajo con curvas del tipo Bézier suele ser muy complicado, tan solo para dibujar unas pocas curvas. Dependiendo la dificultad de lo que se pretenda dibujar, a veces es preferible optar por la opción que sea más sencilla. Sin embargo, hay contribuciones de usuarios muy útiles que podrían ser de gran ayuda. Entre una de ellas se encuentra la Bézier Tool (que puede descargarse desde <http://processing.org/reference/tools/>) la cual nos da un simple entorno de trabajo visual, donde podemos dibujar curvas Bézier como si fuese un programa de gráficos vectoriales (como Adobe Illustrator o Inkscape). Al finalizar nuestro dibujo, podemos copiarlo y pegarlo en nuestro entorno de trabajo en forma de código. Lo convierte en una herramienta de suma utilidad cuando se pretende trabajar con curvas del tipo Bézier.

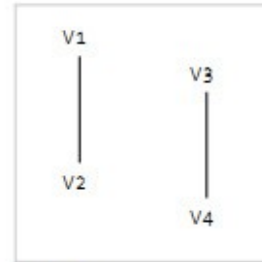
Apéndice de Vértices



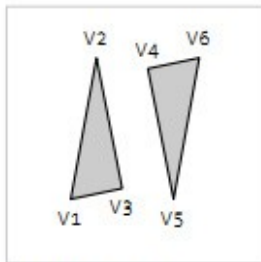
POINTS



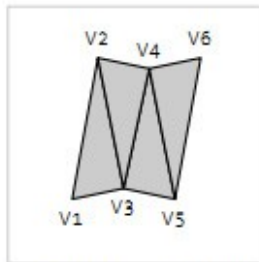
LINES



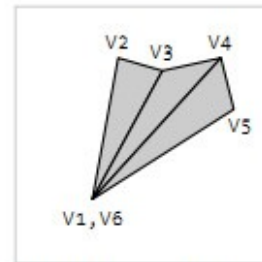
LINES



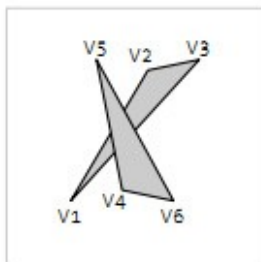
TRIANGLES



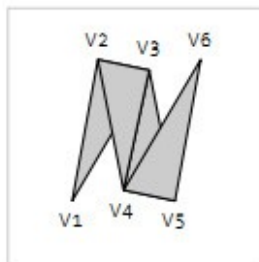
TRIANGLE_STRIP



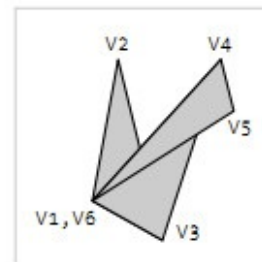
TRIANGLE_FAN



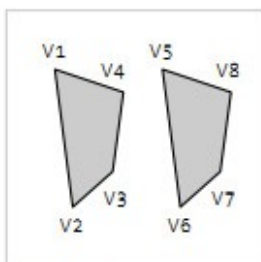
TRIANGLES



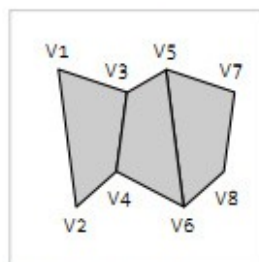
TRIANGLE_STRIP



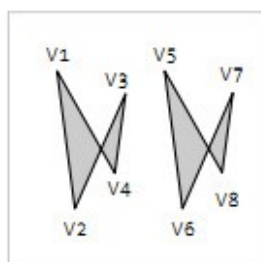
TRIANGLE_FAN



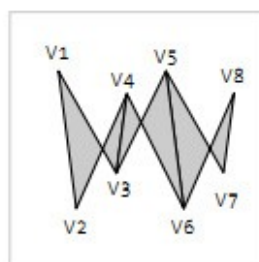
QUADS



QUAD_STRIP

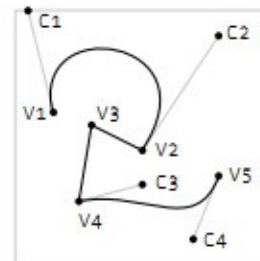
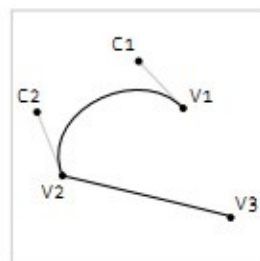
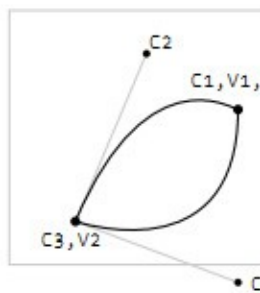
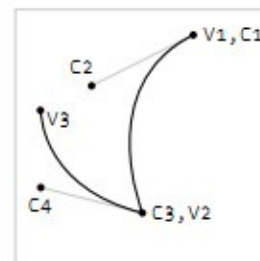
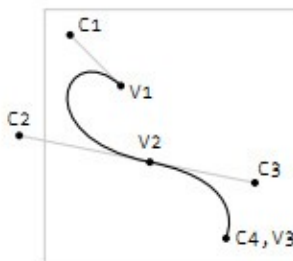
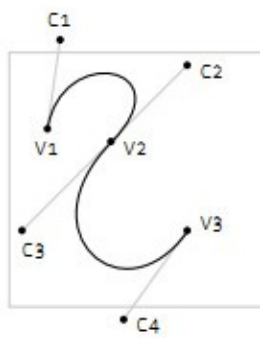
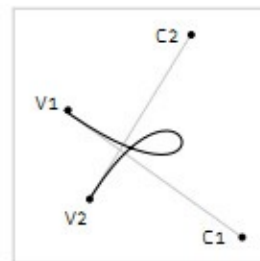
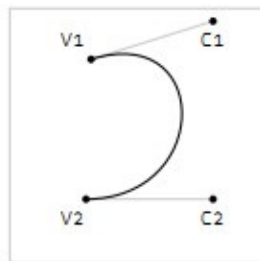
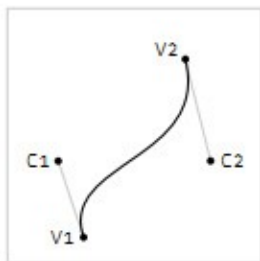
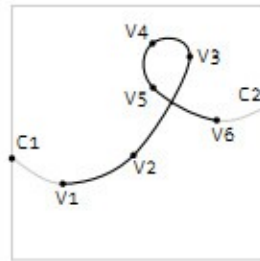
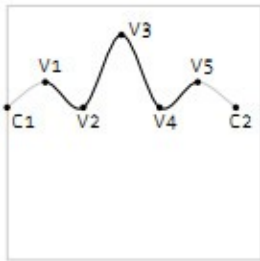
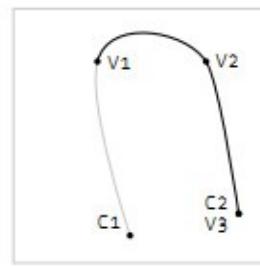
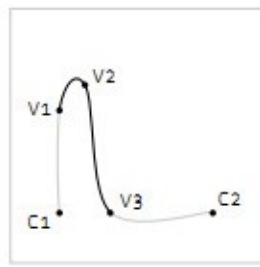
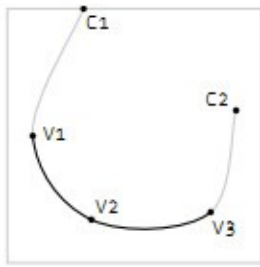


QUADS



QUAD_STRIP

Apéndice de Curvas



Unidad 8

Matemáticas: Curvas

Elementos que se introducen en esta Unidad:

sq(), sqrt(), pow(), norm(), lerp(), map()

Las ecuaciones de matemática básica pueden ser usadas para **dibujar la posición y varios atributos** de una forma. Esas ecuaciones han sido discutidas previamente. Estas ecuaciones se utilizan para **acelerar o desacelerar** las formas en movimiento y su recorrido a base de curvas.

-Exponentes y Raíces

La función `sq()` se utiliza para calcular el cuadrado de un número. O sea, ese mismo número multiplicado por sí mismo. Como bien se sabe, aunque el número sea negativo, devolverá un resultado positivo. Acepta solo un valor como parámetro:

```
sq(valor)
```

El *valor* puede ser cualquier número. Cuando `sq()` es usado, el valor puede ser asignado a una variable:

```
float a = sq(1);           //Asigna 1 a la variable a: Equivale a 1 * 1
float b = sq(-5);         //Asigna 25 a la variable b: Equivale a -5 * -5
float c = sq(9);          //Asigna 81 a la variable c: Equivale a 9 * 9
```

La función `sqrt()` se utiliza para calcular la raíz cuadrada de un número. O sea, el número que multiplicado por sí mismo da determinado valor. Como bien se sabe, aunque el número sea negativo, devolverá un resultado positivo. Acepta solo un valor como parámetro:

```
sqrt(valor)
```

El *valor* puede ser cualquier número. Cuando `sqrt()` es usado, el valor puede ser asignado a una variable:

```
float a = sqrt(6561);     //Asigna 81 a la variable a
float b = sqrt(625);      //Asigna 25 a la variable b
float c = sqrt(1);        //Asigna 1 a la variable c
```

La función `pow()` es usada para calcular un número elevado por el exponente que uno desee. Acepta dos parámetros:

```
pow(número, exponente)
```

El *número* es aquel número a multiplicar. El *exponente* es la cantidad de veces que se va a multiplicar.

```
float a = pow(1, 3);      //Asigna 1.0 a la variable a: Equivale a 1*1*1
float b = pow(3, 4);      //Asigna 81.0 a la variable b: Equivale a 3*3*3*3
float c = pow(3, -2);     //Asigna 0.11 a la variable c: Equivale a 1 / 3*3
float d = pow(-3, 3);     //Asigna -27.0 a la variable d: Equivale a -3*-3*-3
```

Cualquier número (excepto 0) que es elevado a la 0 da como resultado 1. Si se eleva por 1 da como resultado el mismo número:

```
float a = pow(8, 0);     //Asigna 1
float b = pow(3, 1);     //Asigna 3
float c = pow(4, 1);     //Asigna 4
```

-Normalización y Mapeo

Hay veces en que necesitamos convertir una serie de números en un rango de 0.0 a 1.0. Esto es a lo que se denomina *Normalización*. Cuando números entre 0.0 y 1.0 son multiplicados entre sí, nunca dará un resultado menor a 0.0 ni mayor a 1.0. Esto permite multiplicar un número por otro o por sí mismo tantas veces como queramos sin que se salga del rango. Como la normalización suele devolver números decimales, utilizaremos variables del tipo `float` si lo requerimos.

Para normalizar un número, deberemos dividirlo por el máximo valor del rango:

<u>Valor Inicial</u>	<u>Cálculo</u>	<u>Valor Normalizado</u>
0.0	0.0 / 255.0	0.0
102.0	102.0 / 255.0	0.4
255.0	255.0 / 255.0	1.0

Podemos, pues, simplificar la tarea con la función `norm()`. Esta acepta tres parámetros:

```
norm(valor, menor, mayor)
```

El *valor* es el número a normalizar en uno entre 0.0 y 1.0. El *menor* y el *mayor* son el número menor y el mayor que componen el rango a utilizar. Si el resultado está fuera de rango resultará en número menor que 0 o mayor que 1.

```
float x = norm(0.0, 0.0, 255.0); //Asigna 0.0 a x
float y = norm(102.0, 0.0, 255.0); //Asigna 0.4 a y
float z = norm(255.0, 0.0, 255.0); //Asigna 1.0 a z
```

Luego de que un valor es normalizado entre 0.0 y 1.0, ese mismo valor **puede convertirse** en un valor normalizado de otro rango. Por ejemplo, si normalizamos un número su valor se encontrará entre 0.0 y 1.0, al multiplicar ese número por 500.0 obtendremos un valor normalizado entre un rango de 0.0 y 500.0. Para normalizar números que están entre 0.0 y 1.0, a un rango de 200.0 y 250.0, es necesario multiplicar primero por 50 y luego sumarle 200.

<u>Rango Inicial</u>	<u>Rango Deseado</u>	<u>Cálculo</u>
0.0 a 1.0	0.0 a 255.0	$x * 255.0$
0.0 a 1.0	-1.0 a 1.0	$(x * 2.0) - 1.0$
0.0 a 1.0	-20.0 a 60.0	$(x * 80.0) - 20.0$

La función `lerp()` es usada para simplificar esos cálculos. Tiene tres parámetros:

```
lerp(valor1, valor2, amt)
```

El *valor1* y el *valor2* definen el rango al cual se quiere normalizar un valor normalizado entre 0.0 y 1.0. El *amt* es aquel valor normalizado entre 0.0 y 1.0. Es decir, hace que las conversiones vistas en la tabla anterior sean mas rápidas y sencillas para el usuario.

```
float x = lerp(-20.0, 60.0, 0.0); //Asigna -20.0 a x
float y = lerp(-20.0, 60.0, 0.5); //Asigna 20.0 a y
float z = lerp(-20.0, 60.0, 1.0); //Asigna 60.0 a z
```

Por otro lado, la función `map()` permite convertir directamente un número que está en un rango, a otro rango. Esta función utiliza tres parámetros:

```
map(valor, menor1, mayor1, menor2, mayor2)
```

El *valor* es el número a re-mapear. Similar a la función *norm()*, *menor1* y *mayor1* definen el rango en que se encuentra ese valor. El *menor2* y *mayor2* definen el nuevo rango.

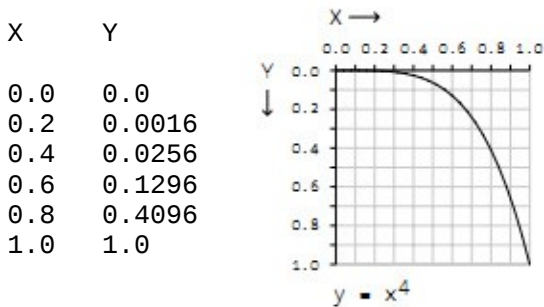
```
float x = map(20.0, 0.0, 255.0, -1.0, 1.0); // Asigna -0.84 a x
float y = map(0.0, 0.0, 255.0, -1.0, 1.0); // Asigna -1.0 a y
float z = map(255.0, 0.0, 255.0, -1.0, 1.0); // Asigna 1.0 a z
```

-Curvas Simples

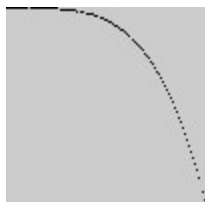
Las funciones exponenciales son útiles para crear curvas muy simples y rápidamente. Al normalizar valores de la función *pow()*, podemos obtener rápidamente curvas ascendentes y descendentes, con números que no exceden el valor de 1. La ecuación tendría la siguiente forma:

$$y = x^n$$

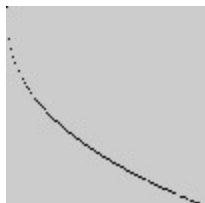
Donde el valor de *x* sería un valor normalizado entre 0.0 y 1.0, y el valor de *n* un número entero. El siguiente es un simple ejemplo de como se ubican estos valores en una grilla de ejes cartesianos:



A continuación se muestran una serie de ejemplos de la capacidad de las ecuaciones exponenciales, con normalización y utilizando ciclos FOR.



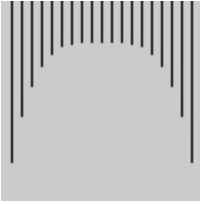
```
for (int x = 0; x < 100; x++) {
    float n = norm(x, 0.0, 100.0); //Rango de 0.0 a 1.0
    float y = pow(n, 4);           //Calcula la Curva
    y *= 100;                       //Rango de 0.0 a 100.0
    point(x, y);
}
```



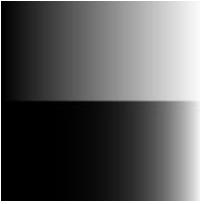
```
for (int x = 0; x < 100; x++) {
    float n = norm(x, 0.0, 100.0); //Rango de 0.0 a 1.0
    float y = pow(n, 0.4);         //Calcula la Curva
    y *= 100;                       //Rango de 0.0 a 100.0
    point(x, y);
}
```



```
//Dibuja círculos de puntos alrededor de la curva y = x^4
noFill();
smooth();
for (int x = 0; x < 100; x += 5) {
    float n = norm(x, 0.0, 100.0); //Rango de 0.0 a 1.0
    float y = pow(n, 4);           //Calcula la Curva
    y *= 100;                       //Escala y rango de 0.0 a 100.0
    strokeWeight(n * 5);           //Incrementa
    ellipse(x, y, 120, 120);
}
```

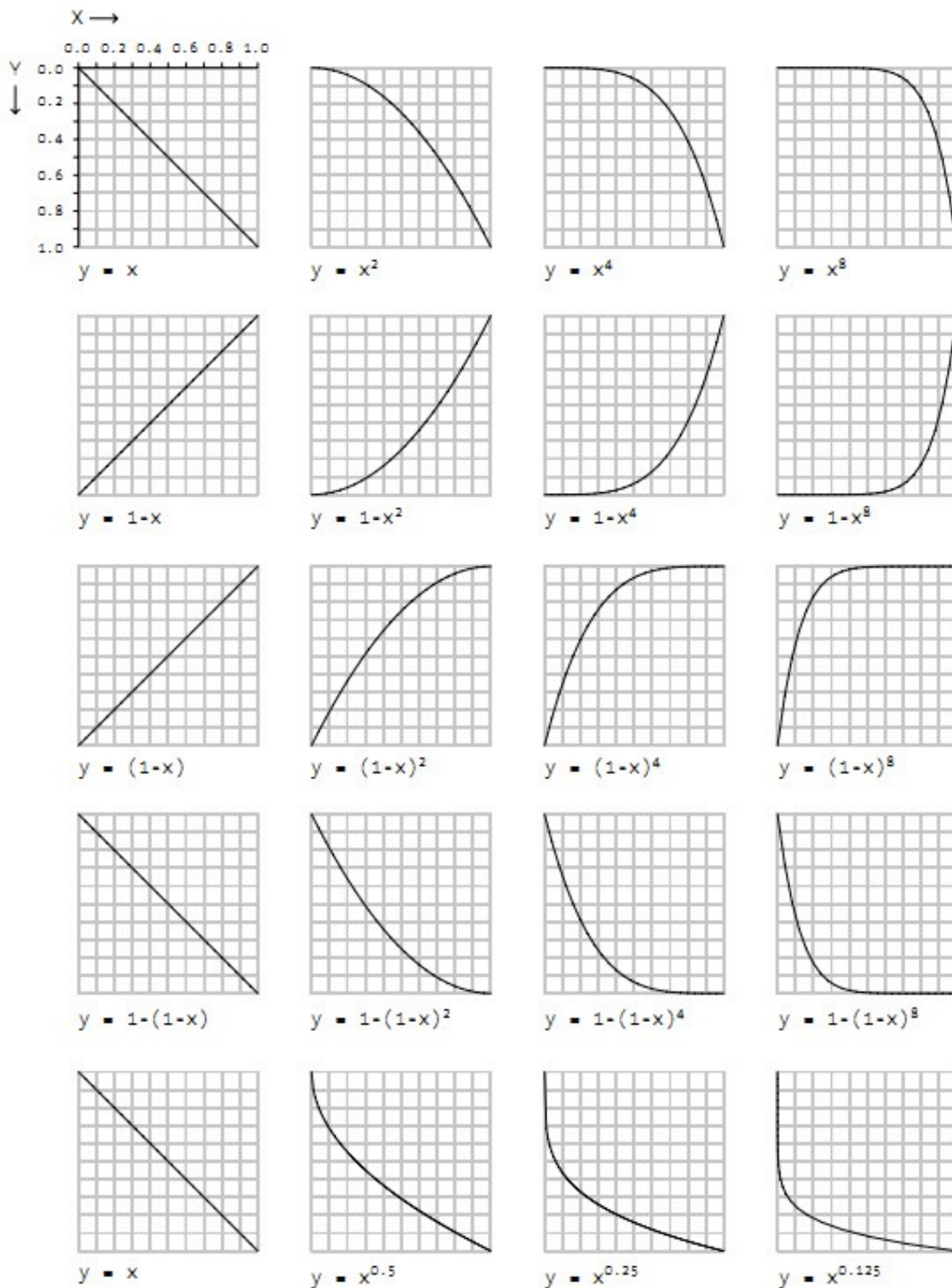


```
//Dibuja una línea de arriba a los puntos de
//la curva  $y = x^4$  de  $x$  en un rango de  $-1.0$  a  $1.0$ 
for (int x = 5; x < 100; x += 5) {
    float n = map(x, 5, 95, -1, 1);
    float p = pow(n, 4);
    float ypos = lerp(20, 80, p);
    line(x, 0, x, ypos);
}
```



```
//Crea un gradiente de  $y = x$  y de  $y = x^4$ 
for (int x = 0; x < 100; x++) {
    float n = norm(x, 0.0, 100.0);
    float val = n * 255.0;
    stroke(val);
    line(x, 0, x, 50);
    float valSquare = pow(n, 4) * 255.0;
    stroke(valSquare);
    line(x, 50, x, 100);
}
```

Apéndice de Curvas Matemáticas



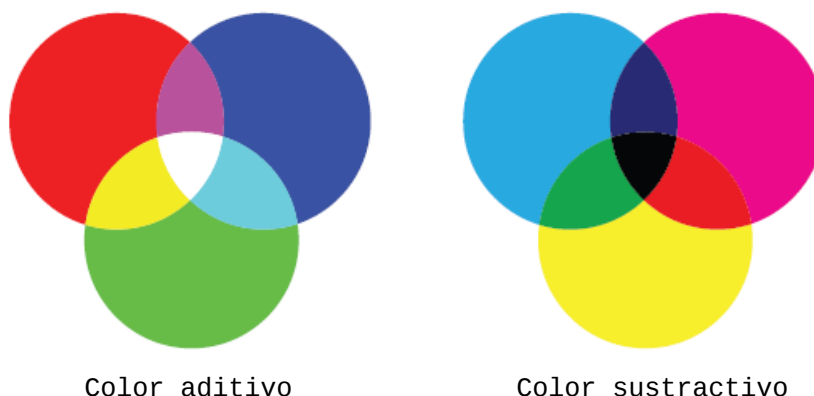
Unidad 9

Color: Color por Números

Elementos que se introducen en esta Unidad:

color, color(), colorMode()

Trabajar con colores a través de una pantalla es diferente a hacerlo en papel o lienzo. Mientras que se apliquen las mismas reglas, el conocimiento de los pigmentos para pintar (rojo cadmio, azul prusiano, ocre tostado) y para imprimir (cian, amarillo, magenta) no necesitan ser traducidos para crear colores digitales. Por ejemplo, unir todos los colores en un monitor de ordenador tiene como resultado el color blanco, mientras que unir todos los colores con pintura tiene como resultado el color negro (o un marrón extraño). Un monitor de ordenador mezcla los colores con luz. La pantalla es una superficie negra y se añade luz coloreada. Esto se conoce como colores aditivos, en contraste con el modelo de color sustractivo para tinta en papel y lienzo. Esta imagen expone la diferencia entre los dos modelos:



La forma más común de especificar un color en el ordenador es mediante valores RGB. Un valor RGB ajusta la cantidad de luz roja, verde y azul en un píxel de la pantalla. Si miras de cerca la pantalla de tu televisor o el monitor de tu ordenador, verás que cada píxel se compone de tres elementos independientes de luz roja, verde y azul; pero debido a que nuestros ojos sólo pueden ver una cantidad limitada de detalle, los tres colores se mezclan para formar uno solo. La intensidad de cada elemento de color está especificada con valores entre 0 y 255, donde 0 es el mínimo y 255 el máximo.

-Ajuste de color

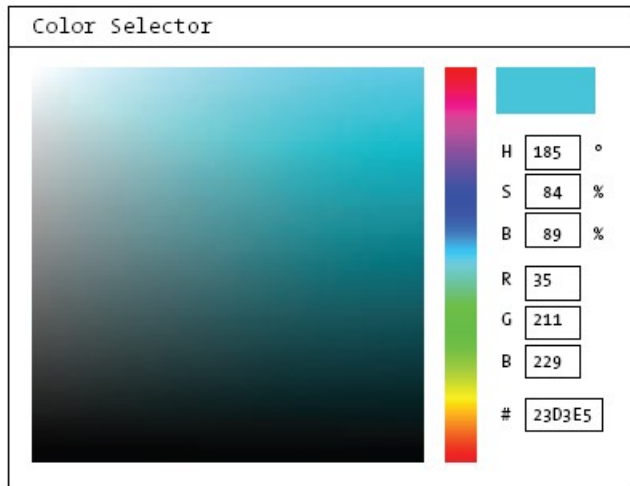
En Processing, los colores están definidos por los parámetros de las funciones `background()`, `fill()` y `stroke()`:

```
background(valor1, valor2, valor3)
fill(valor1, valor2, valor3)
fill(valor1, value2, valor3, alfa)
stroke(valor1, valor2, valor3)
stroke(valor1, valor2, valor3, alfa)
```

Por defecto, el parámetro *value1* define el componente de color rojo, *valor2* el componente verde y *value3* el azul. El parámetro opcional *alfa* para `fill()` o `stroke()` define la opacidad. El valor 255 del parámetro *alfa* indica que el color es totalmente opaco, y el valor 0 indica que es totalmente transparente (no será visible).



```
background(174, 221, 60);
```

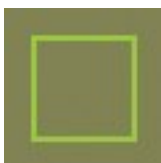



Selector de color

Podemos seleccionar el color con el cursor simplemente introduciendo los valores que deseamos.



```
background(129, 130, 87);
noStroke();
fill(174, 221, 60);
rect(17, 17, 66, 66);
```



```
background(129, 130, 87);
noFill();
strokeWeight(4);
stroke(174, 221, 60);
rect(19, 19, 62, 62);
```



```
background(116, 193, 206);
noStroke();
fill(129, 130, 87, 102); //Menos opacidad
rect(20, 20, 30, 60);
fill(129, 130, 87, 204); //Más opacidad
rect(50, 20, 30, 60);
```



```
background(116, 193, 206);
int x = 0;
noStroke();
for (int i = 51; i <= 255; i += 51) {
  fill(129, 130, 87, i);
  rect(x, 20, 20, 60);
  x += 20;
}
```



```
background(56, 90, 94);
smooth();
strokeWeight(12);
stroke(242, 204, 47, 102); //Menos opacidad
line(30, 20, 50, 80);
stroke(242, 204, 47, 204); //Más opacidad
line(50, 20, 70, 80);
```



```
background(56, 90, 94);
smooth();
int x = 0;
strokeWeight(12);
```

```

for (int i = 51; i <= 255; i += 51) {
    stroke(242, 204, 47, i);
    line(x, 20, x+20, 80);
    x += 20;
}

```

La opacidad se puede usar para crear nuevos colores mediante superposición de formas. Los colores originados dependen del orden en que se dibujen las formas.



```

background(0);
noStroke();
smooth();
fill(242, 204, 47, 160); //Amarillo
ellipse(47, 36, 64, 64);
fill(174, 221, 60, 160); //Verde
ellipse(90, 47, 64, 64);
fill(116, 193, 206, 160); //Azul
ellipse(57, 79, 64, 64);

```



```

background(255);
noStroke();
smooth();
fill(242, 204, 47, 160); //Amarillo
ellipse(47, 36, 64, 64);
fill(174, 221, 60, 160); //Verde
ellipse(90, 47, 64, 64);
fill(116, 193, 206, 160); //Azul
ellipse(57, 79, 64, 64);

```

-Dato de color

El tipo dato de color se usa para almacenar colores en un programa, y la función `color()` se usa para asignar una variable de color. La función `color()` puede crear valores de gris, valores de gris con opacidad, valores de color y valores de color con opacidad. Las variables del tipo dato de color puede almacenar todas estas configuraciones:

```

color(gris)
color(gris, alfa)
color(valor1, valor2, valor3)
color(valor1, valor2, valor3, alfa)

```

Los parámetros de la función `color()` define un color. El parámetro *gris* usado sólo o con *alfa*, define un rango de tonos de blanco a negro. El parámetro *alfa* define la opacidad con valores de entre 0 (transparente) a 255 (opaco). Los parámetros *valor1*, *valor2* y *valor3* definen valores para los distintos componentes. Las variables del tipo de dato de color se definen y se asignan de la misma forma que los tipos de dato `int` y `float` discutidos en la unidad Datos: Variables.

```

color c1 = color(51); //Crea gris.
color c2 = color(51, 204); //Crea gris con opacidad.
color c3 = color(51, 102, 153); //Crea azul.
color c4 = color(51, 102, 153, 51); //Crea azul con opacidad.

```

Después de que una variable *color* se haya definido, se puede usar como parámetro para las funciones `background()`, `fill()` y `stroke()`.



```
color ruby = color(211, 24, 24, 160);
color pink = color(237, 159, 176);
background(pink);
noStroke();
fill(ruby);
rect(35, 0, 20, 100);
```

RGB, HSB

Processing usa el formato de color RGB como predeterminado para trabajar, pero se puede usar en su lugar la especificación HSB para definir colores en función del matiz, saturación y brillo. El matiz de un color es lo que normalmente piensa la gente del nombre del color: amarillo, rojo, azul, naranja, verde, violeta. Un matiz puro es un color sin diluir muy intenso. La saturación es el grado de pureza en un color. Va desde el matiz sin diluir hasta el matiz completamente diluido, donde el color no tiene brillo. El brillo de un color es su relación de luz y oscuridad. La función `colorMode()` establece el formato de color para un programa:

```
colorMode(modos)
colorMode(modos, rango)
colorMode(modos, rango1, rango2, rango3)
```

Los parámetros de `colorMode()` cambian la forma en que Processing interpreta el dato de color. El parámetro *modo* puede ser tanto RGB como HSB. El rango de parámetros permite a Processing usar valores diferentes al predeterminado (0 a 255). En los gráficos de ordenador se usa frecuentemente un rango de valores entre 0.0 y 1.0. Tanto si un parámetro de un sólo rango establece el rango para todos los colores, como si los parámetros *rango1*, *rango2* y *rango3* establecen el rango de cada uno – tanto rojo, verde, azul como matiz, saturación, brillo, dependiendo del valor del parámetro *modo*.

```
//Establece el rango para los valores rojo, verde y azul de 0.0 a 1.0
colorMode(RGB, 1.0);
```

Un ajuste útil para el formato HSB es establecer los parámetros de *rango1*, *rango2* y *rango3* respectivamente a 360, 100 y 100. Los valores de matiz desde 0 a 360 son los grados alrededor de la rueda de color, y los valores de saturación y brillo desde 0 a 100 son porcentajes. Este ajuste coincide con los valores usados en muchos selectores de color y, por tanto, hace más sencilla la transición de dato de color entre otros programas y Processing:

```
//Establece el rango para el matiz con valores desde 0 a 360 y la
//saturación y el brillo con valores entre 0 y 100
colorMode(HSB, 360, 100, 100);
```

Los siguientes ejemplos revelan la diferencia entre matiz, saturación y brillo.



```
//Cambia el matiz; saturación y brillo constante
colorMode(HSB);
for (int i = 0; i < 100; i++) {
    stroke(i*2.5, 255, 255);
    line(i, 0, i, 100);
}
```



```
//Cambia la saturación; matiz y brillo constante
colorMode(HSB);
for (int i = 0; i < 100; i++) {
    stroke(132, i*2.5, 204);
    line(i, 0, i, 100);
}
```



```
//Cambia el brillo; matiz y saturación constante
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  stroke(132, 108, i*2.5);
  line(i, 0, i, 100);
}
```



```
//Cambia la saturación y el brillo; matiz constante
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  for (int j = 0; j < 100; j++) {
    stroke(132, j*2.5, i*2.5);
    point(i, j);
  }
}
```

Es fácil hacer transiciones suaves entre colores cambiando los valores para `color()`, `fill()` y `stroke()`. El formato HSB tiene una enorme ventaja sobre el formato RGB cuando trabajamos con códigos, porque es más intuitivo. Cambiando los valores de los componentes rojo, verde y azul, a menudo obtenemos resultados inesperados, mientras que estimando los resultados de los cambios del matiz, saturación y brillo seguimos un camino más lógico. Los siguientes ejemplos muestran la transición del verde al azul. El primer ejemplo hace la transición usando el formato RGB. Requiere calcular los tres valores de color, y la saturación del color cambia inesperadamente en el centro. El segundo ejemplo hace la transición usando el formato HSB. Sólo es necesario alterar un número, y el matiz cambia suavemente e independientemente de las otras propiedades del color.



```
//Cambio del azul al verde en formato RGB
colorMode(RGB);
for (int i = 0; i < 100; i++) {
  float r = 61 + (i*0.92);
  float g = 156 + (i*0.48);
  float b = 204 - (i*1.43);
  stroke(r, g, b);
  line(i, 0, i, 100);
}
```



```
// Cambio del azul al verde en formato HSB
colorMode(HSB, 360, 100, 100);
for (int i = 0; i < 100; i++) {
  float newHue = 200 - (i*1.2);
  stroke(newHue, 70, 80);
  line(i, 0, i, 100);
}
```

-Hexadecimal

La notación hexadecimal (hex) es una notación alternativa para definir el color. Este método es popular entre diseñadores que trabajan en la Web porque estándares como HyperText Markup Language (HTML) y Cascading Style Sheets (CSS) usan esta notación. La notación Hex para el color codifica cada número desde el 0 al 255 en un valor de dos dígitos usando los números de 0 a 9 y las letras de A a F. De esta forma, tres valores RGB desde 0 a 255 se pueden escribir como un solo valor hex de seis dígitos. Unas pocas conversiones de ejemplo demuestran esta notación:

RGB
255, 255, 255
0, 0, 0

Hex
#FFFFFF
#000000

102, 153, 204	#6699CC
195, 244, 59	#C3F43B
116, 206, 206	#74CECE

La conversión de los valores RGB a notación hex no es intuitiva. A menudo, el valor se toma del selector de color. Por ejemplo, puedes copiar y pegar un valor hex desde el selector de color de Processing en tu código. Cuando se usan valores de color codificados en notación hex, debemos colocar un # antes del valor para distinguirlo dentro del código.



```
// Código del tercer ejemplo reescrito usando números hex
background(#818257);
noStroke();
fill(#AEDD3C);
rect(17, 17, 66, 66);
```

-Valores de grises

Los ejemplos vistos anteriormente han usado el fondo por defecto de color gris claro, líneas negras, y figuras blancas. Para cambiar estos valores, es necesario introducir sintaxis adicional. La función `background()` establece el color de la ventana de representación con un número entre 0 y 255. Este rango puede ser incómodo si no estás familiarizado con programas de dibujo en el ordenador. El valor 255 es blanco y el valor 0 es negro, con un rango de valores de grises en medio. Si no se define un valor para el fondo, se usa el valor por defecto 204 (gris claro).

La función `fill()` define el valor del relleno de las figuras, y la función `stroke()` define el valor del contorno de las figuras dibujadas. Si no se define un valor de relleno, se usa el valor por defecto 255 (blanco). Si no se define un valor de contorno, se usa el valor por defecto 0 (negro).

```
rect(10, 10, 50, 50);
fill(204);
stroke(102);
rect(20, 20, 50, 50);
fill(153);
stroke(153);
rect(30, 30, 50, 50);
fill(102);
stroke(204);
rect(40, 40, 50, 50);
```

Cuando se ha definido un valor de relleno o contorno, se aplica a todas las figuras dibujadas después. Para cambiar el valor de relleno o contorno, usamos la función `fill()` o `stroke()` de nuevo.

Un parámetro opcional adicional para `fill()` o `stroke()` regula la transparencia. Definiendo el parámetro a 255 hace que la figura sea totalmente opaca, y a 0 totalmente transparente:

```
background(0);
fill(255, 220);
rect(15, 15, 50, 50);
rect(35, 35, 50, 50);
```

El relleno y el contorno de una figura se puede eliminar. La función `noFill()` detiene a Processing de rellenar figuras, y la función `noStroke()` detiene la creación de líneas y contornos de las figuras. Si usamos `noFill()` y `noStroke()` no dibujaremos nada en la pantalla.

-Atributos de dibujo

Además de cambiar los valores de relleno y contorno de las figuras, también es posible cambiar atributos de

la geometría. Las funciones `smooth()` y `noSmooth()` activan y desactivan el suavizado (conocido como filtro *antialiasing*). Cuando usamos una de estas funciones, afectará a todas las funciones dibujadas después. Si usamos primero `smooth()`, usar `noSmooth()` cancelará el ajuste, y viceversa.

```
smooth();
ellipse(30, 48, 36, 36);
noSmooth();
ellipse(70, 48, 36, 36);
```

Los atributos de la línea, están controlados por las funciones `strokeWeight()`, `strokeCap()` y `strokeJoin()`. La función `strokeWeight()` tiene un parámetro numérico que define el grosor de todas las líneas dibujadas después de usar esta función. La función `strokeCap()` requiere un parámetro que puede ser `ROUND`, `SQUARE` o `PROJECT`. `ROUND` redondea los puntos finales, y `SQUARE` los cuadra. `PROJECT` es una mezcla de ambos: redondea las esquinas cuadradas suavemente. Esta función se usa en líneas. La función `strokeJoin()` tiene un parámetro que puede ser `BEVEL`, `MITTER` o `ROUND`. Estos parámetros determinan la forma del contorno de la figura. `BEVEL` corta en diagonal las esquinas cuadradas, `MITTER` cuadra las esquinas (es el valor por defecto) y `ROUND` redondea las esquinas.

```
smooth();
strokeWeight(12);
strokeCap(ROUND);
line(20, 30, 80, 30); //Línea superior
strokeCap(SQUARE);
line(20, 50, 80, 50); //Línea central
strokeCap(PROJECT);
line(20, 70, 80, 70); //Línea inferior

smooth();
strokeWeight(12);
strokeJoin(BEVEL);
rect(12, 33, 15, 33); //Figura izquierda
strokeJoin(MITER);
rect(42, 33, 15, 33); //Figura central
strokeJoin(ROUND);
rect(72, 33, 15, 33); //Figura derecha
```

Modos de dibujo

Por defecto, los parámetros para `ellipse()` define la coordenada x del centro, la coordenada y del centro, la anchura y la altura. La función `ellipseMode()` cambia la forma en que se usan estos parámetros para dibujar elipses. La función `ellipseMode()` requiere un parámetro que puede ser `CENTER`, `RADIUS`, `CORNER` o `CORNERS`. El modo por defecto es `CENTER`. El modo `RADIUS` también usa el primer y segundo parámetro de `ellipse()` para establecer el centro, pero el tercer parámetro debe ser la mitad del ancho, y el cuarto parámetro debe ser la mitad del alto. El modo `CORNER` hace que `ellipse()` funcione de manera parecida a `rect()`. Causa que el primer y segundo parámetro se trasladen a la esquina superior izquierda del rectángulo que circunscribe la elipse y usa el tercer y cuarto parámetro para definir la anchura y la altura. El modo `CORNERS` tiene un efecto similar a `CORNER`, pero causa que el tercer y cuarto parámetro de `ellipse()` se definan en la esquina inferior derecha del rectángulo.

```
smooth();
noStroke();
ellipseMode(RADIUS);
fill(126);
ellipse(33, 33, 60, 60); //Elipse gris
fill(255);
```

```
ellipseMode(CORNER);
ellipse(33, 33, 60, 60);    //Elipse blanca
fill(0);
ellipseMode(CORNERS);
ellipse(33, 33, 60, 60);    //Elipse negra
```

Con un estilo similar, la función `rectMode()` afecta a cómo se dibujan los rectángulos. Requiere un parámetro que puede ser `CORNER`, `CORNERS` o `CENTER`. El modo por defecto es `CORNER`, y `CORNERS` causa que el tercer y cuarto parámetro de `rect()` se definan en la esquina contraria a la primera. El modo `CENTER` causa que el primer y segundo parámetro de `rect()` definan el centro del rectángulo y usa el tercer y cuarto parámetro para la anchura y la altura.

```
noStroke();
rectMode(CORNER);
fill(126);
rect(40, 40, 60, 60);        //Rectángulo gris
rectMode(CENTER);
fill(255);
rect(40, 40, 60, 60);        //Rectángulo blanco
rectMode(CORNERS);
fill(0);
rect(40, 40, 60, 60);        //Rectángulo negro
```

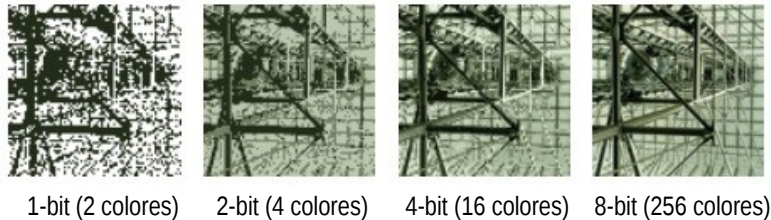
Unidad 10

Imagen: Visualización y Tinta

Elementos que se introducen en esta Unidad:

`PImage`, `loadImage()`, `image()`
`tint()`, `noTint()`

Las fotografías digitales son muy diferentes si las comparamos con las fotografías analógicas capturadas en film. Al igual que las pantallas de ordenador, las imágenes digitales son rejillas rectangulares de color. Las dimensiones de las imágenes digitales se miden en unidades de píxeles. Si una imagen es de 320 píxeles de ancho y 240 píxeles de alto, tiene 76.800 píxeles totales. Si una imagen es de 1280 píxeles de ancho y 1024 píxeles de alto, el número total de píxeles es de 1310720 (1,3 megapíxeles). Cada imagen digital tiene una profundidad de color. La profundidad de color se refiere al número de bits utilizado para almacenar cada píxel. Si la profundidad de color de una imagen es 1, cada píxel puede ser uno de dos valores, por ejemplo, negro o blanco. Si la profundidad de color es de 4, cada píxel puede ser uno de 16 valores. Si la profundidad de color de una imagen es de 8, cada píxel puede ser uno de 256 valores. En cuanto a la imagen, se muestra con diferentes profundidades de color. Esto afecta a la apariencia:



Hoy en día, la mayoría de las computadoras trabajan con una profundidad de 24 bits, lo que quiere decir uno de 16777216 colores. Se lo llama comúnmente como *millones de colores*.

Las imágenes digitales se **componen de números** que representan los colores. El formato de archivo de una imagen determina como los números se ordenan en el archivo. Algunos formatos de archivo de almacenamiento de los datos de color, utilizan ecuaciones de matemáticas complejas para comprimir los datos y reducir el tamaño resultante del archivo final. Un programa que se carga un archivo de imagen debe conocer el formato de la misma para que pueda traducir los datos de archivo en la imagen esperada. Existen diferentes tipos de formatos de imágenes digitales que sirven para necesidades específicas. Processing pueden cargar archivos GIF, JPEG, y PNG, junto con algunos otros formatos como se describe en la referencia. Si no tiene su imagen en uno de estos formatos, se puede convertir a otros tipos de formatos con programas como GIMP o Adobe Photoshop.

<u>Format</u>	<u>Extension</u>	<u>Profundidad de color</u>	<u>Transparencia</u>
GIF	.gif	1-bit a 8-bit	1-bit
JPEG	.jpg	24-bit	No Posee
PNG	.png	1-bit a 24-bit	8-bit

-Visualización

Processing puede cargar imágenes, mostrarlas en pantalla, cambiar su posición, tamaño, color, tinta y transparencia. El tipo de dato que se utiliza para almacenar imágenes es conocido por el nombre de `PImage`. Al igual que los números enteros son almacenados en variables del tipo `int`, y los valores `true` y `false` en variables del tipo `boolean`, las imágenes son almacenadas en variables tipo `PImage`. Antes de mostrarla en al ventana, es necesario cargar la imagen con la función `loadImage()`. Se debe revisar con cuidado que además de ingresar el nombre correcto también este la extensión de formato correcta (por ejemplo, *kant.jpg*, *woah.gif*, *hola.png*). Las imágenes a cargar deben encontrarse en la carpeta *data*, dentro de la carpeta del *sketch*. Existe una forma muy sencilla de añadir una imagen a esa carpeta sin cometer ningún error. En la

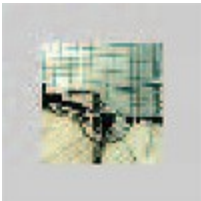
barra de menú, dentro de la opción "Sketch", se encuentra el comando "Add File". Se podrá entonces navegar por su ordenador y buscar el archivo deseado. Una vez que se encuentra, se le da click a "Open" para añadir el archivo a la carpeta *data* de su sketch. Con la imagen dentro de la carpeta, habiéndose declarado un nuevo *PImage* y, posteriormente, cargado en ese *PImage* un archivo de imagen, este puede ser mostrado en pantalla con la función `image()`:

```
image(nombre, x, y)
image(nombre, x, y, ancho, alto)
```

Los parámetros de `image()` determinan la posición y el tamaño de la imagen (similar a otras funciones vistas previamente). El *nombre* es tan solo el nombre de la variable *PImage* donde se encuentra almacenada nuestra imagen. Si no se utilizan los parámetros *ancho* y *alto*, la imagen será mostrada en su tamaño original.



```
PImage img;
//La imagen debe estar guardada en la carpeta "data"
img = loadImage("arch.jpg");
image(img, 0, 0);
```



```
PImage img;
//La imagen debe estar guardada en la carpeta "data"
img = loadImage("arch.jpg");
image(img, 20, 20, 60, 60);
```

-Color de la Imagen y Transparencia

Las imágenes pueden ser coloreadas con la función `tint()`. Esta actúa del mismo modo que `fill()` y `stroke()`, solo que afecta únicamente a las imágenes:

```
tint(gris)
tint(gris, alfa)
tint(valor1, valor2, valor3)
tint(valor1, valor2, valor3, alfa)
tint(color)
```

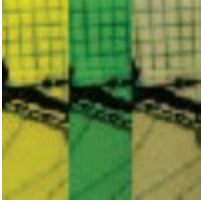
Todas las imágenes que se dibujan luego de declarada la función `tint()`, será pintada por ella. Si en algún momento se desea que no se pinte, se puede utilizar la función `noTint()`.



```
PImage img;
img = loadImage("arch.jpg");
tint(102); //Pinta de gris
image(img, 0, 0);
noTint(); //Desactiva tint()
image(img, 50, 0);
```



```
PImage img;
img = loadImage("arch.jpg");
tint(0, 153, 204); //Pinta de azul
image(img, 0, 0);
noTint(); //Desactiva tint()
image(img, 50, 0);
```



```
color amarillo = color(220, 214, 41);
color verde = color(110, 164, 32);
color tan = color(180, 177, 132);
PImage img;
img = loadImage("arch.jpg");
tint(amarillo);
image(img, 0, 0);
tint(verde);
image(img, 33, 0);
tint(tan);
image(img, 66, 0);
```

Los parámetros de `tint()` están ligados a lo establecido por `colorMode()`. Hay que tener cuidado de no cambiar el modo de color o el rango del mismo, ya que eso afectará los parámetros de `tint()`.

Para generar transparencia simplemente utilizaremos la misma función `tint()` pero con un valor de blanco (255 en un modo RGB).



```
PImage img;
img = loadImage("arch.jpg");
background(255);
tint(255, 102); //Alfa de 102 sin cambiar el color de pintura
image(img, 0, 0, 100, 100);
tint(255, 204, 0, 153); //Pinta amarillo con una
image(img, 20, 20, 100, 100); //transparencia de 153
```



```
PImage img;
img = loadImage("arch.jpg");
background(255);
tint(255, 51);
//Dibuja las imágenes 10 veces moviendo las mismas
//a la derecha
for (int i = 0; i < 10; i++) {
    image(img, i*10, 0);
}
```

Las imágenes de formato GIF y PNG pueden almacenar la transparencia de la imagen. Mientras que las GIF solo pueden convertir un color (ya que solo posee 1-bit de transparencia). En cambio, las imágenes PNG tienen una profundidad de transparencia de 8-bit, por lo tanto admiten 256 colores de transparencia.



```
//Carga una imagen GIF con 1-bit de transparencia
PImage img;
img = loadImage("archTrans.gif");
background(255);
image(img, 0, 0);
image(img, -20, 0);
```



```
//Carga una imagen PNG con 8-bit de transparencia
PImage img;
img = loadImage("arch.png");
background(255);
image(img, 0, 0);
image(img, -20, 0);
```

Unidad 11

Datos: Texto

Elementos que se introducen en esta Unidad:

char, String

Desde la aparición del ordenador se plantearon lenguajes de programación para sus acciones. A partir de ese momento, hubo mucha investigación en la entrada y la salida de texto. De esta forma se expandió el campo de la programación a lo que fueron los primeros intentos por IA (Inteligencia Artificial). Una serie de programas en los que uno escribía lo que deseaba y el mismo programa, a duras penas, respondía algo más o menos acorde. Esta clase de programas ha evolucionado, más allá de que existen todavía problemas con esta clase de ideas.

Más adelante, se experimentó con programas (preferentemente en la web) de traducción de idiomas, con resultados extraordinarios por parte de algunos servidores, que aún existen en la actualidad.

Esta unidad no discute temas acerca de traducción de idiomas o inteligencia artificial, sin embargo el texto es el tipo de dato más común creado y modificado por el programa. El texto creado por e-mails, publicaciones y sitios web es un vasto recurso de información que puede ser almacenada y presentada en forma de datos.

-Caracteres

Los datos del tipo char pueden almacenar tipografías como A, e, 5 y %. El nombre char es el diminutivo para *character* cuyo significado es carácter, y este tipo de datos se distingue por se escritos entre comillas (' '). Las variables tipo char son declaradas como variables tipo int o tipo float.

```
char a = 'n';      //Asigna 'n' a la variable a
char b = n;       //ERROR - Sin comillas, n es una variable
char c = "n";     //ERROR - Las comillas dobles (" ") definen a n como un
                  //String, no un char
char d = 'not';   //ERROR - Las variables tipo char pueden almacenar
                  //solo un carácter
```

El siguiente ejemplo crea una nueva variable tipo char, asigna valores y los imprime en la consola:

```
char letra = 'A'; //Declara una variable letra y asigna el valor de 'A'
println(letra);  //Imprime "A" en la consola
letra = 'B';     //Asigna 'B' a la variable letra
println(letra);  //Imprime "B" en la consola
```

Muchos valores char tienen un valor número que les corresponde por el código de la tabla ASCII. Por ejemplo, A es 65, B es 66, C es 67, y así. Puede probar cada código usando la función `println()`:

```
char letra = 'A';
println(letra); //Imprime "A" en la consola
int n = letra;  //Asigna el valor numérico de A a la variable n
println(n);    //Imprime 65 en la consola
```

Es posible articular funciones matemáticas con valores alfabéticos. En el siguiente ejemplo se utiliza un ciclo FOR para imprimir el alfabeto de la A a la Z, por aumentar los valores char con la función de incremento:

```
char letra= 'A';      //Declara una variable tipo char y asigna el valor de A
for (int i = 0; i < 26; i++) {
    print(letra);     //Imprime un carácter en la consola
    letra++;         //Suma 1 al valor del carácter
}
println('.');        //Agrega un punto al final del alfabeto
```

-Palabras y Oraciones

Al utilizar una variable de tipo `String` pueden almacenarse palabras y oraciones completas. Para distinguir los datos `String` del resto del programa se utilizan comillas dobles (`"`). Las dobles comillas diferencian `"s"` como `String` de `'s'` como carácter. Los datos tipos `String` son diferentes a los del tipo `int`, `float` o `char`, ya que se trata de un *objeto*, o sea un tipo de dato que puede almacenar muchos elementos y funciones. Las variables tipo `PImage` y `PFon`t también son objetos. Las variables tipo `String` se declaran y se asignan como cualquiera:

```
String a = "Anónimo"; //Asigna "Anónimo" a la variable a
String b = 'E';       //ERROR - Las comillas ' ' deben ser dobles.
String c = "E";       //Asigna "E" a c
string d = "E";       //ERROR - String debe comenzar con mayúsculas
```

Los siguientes ejemplos muestran algunos usos básicos de las variables tipo `String`:

```
//Los datos tipo String pueden contener largos textos
String s1 = "Rakete bee bee?";
String s2 = "Rrrrrrrrrrrrrrrrrmmmmppffff tillffff tooooo?";
println(s1); //Imprime "Rakete bee bee?"
println(s2); //Imprime "Rrrrrrrrrrrrrrrrrmmmmppffff tillffff tooooo?"

//Datos tipo String combinados con el operador +
String s3 = "Rakete ";
String s4 = "rinzekete";
String s5 = s3 + s4;
println(s5); //Imprime "Rakete rinzekete"
```

Unidad 12

Datos: Conversión y Objetos

Elementos que se introducen en esta Unidad:

```
boolean(), byte(), char(), int(), float(), str()
"." (Punto como Operador)
PImage.width, PImage.height
String.length(), String.startsWidth(), String.endsWidth(),
String.charAt(), String.toCharArray(), String.subString(),
String.toLowerCase(), String.toUpperCase()
String.equals()
```

Cuando creamos una variable, especificamos el tipo de datos con el que va a trabajar. Si la variable almacena números, su utilizará una variable tipo `int` o tipo `float`. Si la variable almacena texto, se usará el tipo `String` o, si se tratara de solo un carácter, del tipo `char`. Así, variables que almacenan valores `true` o `false` son del tipo `boolean`, para almacenar una imagen digital la variable será del tipo `PImage` y para almacenar una tipografía del tipo `PFont`. Luego de ser declarada, se le asigna un valor del tipo de la variable. Sin embargo, muchas veces es necesario convertir una variable de determinado tipo en otro tipo.

Los tipos de datos `int`, `float`, `char` y `boolean` son llamadas variables primitivas, ya que almacenan un **simple** elemento. En cambio, las del tipo `String`, `PImage` y `PFont` son diferentes, las variables que se crean son **objetos**. Los objetos son usualmente compuestos por muchas variables primitivas y otros objetos, y pueden poseer funciones dentro de ellos mismos. Los objetos se distinguen por comenzar la primer letra en mayúsculas.

-Conversión de Datos

Algunos tipos de datos pueden convertirse en otro tipo automáticamente con solo se reasignados a una variable de otro tipo. Por ejemplo, una variable del tipo `int` puede convertirse automáticamente en una del tipo `float`, pero una del tipo `float` no puede convertirse automáticamente en una del tipo `int`.

```
float f = 12.6;
int i = 127;
f = i; //Convierte 127 en 127.0
i = f; //ERROR - No se puede convertir automáticamente un float en un int
```

¿Qué ocurre si necesitamos convertir otra clase de variable en otros tipos? Para esto existen una gran cantidad de funciones de conversión de un tipo de dato a otro. Sin embargo, hay que aclarar que, como los datos almacenan diferentes cantidad de bit, es muy probable que en algunos casos se pierda una gran cantidad de información (muy útil para ahorrar memoria del programa). Las funciones de conversión son las siguientes: `boolean()`, `byte()`, `char()`, `float()`, `int()`, y `str()`. Cada una transforma cualquier clase de dato en el correspondiente a cada función.

La función `boolean()` convierte el número 0 en un valor `false` y cualquier otro número en un valor `true`.

```
int i = 0;
boolean b = boolean(i); //Asigna false a b
int n = 12;
b = boolean(n); //Asigna true a b
String s = "false";
b = boolean(s); //Asigna false a b
```

La función `byte()` convierte cualquier valor en una representación de `byte`. La profundidad numérica es tan solo de -127 a 128. Sin embargo, si el número está fuera del rango, la función lo convierte en una representación en `byte`.

```
float f = 65.0;
byte b = byte(f);           //Asigna 65 a b
char c = 'E';
b = byte(c);               //Asigna 69 a b
f = 130.0;
b = byte(f);               //Asigna -126 a b
```

La función `char()` convierte cualquier valor numérico en un carácter. El número corresponde a una representación de la tabla ASCII.

```
int i = 65;
byte y = 72.0;
char c = char(i);         //Asigna 'A' a c
c = char(y);              //Asigna 'H' a c
```

La función `float()` convierte cualquier clase de dato en un número decimal. Es usado frecuentemente cuando hacemos cálculos matemáticos.

```
int i = 2;
int j = 3;
float f1 = i/j;           //Asigna 0.0 a f1
float f2 = i/float(j);   //Asigna 0.6666667 a f2
```

La función `int()` convierte cualquier clase de datos en un número entero. Es muy utilizado para redondear un resultado `float` y así este no ocupe demasiada memoria innecesariamente. En muchos casos hay funciones que devuelven valores `float` y es necesario convertirlos.

```
float f = 65.3;
int i = int(f);           //Asigna 65 a i
char c = 'E';
i = int(c);               //Asigna 69 a i
```

La función `str()` convierte cualquier clase de datos en una representación `String`.

```
int i = 3;
String s = str(i);        //Asigna "3" a s
float f = -12.6;
s = str(f);               //Asigna "-12.6" a s
boolean b = true;
s = str(b);               //Asigna "true" a s
```

La función `nf()` (que se verá más adelante) provee de más control cuando convertimos un `int` o un `float` a un `String`.

-Objetos

Las variables tipo `String`, `PImage` y `PFon`t son del tipo objeto. Las variables dentro de un objeto se denominan *campos* y las funciones se denominan *métodos*. Se puede acceder a los campos y a los métodos con el operador *punto* (`.`). El operador punto funciona como puente entre el nombre del objeto y el elemento dentro del mismo. `PImage`, `PFon`t y `String` son objetos, y por tanto poseen elementos.

Los objetos tipo `PImage` poseen dos funciones que almacenan el ancho y el alto de la imagen cargada. Se llaman, de manera apropiada, `width` (ancho) y `height` (alto). Para acceder a ellas, se escribe el nombre del objeto seguido por el operador punto y el nombre de la función.

```

PImage img = loadImage("buenos_aires.jpg");    //Carga una Imagen de 320 x 240
pixeles
int w = img.width;                            //Asigna 320 to w
int h = img.height;                          //Asigna 240 to h
println(w);                                  //Imprime "320"
println(h);                                  //Imprime "240"

```

Más adelante, se discutirán una gran variedad de funciones del objeto PImage.

Por otro lado, el objeto String posee también una gran cantidad de elementos que lo componen. Uno de ellos es el método length(), el cual regresa de valor la cantidad de caracteres que lo componen:

```

String s1 = "Tocar Piano";
String s2 = "T";
println(s1.length());           //Imprime "11"
println(s2.length());           //Imprime "1"

```

Los métodos startswith() y endswith() evalúan cuando un String empieza o termina, siempre y cuando se utilice el parámetros adecuados:

```

String s1 = "Buenos Aires";
println(s1.startsWith("B"));    //Imprime "true"
println(s1.startsWith("Aires")); //Imprime "false"
println(s1.endsWith("Aires")); //Imprime "true"

```

El método charAt() es usado para leer un solo carácter de un String. Recibe como parámetro un número que indica la ubicación del mismo:

```

String s = "Verde";
println(s.charAt(0));           //Imprime "V"
println(s.charAt(2));           //Imprime "r"
println(s.charAt(4));           //Imprime "e"

```

El método toCharArray() crea un nuevo array de caracteres del contenido del String:

```

String s = "Azul";
char[] c = s.toCharArray();
println(c[0]);                  //Imprime "A"
println(c[1]);                  //Imprime "z"

```

El método substring() es usado para leer a porción del String. A diferencia de charAt() que solo lee un carácter, substring() puede leer un String de caracteres. Se utiliza de parámetro un número que indica la posición de un carácter y devuelve un String desde esa posición para adelante. Si se utilizan dos parámetros determina una porción específica.

```

String s = "Giallo";
println(s.substring(2));        //Imprime "allo"
println(s.substring(4));        //Imprime "lo"
println(s.substring(1, 4));     //Imprime "ial"
println(s.substring(0, s.length()-1)); //Imprime "Giall"

```

El método toLowerCase() puede usarse para devolver una copia del String completamente en minúsculas. De la misma forma, toUpperCase() hace lo mismo pero devuelve el texto en mayúsculas.

```
String s = "Negro";  
println(s.toLowerCase());    //Imprime "negro"  
println(s.toUpperCase());    //Imprime "NEGRO"
```

Como un `String` es un objeto, no es posible comparar resultados con el operador relacional de igualdad (`==`). Sin embargo, el método `equals()` del objeto `String` sirve precisamente para eso. Compara el contenido de dos variables `String`.

```
String s1 = "Blanco";  
String s2 = "Blanco";  
String s3 = "Negro";  
println(s1.equals(s2));      //Imprime "true"  
println(s1.equals(s3));      //Imprime "false"
```


Tipografía: Visualización

Elementos que se introducen en esta Unidad:

PFont, loadFont(), textFont(), text()
 textSize(), textLeading(), textAlign(), textWidth()

Las letras en una pantalla son creadas por establecer el color de los píxeles en una determinada ubicación. Por lo tanto, la calidad de la tipografía se ve limitada por la resolución de la pantalla. Como las pantallas tienen una resolución baja en comparación con el papel, se han desarrollado técnicas para mejorar la apariencia del texto en la misma. Las fuentes de las primeras computadoras de Apple Macintosh estaban compuestas por pequeñas imágenes de mapa de bits, creadas específicamente en tamaños como de 10, 12, y 24 puntos. Utilizando esta tecnología, una variación de cada fuente fue diseñada para cada tamaño de un tipo de letra en particular. Por ejemplo, el carácter de un tipo de letra en el San Francisco, utiliza una imagen diferente para mostrar el carácter en un tamaño de 12 y 18. Por otra parte, cuando la impresora LaserWriter se introdujo en 1985, la tecnología Postscript definió fuentes con una descripción matemática del esquema de cada carácter. Este tipo permitió, en la pantalla, una escala de tamaños grandes. Apple y Microsoft desarrollaron, más tarde, el TrueType, otro formato de fuente. Recientemente, estas tecnologías se fusionaron en el formato OpenType. Mientras tanto, el método para suavizar texto de pantalla en blanco-y-negro se introdujo. Estas técnicas del tipo *anti-aliasing* utilizan píxeles grises en el borde de los caracteres para compensar la baja resolución de pantalla.

-Cargando Fuentes y Dibujando Texto

Antes que las letras sean mostradas en la ventana de Processing, es necesario cargar la fuente y para esto debemos convertir el archivo en uno de formato VLW. Para esto, Processing nos ofrece una sencilla herramienta. En el menú "Tools" (herramientas) seleccionar la opción "Create Font" (Crear Fuente). Un ventana se abrirá y le mostrará las fuentes que tiene actualmente instaladas en su equipo y que pueden ser convertidas al formato específico. Seleccione la fuente deseada de la lista y, a continuación, haga click en "OK". La fuente se copiará y convertirá al formato VLW. Luego, automáticamente, será movida a la carpeta "Data" de su Sketch (de forma similar a como se carga una imagen).

Como en las primeras computadoras de Apple Macintosh, el formato VLW permite cargar el alfabeto de una fuente como una serie de simples imágenes. En la ventana del "Create Font" también se encuentra una opción que permite seleccionar el tamaño al exportar al fuente en un formato VLW. Esto es útil, ya que fuentes muy grandes requieren mayor cantidad de memoria. Si se va a utilizar una fuente no mas grande que 12, exportarla en tamaño 96 será totalmente innecesario.

Después que la fuente es creada, antes de que el programa la pueda utilizar es necesario cargarla en el mismo. Processing posee un tipo de dato llamado PFont, en el que deben cargarse las fuentes. Se crea una nueva variable tipo PFont y se utiliza las función loadFont() para cargarla. La función de textFont() se utiliza para seleccionar la fuente que se quiere utilizar. Por último, el texto es escrito con la función text().

```
text(datos, x, y)
text(stringdatos, x, y, ancho, alto)
```

El parámetro *datos* puede ser un String, char, int o float. El parámetro *stringdatos* es un parámetro que solo acepta datos tipo String. Los parámetros *x* e *y* cambian la posición horizontal y vertical del texto (como en cualquier figura o imagen). Existen los parámetros opcionales *ancho* y *alto*, los cuales modifican el tamaño del texto. La función fill() controla el color del texto y la transparencia, así como también afecta a figuras como ellipse() o rect(). El texto no es afectado por stroke().

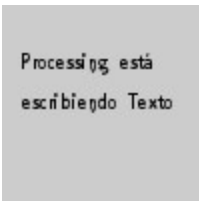
Los siguientes ejemplos utilizan una fuente llamada Aressence, para que pueda ver su texto asegúrese de cargar su propia fuente y cargarla correctamente (con la extensión) en la función loadFont().



```
PFont font; // Declare the variable
font = loadFont("Aressence-32.vlw"); //Carga la fuente
textFont(font); //Selecciona la fuente actual
fill(0);
text("LAX", 0, 40); //Escribe "LAX" en la posición (0,40)
text("AMS", 0, 70); //Escribe "AMS" en la posición (0,70)
text("FRA", 0, 100); //Escribe "FRA" en la posición (0,100)
```



```
PFont font;
font = loadFont("Aressence-32.vlw");
textFont(font);
fill(0);
text(19, 0, 36); //Escribe 19 en la posición (0,36)
text(72, 0, 70); //Escribe 72 en la posición (0,70)
text('R', 62, 70); //Escribe 'R' en la posición (62,70)
```



```
PFont font;
font = loadFont("Aressence-12.vlw");
textFont(font);
fill(0);
String s = "Processing está escribiendo Texto";
text(s, 10, 20, 80, 50);
```



```
PFont font;
font = loadFont("Aressence-32.vlw");
textFont(font);
fill(255); //Blanco
text("DAY", 0, 40);
fill(0); //Negro
text("CVG", 0, 70);
fill(102); //Gris
text("ATL", 0, 100);
```



```
PFont font;
font = loadFont("Aressence-72.vlw");
textFont(font);
fill(0, 160); //Negro con baja Opacidad
text("1", 0, 80);
text("2", 15, 80);
text("3", 30, 80);
text("4", 45, 80);
text("5", 60, 80);
```

También pueden utilizarse dos fuentes distintas en el mismo programa, pero deberán crearse dos variables tipo PFont que las almacenen:



```
PFont font1, font2;
font1 = loadFont("Aressence-32.vlw");
font2 = loadFont("Arhermann-32.vlw");
fill(0);
textFont(font1);
text("GNU", 6, 45);
textFont(font2);
text("GNU", 6, 80);
```

-Atributos de Texto


Processing incluye funciones para controlar la forma en la que el texto se muestra (por ejemplo, alterando su tamaño o alineación). Processing, además, puede calcular el ancho de algún carácter o grupo de caracteres y

ponerlo en función de alguna figura o forma.


Las fuentes en Processing son tomadas como imágenes, y no como vectores. Por eso si la fuente que cargamos está en 12 de tamaño, y luego la agrandamos a 96, se verá desenfocada. La función correcta para cambiar el tamaño de la fuente es la de `textSize()`.

```
textSize(tamaño)
```

El parámetro *tamaño* define la dimensión que ocuparán las letras en unidades de píxeles.



```
//Reducción del tamaño de la letra
PFont font;
font = loadFont("Aressence-32.vlw");
textFont(font);
fill(0);
text("LNZ", 0, 40); //Grande
textSize(18);
text("STN", 0, 75); //Mediano
textSize(12);
text("BOS", 0, 100); //Pequeño
```



```
//Agrandar tamaño de letra
PFont font;
font = loadFont("Aressence-12.vlw");
textFont(font);
textSize(32);
fill(0);
text("LNZ", 0, 40); //Grande
textSize(18);
text("STN", 0, 75); //Mediano
textSize(12);
text("BOS", 0, 100); //Pequeño
```

La función `textLeading()` permite alterar el espacio entre líneas:

```
textLeading(dist)
```

La propiedad *dist* define la distancia del espacio en unidad de píxeles.

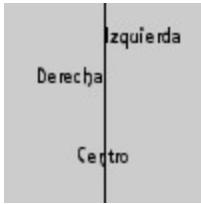
```
PFont font;
font = loadFont("Aressence-12.vlw");
textFont(font);
String lines = "L1 L2 L3";
textLeading(10);
fill(0);
text(lines, 5, 15, 30, 100);
textLeading(20);
text(lines, 36, 15, 30, 100);
textLeading(30);
text(lines, 68, 15, 30, 100);
```

Además, el texto puede ser marginado al centro, a la izquierda o a la derecha utilizando la función `textAlign()`:

```
textAlign(MODO)
```

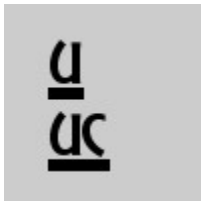
En este caso, el modo es una constante que puede ser `CENTER` (centro), `RIGHT` (derecha) y `LEFT`

(izquierda), y se refiere simplemente a la alineación que tendrá el texto mostrado en pantalla.



```
PFont font;  
font = loadFont("Aressence-12.vlw");  
textFont(font);  
line(50, 0, 50, 100);  
fill(0);  
textAlign(LEFT);  
text("Izquierda", 50, 20);  
textAlign(RIGHT);  
text("Derecha", 50, 40);  
textAlign(CENTER);  
text("Centro", 50, 80);
```

La función `textwidth()` calcula y regresa el ancho de pixel de un carácter o texto. El calculo se realiza dependiendo de las propiedades de `textFont()` y `textSize()`. Ya que cada carácter posee un ancho diferente entre sí, es imposible saber la medida exacta de una cadena de caracteres. Para esos casos donde necesitamos una medida específica, o conocer el tamaño de nuestra cadena, se utiliza esta función.



```
PFont font;  
font = loadFont("Aressence-32.vlw");  
textFont(font);  
fill(0);  
char c = 'U';  
float cw = textWidth(c);  
text(c, 22, 40);  
rect(22, 42, cw, 5);  
String s = "UC";  
float sw = textWidth(s);  
text(s, 22, 76);  
rect(22, 78, sw, 5);
```

Unidad 14

Matemáticas: Trigonometría

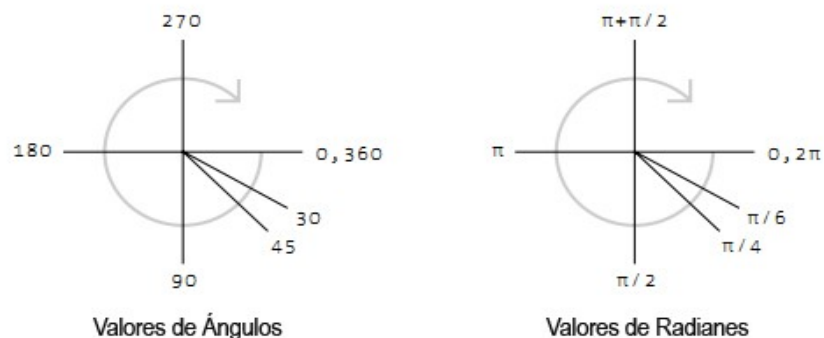
Elementos que se introducen en esta Unidad:

PI, QUARTER_PI, HALF_PI, TWO_PI, radians(), degrees()
sin(), cos(), arc()

La trigonometría permite definir relaciones entre lados y ángulos de los triángulos. Las funciones trigonométricas de seno y coseno nos permiten generar números repetitivos para dibujar ondas, arcos, círculos y espirales.

-Ángulos y Ondas

Los grados son una forma común de medir ángulos. Un ángulo recto es de 90° , a mitad de camino, alrededor de un círculo, es 180° , y el círculo completo es de 360° . Al trabajar con trigonometría, los ángulos se miden en unidades llamadas *radianes*. Utilizando los radianes, el ángulo de un valor se expresa en relación con el valor de π de matemática, escrito en caracteres latinos como "pi". En términos de unidades radianes, el ángulo recto es $\pi/2$, a mitad de camino alrededor de un círculo es simplemente π , y el punto de partida es 2π .



El valor numérico de π es una constante que es infinitamente larga y sin ningún patrón de repetición. Es la razón de la circunferencia de un círculo a su diámetro. Al escribir el código en Processing, se puede utilizar la constante matemática *PI* para representar a este número. Otros valores de uso de π se expresan con las constantes *QUARTER_PI*, *HALF_PI* y *TWO_PI*. Si se ejecuta la siguiente línea de código, se podrá ver el valor de π con sus 8 dígitos significativos:

```
println(PI); //Imprime el valor de  $\pi$  en la consola
```

En el uso cotidiano, el valor numérico de π suele denominarse como 3,14. De esta manera, los ángulos pueden convertirse en radianes a través de la función `radians()`, y viceversa con la función `degrees()`.

```
float r1 = radians(90);  
float r2 = radians(180);  
println(r1); //Imprime "1.5707964"  
println(r2); //Imprime "3.1415927"  
float d1 = degrees(PI);  
float d2 = degrees(TWO_PI);  
println(d1); //Imprime "180.0"  
println(d2); //Imprime "360.0"
```

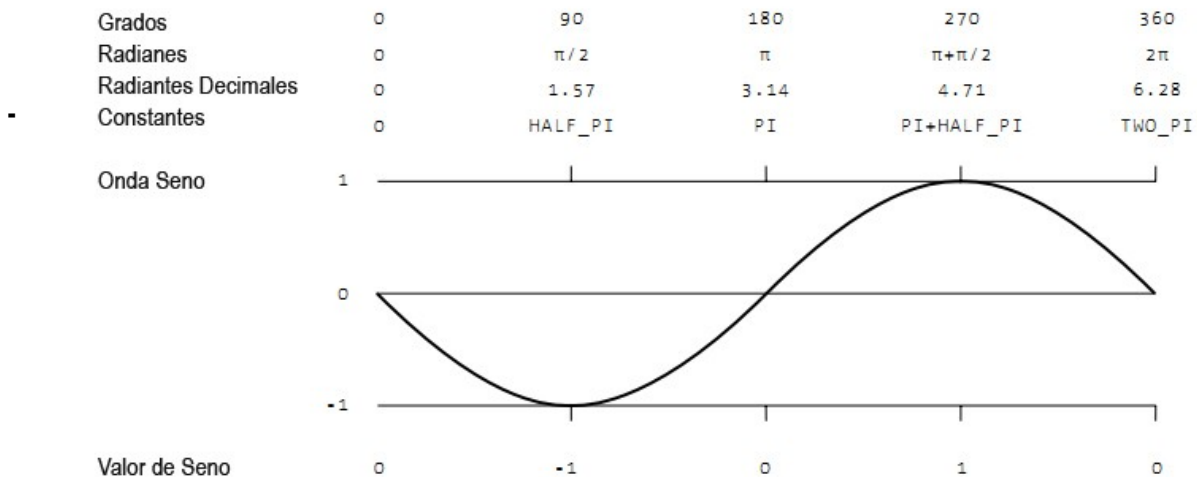
Si uno tiene preferencia por el trabajo en ángulos, se vuelve muy práctico el utilizar la función de conversión

para estar más cómodo a la hora de programar.

Por otro lado, las funciones `sin()` y `cos()` determinan el valor de seno y coseno, respectivamente, de un ángulo determinado. Ambas funciones aceptan tan solo 1 parámetro:

```
sin(angulo)
cos(angulo)
```

El parámetro *angulo* es siempre un valor en radianes específico. El valor que devuelve es del tipo decimal entre -1.0 y 1.0. Las relaciones establecidas entre el valor en radianes y el valor de su seno son mostradas en el siguiente gráfico:



A medida que el valor del ángulo incrementa su valor, el valor de seno se repite. En el valor de ángulo 0, el valor de seno también es 0. A medida que comienza a aumentar el ángulo, el seno decrece, hasta llegar a 90°, donde el seno aumenta, pasando por el valor 180° (en cual el seno vuelve a ser 0) hasta el 270° (donde el seno llega a su punto más alto). Por último, cuando el valor llega a 360°, el seno vuelve a ser 0, y el ciclo vuelve a repetirse. Los valores del seno pueden ser visto en su totalidad simplemente por ser ubicados dentro de una estructura FOR:

```
for (float angle = 0; angle < TWO_PI; angle += PI/24.0) {
    println(sin(angle));
}
```

Como los valores de seno son entre -1.0 y 1.0 son muy fáciles de usar en una composición. Por ejemplo, al multiplicarlos por 50, obtendremos un rango de entre -50.0 y 50.0:

```
for (float angle = 0; angle < TWO_PI; angle += PI/24.0) {
    println(sin(angle) * 50.0);
}
```

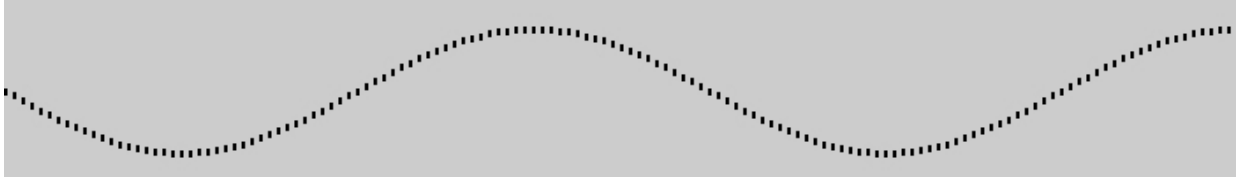
Para convertir valores de seno en números positivos, hay que sumar el valor 1.0 para que el rango pase a ser de 0.0 a 2.0. Luego, al dividir esos valores por 2.0 obtendremos un rango entre 0.0 y 1.0. La función de `map()` también puede utilizarse para simplificar el trabajo. En el siguiente ejemplo se ponen los valores de seno en un rango entre 0 y 1000:

```

for (float angle = 0; angle < TWO_PI; angle += PI/24.0) {
    float newValue = map(sin(angle), -1, 1, 0, 1000);
    println(newValue);
}

```

Si utilizáramos estos valores seno en una coordenada-y, para dibujar una serie de puntos, obtendremos como resultado una onda.

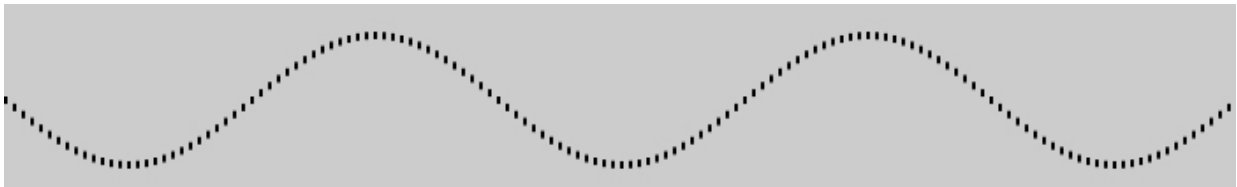


```

size(700, 100);
noStroke();
fill(0);
float angle = 0.0;
for (int x = 0; x <= width; x += 5) {
    float y = 50 + (sin(angle) * 35.0);
    rect(x, y, 2, 4);
    angle += PI/40.0;
}

```

Al reemplazar algunos números con variables, permiten controlar la forma de onda de una forma más exacta. La variable *offset* controla la coordenada-y de la onda, la variable *scaleVal* controla el ancho de a onda y la variable *angleInc* controla la velocidad con la cual aumenta el ángulo de la onda, creando una onda de mayor o menor frecuencia.

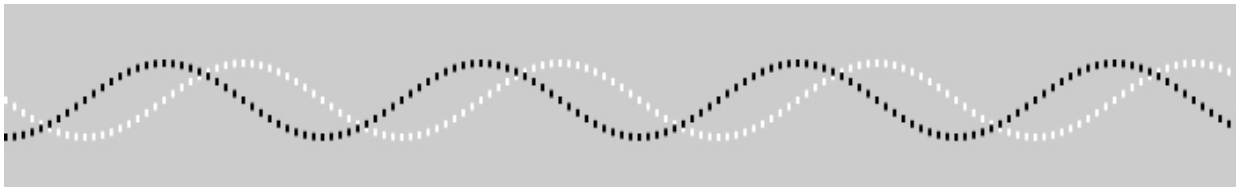


```

size(700, 100);
noStroke();
smooth();
fill(0);
float offset = 50.0; //Compensación en Y
float scaleVal = 35.0; //Escala el valor de la magnitud de
Onda
float angleInc = PI/28.0; //Incremento entre ángulos
float angle = 0.0; //El ángulo recibe valores desde 0
for (int x = 0; x <= width; x += 5) {
    float y = offset + (sin(angle) * scaleVal);
    rect(x, y, 2, 4);
    angle += angleInc;
}

```

En el caso de la función `cos()`, regresa el coseno de un ángulo. En otras palabras, son los mismos valores que en seno, con la diferencia que hay una compensación de 90°.

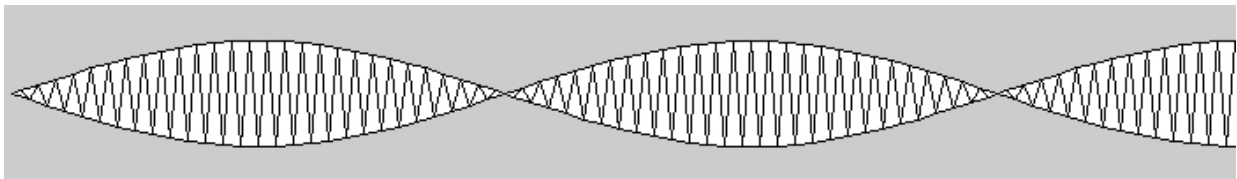


```

size(700, 100);
noStroke();
smooth();
float offset = 50.0;
float scaleVal = 20.0;
float angleInc = PI/18.0;
float angle = 0.0;
for (int x = 0; x <= width; x += 5) {
    float y = offset + (sin(angle) * scaleVal);
    fill(255);
    rect(x, y, 2, 4);
    y = offset + (cos(angle) * scaleVal);
    fill(0);
    rect(x, y, 2, 4);
    angle += angleInc;
}

```

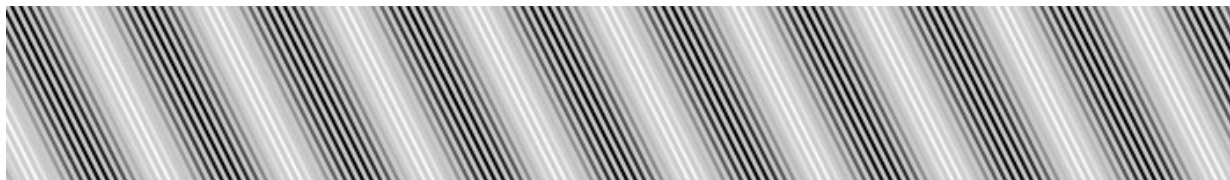
Los siguientes ejemplos muestran una manera de generar formas utilizando la función `sin()`.



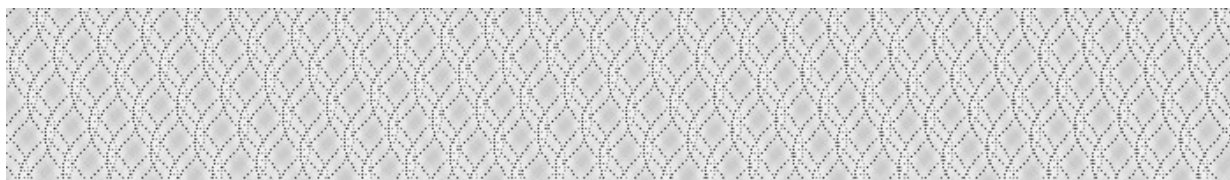
```

size(700, 100);
float offset = 50;
float scaleVal = 30.0;
float angleInc = PI/56.0;
float angle = 0.0;
beginShape(TRIANGLE_STRIP);
for (int x = 4 ; x <= width+5; x += 5) {
    float y = sin(angle) * scaleVal;
    if ((x % 2) == 0) { //Genera un loop
        vertex(x, offset + y);
    } else {
        vertex(x, offset - y);
    }
    angle += angleInc;
}
endShape();

```

```
size(700, 100);
smooth();
strokeWeight(2);
float offset = 126.0;
float scaleVal = 126.0;
float angleInc = 0.42;
float angle = 0.0;
for (int x = -52; x <= width; x += 5) {
    float y = offset + (sin(angle) * scaleVal);
    stroke(y);
    line(x, 0, x+50, height);
    angle += angleInc;
}
```



```
size(700, 100);
smooth();
fill(255, 20);
float scaleVal = 18.0;
float angleInc = PI/28.0;
float angle = 0.0;
for (int offset = -10; offset < width+10; offset += 5) {
    for (int y = 0; y <= height; y += 2) {
        float x = offset + (sin(angle) * scaleVal);
        noStroke();
        ellipse(x, y, 10, 10);
        stroke(0);
        point(x, y);
        angle += angleInc;
    }
    angle += PI;
}
```

-Círculos, Arcos y Espirales

Los círculos pueden ser dibujados directamente desde ondas seno y coseno. En el siguiente ejemplo, el ángulo aumenta de a 12° hasta completar los 360° que tiene un círculo. En cada paso, la función `cos()` se utiliza para determinar la posición en x, y la función `sin()` se utiliza para la posición en y. Ya que ambas funciones devuelven tan solo valores entre -1.0 y 1.0, el resultado es multiplicado por la variable *radio*, la cual vale 38. Por último, al agregar 50 a cada valor, el círculo se mantendrá en la posición (50,50).



```
noStroke();
smooth();
int radio = 38;
for (int deg = 0; deg < 360; deg += 12) {
    float angle = radians(deg);
    float x = 50 + (cos(angle) * radio);
```

```

        float y = 50 + (sin(angle) * radio);
        ellipse(x, y, 6, 6);
    }

```

Si el ángulo al incrementar no finaliza en los 360°, entonces puede formarse un simple arco:



```

noStroke();
smooth();
int radius = 38;
for (int deg = 0; deg < 220; deg += 12) {
    float angle = radians(deg);
    float x = 50 + (cos(angle) * radius);
    float y = 50 + (sin(angle) * radius);
    ellipse(x, y, 6, 6);
}

```

Para simplificar el trabajo con arcos, Processing incluye la función `arc()`.

`arc(x, y, ancho, alto, iniciar, parar)`

Los arcos son dibujados a partir de una elipse invisible, cuyos parámetros están determinados por *x*, *y*, *ancho* y *alto*. Los parámetros *iniciar* y *parar* necesitan de unidades en radianes para generar la curvatura del arco. Los siguientes ejemplos muestran un sencillo uso de los mismos:



```

strokeWeight(2);
arc(50, 55, 50, 50, 0, HALF_PI);
arc(50, 55, 60, 60, HALF_PI, PI);
arc(50, 55, 70, 70, PI, TWO_PI - HALF_PI);
noFill();
arc(50, 55, 80, 80, TWO_PI - HALF_PI, TWO_PI);

```

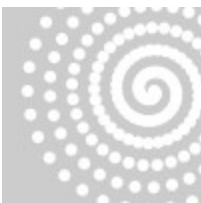


```

smooth();
noFill();
randomSeed(0);
strokeWeight(10);
stroke(0, 150);
for (int i = 0; i < 160; i += 10) {
    float begin = radians(i);
    float end = begin + HALF_PI;
    arc(67, 37, i, i, begin, end);
}

```

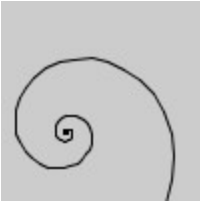
Si se desea crear espirales, simplemente se multiplica el seno y coseno por una escala de valores ascendentes o descendentes.



```

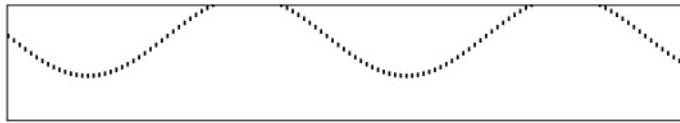
noStroke();
smooth();
float radius = 1.0;
for (int deg = 0; deg < 360*6; deg += 11) {
    float angle = radians(deg);
    float x = 75 + (cos(angle) * radius);
    float y = 42 + (sin(angle) * radius);
    ellipse(x, y, 6, 6);
    radius = radius + 0.34;
}

```

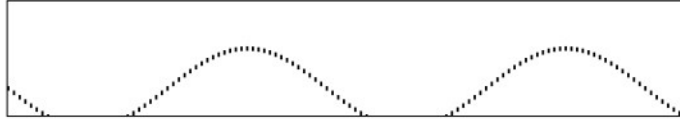


```
smooth();
float radius = 0.15;
float cx = 33; //Centro de coordenadas x e y
float cy = 66;
float px = cx; //Empieza con el centro como
float py = cy; //las coordenadas anteriores
for (int deg = 0; deg < 360*5; deg += 12) {
    float angle = radians(deg);
    float x = cx + (cos(angle) * radius);
    float y = cy + (sin(angle) * radius);
    line(px, py, x, y);
    radius = radius * 1.05;
    px = x;
    py = y;
}
```

Apéndice de Ángulos Matemáticos



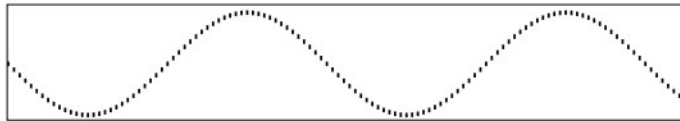
offset = 25



offset = 75



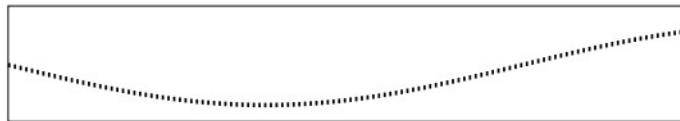
scaleVal = 5.0



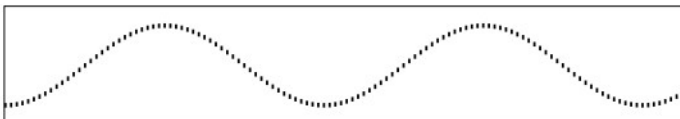
scaleVal = 45.0



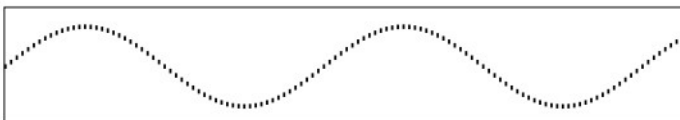
angleInc = $\pi/12.0$



angleInc = $\pi/90.0$



angle = HALF_PI



angle = PI

Matemáticas: Aleatoriedad

Elementos que se introducen en esta Unidad:

random(), randomSeed(), noise(), noiseSeed()

Lo aleatorio tiene una fundamental importancia en nuestra historia, especialmente en el arte moderno. Con acciones como arrastrar objetos, romperlos, y demás, los artistas generan una pieza con cierto nivel de valor aleatorio. Por ejemplo, una técnica muy usada de este estilo es llenar pequeñas bolsas con pintura y ubicarlas en un lienzo. Posteriormente, se lanzan dardos que rompen estas bolsas y dejan caer la pintura de una forma completamente inesperada, y con resultados de texturas muy interesantes. Es así, como se presenta un contraste muy elevado entre una estructura rígida y el completo caos.

-Valores Inesperados

La función `random()` permite devolver un valor completamente aleatorio e inesperado de un rango especificado por parámetros:

```
random(alto)
random(bajo, alto)
```

La función regresa un valor aleatorio decimal (`float`) desde el 0 hasta el parámetro *alto*. Otra forma de ingresar parámetros a la función es a través de dos parámetros en lugar de uno solo. El valor *bajo* será el primer valor del rango, y el valor *alto* será el último. Eso significa que si ingresamos el valor 3 primero, y luego el valor 5, nos devolverá un valor aleatorio decimal entre, por supuesto, 3 y 5. Además, acepta perfectamente valores negativos.

El valor siempre será del tipo `float`. Si se deseara obtener un valor aleatorio pero entero (`int`), deberemos recurrir a la función `int()` para convertirlo.

```
float f = random(5.2);           //Asigna a f un valor decimal entre 0 y 5.2
int i = random(5.2);            //ERROR! No se puede asignar un valor aleatorio a
                                //una variable int
int j = int(random(5.2));       //Asigna a j un valor entero entre 0 to 5
```

Ya que, como queda dicho, los valores que devuelve valores impredecibles, cada vez que el programa se ejecuta obtendremos diferentes resultados. Este número puede ser utilizado para controlar algún aspecto del programa.



```
smooth();
strokeWeight(10);
stroke(0, 130);
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
```

Si utilizamos la versión de `random()` que acepta dos parámetros, tendremos más control sobre algunas propiedades.



```
smooth();
strokeWeight(20);
stroke(0, 230);
float r = random(5, 45);
stroke(r * 5.6, 230);
```

```

line(0, r, 100, random(55, 95));
r = random(5, 45);
stroke(r * 5.6, 230);
line(0, r, 100, random(55, 95));
r = random(5, 45);
stroke(r * 5.6, 230);
line(0, r, 100, random(55, 95));

```

Podemos, además, crear combinaciones. Por ejemplo, podemos utilizar un ciclo FOR para crear texturas muy interesantes y sacar mayor provecho a la aleatoriedad.

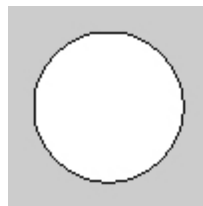
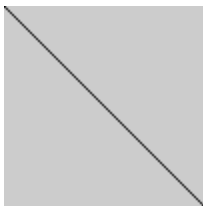


```

background(0);
smooth();
stroke(255, 60);
for (int i = 0; i < 100; i++) {
  float r = random(10);
  strokeWeight(r);
  float offset = r * 5.0;
  line(i-20, 100, i+offset, 0);
}

```

Los valores aleatorios determinan el flujo de un programa. Por lo tanto, podemos implementarlos junto con alguna expresión relacional, por ejemplo un IF. En el siguiente ejemplo, se evalúa si el valor es mayor o menor que 50, y dibuja una figura diferente dependiendo el caso:

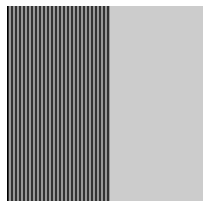
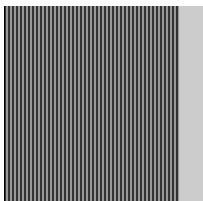


```

float r = random(100);
if (r < 50.0) {
  line(0, 0, 100, 100);
} else {
  ellipse(50, 50, 75, 75);
}

```

También podemos utilizarlo para limitar la frecuencia con la que se dibuja alguna figura:



```

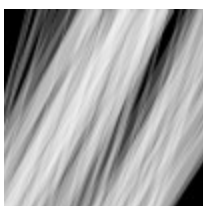
int num = int(random(50)) + 1;
for (int i = 0; i < num; i++) {
  line(i * 2, 0, i * 2, 100);
}

```

A veces deseamos conseguir un valor aleatorio, pero queremos forzar a que siempre sea el mismo. Para eso se utiliza la función `randomSeed()`. Esta es la manera en la que produciremos dichos números:

```
randomSeed(valor)
```

El parámetro *valor* tiene que ser del tipo entero (`int`). Utilizar el mismo número producirá siempre el mismo orden de valores. Es decir, mantendremos un valor aleatorio, pero forzaremos a que siempre se trate del mismo valor.



```

int s = 6; //Valor del Seed
background(0);
smooth();
stroke(255, 60);
randomSeed(s); //Produce el mismo número cada vez
for (int i = 0; i < 100; i++) {

```

```

float r = random(10);
strokeWeight(r);
float offset = r * 5;
line(i-20, 100, i+offset, 0);
}

```

-Noise

La función `noise()` nos permite tener un amplio control en los valores aleatorios que creamos. Principalmente, se utiliza para crear movimientos orgánicos. Hay tres versiones de la función, acepta un parámetro, dos parámetros e incluso tres parámetros:

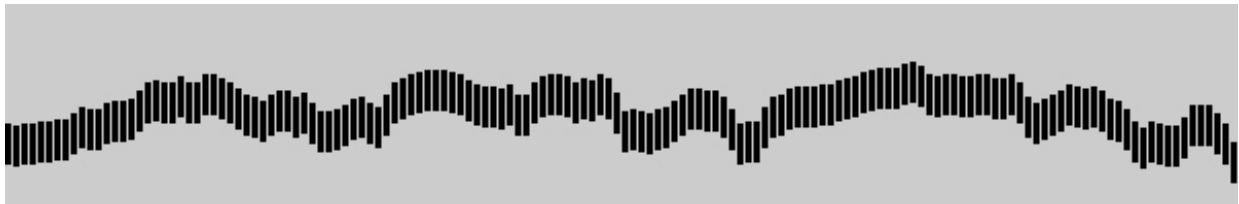
```

noise(x)
noise(x, y)
noise(x, y, z)

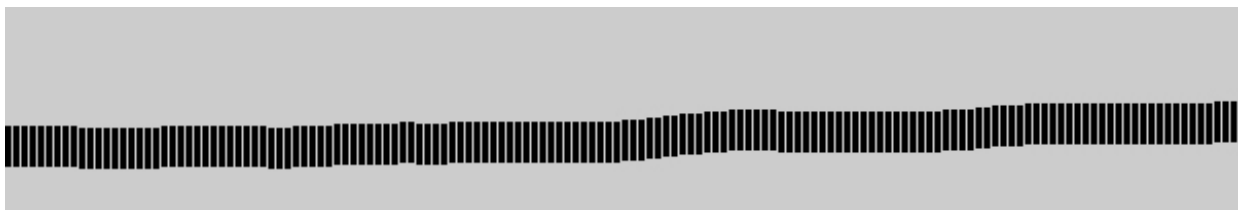
```

La versión de un solo parámetro permite crear una simple secuencia de números aleatorios. Por ende, más cantidad de parámetros permiten crear mas dimensiones de aleatoriedad. Dos parámetros pueden ser usados para crear texturas en dos dimensiones, mientras que tres parámetros pueden utilizarse para crear figuras en tres dimensiones o movimientos de animación variados. Independientemente del valor ingresado, siempre se devuelve un valor entre 0.0 y 1.0. Si se desea otra clase de valor, se puede recurrir a alguna de las operaciones aritméticas ya vistas.

Los números generados por esta función, pueden ser utilizados para producir leves cambios dentro de un mismo rango. Por ejemplo, movimientos muy cortos pero aleatorios de la misma figura (que sumados a valores seno y/o coseno producirán una variedad de movimientos orgánicos). Por lo general, los mejores valores para trabajar son los que van de entre 0.005 y 0.3, pero depende de que es lo que se quiera conseguir. Además, posee en adición la función `noiseSeed()` la cual trabaja exactamente igual que `randomSeed()`. En el siguiente ejemplo se incorporará, también, una variable *inc*, la cual se utiliza para controlar las diferencias entre valores aleatorios:



`inc = 0.1`



`inc = 0.01`

```

size(600, 100);
float v = 0.0;
float inc = 0.1;
noStroke();
fill(0);
noiseSeed(0);
for (int i = 0; i < width; i = i+4) {
    float n = noise(v) * 70.0;
    rect(i, 10 + n, 3, 20);
}

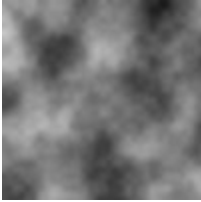
```

```

    v = v + inc;
}

```

Si se utiliza el segundo parámetro, se nos da la posibilidad de crear texturas en dos dimensiones. En el siguiente ejemplo, se utiliza junto con variaciones de la variable *inc* y un ciclo FOR pueden utilizarse para crear sistemas muy interesantes:

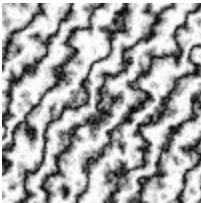


```

float xnoise = 0.0;
float ynoise = 0.0;
float inc = 0.04;
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        float gray = noise(xnoise, ynoise) * 255;
        stroke(gray);
        point(x, y);
        xnoise = xnoise + inc;
    }
    xnoise = 0;
    ynoise = ynoise + inc;
}

```

Ahora bien, las combinaciones con las estructuras ya vistas y el efecto de la aleatoriedad del tipo *noise* son ilimitadas y de mucha variedad. Un efecto interesante es combinarlo con un valor de seno para generar una textura con turbulencia.



```

float power = 3; //Intensidad de la turbulencia
float d = 8; //Densidad de la turbulencia
noStroke();
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        float total = 0.0;
        for (float i = d; i >= 1; i = i/2.0) {
            total += noise(x/d, y/d) * d;
        }
        float turbulence = 128.0 * total / d;
        float base = (x * 0.2) + (y * 0.12);
        float offset = base + (power * turbulence /
            256.0);
        float gray = abs(sin(offset)) * 256.0;
        stroke(gray);
        point(x, y);
    }
}

```


Transformaciones: Matrices y Traslaciones

Elementos que se introducen en esta Unidad:

`translate()`, `pushMatrix()`, `popMatrix()`

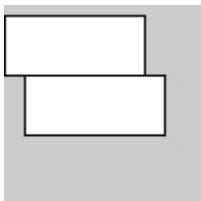
El sistema de coordenadas introducido en la unidad de Formas utiliza la esquina superior izquierda para visualizar en la ventana de representación, con un origen en un sistema de coordenadas X e Y. Dicho sistema puede ser modificado con transformaciones. Las coordenadas pueden, entonces, ser trasladadas, rotadas y escaladas, de esta forma las figuras variarán su posición, rotación y tamaño.

-Traslación

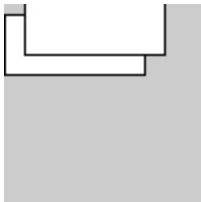
La función `translate()` mueve el origen de la esquina superior izquierda (la cual por defecto se encuentra en el 0, 0) a otra localización. Acepta dos parámetros. El primero es la coordenada en x y el segundo en y.

`translate(x, y)`

Los valores de x y de y que se ingresan afectan a cualquier figura o forma que se pretenda dibujar después. Si 10 es el parámetro de x, y 30 es el parámetro de y, un punto dibujado en la posición (0, 5) se mostrará en la posición (10, 35).

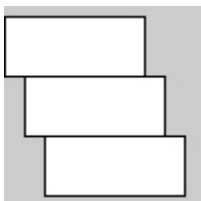


```
//El mismo rectángulo es dibujado, pero solo el segundo es
//afectado por la función translate, ya que este es dibujado
//luego de su declaración
rect(0, 5, 70, 30);
translate(10, 30);    //Mueve 10 pixeles a la derecha y 30
rect(0, 5, 70, 30);    //abajo
```



```
//Si se utilizara un número negativo,
//la posición se mueve de manera opuesta
rect(0, 5, 70, 30);
translate(10, -10);    //Mueve 10 pixeles a la derecha y
rect(0, 5, 70, 30);    //otros 10 arriba
```

Hay que tener en cuenta que la función `translate()` es aditiva. Si `translate(10, 30)` se ejecuta dos veces, entonces la posición de base será (20, 60).

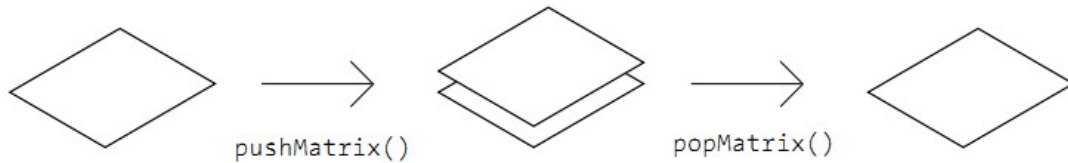


```
rect(0, 5, 70, 30);
translate(10, 30); //Mueve 10 pixeles a la derecha y 30 abajo
rect(0, 5, 70, 30)
translate(10, 30); //Repite traslación, por lo tanto
rect(0, 5, 70, 30); //mueve 20 pixeles a la derecha y 60 abajo
```

-Controlando Transformaciones

La transformación por matrix es un conjunto de números que define como, en términos geométricos, la figura o forma es dibujada en pantalla. Transformaciones y funciones como `translate()`, alteran los números en esta matrix, provocando que se dibuje diferente. En el ejemplo anterior, hablamos de como estas transformaciones poseían la característica de acumularse entre sí. Es así que la función `pushMatrix()` nos sirve para guardar el estado de todas las transformaciones en ese momento, así el programa puede devolverlo más tarde. Para regresar al estado anterior, se llama a la función `popMatrix()`.

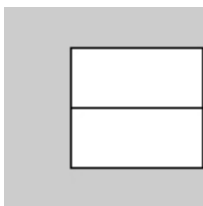
Vamos a pensar, entonces, a una matrix como una hoja de papel que lleva escrito en su superficie una lista con las transformaciones que deseamos (trasladar, rotar, escalar). Cuando una función `translate()` se ejecuta, esta se agrega al papel. Para guardar la actual matrix, para utilizarla luego, agregamos una nueva hoja de papel por encima de la pila y copiamos la información. Cualquier cambio que generemos en la hoja de arriba, preservará, sin embargo, los números originales que están debajo. Para volver al sistema anterior de coordenadas, simplemente retiramos la hoja que quedó encima (en la cual realizamos los cambios).



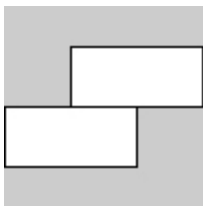
Sirve pensarlo, para quienes manejan programas al estilo del Adobe Photoshop o GIMP, como que la matrix es tan solo una capa. A dicha capa podemos darle características diversas (trasladar, rotar, escalar).

Es esencial como se ingresan dichas coordenadas en una matrix, pero se trata tan solo de una terminología mas técnica y mecánica. La función `pushMatrix()` se utiliza para añadir una nueva coordenada al grupo, mientras que `popMatrix()` se utiliza para remover dicho grupo. Por ende, la función `pushMatrix()` no puede ser utilizada sin `popMatrix()` y viceversa.

Para esto, compararemos los siguientes ejemplos. Ambos dibujan dos rectángulos, pero con diferentes resultados. En el segundo ejemplo uno de los rectángulos se encuentra en una matrix asociada a la función `translate()`.

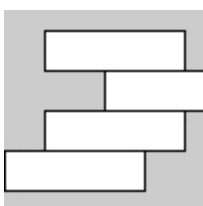


```
translate(33, 0); //Mueve 33 pixeles a la derecha
rect(0, 20, 66, 30);
rect(0, 50, 66, 30);
```



```
pushMatrix();
translate(33, 0); //Mueve 33 pixeles a la derecha
rect(0, 20, 66, 30);
popMatrix(); //Remueve el movimiento
//La siguiente forma no es afectada por translate(), ya que la
//transformación está escrita entre pushMatrix() y popMatrix()
rect(0, 50, 66, 30);
```

Al utilizar una mayor cantidad de matrices, uno puede tener más control de todo lo que pasa en el programa. En el siguiente ejemplo, el primer rectángulo es afectado por la primer traslación, el segundo por la primera y por la segunda, el tercero solo por la primera, ya que la segunda traslación está asociada a una matrix específica.



```
pushMatrix();
translate(20, 0);
rect(0, 10, 70, 20); //Dibuja en (20, 30)
pushMatrix();
translate(30, 0);
rect(0, 30, 70, 20); //Dibuja en (50, 30)
popMatrix();
```

```
rect(0, 50, 70, 20);          //Dibuja en (20, 50)
popMatrix();
rect(0, 70, 70, 20);          //Dibuja en (0, 70)
```

Las transformaciones restantes son incluidas en la siguiente unidad (Unidad 17).

Unidad 17

Transformaciones: Rotación y Escala

Elementos que se introducen en esta Unidad:

`rotate()`, `scale()`

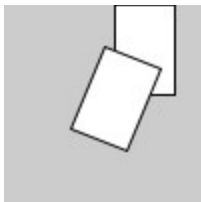
Las funciones de transformación son poderosas maneras para cambiar la geometría de un sector específico del programa. Ya sea por el uso de tan solo una, o por la combinación de varias, se requiere entender como trabaja cada una por separado. El orden en que las transformaciones son escritas, afecta de forma muy notoria el comportamiento del programa.

-Rotando y Escalando

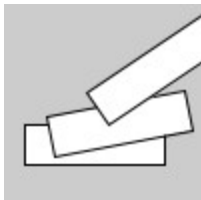
La función `rotate()` permite rotar las coordenadas de un sistema, de modo tal que las figuras y formas sean dibujadas con cierto ángulo. Recibe tan solo un parámetro, el cual modifica la rotación de acuerdo a un ángulo:

```
rotate(angulo)
```

El parámetro *angulo* asume que el valor se encuentra en radianes. Las figuras son siempre rotadas en relación a la posición de origen (0, 0). Como en todas las transformaciones, el efecto de rotación también es acumulativo. Si hay una rotación de $\pi/4$ radianes y otra de $\pi/4$ radianes, entonces la figura que se dibuje se rotará en $\pi/2$ radianes.



```
smooth();  
rect(55, 0, 30, 45);  
rotate(PI/8);  
rect(55, 0, 30, 45);
```



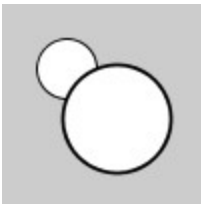
```
smooth();  
rect(10, 60, 70, 20);  
rotate(-PI/16);  
rect(10, 60, 70, 20);  
rotate(-PI/8);  
rect(10, 60, 70, 20);
```

Estos ejemplos dejan en claro que la rotación a base de la posición de origen tiene sus limitaciones. Mas adelante se explicará como combinar transformaciones.

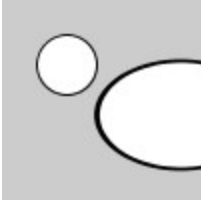
Por otro lado, la función `scale()` permite controlar las coordenadas del sistemas para magnificarlas y dibujar figuras de mayor escala. Esta soporta uno o dos parámetros:

```
scale(tamaño)  
scale(xtamaño, ytamaño)
```

La versión de un solo parámetro permite escalar la figura en todas sus dimensiones, mientras que la de dos parámetros abarca la escala en el eje x y en el eje y de manera individual. Los parámetros a ingresar se expresan como valores decimales con una relación de porcentajes. Por ejemplo, 2.0 hace alusión a 200%, 1.5 a 150%, y 0.5 a 50%.

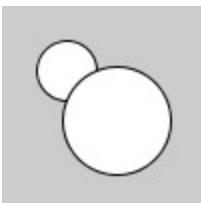


```
smooth();  
ellipse(32, 32, 30, 30);  
scale(1.8);  
ellipse(32, 32, 30, 30);
```



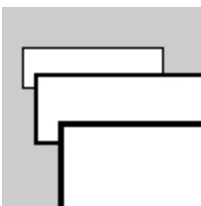
```
smooth();  
ellipse(32, 32, 30, 30);  
scale(2.8, 1.8);  
ellipse(32, 32, 30, 30);
```

En los ejemplos anteriores se ve que el contorno de la figura es afectado por la escala. Para solucionar este inconveniente puede usarse algo similar al siguiente código:



```
float s = 1.8;  
smooth();  
ellipse(32, 32, 30, 30);  
scale(s);  
strokeWeight(1.0 / s);  
ellipse(32, 32, 30, 30);
```

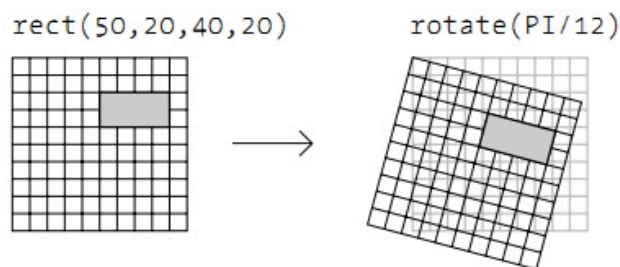
Y de la misma forma que `translate()` y `rotate()`, en `scale()` los valores también son acumulativos:



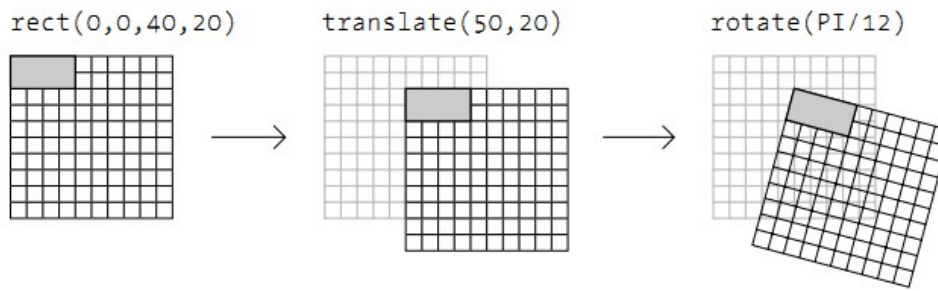
```
rect(10, 20, 70, 20);  
scale(1.7);  
rect(10, 20, 70, 20);  
scale(1.7);  
rect(10, 20, 70, 20);
```

-Combinando Transformaciones

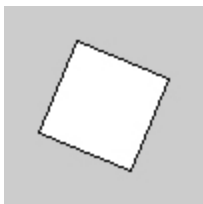
Cuando una figura es afectada por `translate()`, `rotate()` o `scale()`, siempre todo es en relación a la coordenada de origen. Por ejemplo, al rotar un rectángulo en la coordenada (50, 20), la rotación se hará sobre ese mismo eje:



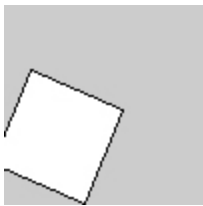
Para hacer que la figura rote sobre su propia esquina, se debe cambiar la coordenada a la posición (0, 0). Para esto, utilizaremos la función `translate()`. Cuando la figura rote, lo hará sobre su esquina superior izquierda.



Hay dos maneras de pensar las transformaciones. Un método es utilizar las funciones para transformar, y luego corregir el problema de posición con las coordenadas propias de la figura. El otro método consiste en utilizar las propias funciones de transformación para corregir lo no deseado. Utilizando este método hay que tener cuidado en el orden que se ubica cada función. En el siguiente ejemplo se muestra el mismo código dos veces pero con las funciones invertidas:

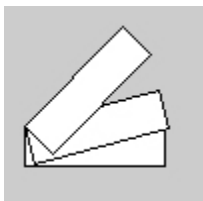


```
translate(width/2, height/2);
rotate(PI/8);
rect(-25, -25, 50, 50);
```

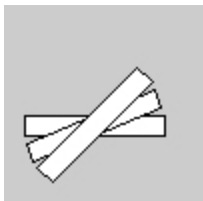


```
rotate(PI/8);
translate(width/2, height/2);
rect(-25, -25, 50, 50);
```

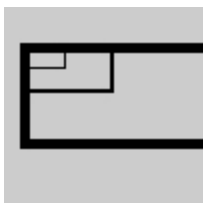
Los siguientes ejemplos demuestran el potencial de combinar funciones de transformación, pero también queda en claro que estas acciones requieren un mayor planeamiento.



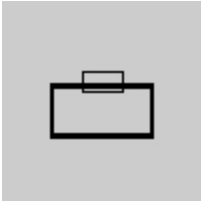
```
translate(10, 60);
rect(0, 0, 70, 20);
rotate(-PI/12);
rect(0, 0, 70, 20);
rotate(-PI/6);
rect(0, 0, 70, 20);
```



```
translate(45, 60);
rect(-35, -5, 70, 10);
rotate(-PI/8);
rect(-35, -5, 70, 10);
rotate(-PI/8);
rect(-35, -5, 70, 10);
```



```
noFill();
translate(10, 20);
rect(0, 0, 20, 10);
scale(2.2);
rect(0, 0, 20, 10);
scale(2.2);
rect(0, 0, 20, 10);
```



```
noFill();
translate(50, 30);
rect(-10, 5, 20, 10);
scale(2.5);
rect(-10, 5, 20, 10);
```

La propiedad acumulativa de las funciones de transformación puede ser muy útil para generar efectos muy interesantes con una estructura FOR.



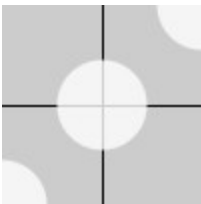
```
background(0);
smooth();
stroke(255, 120);
translate(66, 33); //Configura posición inicial
for (int i = 0; i < 18; i++) { //18 repeticiones
  strokeWeight(i); //Incrementa grosor del contorno
  rotate(PI/12); //Acumula rotación
  line(0, 0, 55, 0);
}
```



```
background(0);
smooth();
noStroke();
fill(255, 48);
translate(33, 66); //Configura posición inicial
for (int i = 0; i < 12; i++) { //12 repeticiones
  scale(1.2); //Acumula escala
  ellipse(4, 2, 20, 20);
}
```

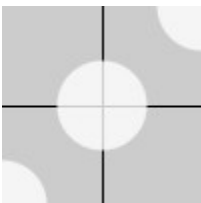
-Nuevas Coordenadas

La posición de origen por defecto se encuentra en el (0, 0), esquina superior izquierda, y hace referencia a una posición de los píxeles. Las funciones de transformación pueden alterar este sistema de coordenadas, en base a algún efecto que deseamos producir.



```
//Mueve el origen del (0,0) al centro
size(100, 100);
translate(width/2, height/2);
line(-width/2, 0, width/2, 0); //Dibuja eje X
line(0, -height/2, 0, height/2); //Dibuja eje Y
smooth();
noStroke();
fill(255, 204);
ellipse(0, 0, 45, 45); //Dibuja en el origen
ellipse(-width/2, height/2, 45, 45);
ellipse(width/2, -height/2, 45, 45);
```

Al combinar las funciones de translate() y scale(), puede cambiarse un rango de valores.



```
//Mueve el origen del (0,0) al centro
//y cambia el tamaño del sistema de coordenadas
size(100, 100);
scale(width/2, height/2);
translate(1.0, 1.0);
strokeWeight(1.0/width);
line(-1, 0, 1, 0); //Dibuja eje X
```

```

line(0, -1, 0, 1);           //Dibuja eje X
smooth();
noStroke();
fill(255, 204);
ellipse(0, 0, 0.9, 0.9);    //Dibuja en el origen
ellipse(-1.0, 1.0, 0.9, 0.9);
ellipse(1.0, -1.0, 0.9, 0.9);

```

Además, puede usarse para cambiar el sistema de coordenadas a la esquina inferior izquierda. Este es el sistema de coordenadas que usan el Adobe Illustrator y el PostScript.



```

//Mueve el origen del (0,0) a la esquina inferior izquierda
size(100, 100);
translate(0, height);
scale(1.0, -1.0);
line(0, 1, width, 1);      //Dibuja eje X
line(0, 1, 0, height );   //Dibuja eje Y
smooth();
noStroke();
fill(255, 204);
ellipse(0, 0, 45, 45);    //Dibuja en el origen
ellipse(width/2, height/2, 45, 45);
ellipse(width, height, 45, 45);

```


Unidad 18

Estructuras: Continuidad

Elementos que se introducen en esta Unidad:

`draw()`, `frameRate()`, `frameCount`, `setup()`, `noLoop()`

Todos los programas, mostrados en las unidades anteriores, ejecutan el código y, posteriormente, lo detienen. Los programas con animaciones o que modifican su contenido en vivo, requieren estar ejecutándose continuamente. Dichos programas, se ejecutan de manera continua y permiten, por lo tanto, animaciones o bien conseguir datos a través de dispositivos de entrada.

-Evaluación Continua

Un programa que estará ejecutándose continuamente requiere de la función `draw()`. El código dentro de un bloque `draw()` es ejecutado en un continuo loop (repetición) hasta que se detenga de ejecutar el programa o se cierre la ventana de representación. Un programa puede tener **solo un** bloque `draw()`. Cada vez que el código dentro del bloque `draw()` es terminado de leer, lo muestra en la ventana de representación y vuelve a leer desde la primer línea (siempre dentro del bloque `draw()`).

Por defecto, las imágenes son dibujadas a 60 cuadros por segundo (fps). La función `frameRate()` nos permite cambiar la cantidad de esos cuadros por segundo. El programa se ejecutará a la velocidad que se le dé al `frameRate()`, aún así, muchos programadores ambiciosos suelen exceder la cantidad de cuadros por segundos en relación al poder de su ordenador. Se recomienda no exceder los 60 cuadros que ya vienen por defecto.

La variable `frameCount` siempre contiene el número de cuadros que se **está ejecutando** desde que el programa fue iniciado. Un programa que posee un bloque `draw()` siempre variará el número de cuadros (1, 2, 3, 4, 5, 6...) hasta que el programa sea detenido, la computadora agote su capacidad de procesamiento o se pierda la fuente de tensión que la alimenta.

```
//Imprime cada número de cuadro en la consola
void draw() {
    println(frameCount);
}

//Se ejecuta a 4 cuadros por segundo, imprime cada cuadro en la consola
void draw() {
    frameRate(4);
    println(frameCount);
}
```

Con cambiar algún parámetro dentro de la estructura `draw()`, produciremos un simple efecto de movimiento. Por ejemplo, a través de una variable, cambiar la posición en la que se encuentra una línea.



```
float y = 0.0;
void draw() {
    frameRate(30);
    line(0, y, 100, y);
    y = y + 0.5;
}
```

Cuando este código es ejecutado, las variables son reemplazadas con los valores actuales y vuelve a correr las acciones en este orden:

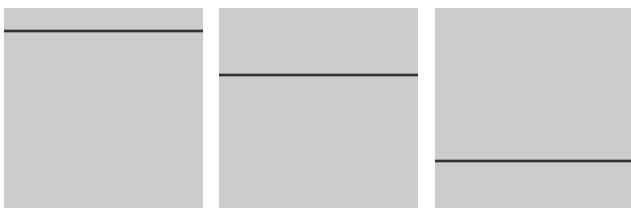
```

float y = 0.0
frameRate(30)
----- Entra al draw()
line(0, 0.0, 100, 0.0)
y = 0.5
frameRate(30)
----- Entra al draw() por
segunda vez
line(0, 0.5, 100, 0.5)
y = 1.0
frameRate(30)
----- Entra al draw() por
tercera vez
line(0, 1.0, 100, 1.0)
y = 1.5
Etc...

```

La variable de control mostrada anteriormente (llamada *y*) debe ser declarada fuera del bloque `draw()`. De otro modo, cada vez que el bloque se actualice volverá a crear la variable, es decir, es como si es valor se *resteará*.

Como se puede comprobar anteriormente, el fondo del programa no se actualiza automáticamente con cada vuelta del bloque `draw()`. Esto produce un efecto de barrido. Sin embargo, ¿qué pasa si lo que queremos es solo una línea moviéndose, en lugar de una especie de barrido?. Recurrirémos a declararlo como **una de las primeras líneas** dentro del `draw()` y con un color en específico. La solución se piensa como si se estuviese dibujando cada cuadro a mano. Si queremos que la posición antigua de una figura desaparezca, volvemos a pintar el fondo, este pintará la figura y, posteriormente, se dibujará la figura en la posición actualizada.




```

float y = 0.0;
void draw() {
    frameRate(30);
    background(204);
    y = y + 0.5;
    line(0, y, 100, y);
}

```

La variable que controla la posición de la línea puede ser usada para otros propósitos. Por ejemplo, alterar el color del fondo:

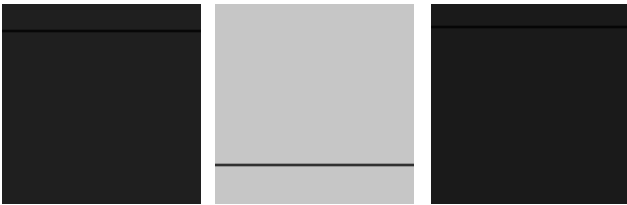


```

float y = 0.0;
void draw() {
    frameRate(30);
    background(y * 2.5);
    y = y + 0.5;
    line(0, y, 100, y);
}

```

Pasado unos pocos segundos del programa, la línea desaparece por debajo. Con una simple estructura `IF` solucionamos ese problema:



```
float y = 0.0;
void draw() {
  frameRate(30);
  background(y * 2.5);
  y = y + 0.5;
  line(0, y, 100, y);

  if (y > 100) {
    y = 0;
  }
}
```

```
int y = 0;
```

Labels: Tipo de Dato (int), Nombre de Variable (y), Operador de Asignación (=), Terminador de Acción (;)

```
void setup() {
  size(300, 300);
}
```

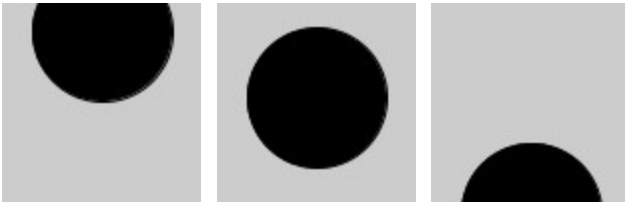
Label: El área entre { } es un bloque

```
void draw() {
  line(0, y, 300, y);
  y = y + 4;
}
```

Labels: Regresa el valor (void), Función (draw), Parámetros (0, y, 300, y), Expresión (y + 4)

-Controlando el Flujo

Hay funciones que, incluso en programas dinámicos, solo requieren ser ejecutadas una vez. El bloque `setup()` permite que cualquier función que se escriba dentro de él se ejecute solo una vez. Muy útil para funciones como `size()` o `loadImage()`, que suelen ser ejecutadas únicamente la primera vez. El bloque `setup()` se escribe siempre antes que el bloque `draw()` y, al igual que con el bloque `draw()`, puede haber **solo un** bloque `setup()` por programa. Cuando un programa es ejecutado, primero se lee lo que está fuera de los bloques `setup()` y `draw()`, luego se lee todo lo que está dentro del `setup()` y, finalmente, lo que está dentro del `draw()`. En el siguiente ejemplo, el tamaño de la ventana, el filtro de suavizado y el relleno no cambiarán, así que se incluirán en el bloque `setup()`.



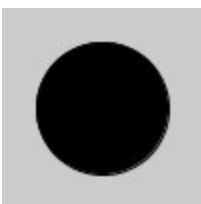
```
float y = 0.0;
void setup() {
  size(100, 100);
  smooth();
  fill(0);
}

void draw() {
  background(204);
  ellipse(50, y, 70, 70);
  y += 0.5;
  if (y > 150) {
    y = -50.0;
  }
}
```

Cuando este código es ejecutado, las variables son reemplazadas con los valores actuales y vuelve a correr las acciones en este orden:

```
float y = 0.0
size(100, 100)
----- Entra al setup()
smooth()
fill(0)
background(204)
----- Entra al draw()
ellipse(50, 0.0, 70, 70)
y = 0.5
background(204)
----- Entra al draw() por segunda vez
ellipse(50, 0.5, 70, 70)
y = 1.0
background(204)
----- Entra al draw() por tercera vez
ellipse(50, 1.0, 70, 70)
y = 1.5
Etc....
```

Cuando el valor de `y` es mayor a 150, el código en la estructura IF convierte el valor a -50. La variable que cambia con cada repetición del bloque `draw()` debe ser declarada **fuera** de los bloques `setup()` y `draw()`. Si está es declarada dentro del `draw()`, se estará re-creando con cada repetición del mismo, por lo tanto su valor nunca cambiaría. Por otro lado, fuera de los bloques `setup()` y `draw()` solo pueden declararse y asignarse variables. Si se declarara una función, causará un error. Si un programa solo dibujará un cuadro, este puede declararse en el `setup()`. La única diferencia entre `setup()` y `draw()` es que el `setup()` se ejecuta **solo una vez**.



```
void setup() {
  size(100, 100);
  smooth();
  fill(0);
  ellipse(50, 50, 66, 66);
}
```

Utilizando la función `noLoop()`, podemos hacer que el `draw()` deje de repetir y simplemente dibuje un cuadro. El siguiente ejemplo es similar al anterior, con la diferencia que la elipse es dibujada en el `draw()`:



```
void setup() {
    size(100, 100);
    smooth();
    fill(0);
    noLoop();
}
void draw() {
    ellipse(50, 50, 66, 66);
}
```

-Extensión de la Variable

Cuando trabajamos con los bloques `setup()` y `draw()` es necesario tomar consciencia de donde declaramos y asignamos cada variable. La localización de la variable determina su *extensión*. La regla para conocer la extensión que posee una variable es muy simple: si la variable está dentro de una estructura, esa variable es local y puede usarse solamente dentro de esa estructura; si la variable está en el programa, fuera de cualquier estructura, esa variable es global y puede usarse en cualquier estructura. Las variables declaradas en el `setup()` pueden usarse **solo** en el `setup()`. Las variables declaradas en el `draw()` pueden usarse **solo** en el `draw()`.

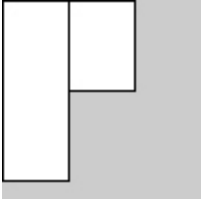
```
int d = 51; //La variable d es global, puede ser usada donde sea
void setup() {
    size(100, 100);
    int val = d * 2; //La variable local val, puede ser usada solo en el
    fill(val); //setup()
}
void draw() {
    int y = 60; //La variable local y, puede ser usada solo en el draw()
    line(0, y, d, y);
    y -= 25;
    line(0, y, d, y);
}
```

Cuando una variable es creada dentro de un bloque, al salir del bloque, esa variable es **destruida**. Esto significa que no puede utilizarse fuera de ese bloque.

```
void draw() {
    int d = 80; //Esta variable puede usarse donde sea en el draw()
    if (d > 50) {
        int x = 10; //Esta variable puede usarse solo en este bloque IF
        line(x, 40, x+d, 40);
    }
    line(0, 50, d, 50);
    line(x, 60, x+d, 60); //ERROR! No se puede leer esta variable fuera del
} //bloque

void draw() {
    for (int y = 20; y < 80; y += 6) { //La variable puede usarse
        line(20, y, 50, y); //solo en el bloque FOR
    }
    line(y, 0, y, 100); //ERROR! No se puede acceder a y fuera del FOR
}
```

La implementación de variables locales y globales permite que se pueda tener dos o más variables con el mismo nombre. Aún así, no se recomienda el trabajo con variables del mismo nombre ya que puede producir confusión en quien programa.



```
int d = 45;           //Asigna 45 a la variable d
void setup() {
  size(100, 100);
  int d = 90;        //Asigna 90 a la variable local d
  rect(0, 0, 33, d); //Usa la variable local d con
                    //valor 90
}
void draw() {
  rect(33, 0, 33, d); //Usa d con valor de 45
}
```

Una variable dentro de un bloque con el mismo nombre que una variable fuera del bloque es, comúnmente, un error muy difícil de encontrar.

Unidad 19

Estructuras: Funciones

Elementos que se introducen en esta Unidad:

void, return

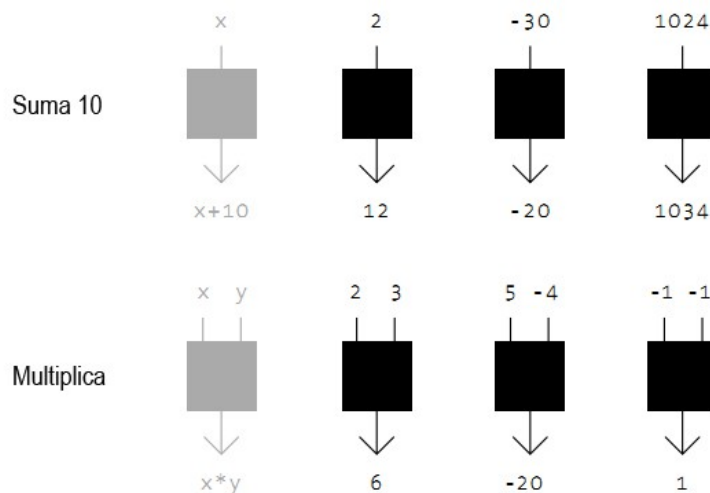
Una función contiene en sí misma un modulo de modificación de programa. Se han utilizado hasta ahora funciones que vienen incluidas en Processing, tales como `size()`, `line()`, `stroke()`, y `translate()`, para escribir los programas. Sin embargo, es posible escribir nuestras propias funciones. Las funciones hacen que el código redundante sea mas conciso al mostrarlo en forma de bloque, por extraer elementos comunes. Al mantener forma de bloque, permite que el mismo código se utilice una infinidad de veces sin necesidad de repetir las líneas. Además, mantiene una estructura mas sencilla de leer en caso de tener algún error.

Las funciones de Processing por lo general aceptan una determinada cantidad de parámetros. Por ejemplo, `line()` acepta cuatro parámetros, los cuales definen la posición de dos puntos. Al cambiar los números, se cambia la posición de la línea. También el comportamiento de la función puede depender de la cantidad de parámetros que le sean enviados. Por ejemplo, la función `fill()` al recibir un parámetro pintará en escala de grises, con dos parámetros serán grises con transparencia y con tres parámetros RGB.

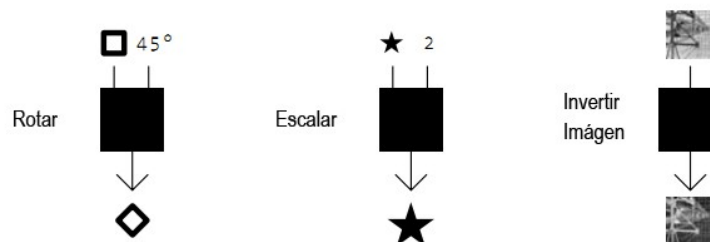
Una función puede ser imaginada como una caja con una serie de mecanismos dentro que permiten actuar sobre una serie de datos. Por lo general hay datos de entrada y código dentro de la caja, los cuales producen datos de salida:



Por ejemplo, una función puede utilizarse para sumar 10 a un número o multiplicar dos:



El ejemplo anterior es sumamente simple, pero el concepto puede extenderse a cuestiones que no sean necesariamente tan obvias:



-Abstracción

En términos de programa, la palabra abstracción tiene un significado distinto a el simple hecho de como uno puede pintar o dibujar. Se refiere a la acción de esconder detalles, enfocándose únicamente en los **resultados**. La interfaz de un automóvil permite que el transporte se mueva por el accionar de los pedales y el volante, ignorando los procesos químicos internos de los pistones y la combustión.

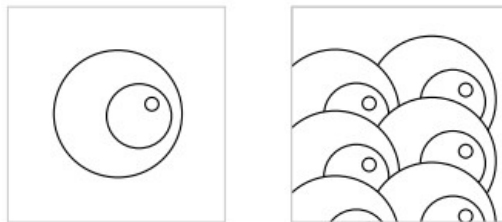
La idea de abstracción puede pensarse también desde el cuerpo humano. Por ejemplo, podemos controlar nuestra respiración, pero por lo general este proceso es involuntario. Imaginemos por un momento que tuviéramos total control de cada accionar de nuestro cuerpo. Si controláramos nuestra respiración, los latidos del corazón, las reacciones químicas que se producen en nuestro cuerpo y la retroalimentación de las neuronas, nos sería completamente imposible hacer algo tan simple como leer un libro o escribir un código para un programa. Nuestro cerebro abstrae estos aspectos para que podamos concentrarnos en otros aspectos de la vida.

De esta forma, la abstracción es fundamental a la hora de escribir un programa. En Processing contamos con abstracciones, funciones como `line()`, `ellipse()` y `fill()`, ocultan los procesos que realizan para que podemos concentrarnos en su propia implementación. Si nos interesa dibujar una línea, posiblemente nos interese su posición, tamaño, color y grosor, y no el tener una gran cantidad de código que ejecute dicho sistema.

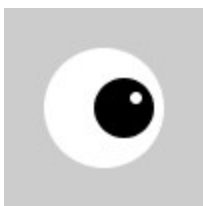
-Creando Funciones de Usuario

Antes de explicar en detalle como crear una función de usuario, explicaremos el porque alguien querría crear su propia función. El siguiente programa explica como acortar un sistema, para que sea más modular, a través de una simple función. Esto hace que el código sea más sencillo de leer, modificar, expandir.

Algo muy común es dibujar una figura en pantalla repetidas veces. Poseemos el código que permite crear la forma que se muestra a continuación, y requerimos poder generar la textura que se ve a la derecha:



Lo primero que haremos será dibujar simplemente la figura en pantalla, solo para corroborar que el código funcione:



```
void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}
void draw() {
  fill(255);
  ellipse(50, 50, 60, 60); //Círculo Blanco
  fill(0);
  ellipse(50+10, 50, 30, 30); //Círculo Negro
  fill(255);
  ellipse(50+16, 45, 6, 6); //Círculo Blanco y Pequeño
}
```

El programa anterior muestra una forma muy limpia y clara de dibujar una simple figura. Pero si deseáramos

que fuesen dos, nos veremos obligados a multiplicar el número de líneas de código. Si el código tiene 6 líneas, tendremos 12. Para diseñar nuestra textura, necesitaremos 6 figuras, por lo tanto son 36 líneas de código. Imaginemos un caso con 30 figuras, dentro del bloque `draw()` se presentarían 180 líneas de simple código.



```
void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}
void draw() {
  //Figura de la Derecha
  fill(255);
  ellipse(65, 44, 60, 60);
  fill(0);
  ellipse(75, 44, 30, 30);
  fill(255);
  ellipse(81, 39, 6, 6);
  //Figura de la Izquierda
  fill(255);
  ellipse(20, 50, 60, 60);
  fill(0);
  ellipse(30, 50, 30, 30);
  fill(255);
  ellipse(36, 45, 6, 6);
}
```

Como las figuras son idénticas, podemos utilizar una función para simplificar la tarea. Simplemente agregaremos los datos de entrada de la posición en `x` y en `y`.



```
void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}
void draw() {
  ojo(65, 44);
  ojo(20, 50);
}
void ojo(int x, int y) {
  fill(255);
  ellipse(x, y, 60, 60);
  fill(0);
  ellipse(x+10, y, 30, 30);
  fill(255);
  ellipse(x+16, y-5, 6, 6);
}
```

La función es de 8 líneas de código, pero tan solo lo escribiremos una vez. Con esta estrategia, es posible dibujar 30 ojos con 38 líneas de código.

Cada vez que la función es usada con el `draw()`, las líneas de código dentro de la función son ejecutadas. El flujo casual del programa es interrumpido por la función. Una vez terminada la ejecución del código dentro de la función, se dispone a leer lo que sigue en el bloque `draw()`:

```

----- Inicia leyendo el código del setup()
size(100, 100)
noStroke()
smooth()
noLoop()
fill(255)
----- Entra al draw(), se desvía a la función ojo
ellipse(65, 44, 60, 60)
fill(0)
ellipse(75, 44, 30, 30)
fill(255)
ellipse(81, 39, 6, 6)
fill(255)
----- Vuelve al draw(), se desvía a la función ojo por segunda vez
ellipse(20, 50, 60, 60)
fill(0)
ellipse(30, 50, 30, 30)
fill(255)
ellipse(36, 45, 6, 6)
----- Finaliza el Programa

```

Ahora que la función está trabajando, podemos crear tantas figuras como queramos. De esta forma dejaremos de preocuparnos por como la figura es dibujada, y nos concentraremos en enviar correctamente la posición a través de los dos parámetros:



```

void setup() {
    size(100, 100);
    noStroke();
    smooth();
    noLoop();
}
void draw() {
    ojo(65, 44);
    ojo(20, 50);
    ojo(65, 74);
    ojo(20, 80);
    ojo(65, 104);
    ojo(20, 110);
}
void ojo(int x, int y) {
    fill(255);
    ellipse(x, y, 60, 60);
    fill(0);
    ellipse(x+10, y, 30, 30);
    fill(255);
    ellipse(x+16, y-5, 6, 6);
}

```

Lo importante a la hora de escribir una función, es tener la idea clara de lo que se va a hacer. ¿Dibujará una figura? ¿Calculará un número? ¿Aplicará un filtro de imagen? Una vez que se piensa en que se quiere hacer, hay que pensar en los parámetros que recibirá. De este modo se construirá una función en pocos pasos. En el siguiente ejemplo, el mismo explora algunos detalles de la función antes de escribirla. Después, se construye la función, donde se agrega un parámetro cada vez que se comprueba el código:



```

void setup() {
    size(100, 100);
    smooth();
    noLoop();
}

```

```

}
void draw() {
  //Dibujar una gruesa X color gris claro
  stroke(160);
  strokeWeight(20);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
  //Dibuja una normal X color negro
  stroke(0);
  strokeWeight(10);
  line(30, 20, 90, 80);
  line(90, 20, 30, 80);
  //Dibuja una suave X color blanco
  stroke(255);
  strokeWeight(2);
  line(20, 38, 80, 98);
  line(80, 38, 20, 98);
}

```

Para conseguir una función que pueda dibujar las tres X, primero crearemos el código para que solo dibuje una. Llamaremos a la función `dibujarX()` para que sea más claro. Dentro, escribiremos el código para dibujar la X de color gris claro (el mismo que está arriba) y la llamaremos a la función desde el `draw()`. Como la función no recibe parámetros, dibujará la X tal cual está en el código:



```

void setup() {
  size(100, 100);
  smooth();
  noLoop();
}
void draw() {
  dibujarX();
}
void dibujarX() {
  //Dibujar una gruesa X color gris claro
  stroke(160);
  strokeWeight(20);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
}

```

Para dibujar una X diferente, necesitaremos agregar un parámetro. En el siguiente ejemplo, la variable `gris` a sido añadida para poder recibir un parámetro que actúe como variable de control de la escala de grises de la X. Este parámetro variable, debe incluir el tipo y un nombre.



```

void setup() {
  size(100, 100);
  smooth();
  noLoop();
}
void draw() {
  dibujarX(0); //Se envía como parámetro el valor 0
}
void dibujarX(int gris) { //Se declara la variable gris
  stroke(gris); //y se le asigna el valor del
  strokeWeight(20); //parámetro
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
}

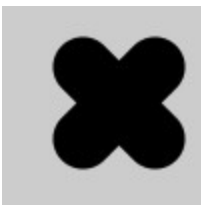
```

Esto está bien para un ejemplo sencillo, pero una función puede no necesariamente aceptar solo un parámetro. Podemos crear una cantidad ilimitada de estos. Cada parámetro debe tener un lugar entre los paréntesis, posterior al llamado de la función. Debe especificarse el tipo de dato y el nombre de cada uno de ellos. En el siguiente ejemplo, se agrega el parámetro *grosor*:



```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}
void draw() {
  dibujarX(0, 30); //Pasa los valores 0 y 30 a dibujarX()
}
void dibujarX(int gris, int grosor) {
  stroke(gris);
  strokeWeight(grosor);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
}
```

En el siguiente ejemplo, se extiende sus propiedades con tres parámetros más adicionales a los anteriores:



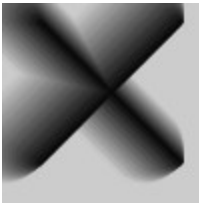
```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}
void draw() {
  dibujarX(0, 30, 40, 30, 36);
}
void dibujarX(int gris, int grosor, int x, int y, int largo) {
  stroke(gris);
  strokeWeight(grosor);
  line(x, y, x+largo, y+largo);
  line(x+largo, y, x, y+largo);
}
```

Hay que ser cuidadosos a la hora de programar funciones, haciendo paso por paso, sin perder el objetivo original. En este caso, podemos concluir generando el dibujo de las X originales de la siguiente forma:



```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}
void draw() {
  dibujarX(160, 20, 0, 5, 60);
  dibujarX(0, 10, 30, 20, 60);
  dibujarX(255, 2, 20, 38, 60);
}
void dibujarX(int gris, int grosor, int x, int y, int largo) {
  stroke(gris);
  strokeWeight(grosor);
  line(x, y, x+largo, y+largo);
  line(x+largo, y, x, y+largo);
}
```

Ahora que tenemos programada nuestra función `dibujarX()`, es posible crear cosas muy interesantes que serían poco prácticas sin la implementación de una de ellas. Por ejemplo, con un llamado de la función dentro de una estructura `FOR`, podemos iterar con pequeñas variaciones el modo en que se muestra la figura:



```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}
void draw() {
  for (int i = 0; i < 20; i++) {
    dibujarX(200- i*10, (20-i)*2, i, i/2, 70);
  }
}
void dibujarX(int gris, int grosor, int x, int y, int largo) {
  stroke(gris);
  strokeWeight(grosor);
  line(x, y, x+largo, y+largo);
  line(x+largo, y, x, y+largo);
}
```



```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}
void draw() {
  for (int i = 0; i < 70; i++) { //Dibuja 70 Figuras
    dibujarX(int(random(255)), int(random(30)),
             int(random(width)), int(random(height)), 100);
  }
}
void dibujarX(int gris, int grosor, int x, int y, int largo) {
  stroke(gris);
  strokeWeight(grosor);
  line(x, y, x+largo, y+largo);
  line(x+largo, y, x, y+largo);
}
```

Para el siguiente ejemplo usaremos una función que nombraremos `hoja()` y otra que llamaremos `parra()`. Con esto se pretende demostrar como una función puede correr dentro de otra. Para empezar definiremos que parámetros requerimos:

<code>float x</code>	Coordenada X
<code>float y</code>	Coordenada Y
<code>float ancho</code>	Ancho de la figura en pixeles
<code>int dir</code>	Dirección, siendo 1 (izquierda) o -1 (derecha)

El programa para dibujar la figura es muy sencillo:



```
void setup() {
  size(100, 100);
  smooth();
  noStroke();
  noLoop();
}
void draw() {
```

```

        hoja(26, 83, 60, 1);
    }
    void hoja(int x, int y, int ancho, int dir) {
        pushMatrix();
        translate(x, y);
        scale(ancho);
        beginShape();
        vertex(1.0*dir, -0.7);
        bezierVertex(1.0*dir, -0.7, 0.4*dir, -1.0, 0.0, 0.0);
        bezierVertex(0.0, 0.0, 1.0*dir, 0.4, 1.0*dir, -0.7);
        endShape();
        popMatrix();
    }

```

La función `parra()`, posee parámetros para cambiar la posición, el número de hojas y el tamaño de las mismas:

<code>int x</code>	Coordenada X
<code>int numHojas</code>	Número total de hojas en la parra
<code>float hojasAncho</code>	Ancho de las hojas en pixeles

Con esta función se dibuja una línea vertical y luego determina el espacio entre cada hoja:



```

void setup() {
    size(100, 100);
    smooth();
    noLoop();
}
void draw() {
    parra(33, 9, 16);
}
void parra(int x, int numHojas, int hojasAncho) {
    stroke(255);
    line(x, 0, x, height);
    noStroke();
    int gap = height / numHojas;
    int direccion = 1;
    for (int i = 0; i < numHojas; i++) {
        int r = int(random(gap));
        hoja(x, gap*i + r, hojasAncho, direccion);
        direccion = -direccion;
    }
}
//Copiar y pegar la función hoja() aquí

```

-Sobrecarga de Funciones

Las funciones pueden tener el mismo nombre, siempre y cuando tengan distinta cantidad de parámetros. Al crear funciones con el mismo nombre, se lo denomina *sobrecarga de funciones*. Esto es lo que permite que Processing tenga funciones mas de una versión de funciones como `fill()`, `image()` y `text()`, cada una acepta diferentes parámetros. Por ejemplo, `fill()` puede tener uno, dos, tres o hasta cuatro parámetros. Cada versión de `fill()` configura el color de relleno de una figura, ya sea en escala de grises, grises con transparencia, RGB o RGB con transparencia, respectivamente.

Un programa puede, también, tener dos funciones con el mismo nombre y el mismo número de parámetros, pero sus parámetros son de tipos de datos diferentes. Por ejemplo, hay tres versiones diferentes de `fill()` que aceptan un solo parámetro. La primer versión utiliza un parámetro tipo `int`, para escala de grises. La segunda versión utiliza uno del tipo `float`, también para escala de grises. Y la tercer versión acepta un

parámetro del tipo `color`, para configurar directamente desde un color. En el lenguaje de Processing, sería muy frustrante configurar un nombre distinto solo para obtener el mismo resultado.

El siguiente ejemplo utiliza tres versiones distintas de `dibujarX()`, pero con el mismo nombre.



```
void setup() {
    size(100, 100);
    smooth();
    noLoop();
}
void draw() {
    dibujarX(255); //Ejecuta la primer función dibujarX()
    dibujarX(5.5); //Ejecuta la segunda función dibujarX()
    dibujarX(0, 2, 44, 48, 36); //Ejecuta la tercer función
                                //dibujarX()
}
//Dibuja una X con un valor de gris
void dibujarX(int gris) {
    stroke(gris);
    strokeWeight(20);
    line(0, 5, 60, 65);
    line(60, 5, 0, 65);
}
//Dibuja una X negra con determinado valor de grosor
void dibujarX(float grosor) {
    stroke(0);
    strokeWeight(grosor);
    line(0, 5, 60, 65);
    line(60, 5, 0, 65);
}
//Dibuja una X con determinado valro de gris, grosor
//posición y tamaño
void dibujarX(int gris, int grosor, int x, int y, int s) {
    stroke(gris);
    strokeWeight(grosor);
    line(x, y, x+s, y+s);
    line(x+s, y, x, y+s);
}
```

-Calculando y Regresando Valores

En los ejemplos anteriores, la salida producida por una función dibuja una figura o una forma en la pantalla. No obstante, hay veces que preferimos que la propia salida que se produce sea un tipo de dato. A esto se lo llama *regresar un valor*. Todas las funciones puede regresar un valor, como un `int` o un `float`. Si la función no regresa ningún valor en particular, se utiliza la función especial llamada `void` (cuyo significado es *vacío*, es decir, la función regresa un valor vacío). El tipo de dato que regresa la función se declara a la izquierda del nombre de la misma.

La palabra `return` es usada para salir de la función y regresar a la posición de la cual fue llamada. Si una función regresará un valor, se utiliza `return` para indicar que valor se debe devolver. La acción de utilizar `return` es usualmente usada en la última línea de la función, puesto que inevitablemente saldrá de la función una vez usada. Por lo general, con las funciones de Processing, solemos utilizar funciones que devuelven valores. Por ejemplo, `random()` devuelve un valor tipo `float`, y la función `color()` devuelve un valor tipo `color`.

Si la función va a devolver un valor, debe llamarse dentro de una variable:

```
float d = random(0, 100);
ellipse(50, 50, d, d);
```

Además, debemos ser cuidadosos con el tipo de dato que estamos utilizando para guardar el valor:

```
int d = random(0, 100);          //ERROR! random() regresa un valor tipo float
ellipse(50, 50, d, d);
```

Si consultamos la referencia de cada función, podemos tener una idea de que clase de dato maneja cada función en particular. El problema ocurre cuando utilizamos funciones de usuario.

Para escribir nuestras propias funciones, simplemente reemplazaremos el tipo de datos `void` por el tipo de datos que pretendemos devolver. No hay que olvidar incluir la palabra-clave `return` dentro, de preferencia en la última línea. El siguiente ejemplo muestra una manera muy simple de escribir una función que devuelve datos:

```
void setup() {
    size(100, 100);
    float f = average(12.0, 6.0);          //Asigna 9.0 a f
    println(f);
}
float average(float num1, float num2) {
    float av = (num1 + num2) / 2.0;
    return av;
}

void setup() {
    size(100, 100);
    float c = fahrenheitACelsius(451.0);  //Asigna 232.77779 a c
    println(c);
}
float fahrenheitACelsius(float t) {
    float f = (t-32.0) * (5.0/9.0);
    return f;
}
```


Formas: Parámetros y Recursión

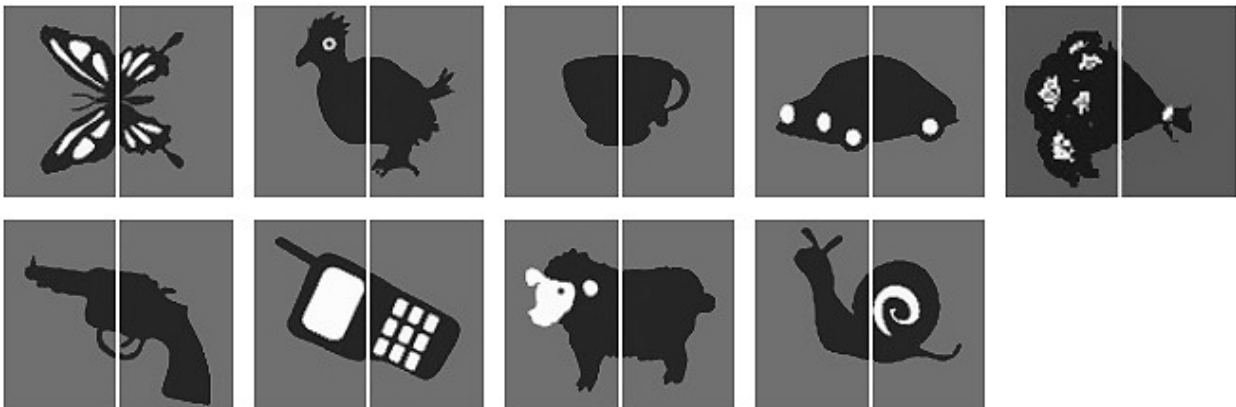
Los programas proveen de herramientas mínimas para la exploración y la búsqueda de sistemas con comportamientos particulares. Una de estas opciones es recurrir al uso de funciones. Cuando una figura es dibujada a partir de los parámetros que recibe, se dice que esa forma está *parametrizada*. Además, una función puede contener en sí misma una línea de código que utilice la misma función dentro de sí misma. Esta técnica es conocida como *recursión*.

-Formas Parametrizadas

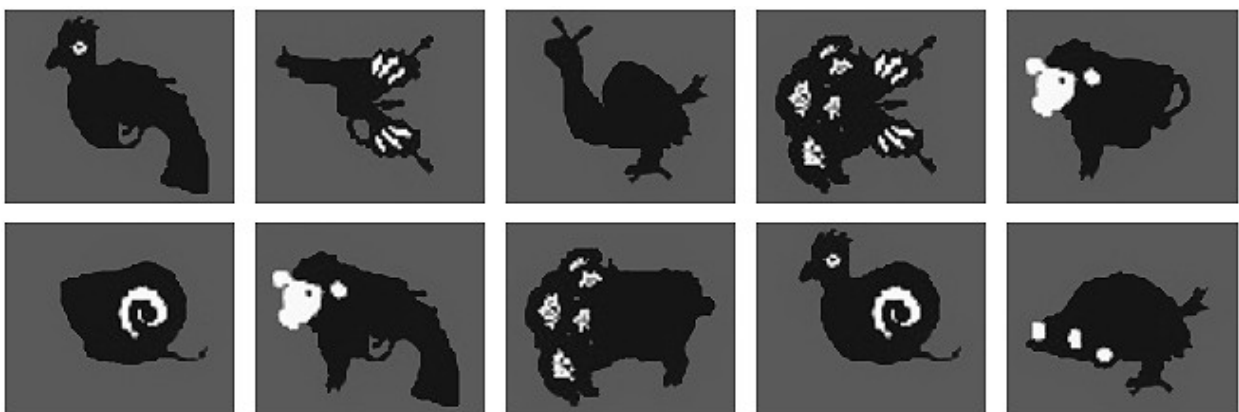
La función `hoja()` (vista anteriormente) es un claro ejemplo de una forma parametrizada. Dependiendo los parámetros que reciba, produce una forma diferente. La forma está condicionada a sus parámetros:



Las cartas Ars Magna creadas por Tatsuya Saito, muestra un ejemplo de un sistema de imágenes modulares. Nueve imágenes divididas en dos:



Una de las cartas frontales es usada en combinación con una de las traseras para crear resultados inesperados:



Un simple programa que puede emplearse para generar dichos resultados podría ser el siguiente:

```

size(120, 100);
int frente = int(random(1, 10)); //Seleccionar una carta del frente
int atrás = int(random(1, 10)); //Seleccionar una carta de atrás
PImage imgFrente = loadImage(frente + "f.jpg");
PImage imgAtras = loadImage(atrás + "b.jpg");
image(imgFrente, 0, 0);
image(imgAtras, 60, 0);

```

El sistema de Ars Magna nos permite crear, sin embargo, un número finito de combinaciones. Otra forma de crear formas parametrizadas es por incluir valores de entrada a través de algún dispositivo. Este es uno de los más grandes avances de crear visual con código.

Una simple función `arco()`, creada usando `bezierVertex()`, puede generar cambios continuos modulares por cambiar un simple parámetro. El parámetro para la función `arco()` es un número decimal (float), así que pueden generarse variaciones sumamente pequeñas.



```

float c = 25.0; //Al cambiar los valores de c
void setup() { //cambia la curvatura
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  arco(c);
}

void arco(float curvatura) {
  float y = 90.0;
  strokeWeight(6);
  noFill();
  beginShape();
  vertex(15.0, y);
  bezierVertex(15.0, y-curvatura, 30.0, 55.0, 50.0, 55.0);
  bezierVertex(70.0, 55.0, 85.0, y-curvatura, 85.0, y);
  endShape();
}

```

En un sistema parametrizado, como la función `arco()`, el valor de un parámetro puede afectar a más de una variable. Esto es llamado *coupling* (parejas). Si cambiamos un poco el código dentro de la función `arco()`, el valor de entrada `curvatura` puede controlar el grosor de la línea, además de la curva.



```

void arco(float curvatura) {
  float y = 90.0;
  float sw = (65.0 - curvatura) / 4.0;
  strokeWeight(sw);
  noFill();
  beginShape();
  vertex(15.0, y);
  bezierVertex(15.0, y-curvatura, 30.0, 55.0, 50.0, 55.0);
  bezierVertex(70.0, 55.0, 85.0, y-curvatura, 85.0, y);
  endShape();
}

```

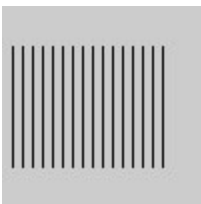
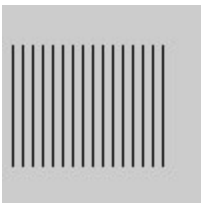
Esto se trata tan solo de un ejemplo muy sencillo. El mismo parámetro puede utilizarse para cambiar aspectos como la visualización, la rotación o la escala. El siguiente programa es prueba de ello:



```
int x = 20; //Coordenada X
int u = 14; //Unidades
float a = -0.12; //Ángulo
void setup() {
  size(100, 100);
  stroke(0, 153);
  smooth();
  noLoop();
}
void draw() {
  background(204);
  trazo(x, u, a);
}
void trazo(int xpos, int unidad, float angulo) {
  pushMatrix();
  translate(xpos, 0);
  for (int i = unidad; i > 0; i--) { //Cuenta Regresiva
    strokeWeight(i);
    line(0, 0, 0, 8);
    translate(0, 8);
    rotate(angulo);
  }
  popMatrix();
}
```

-Recursión

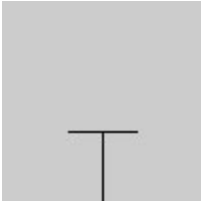
Un ejemplo práctico de algo recurrente es situarse entre dos espejos y ver el “infinito”. En términos del programa, la recursión se produce cuando una función puede llamarse a sí misma si tener ninguna clase de bloqueo. Para evitar que se llama a sí misma eternamente, es necesario agregar una salida. El siguiente programa genera el mismo resultado de dos maneras distintas. Una es por un ciclo FOR y la otra por recursión.



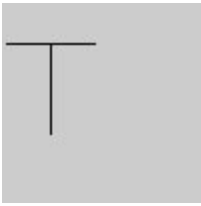
```
int x = 5;
for (int num = 15; num >= 0; num -= 1) {
  line(x, 20, x, 80);
  x += 5;
}

void setup() {
  dibujarLineas(5, 15);
}
void dibujarLineas(int x, int num) {
  line(x, 20, x, 80);
  if (num > 0) {
    dibujarLineas(x+5, num-1);
  }
}
```

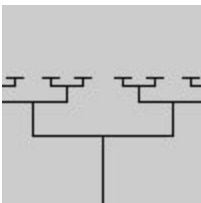
El ejemplo de recursión consume muchos más recursos y puede empeorar el rendimiento de nuestro programa. Para cálculos muy simples, una estructura *FOR* es una gran solución. No obstante, la recursión abre un abanico de posibilidades bastante interesantes. Los siguientes ejemplos utilizan una función llamada `dibujarT()`, para así mostrar los diferentes efectos de la recursión:



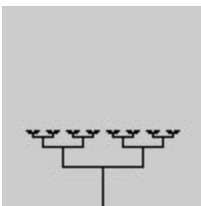
```
int x = 50;           //Coordenada X del centro
int y = 100;         //Coordenada Y de abajo
int a = 35;          //Mitad del ancho
void setup() {
    size(100, 100);
    noLoop();
}
void draw() {
    dibujarT(x, y, a);
}
void dibujarT(int xpos, int ypos, int apex) {
    line(xpos, ypos, xpos, ypos-apex);
    line(xpos-(apex/2), ypos-apex, xpos+(apex/2), ypos-
        apex);
}
```



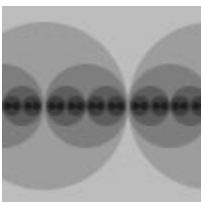
La función `dibujarT()` puede hacerse una recursión si se llama a los elementos del bloque nuevamente. Simplemente se necesitará agregar una variable `num` que permita aumentar o decrecer la cantidad de veces que se llama a dicha función:



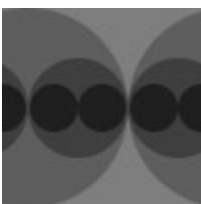
```
int x = 50;           //Coordenada X del centro
int y = 100;         //Coordenada Y de abajo
int a = 35;          //Mitad del ancho
int n = 3;           //Número de recursiones
void setup() {
    size(100, 100);
    noLoop();
}
void draw() {
    dibujarT(x, y, a, n);
}
void dibujarT(int x, int y, int apex, int num) {
    line(x, y, x, y-apex);
    line(x-apex, y-apex, x+apex, y-apex);
    //Esta expresión relacional debe permitir devolver un
    //valor falso para detener la recursión y dibujar en
    //pantalla.
    if (num > 0) {
        dibujarT(x-apex, y-apex, apex/2, num-1);
        dibujarT(x+apex, y-apex, apex/2, num-1);
    }
}
```



La estructura binaria en forma de árbol, puede ser visualizada de diversas maneras. El siguiente programa dibujar un círculo en cada nodo y elimina las uniones.



```
int x = 63;           //Coordenada X
int r = 85;          //Radio de Inicio
int n = 6;           //Número de recursiones
void setup() {
    size(100, 100);
    noStroke();
    smooth();
    noLoop();
}
void draw() {
    dibujarCirculo(x, r, n);
}
void dibujarCirculo(int x, int radius, int num) {
```

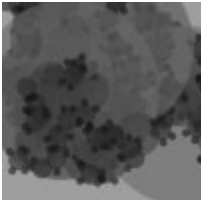
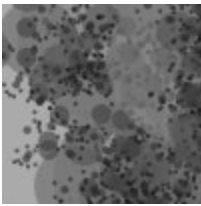


```

float tt = 126 * num/4.0;
fill(tt);
ellipse(x, 50, radius*2, radius*2);
if (num > 1) {
    num = num - 1;
    dibujarCirculo(x - radius/2, radius/2, num);
    dibujarCirculo(x + radius/2, radius/2, num);
}
}

```

Una pequeña modificación a las variables producen diferentes resultados en la visualización. Si a cada círculo se le agregara una valor aleatorio para marcar la posición, el resultado sería una imagen con una mezcla equilibrada entre orden y desorden. En el siguiente ejemplo, en cada recursión, la escala de los círculos decrece, la distancia con el círculo anterior también, y aumenta su nivel de oscuridad. Al cambiar el número utilizado con el `randomSeed()`, la composición varía.



```

int x = 63;          //Coordenada X
int y = 50;          //Coordenada Y
int r = 80;          //Radio de Inicio
int n = 7;           //Número de recursiones
int rs = 12;         //Valor para randomSeed
void setup() {
    size(100, 100);
    noStroke();
    smooth();
    noLoop();
    randomSeed(rs);
}
void draw() {
    dibujarCirculo(x, y, r, n);
}
void dibujarCirculo(float x, float y, int radius, int num) {
    float value = 126 * num / 6.0;
    fill(value, 153);
    ellipse(x, y, radius*2, radius*2);
    if (num > 1) {
        num = num - 1;
        int branches = int(random(2, 6));
        for (int i = 0; i < branches; i++) {
            float a = random(0, TWO_PI);
            float nuevox = x + cos(a) * 6.0 * num;
            float nuevoy = y + sin(a) * 6.0 * num;
            dibujarCirculo(nuevox, nuevoy, radius/2,
                           num);
        }
    }
}
}

```

Unidad 21

Valores de Entrada: Mouse

Elementos que se introducen en esta Unidad:

mouseX, mouseY, pmouseX, pmouseY, mousePressed, mouseButton
cursor(), noCursor()

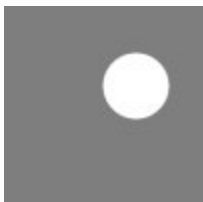
La pantalla del ordenador es tan solo un puente entre nuestro cuerpo físico y una cantidad abismal de circuitos eléctricos que habitan dentro de una computadora. Controlamos los elementos de la pantalla a través de “prótesis” físicas, tales como pantallas táctiles, trackballs o joysticks. Sin embargo, el más común de todos estos dispositivos posiblemente sea el *mouse* (ratón). El mouse de los ordenadores data de finales de 1960, cuando Douglas Engelbart lo presentó como un dispositivo del oN-Line System (NLS), uno de los primeros sistemas de ordenadores con visualización de video. El mouse fue incluido como concepto en Xerox Palo Alto Research Center (PARC), pero no fue hasta 1984 que la Apple Macintosh lo convirtió en catalizador de su actual uso. El diseño del mouse ha tenido diversas modificaciones a través de los años. No obstante, su forma de uso sigue siendo la misma. Se encarga de cambiar en pantalla la posición X-Y del cursor.

-Datos del Mouse

En Processing, las variables que nos permiten obtener datos del mouse son `mouseX` y `mouseY` (nótese el uso de mayúsculas en la X y en la Y) tomando como referencia la esquina superior izquierda como eje 0,0. Para ver los valores actuales del mouse en la consola, se recurre a un simple programa:

```
void draw() {  
    frameRate(12);  
    println(mouseX + " : " + mouseY);  
}
```

Cuando un programa inicia, el valor de `mouseX` y `mouseY` es 0. Si el cursor se mueve dentro de la ventana de representación, el valor cambia a la actual posición del mismo. Si el cursor se encuentra a la izquierda, el valor de `mouseX` será 0 y comenzará a incrementar a medida que este se mueve hacia la derecha. En cambio, si el cursor esta arriba, el valor de `mouseY` será 0 y comenzará a aumentar a medida que este se mueve hacia abajo. Si `mouseX` y `mouseY` se encuentran en un programa donde no existe una estructura `draw()` o está activada la función `noLoop()` en el `setup()`, los valores de ambos serán siempre 0. Generalmente, la posición del mouse es utilizada para controlar la posición de algunos elementos de pantalla. Lo interesante se produce cuando existen diferentes relaciones entre unos elementos y otros a base de conseguir datos de entrada por la posición del mouse. Para invertir los valores del mouse, simplemente hay que restarle a `mouseX` el ancho de pantalla (`width`) y a `mouseY` el alto de pantalla (`height`).



```
// Un círculo sigue al cursor  
void setup() {  
    size(100, 100);  
    smooth();  
    noStroke();  
}  
void draw() {  
    background(126);  
    ellipse(mouseX, mouseY, 33, 33);  
}
```





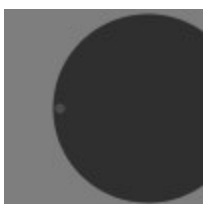
```
//Agregando operaciones
void setup() {
  size(100, 100);
  smooth();
  noStroke();
}
void draw() {
  background(126);
  ellipse(mouseX, 16, 33, 33);      //Circulo de Arriba
  ellipse(mouseX+20, 50, 33, 33);  //Circulo de el Medio
  ellipse(mouseX-20, 84, 33, 33);  //Circulo de Abajo
}
```



```
//Al multiplicar y dividir se crean posiciones escalares
void setup() {
  size(100, 100);
  smooth();
  noStroke();
}
void draw() {
  background(126);
  ellipse(mouseX, 16, 33, 33);      //Circulo de Arriba
  ellipse(mouseX/2, 50, 33, 33);    //Circulo de el Medio
  ellipse(mouseX*2, 84, 33, 33);    //Circulo de Abajo
}
```



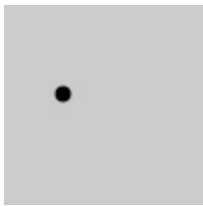
```
//Invertir la posición del cursor para crear segundas
//respuestas
void setup() {
  size(100, 100);
  noStroke();
  smooth();
}
void draw() {
  float x = mouseX;
  float y = mouseY;
  float ix = width - mouseX;        //Invertir X
  float iy = mouseY - height;       //Invertir Y
  background(126);
  fill(255, 150);
  ellipse(x, height/2, y, y);
  fill(0, 159);
  ellipse(ix, height/2, iy, iy);
}
```



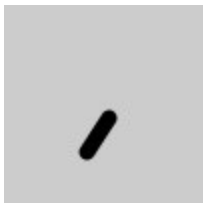
Las variables de Processing, pmouseX y pmouseY imprimen como valor la posición del mouse previa al cuadro que se esta ejecutando. Si el mouse no se mueve, el valor siempre será el mismo. Sin embargo, si el mouse se mueve rápidamente, los valores pueden oscilar entre diversos parámetros. Para ver esta diferencia, se puede ejecutar un simple programa que alterne los movimientos lentos y rápidos del mouse:

```
void draw() {
  frameRate(12);
  println(pmouseX - mouseX);
}
```

Al dibujar una línea desde la anterior posición del mouse hasta la posición actual, se revela la velocidad y la dirección del trazado. Cuando el mouse está quieto, si dibuja un punto. No obstante, al mover el mouse, se dibujan largas líneas.



```
// Dibuja un línea entre la anterior posición y la actual
//posición
void setup() {
  size(100, 100);
  strokeWeight(8);
  smooth();
}
void draw() {
  background(204);
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```



Los valores de `mouseX` y `mouseY` pueden utilizarse para controlar la escala, posición y rotación de los elementos del programa. Por ejemplo, pueden emplearse junto a la función `translate()`.



```
// Utilizando translate() para mover la figura
void setup() {
  size(100, 100);
  smooth();
  noStroke();
}
void draw() {
  background(126);
  translate(mouseX, mouseY);
  ellipse(0, 0, 33, 33);
}
```



Antes de utilizar los valores obtenidos por `mouseX` y `mouseY`, hay que pensar primero en la clase de parámetros que aceptan dichas funciones de transformación. Por ejemplo, la función `rotate()` solo acepta valores en radianes. Para hacer que una figura rote en 360 grados, es necesario convertir los valores de `mouseX` en valores de 0.0 a 2π . En el siguiente ejemplo, se utiliza la función `map()` para convertir dichos valores. El valor obtenido es utilizado en la función `rotate()`.



```
// Utilizando rotate() para rotar la figura
void setup() {
  size(100, 100);
  strokeWeight(8);
  smooth();
}
```



```
void draw() {
  background(204);
  float angle = map(mouseX, 0, width, 0, TWO_PI);
  translate(50, 50);
  rotate(angle);
  line(0, 0, 40, 0);
}
```


De la misma forma, se puede utilizar una estructura IF para reconocer individualmente diferentes sectores de la pantalla:



```
// La posición del cursor selecciona una mitad de la pantalla
void setup() {
  size(100, 100);
  noStroke();
  fill(0);
}
void draw() {
  background(204);
  if (mouseX < 50) {
    rect(0, 0, 50, 100); //Izquierda
  } else {
    rect(50, 0, 50, 100); //Derecha
  }
}
```

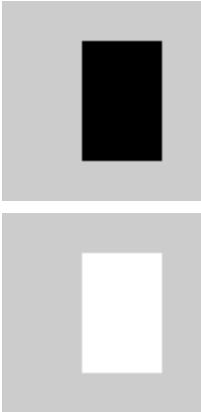


```
// La posición del cursor selecciona la izquierda, derecha o
//el centro de la pantalla
void setup() {
  size(100, 100);
  noStroke();
  fill(0);
}
void draw() {
  background(204);
  if (mouseX < 33) {
    rect(0, 0, 33, 100); //Izquierda
  } else if ((mouseX >= 33) && (mouseX <= 66)) {
    rect(33, 0, 33, 100); //Medio
  } else {
    rect(66, 0, 33, 100); //Derecha
  }
}
```



```
// La posición del cursor selecciona un cuadrante de la
//pantalla
void setup() {
  size(100, 100);
  noStroke();
  fill(0);
}
void draw() {
  background(204);
  if ((mouseX <= 50) && (mouseY <= 50)) {
    rect(0, 0, 50, 50); //Arriba-Izquierda
  } else if ((mouseX <= 50) && (mouseY > 50)) {
    rect(0, 50, 50, 50); //Abajo-Izquierda
  } else if ((mouseX > 50) && (mouseY < 50)) {
    rect(50, 0, 50, 50); //Arriba-Derecha
  } else {
    rect(50, 50, 50, 50); //Abajo-Derecha
  }
}
```

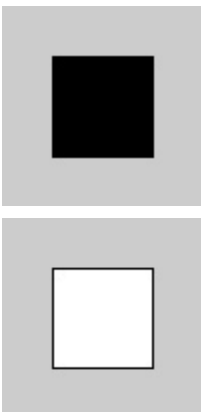




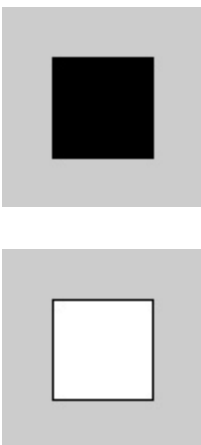
```
// La posición del cursor selecciona un área rectangular
void setup() {
  size(100, 100);
  noStroke();
  fill(0);
}
void draw() {
  background(204);
  if ((mouseX > 40) && (mouseX < 80) && (mouseY > 20)
  && (mouseY < 80)) {
    fill(255);
  } else {
    fill(0);
  }
  rect(40, 20, 40, 60);
}
```

-Botones del Mouse

Los dispositivos de entrada de las computadoras, por lo general, poseen entre 2 y 3 botones, y Processing puede detectarlos. La detección de la posición del mouse, sumado a los botones, permiten utilizar al mouse como un importante dispositivo de entrada en un programa interactivo. La variable `mousePressed` devuelve un valor `true` cuando un botón del mouse es oprimido, por el contrario devuelve un `false`. Además, permite detectar que botón fue oprimido a través de la variable `mouseButton`, pudiendo ser su valor `LEFT` (izquierdo), `RIGHT` (derecho) y `CENTER` (centro), dependiendo el botón que se desee detectar. Estas variables pueden utilizarse independientes o en combinación:



```
// El cuadrado cambia a blanco cuando el botón es presionado
void setup() {
  size(100, 100);
}
void draw() {
  background(204);
  if (mousePressed == true) {
    fill(255); //Blanco
  } else {
    fill(0); //Negro
  }
  rect(25, 25, 50, 50);
}
```



```
// El cuadro se vuelve negro cuando se oprime el botón
//izquierdo y blanco cuando se oprime el derecho, y gris
//cuando no se oprime ninguno.
void setup() {
  size(100, 100);
}
void draw() {
  if (mousePressed == true) {
    if (mouseButton == LEFT) {
      fill(0); //Negro
    } else if (mouseButton == RIGHT) {
      fill(255); //Blanco
    }
  } else {
    fill(126); //Gris
  }
}
```

```

    }
    rect(25, 25, 50, 50);
}

```

Nota: Al utilizar software que detecte dispositivos de entrada, se suele correr el riesgo de que el usuario no posea el dispositivo indicado (especialmente si se planea subir el proyecto a la web). Por ejemplo, existen usuarios que poseen un dispositivo de mouse con dos botones y otros que tienen tres botones. Es importante tener esto en mente a la hora de programar con dispositivos de entrada.

-Icono del Cursor

El cursor puede ser ocultado con la función `noCursor()`, y del mismo modo puede reemplazarse por la función `cursor()`. Cuando la función `noCursor()` se está ejecutando, el cursor se encuentra totalmente oculto, sin embargo, la posición puede conseguirse con `mouseX` y `mouseY`.

```

// Dibuja una elipse para mostrar la posición del cursor oculto
void setup() {
    size(100, 100);
    strokeWeight(7);
    smooth();
    noCursor();
}
void draw() {
    background(204);
    ellipse(mouseX, mouseY, 10, 10);
}

```

Si la función `noCursor()` se está ejecutando, el cursor estará completamente oculto hasta que se llame a la función `cursor()`:

```

// Esconde el cursor hasta que se oprima el botón del mouse
void setup() {
    size(100, 100);
    noCursor();
}
void draw() {
    background(204);
    if (mousePressed == true) {
        cursor();
    }
}

```

Además, la función `cursor()` acepta determinados parámetros que permiten cambiar el cursor determinado por defecto por otro diferente. Los parámetros auto-descriptivos son: `ARROW` (flecha), `CROSS` (cruz), `HAND` (mano), `MOVE` (mover), `TEXT` (texto) y `WAIT` (espera).

```

// Dibujar el cursor como una mano cuando el botón es oprimido
void setup() {
    size(100, 100);
    smooth();
}
void draw() {
    background(204);
    if (mousePressed == true) {
        cursor(HAND);
    } else {
        cursor(MOVE);
    }
}

```

```
    line(mouseX, 0, mouseX, height);  
    line(0, mouseY, height, mouseY);  
}
```

Las imágenes que se muestran como cursor son las que están instaladas por defecto en el ordenador, y varían entre sistemas operativos.

Dibujo: Formas Estáticas

El hecho de dibujar implica trasladar las percepciones individuales y la imaginación en formas visuales de representación. Las diferencias entre los dibujos de las personas demuestran que cada “mano” y cada mente es única. Los dibujos van desde las redes mecánicas de Sol LeWitt, pasando por las líneas lúcidas de Paul Klee, la figuración abstracta de Mariano Ferrante, las figuraciones cromáticas de Xul Solar, y mucho más allá. Es infinitamente abarcador el universo que envuelve al dibujo.

El dibujo por computadora se inicia a plantear en la década de 1960. Ivan Sutherland había creado en 1963, el software de *Sketchpad* (tabla de dibujo) para su disertación de PhD. El Sketchpad se convirtió en el antecesor de programas como el Autocad, el Adobe Illustrator o el Inkscape.

-Herramientas Simples

Una forma muy sencilla de dibujar con Processing es no incluir la función `background()` dentro del bloque `draw()`. Esta omisión permite acumular píxeles cuadro a cuadro.



```
//Dibuja un punto en la posición del cursor
void setup() {
    size(100, 100);
}
void draw() {
    point(mouseX, mouseY);
}
```



```
//Dibuja desde la anterior posición del mouse a la actual
//para crear líneas continuas
void setup() {
    size(100, 100);
}
void draw() {
    line(mouseX, mouseY, pmouseX, pmouseY);
}
```



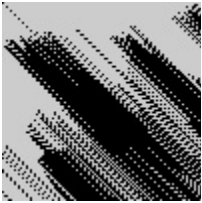
```
//Dibuja una línea cuando el botón del mouse es oprimido
void setup() {
    size(100, 100);
}
void draw() {
    if (mousePressed == true) {
        line(mouseX, mouseY, pmouseX, pmouseY);
    }
}
```



```
void setup() { //Dibuja líneas con diferentes valores
    size(100, 100); //de grises
}
void draw() {
    if (mousePressed == true) { //Si el mouse está
        stroke(255); //presionado el contorno
    } else { //será blanco. Sino,
        stroke(0); //será negro.
    }
    line(mouseX, mouseY, pmouseX, pmouseY);
}
```

El hecho de dibujar con software no restringe a solo seguir valores de entrada con el mouse. Incluyendo algo

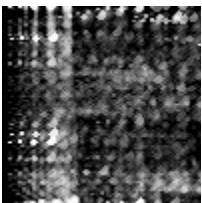
tan simple como una estructura FOR, es posible dibujar líneas más complejas con poca cantidad de código.



```
void setup() {
    size(100, 100);
}
void draw() {
    for (int i = 0; i < 50; i += 2) {
        point(mouseX+i, mouseY+i);
    }
}
```



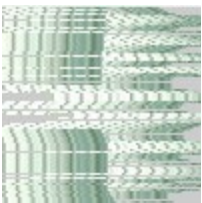
```
void setup() {
    size(100, 100);
}
void draw() {
    for (int i = -14; i <= 14; i += 2) {
        point(mouseX+i, mouseY);
    }
}
```



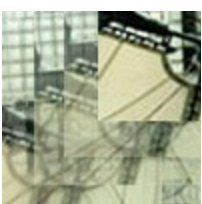
```
void setup() {
    size(100, 100);
    noStroke();
    fill(255, 40);
    background(0);
}
void draw() {
    if (mousePressed == true) {
        fill(0, 26);
    } else {
        fill(255, 26);
    }
    for (int i = 0; i < 6; i++) {
        ellipse(mouseX + i*i, mouseY, i, i);
    }
}
```

-Dibujando con Imágenes

De la misma forma que se emplean las figuras pre-diseñadas de Processing como herramientas de dibujo, las imágenes que nosotros mismos cargamos pueden ser empleadas para la misma labor.



```
// Dibujando con una imagen
PImage lineaImagen;
void setup() {
    size(100, 100);
    // Esta imagen es de 100 pixeles de ancho, pero solo 1
    //pixel de alto
    lineaImagen = loadImage("imagenlinea.jpg");
}
void draw() {
    image(lineaImagen, mouseX-lineImage.width/2, mouseY);
}
```



```
// Dibujando con una imagen que posee transparencia
PImage alphaImg;
void setup() {
    size(100, 100);
    // Esta imagen tiene transparencia
}
```

```
        alphaImg = loadImage("alphaArch.png");
    }
    void draw() {
        int ix = mouseX - alphaImg.width/2;
        int iy = mouseY - alphaImg.height/2;
        image(alphaImg, ix, iy);
    }
}
```

Valores de Entrada: Teclado

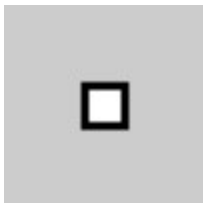
Elementos que se introducen en esta Unidad:

keyPressed, key, keyCode

Los teclados son comúnmente utilizados como dispositivos de entrada de caracteres para componer documentos de texto, Email, mensajería instantánea, y relacionados. Sin embargo, el potencial de un teclado de computadora va mucho más allá que su uso común. La expansión de la máquina de escribir al teclado permitió utilizar este mismo como medio para ejecutar programas, moverse entre aplicaciones y navegar en 3D en los diversos video-juegos. Es posible ignorar la impresión de caracteres y concentrarse solo en el ritmo en el que se oprimen las teclas. De esta forma, podremos crear un programa que obtenga como valor de entrada la velocidad con la que se escribe, y utilizar dichos valores para controlar la velocidad de un evento. En los últimos años, el teclado ha sido revalorizado por otra clase de dispositivos que no responden al estilo de escritura QWERTY. Métodos de tamaño más pequeño, como el teclado numérico de los teléfonos (a pesar de que dicha interfaz halla fallado en la industria y ahora los teléfonos móviles utilicen teclado tipo QWERTY). Por otro lado, el reconocimiento de voz está avanzando fuertemente, y se proclama como una alternativa futura a la escritura por medio del teclado.

-Datos del Teclado

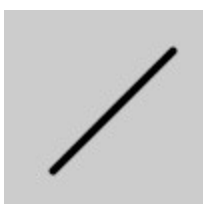
Processing puede registrar la última tecla presionada por un dispositivo de teclado. Por lo tanto, la variable que se emplea es denominada `keyPressed`, y devuelve un valor de tipo `boolean`. Dicho valor será `true` solo cuando una tecla halla sido presionada, de lo contrario será `false`. Mientras la tecla es presionada, este valor se mantendrá `true`, sin embargo, se convertirá en un `false` cuando la tecla sea liberada.



```
// Dibuja una línea si alguna tecla está siendo presionada
void setup() {
  size(100, 100);
  smooth();
  strokeWeight(4);
}
void draw() {
  background(204);
  if (keyPressed == true) { // Si una tecla es
    line(20, 20, 80, 80); // presionada, dibuja una
  } else { // línea. Sino,
    rect(40, 40, 20, 20); // dibuja un rectángulo
  }
}
```



```
// Mueve una línea mientras una tecla es presionada
int x = 20;
void setup() {
  size(100, 100);
  smooth();
  strokeWeight(4);
}
void draw() {
  background(204);
  if (keyPressed == true){ // Si una tecla es presionada
    x++; // agrega 1 a x
  }
  line(x, 20, x-60, 80);
}
```

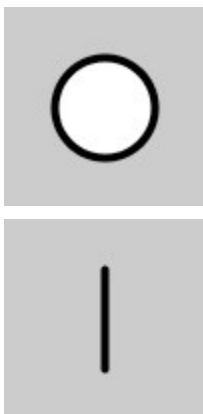


La variable `key` es del tipo `char`, permite almacenar un carácter, el último presionado. La variable `key` solo almacenará un valor cada vez. De esta manera, podemos crear un programa que muestre texto utilizando la variable `key` y la función `text()`.



```
PFont fuente;
void setup() {
  size(100, 100);
  fuente = loadFont("ThesisMonoLight-72.vlw");
  textFont(fuente);
}
void draw() {
  background(0);
  text(key, 28, 75);
}
```

La variable `key`, suele emplearse para detectar cual fue la última tecla oprimida. El siguiente ejemplo usa la expresión relacional `key == 'A'`. El empleo de las comillas simples (`' '`) y no de las comillas dobles (`" "`) se debe a que las comillas dobles se interpretan como valores del tipo `String`, y `key` solo devuelve valores del tipo `char`. Por lo tanto, si utilizáramos comillas dobles nos produciría un error. El evento se producirá cuando se oprima la A mayúscula únicamente.

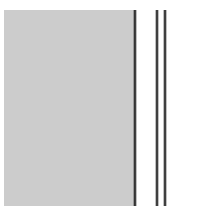


```
void setup() {
  size(100, 100);
  smooth();
  strokeWeight(4);
}
void draw() {
  background(204);
  // Si 'A' es presionada, dibuja una línea
  if ((keyPressed == true) && (key == 'A')) {
    line(50, 25, 50, 75);
  } else { // Sino, dibuja una elipse
    ellipse(50, 50, 50, 50);
  }
}
```

En contadas ocasiones necesitaremos detectar si una tecla, que producirá un evento, es presionada, pero se puede correr el riesgo de que el usuario tenga activada la mayúscula. Por ende, vamos a recurrir a comprobar ambas teclas para un mismo evento. Para esto, utilizaremos la expresión relacional `||` (o).

```
if ((keyPressed == true) && ((key == 'a') || (key == 'A'))) {
```

Ya que cada tecla tiene un valor numérico asignado (ASCII), el valor de una `key` puede utilizarse como variable de control.



```
int x = 0;
void setup() {
  size(100, 100);
}
void draw() {
  if (keyPressed == true) {
```



```
        x = key - 32;
        rect(x, -1, 20, 101);
    }
}
```

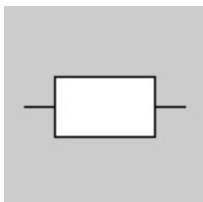


```
float angle = 0;
void setup() {
    size(100, 100);
    smooth();
    strokeWeight(8);
}
void draw() {
    background(204);
    if (keyPressed == true) {
        if ((key >= 32) && (key <= 126)) {
            // Si es una tecla alfanumérica,
            // convertir este valor en un ángulo
            angle = map(key, 32, 126, 0, TWO_PI);
        }
    }
    arc(50, 50, 66, 66, angle-PI/6, angle+PI/6);
}
```

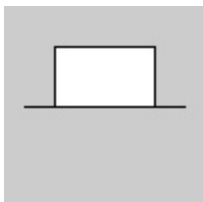


-Teclas Codificadas

Además de leer letras numéricas, caracteres, y símbolos, Processing puede leer valores de otras letras tales como Alt, Control, Shift, Barra-Espaciadora, Enter, Return, Escape y la barra de borrar. La variable `keyCode` almacena constantes encargadas de administrar esa información. Se emplea con la expresión `keyCode == CODED`, donde la constante puede ser `ALT`, `CONTROL`, `SHIFT`, `UP` (arriba), `DOWN` (abajo), `LEFT` (izquierda), y `RIGHT` (derecha). Existen caracteres que no son alfanuméricos, como `BACKSPACE`, `TAB`, `ENTER`, `RETURN`, `ESC`, y `DELETE`, y sin embargo no se los considera caracteres codificados. Esto se debe a que su implementación es muy variable dependiendo la plataforma en la que se realicen. Se verá más adelante.



```
int y = 35;
void setup() {
    size(100, 100);
}
void draw() {
    background(204);
    line(10, 50, 90, 50);
    if (key == CODED) {
        if (keyCode == UP) {
            y = 20;
        } else if (keyCode == DOWN) {
            y = 50;
        }
    } else {
        y = 35;
    }
    rect(25, y, 50, 30);
}
```



Unidad 24

Valores de Entrada: Eventos

Elementos que se introducen en esta Unidad:

mousePressed(), mouseReleased(), mouseMoved(), mouseDragged()
keyPressed(), keyReleased()
loop(), redraw()

Las funciones pueden llamar eventos que alteran el normal flujo del programa cuando una acción como un botón o una tecla es oprimida. Por lo tanto, un evento de una interrupción del flujo normal del programa. Una tecla oprimida, o la posición del mouse, es almacenada hasta que el bloque `draw()` terminé de ejecutarse. En esos casos, se puede producir un disturbio en el modo de dibujar. El código dentro de un evento se ejecuta una vez, cuando el evento ocurre. Por ejemplo, si se utilizara un evento que detecta el presionar un botón del mouse, las acciones ocurrirán solo una vez hasta que el botón sea liberado y presionado nuevamente. Esto permite que los datos que manejen los eventos del mouse o el teclado sean leídos independientemente de lo que esté ocurriendo en el bloque principal.

-Eventos del Mouse

Los eventos del mouse son `mousePressed()`, `mouseReleased()`, `mouseMoved()`, y `mouseDragged()`:

<code>mousePressed()</code>	El código dentro se ejecuta cuando un botón del mouse es oprimido.
<code>mouseReleased()</code>	El código dentro se ejecuta cuando un botón del mouse es liberado.
<code>mouseMoved()</code>	El código dentro se ejecuta cuando el cursor se mueve.
<code>mouseDragged()</code>	El código dentro se ejecuta cuando el cursor se mueve mientras un botón del mouse es oprimido

La función `mousePressed()` se ejecuta de forma diferente a la variable `mousePressed`. El valor de la variable `mousePressed` es un `true` hasta que el botón sea liberado. En cambio, el código dentro de un bloque `mousePressed()` se ejecuta cuando el botón del mouse es oprimido. En el siguiente ejemplo se puede ver como el fondo del programa cambia gradualmente su nivel de gris a medida que un botón del mouse es oprimido.

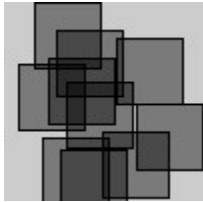
```
float gris = 0;
void setup() {
    size(100, 100);
}
void draw() {
    background(gris);
}
void mousePressed() {
    gris += 20;
}
```

En el siguiente ejemplo se ejecuta lo mismo que el ejemplo anterior, con la diferencia que lo que controla el nivel de gris es el evento de liberar el botón del mouse. La diferencia puede apreciarse manteniendo apretado el botón del mouse por un largo tiempo y así advertir que el fondo se altera cuando dicho botón es liberado.

```
float gray = 0;
void setup() {
    size(100, 100);
}
void draw() {
    background(gray);
}
```

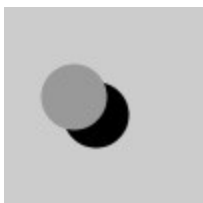
```
void mouseReleased() {
    gray += 20;
}
```

Antes de dibujar dentro de estas funciones, es importante pensar en el flujo del programa. Por ejemplo, hay círculos que son dibujados dentro de `mousePressed()`, y luego dentro del bloque `draw()` ya que retiramos la función `background()`. Sin embargo, si estuviese `background()` declarado, los elementos solo aparecerían un simple cuadro en pantalla y luego dejarían de verse.



```
void setup() {
    size(100, 100);
    fill(0, 102);
}
void draw() { } // Bloque draw() vacío, mantiene al
void mousePressed() { //programa ejecutándose
    rect(mouseX, mouseY, 33, 33);
}
```

El código de las funciones `mouseMoved()` y `mouseDragged()` se ejecuta cuando hay algún cambio de posición en el cursor. El código dentro de `mouseMoved()` se ejecuta al final de cada cuadro cuando la posición del mouse cambia y no es presionado ningún botón. En cambio, el código dentro de `mouseDragged()` se ejecuta cuando la posición del cursor es alterada a la vez que es oprimido el botón del mouse.



```
int dragX, dragY, moveX, moveY;
void setup() {
    size(100, 100);
    smooth();
    noStroke();
}
void draw() {
    background(204);
    fill(0);
    ellipse(dragX, dragY, 33, 33); //Círculo Negro
    fill(153);
    ellipse(moveX, moveY, 33, 33); //Círculo Gris
}
void mouseMoved() { //Mueve el círculo gris
    moveX = mouseX;
    moveY = mouseY;
}
void mouseDragged() { //Mueve el círculo negro
    dragX = mouseX;
    dragY = mouseY;
}
```

-Eventos del Teclado

Cada tecla presionada es registrada y posee dos clases de eventos, `keyPressed()` y `keyReleased()`:

`keyPressed()` El código dentro de este bloque se ejecuta cuando un tecla es presionada
`keyReleased()` El código dentro de este bloque se ejecuta cuando un tecla es liberada

Cada vez que una tecla es presionada podemos registrar el evento y relacionarlo a una serie de acciones. De esta forma, podemos usar `keyPressed()` para controlar valores numéricos y así utilizarlos para, por ejemplo, cambiar la posición de un rectángulo.



```
void setup() {
  size(100, 100);
  noStroke();
  fill(255, 51);
}
void draw() { } //Bloque draw() vacío, mantiene al
void keyPressed() { //programa ejecutándose
  int y = key - 32;
  rect(0, y, 100, 4);
}
```

Cada vez que una tecla es liberada el código dentro de `keyReleased()` es ejecutado. En el siguiente ejemplo, se dibuja una T cuando la tecla T se oprime, y desaparece cuando la tecla se suelta.



```
boolean dibujarT = false;
void setup() {
  size(100, 100);
  noStroke();
}
void draw() {
  background(204);
  if (dibujarT == true) {
    rect(20, 20, 60, 20);
    rect(39, 40, 22, 45);
  }
}
void keyPressed() {
  if ((key == 'T') || (key == 't')) {
    dibujarT = true;
  }
}
void keyReleased() {
  dibujarT = false;
}
```

Los siguientes dos ejemplos trabajan con `keyPressed()` para leer y analizar los valores de entrada por el teclado combinándolos con datos del tipo `String`.



```
// Un ejemplo extremadamente simple de un editor de texto
// permite agregar y remover elementos de una línea
PFont fuente;
String letras = "";
void setup() {
  size(100, 100);
  fuente = loadFont("Consolas-24.vlw");
  textFont(fuente);
  stroke(255);
  fill(0);
}
void draw() {
  background(204);
  float cursorPosition = textWidth(letras);
  line(cursorPosition, 0, cursorPosition, 100);
  text(letras, 0, 50);
}
void keyPressed() {
  if (key == BACKSPACE) { //Barra de Espacio
    if (letras.length() > 0) {
      letras= letras.substring(0, letras .length()-1);
    }
  }
}
```

```

    }
  } else if (textWidth(letras+key) < width){
    letras = letras+key;
  }
}

```



```

// Compara los valores de entrada del teclado
// sea "negro" o "gris" y cambia el fondo de acuerdo al valor.
// Presionar Enter o Return para activar los valores de
//entrada.
PFont fuente;
String letras = "";
int back = 102;
void setup() {
  size(100, 100);
  fuente = loadFont("Consolas-24.vlw");
  textFont(fuente);
  textAlign(CENTER);
}
void draw() {
  background(back);
  text(letras, 50, 50);
}
void keyPressed() {
  if ((key == ENTER) || (key == RETURN)) {
    letras = letras.toLowerCase();
    println(letras);          //Imprime en la consola
                              //el valor de entrada
    if (letras.equals("negro")) {
      back = 0;
    } else if (letras.equals("gris")) {
      back = 204;
    }
    letras = "";              // Limpia la variable
  } else if ((key > 31) && (key != CODED)) {
    //Si la tecla es alfanumérica, la agrega al String
    letras = letras + key;
  }
}
}

```

-Controlando el Flujo

Los programas utilizan el bloque `draw()` para dibujar cuadro a cuadro las acciones que se pretenden tan rápido como sea posible. Con la función `frameRate()`, es posible limitar la cantidad de cuadros que ejecuta una acción cada segundo, y la función `noLoop()` es utilizada para hacer que el bloque `draw()` deje de ejecutarse constantemente. Las funciones adicionales, `loop()` y `redraw()`, proveen de mas opciones cuando se utilizan eventos del mouse y el teclado.

De este modo, se podrá ejecutar un programa que se encuentre con `noLoop()` y utilizar la función `loop()` solo cuando se requiera. Esto sirve para ahorrar una gran cantidad de recursos (especialmente si se piensa utilizar el proyecto en la web). El siguiente ejemplo ejecuta un bloque `draw()` por dos segundos cada vez que el botón del mouse es oprimido. Pasado el tiempo, el programa se pone en pausa.

```

int frame = 0;
void setup() {
  size(100, 100);
  frameRate(30);
}
void draw() {

```

```

    if (frame > 60) { // Si ya pasaron mas de 60 cuadros
        noLoop(); // desde que el mouse fue oprimido, pausar el programa
        background(0); // y volver el fondo negro.
    } else { // Sino, hacer el fondo gris
        background(204); // y dibujar líneas en la
        line(mouseX, 0, mouseX, 100); // posición del mouse.
        line(0, mouseY, 100, mouseY);
        frame++;
    }
}
void mousePressed() {
    loop();
    frame = 0;
}

```

La función `redraw()` ejecuta el código del bloque `draw()` una vez y después detiene su ejecución. Esto es muy útil si nuestro programa no necesita ser actualizado continuamente. A continuación, se presenta un ejemplo donde el bloque `draw()` se ejecuta una vez cuando se oprime el botón del mouse:

```

void setup() {
    size(100, 100);
    noLoop();
}
void draw() {
    background(204);
    line(mouseX, 0, mouseX, 100);
}
void mousePressed() {
    redraw(); // Ejecuta el código en el draw() una vez
}

```

Valores de Entrada: Mouse II

Elementos que se introducen en esta Unidad:

`constrain()`, `dist()`, `abs()`, `atan2()`

La posición del cursor es un punto en la ventana de representación que se actualiza en cada cuadro. Este punto puede ser analizado y modificado en relación a otros elementos para producir nuevos valores. Es posible contraer los valores del mouse en un rango específico, calcular la distancia entre su posición y otro elemento, interpolar entre dos valores, determinar su velocidad, y calcular el ángulo del mouse en relación a otra posición.

-Restringir

La función `constrain()` permite limitar un número en un determinado rango. Recibe tres parámetros:

```
constrain(valor, min, max)
```

El parámetro *valor* es el número a limitar, el parámetro *min* determina el valor mínimo del rango, y el parámetro *max* determina el máximo valor del rango. Si el *valor* es menor o igual al parámetro *min*, entonces el *valor* equivale a *min*. Regresa, entonces el valor de *max* si el *valor* es mayor o igual a *max*.

```
int x = constrain(35, 10, 90);    // Asigna 35 a x
int y = constrain(5, 10, 90);    // Asigna 10 a y
int z = constrain(91, 10, 90);   // Asigna 90 a z
```

Cuando se utiliza junto con `mouseX` y `mouseY`, podemos determinar el rango de valores por el que se va a mover el cursor. Por ejemplo, un área.



```
// Limita la posición del cursor en un área
void setup() {
  size(100, 100);
  smooth();
  noStroke();
}
void draw() {
  background(0);
  // Limita mx entre 35 y 65
  float mx = constrain(mouseX, 35, 65);
  // Limita my entre 40 y 60
  float my = constrain(mouseY, 40, 60);
  fill(102);
  rect(20, 25, 60, 50);
  fill(255);
  ellipse(mx, my, 30, 30);
}
```

-Distancia

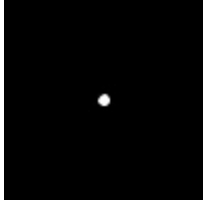
La función `dist()` calcula la distancia entre dos coordenadas. Esta función puede utilizarse para calcular la distancia entre la posición del cursor y un punto en la pantalla. Recibe cuatro parámetros:

```
dist(x1, y1, x2, y2)
```

Los parámetros *x1* y *y1* determinan el primer punto, mientras que *x2* y *y2* determinan el segundo punto. La distancia entre ambos es calculado como un número de tipo decimal (`float`).


```
float x = dist(0, 0, 50, 0); // Asigna 50.0 a x
float y = dist(50, 0, 50, 90); // Asigna 90.0 a y
float z = dist(30, 20, 80, 90); // Asigna 86.023254 a z
```

El valor regresado por `dist()` puede utilizarse para para cambiar las propiedades de una figura:



```
//La distancia entre el centro de la ventana de representación
//y el cursor determinan el tamaño del círculo.
```

```
void setup() {
    size(100, 100);
    smooth();
}
void draw() {
    background(0);
    float d = dist(width/2, height/2, mouseX, mouseY);
    ellipse(width/2, height/2, d*2, d*2);
}
```

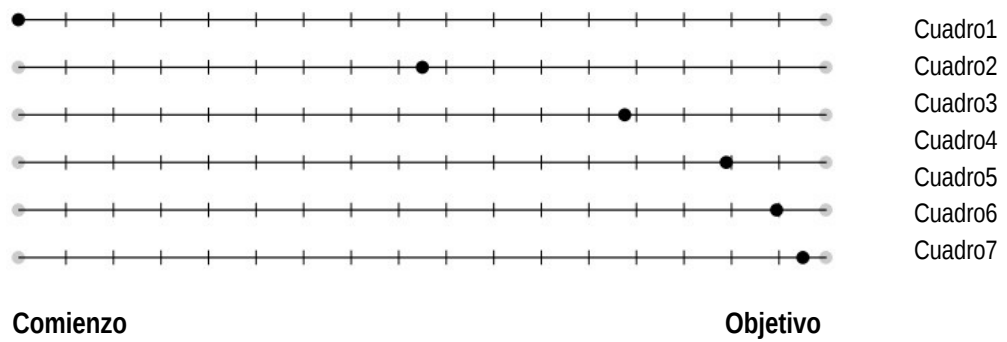


```
// Dibuja una grilla de círculos y calcula la
// la distancia para determinar su tamaño
```

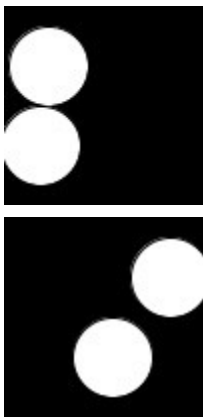
```
float maxDistancia;
void setup() {
    size(100, 100);
    noStroke();
    smooth();
    fill(0);
    maxDistancia = dist(0, 0, width, height);
}
void draw() {
    background(204);
    for (int i = 0; i <= width; i += 20) {
        for (int j = 0; j <= height; j += 20) {
            float mouseDist = dist(mouseX, mouseY,
                i, j);
            float diametro = (mouseDist / maxDistancia)
                * 66.0;
            ellipse(i, j, diametro, diametro);
        }
    }
}
```

-Interpolación

La Interpolación es una técnica de animación, que permite generar movimiento entre dos puntos. El hecho de mover una fracción de la distancia total por cuadro, permite generar desaceleraciones o incluso aceleraciones en la forma que se mueve una figura. El siguiente diagrama muestra lo que ocurre cuando un punto se mueve siempre la mitad entre su actual localización y su próxima localización:

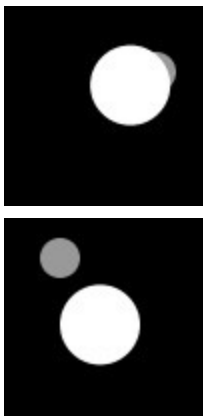


Como la figura disminuye su velocidad al acercarse al objetivo, la figura baja. En el siguiente ejemplo, la variable `x` es la posición actual del círculo, y la variable `objetivoX` es el objetivo al cual se pretende llegar.



```
float x = 0.0;
float interp = 0.05;           //Números de 0.0 y 1.0
void setup() {
    size(100, 100);
    smooth();
}
void draw() {
    background(0);
    float objetivoX = mouseX;
    x += (objetivoX - x) * interp;
    ellipse(mouseX, 30, 40, 40);
    ellipse(x, 70, 40, 40);
}
```

Para aplicar el mismo principio a la posición en `x` y en `y` a la vez, será necesario calcular la distancia entre ambas posiciones. En el siguiente ejemplo el pequeño círculo “sigue” al cursor. Mientras que el círculo grande se posiciona en pantalla por medio de un cálculo de interpolación.



```
float x = 0;
float y = 0;
float interp = 0.05;           //Números del 0.0 al 1.0
void setup() {
    size(100, 100);
    smooth();
}
void draw() {
    background(0);
    float objetivoX = mouseX;
    float objetivoY = mouseY;
    x += (objetivoX - x) * interp;
    y += (objetivoY - y) * interp;
    fill(153);
    ellipse(mouseX, mouseY, 20, 20);
    fill(255);
    ellipse(x, y, 40, 40);
}
```

El siguiente ejemplo introduce la función `abs()` para conseguir el valor absoluto de un número. Es necesario, ya que los valores utilizados en la interpolación pueden ser negativos o positivos, dependiendo de la posición en la que se encuentren con respecto al objetivo. Una estructura `IF` es utilizada para actualizar la posición, solo si está no tiene los mismos valores que el objetivo.

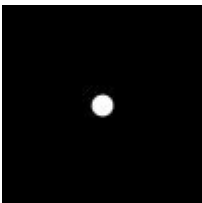


```
float x = 0.0;
float interp = 0.05;    //Números del 0.0 al 1.0
void setup() {
    size(100, 100);
    smooth();
}
void draw() {
    background(0);
    float objetivoX = mouseX;
    //Distancia de la posición al objetivo
    float dx = objetivoX - x;
    //Si la distancia está entre la actual posición y el
    //destino es mayor que 1.0, actualizar posición
    if (abs(dx) > 1.0) {
        x += dx * interp;
    }
    ellipse(mouseX, 30, 40, 40);
    ellipse(x, 70, 40, 40);
}
```

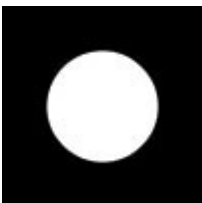


-Velocidad

Se puede calcular la velocidad del cursor por comparar su actual posición con la anterior. Puede conseguirse fácilmente utilizando la función `dist()` y los valores de `mouseX`, `mouseY`, `pmouseX`, y `pmouseY` como parámetros. El siguiente ejemplo calcula la velocidad del mouse y la utiliza como valor de de tamaño de una elipse.



```
void setup() {
    size(100, 100);
    noStroke();
    smooth();
}
void draw() {
    background(0);
    float velocidad= dist(mouseX, mouseY, pmouseX, pmouseY);
    float diametro = velocidad * 3.0;
    ellipse(50, 50, diametro, diametro);
}
```



El anterior ejemplo muestra una respuesta instantánea de la velocidad del mouse. Los números que producen son extremos y pueden ir desde cero a enormes valores de un cuadro al otro. La técnica de interpolación podría utilizarse para aumentar o disminuir gradualmente una propiedad de un objeto, utilizando los valores de velocidad del mouse.



```
float velocidad = 0.0;
float interp = 0.05;    // Números del 0.0 al 1.0
void setup() {
    size(100, 100);
    noStroke();
    smooth();
}
void draw() {
    background(0);
    float objetivo = dist(mouseX, mouseY, pmouseX, pmouseY);
    velocidad += (objetivo - velocidad) * interp;
    rect(0, 33, objetivo, 17);
}
```

```

    rect(0, 50, velocidad, 17);
}

```

-Orientación

La función `atan2()` es usada para calcular el ángulo desde un punto a la coordenada de origen $(0, 0)$. Recibe dos parámetros:

`atan2(y, x)`

El parámetro `y` es la coordenada de `y` de donde se encuentra el ángulo, mientras que el parámetro `x` es la coordenada en `x` de dicho punto. Los valores del ángulo son regresado en radianes, en un rango de π a $-\pi$. Es importante destacar que el orden de los parámetros `x` y `y` están invertidos en este caso, con respecto a las funciones que hemos visto.

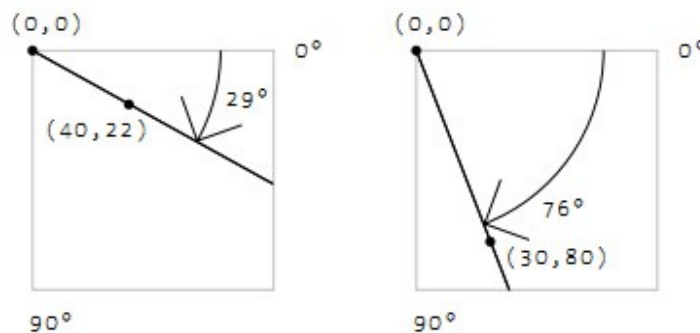
//El ángulo incrementa a medida que se el mouse se mueve de la esquina superior derecha a la esquina inferior izquierda

```

void setup() {
    size(100, 100);
    frameRate(15);
    fill(0);
}
void draw() {
    float angle = atan2(mouseY, mouseX);
    float deg = degrees(angle);
    println(deg);
    background(204);
    ellipse(mouseX, mouseY, 8, 8);
    rotate(angle);
    line(0, 0, 150, 0);
}

```

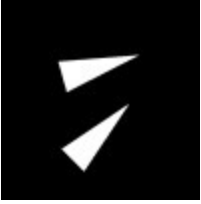
El código se explica con las siguientes imágenes:



Para calcular el `atan2()` relativo a otro punto en lugar del $(0, 0)$, debe restarle dicha coordenada a los parámetros `y` y `x`. Para esto, lo ideal es utilizar la función `translate()` con `atan2()` para controlar la rotación, junto con `pushMatrix()` y `popMatrix()` para controlar la figura.



```
//Rotar los triángulos siempre en el punto  
//del cursor  
float x = 50;  
float y1 = 33;  
float y2 = 66;
```



```
void setup() {  
  size(100, 100);  
  noStroke();  
  smooth();  
}  
  
void draw() {  
  background(0);  
  //Triángulo Superior  
  float angle = atan2(mouseY-y1, mouseX-x);  
  pushMatrix();  
  translate(x, y1);  
  rotate(angle);  
  triangle(-20, -8, 20, 0, -20, 8);  
  popMatrix();  
  pushMatrix();  
  //Triángulo Inferior  
  float angle2 = atan2(mouseY-(y2), mouseX-x);  
  translate(x, y2);  
  rotate(angle2);  
  triangle(-20, -8, 20, 0, -20, 8);  
  popMatrix();  
}
```

Valores de Entrada: Tiempo y Fechas

Elementos que se introducen en esta Unidad:

`second()`, `minute()`, `hour()`, `millis()`, `day()`, `month()`, `year()`

Los seres humanos poseemos una relativa percepción del tiempo, pero las máquinas mantienen una idea fija y regular del tiempo. En tiempos pasados, se utilizaban relojes de sol y de agua para poder tener una forma correcta de visualizar el tiempo. Hoy en día las personas recurren a los relojes digitales, un reloj numérico graduado en doce horas, con minutos y segundos.

La posibilidad de que un programa sea capaz de leer el actual tiempo y fechas, abre la posibilidad a un área interesante. Darle la posibilidad a un programa de depender del tiempo, nos otorga poder hacer que este cambie de color cada día, o un reloj digital que produzca una animación cada hora.

-Segundos, Minutos, Horas

Para empezar, Processing es capaz de leer los datos del reloj de la computadora. Los segundos actuales son leídos con la función `second()`, la cual regresa un valor entre 0 y 59. Los minutos actuales son leídos con la función `minute()`, la cual regresa un valor entre 0 y 59. La hora actual es leída por la función `hour()`, la cual levanta cualquiera de las 24 horas, regresa valores entre 0 y 23. En este sistema, la medianoche es 0, el mediodía es 12, las 9:00 a.m. es 9, y las 5:00 p.m. es 17. Si se ejecuta el siguiente programa podrá verse la hora actual en la consola:

```
int s = second();           //Regresa valores de 0 a 59
int m = minute();          //Regresa valores de 0 a 59
int h = hour();            //Regresa valores de 0 a 23
println(h + ":" + m + ":" + s); //Imprime la hora en la consola
```

Si se ejecuta dentro del bloque `draw()`, podremos ver como el tiempo se va actualizando. El siguiente ejemplo lee continuamente la hora actual y la muestra en la consola:

```
int ultimoSegundo = 0;

void setup() {
  size(100, 100);
}
void draw() {
  int s = second();
  int m = minute();
  int h = hour();
  //Solo imprime una vez, cuando los segundos cambian
  if (s != ultimoSegundo) {
    println(h + ":" + m + ":" + s);
    ultimoSegundo = s;
  }
}
```

Además, se puede crear de forma muy simple un reloj que muestre la hora y se vaya actualizando al utilizar la función `text()`. La función `nF()` se utiliza para separar apropiadamente el espacio de la izquierda y la derecha del número.



```
PFont fuente;

void setup() {
  size(100, 100);
  fuente = loadFont("Pro-20.vlw");
  textFont(fuente);
}

void draw() {
  background(0);
  int s = second();
  int m = minute();
  int h = hour();
  //La función nf() separa los números correctamente
  String t = nf(h,2) + ":" + nf(m,2) + ":" + nf(s,2);
  text(t, 10, 55);
}
```

Hay muchas formas de expresar el paso del tiempo, y no todas deben estar ligadas a la forma de un reloj. En el siguiente ejemplo, las líneas simbolizan las horas, los minutos y los segundos. Para hacerlo simplemente se recurre a usar la función `map()` y así convertir los valores de 0 a 59, a un rango de 0 a 100.



```
void setup() {
  size(100, 100);
  stroke(255);
}

void draw() {
  background(0);
  float s = map(second(), 0, 60, 0, 100);
  float m = map(minute(), 0, 60, 0, 100);
  float h = map(hour(), 0, 24, 0, 100);
  line(s, 0, s, 33);
  line(m, 34, m, 66);
  line(h, 67, h, 100);
}
```

Al tener los valores normalizados, los mismos pueden utilizarse para otra clase de factores. En este caso, se programará un clásico reloj de mano. El inconveniente se produce en la función `hour()`, ya que esta devuelve únicamente valores de 24 horas, y requerimos valores de 12 horas. Sin embargo, podremos conseguir dicha escala por sacar el `% 12`.



```
void setup() {
  size(100, 100);
  stroke(255);
}

void draw() {
  background(0);
  fill(80);
  noStroke();
  //Ángulos para sin() y cos() inician a las 3 en punto;
  //la sustracción de HALF_PI hace que comiencen desde
  //arriba
  ellipse(50, 50, 80, 80);
  float s = map(second(), 0, 60, 0, TWO_PI) - HALF_PI;
  float m = map(minute(), 0, 60, 0, TWO_PI) - HALF_PI;
  float h = map(hour() % 12, 0, 12, 0, TWO_PI) - HALF_PI;
  stroke(255);
  line(50, 50, cos(s) * 38 + 50, sin(s) * 38 + 50);
}
```

```

        line(50, 50, cos(m) * 30 + 50, sin(m) * 30 + 50);
        line(50, 50, cos(h) * 25 + 50, sin(h) * 25 + 50);
    }

```

Como agregado a la lectura del reloj del ordenador, Processing incluye una función que almacena el tiempo desde que inició el programa. Este tiempo es almacenado en mili-segundos, o sea, milésimas de segundos. Dos mil mili-segundos son 2 segundos, 200 mili-segundos son 0,2 segundos. Este número puede ser obtenido con la función `millis()` y puede ser utilizado para calcular el tiempo transcurrido desde que inició el programa.

```

//Se usa millis() para iniciar una línea pasados 3 segundos
int x = 0;

void setup() {
    size(100, 100);
}
void draw() {
    if (millis() > 3000) {
        x++;
    }
    line(x, 0, x, 100);
}

```

La función `millis()` regresa un número del tipo `int`, pero es conveniente convertirlo en un `float` para poder utilizarlo directamente como segundos.

```

int x = 0;

void setup() {
    size(100, 100);
}
void draw() {
    float sec = millis() / 1000.0;
    if (sec > 3.0) {
        x++;
    }
    line(x, 0, x, 100);
}

```

-Fecha

La información de la fecha es leída de manera similar a la del tiempo. El día actual es leído con la función `day()`, la cual devuelve un valor entre 1 y 31. El mes corriente es leído con la función `month()`, la cual devuelve un valor entre 1 y 12, donde 1 es Enero, 6 es Junio y 12 es Diciembre. El año actual es leído con la función `year()`, el cual regresa un valor entero de cuatro dígitos, o sea, el año actual.

```

int d = day();           //Regresa valor de 1 al 31
int m = month();        //Regresa valor de 1 al 12
int y = year();         //Regresa el año en cuatro dígitos (2009, 2010, etc.)
println(d + " " + m + " " + y);

```

El siguiente ejemplo evalúa si el día actual es el primero del mes. En caso de ser cierto, imprime en la consola el mensaje *"Bienvenido a un nuevo Mes"*.

```

void draw() {
    int d = day();           //Valores de 1 a 31
    if (d == 1) {

```



```
        println("Bienvenido a un nuevo Mes.");
    }
}
```

El siguiente ejemplo se ejecuta continuamente y evalúa si el día actual es el primer día del año, o sea, Año Nuevo. En caso de que sea cierto, imprime en la consola "*Hoy es el primer día del Año!*".

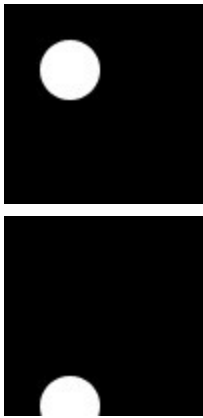
```
void draw() {
    int d = day();           //Valores de 1 a 31
    int m = month();        //Valores de 1 a 12
    if ((d == 1) && (m == 1)) {
        println("Hoy es el primer día del Año!");
    }
}
```

Movimiento: Líneas y Curvas

Una profunda comprensión del movimiento es sumamente útil a la hora de comunicar para el bailarín, el animador y el director de cine. Quien practica el arte de los nuevos medios puede emplear el movimiento para potenciar cualquiera de sus trabajos, desde un sitio web a un video-juego. El elemento fundamental será el tiempo, o, más precisamente, como los elementos cambian en el tiempo. Las imágenes estáticas pueden expresar la noción de movimiento, un medio basado en el tiempo, como un vídeo, un *film*, y el software pueden expresar eso mismo de forma directa. La definición de movimiento a través de código muestra el poder y la flexibilidad del medio.

-Controlando el Movimientos

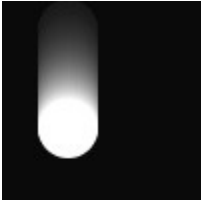
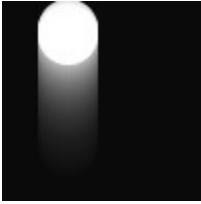
Para poner una figura en movimiento, siempre necesitaremos de al menos una variable que cambie al menos una de sus propiedades en cada cuadro que se ejecuta la función `draw()`. En unidades previas, se han presentado algunas formas de generar movimientos. Para que la figura no deje un rastro, necesitamos limpiar la pantalla antes de dibujar la figura actualizada. Para esto, pondremos la función `background()` al principio de la estructura `draw()`, de esta forma estaremos limpiando cada cuadro antes de dibujar en el. Y por último, tener en cuenta la utilización de la función `frameRate()` para controlar la cantidad de cuadros por segundo que va a procesar el programa (a mayor cantidad, el movimiento será mas fluido, pero se necesitará de una mayor capacidad en el ordenador).



```
float y = 50.0;
float vel = 1.0;
float radio = 15.0;
void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}
void draw() {
  background(0);
  ellipse(33, y, radio, radio);
  y = y + vel;
  if (y > height+radio) {
    y = -radio;
  }
}
```

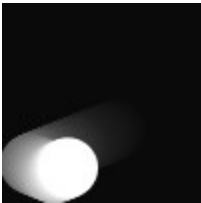
La dinámica permite también crear una serie de efectos visuales muy interesantes. Uno de ellos, y de sencilla programación, es crear un desenfoque mientras se mueve la figura. Es una de las tantas alternativas al utilizar `background()` al principio del `draw()`. Utilizando simplemente un rectángulo del tamaño de la ventana de representación, podremos darle a dicha figura cierta transparencia. Si el número de alfa se acerca a 255, el rastro desenfocado será cada vez menor hasta no distinguirse. Si este número se acerca a 0, dicho desenfoque será mayor.

Podemos, además, crear una variable `direccion` que controle la posición de la figura. De esta forma, cuando la figura se exceda de la ventana de representación, esta variable puede cambiar a -1 y así invertir los valores con los que se mueve la figura. De esta forma, si `direccion` es 1, será -1, y si `direccion` es -1, será 1. La figura resultante se encontrará siempre dentro de las dimensiones de la ventana de representación.



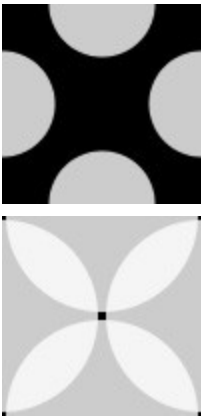
```
float y = 50.0;
float vel = 1.0;
float radio = 15.0;
int direccion = 1;
void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}
void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  ellipse(33, y, radio, radio);
  y += vel * direccion;
  if ((y > height-radio) || (y < radio)) {
    direccion = -direccion;
  }
}
```

Para tener una figura que también cambie la posición relativa a los márgenes izquierdo y derecho de la ventana de representación, requiere un segundo grupo de variables. El siguiente ejemplo trabaja con el mismo principio que los anteriores, solo que fija un límite en todos los márgenes.



```
float x = 50.0;          //Coordenada X
float y = 50.0;          //Coordenada Y
float radio = 15.0;      //Radio del círculo
float velX = 1.0;        //Velocidad del movimiento en el eje X
float velY = 0.4;        //Velocidad del movimiento en el eje Y
int direccionX = 1;      //Dirección del movimiento en el eje X
int direccionY = -1;     //Dirección del movimiento en el eje Y
void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}
void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  ellipse(x, y, radio, radio);
  x += velX * direccionX;
  if ((x > width-radio) || (x < radio)) {
    direccionX = -direccionX;    //Cambia Dirección
  }
  y += velY * direccionY;
  if ((y > height-radio) || (y < radio)) {
    direccionY = -direccionY;    //Cambia Dirección
  }
}
```

También es posible cambiar las propiedades del fondo, color, relleno y tamaño, en lugar de solo la posición de una figura. El siguiente ejemplo modifica el tamaño de unas elipses con la misma idea de cambio de dirección del ejemplo anterior.

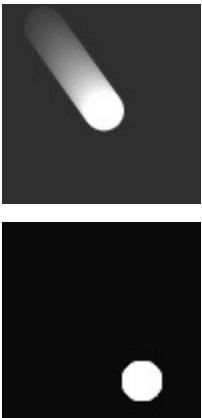


```

float d = 20.0;
float vel = 1.0;
int direccion = 1;
void setup() {
    size(100, 100);
    smooth();
    noStroke();
    fill(255, 204);
}
void draw() {
    background(0);
    ellipse(0, 50, d, d);
    ellipse(100, 50, d, d);
    ellipse(50, 0, d, d);
    ellipse(50, 100, d, d);
    d += vel * direccion;
    if ((d > width) || (d < width/10)) {
        direccion = -direccion;
    }
}

```

En contraste con los movimientos implícitos vistos en los ejemplos anteriores, podemos crear movimientos explícitos. Esta clase de movimientos requieren que se establezca primero una posición de inicio y la distancia del recorrido. Además, necesitaremos establecer el recorrido entre los cuadros. En el código a continuación, las variables `inicioX` e `inicioY`, establecen la posición de partida. En contra parte, las variables `finalX` y `finalY` corresponden a la posición de llegada. Las variables `x` e `y` corresponden a la posición actual de la figura. La variable `step` representa el porcentaje del cambio que tendrá la figura, y la variable `pct` mantiene el viaje del porcentaje total de la distancia recorrida. Por lo tanto, `step` y `pct` deben siempre ser entre 0.0 y 1.0. Cuando el programa se ejecuta, `step` se incrementa y así cambia la posición de las variables `x` e `y`, ese valor es mayor a 1.0, el movimiento se detiene.



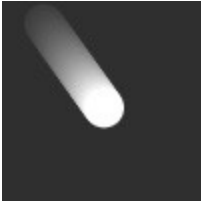
```

float inicioX = 20.0;           //Coordenada inicial X
float inicioY = 10.0;          //Coordenada inicial Y
float finalX = 70.0;           //Coordenada final X
float finalY = 80.0;          //Coordenada final Y
float distX;                    //Distancia con el Eje X
float distY;                    //Distancia con el Eje Y
float x = 0.0;                 //Coordenada X actual
float y = 0.0;                 //Coordenada Y actual
float step = 0.02; //Tamaño de cada movimiento (de 0.0 a 1.0)
float pct = 0.0; //Porcentaje recorrido (de 0.0 a 1.0)
void setup() {
    size(100, 100);
    noStroke();
    smooth();
    distX = finalX - inicioX;
    distY = finalY - inicioY;
}
void draw() {
    fill(0, 12);
    rect(0, 0, width, height);
    pct += step;
    if (pct < 1.0) {
        x = inicioX + (pct * distX);
        y = inicioY + (pct * distY);
    }
    fill(255);
    ellipse(x, y, 20, 20);
}

```

```
}
```

La técnica de interpolación, introducida en unidades previas, puede ser un gran recurso para los movimientos de animación. El siguiente ejemplo adapta el código anterior y agrega la técnica de interpolación.



```
float x = 20.0;           //Coordenada X inicial
float y = 10.0;           //Coordenada Y inicial
float objetivoX = 70.0;   //Coordenada X de Destino
float objetivoY = 80.0;   //Coordenada Y de Destino
float interpolacion = 0.05; //Tamaño de cada movimiento
void setup() {
  size(100, 100);
  noStroke();
  smooth();
}
void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  float d = dist(x, y, objetivoX, objetivoY);
  if (d > 1.0) {
    x += (objetivoX - x) * interpolacion;
    y += (objetivoY - y) * interpolacion;
  }
  fill(255);
  ellipse(x, y, 20, 20);
}
```

-Movimiento a lo largo de las Curvas

En unidades anteriores, se detalló particularmente el uso de curvas simples con formulas matemáticas que Processing posee en su estructura. En lugar de dibujar la curva entera en un solo cuadro, es posible distribuir este proceso en los pasos para llegar de un punto a otro. El siguiente código es un simple agregado a los ejemplos anteriores, donde el trayecto en lugar de hacerse de forma recta, se utiliza un trayecto de curva.



```
float inicioX = 20.0;     //Coordenada inicial X
float inicioY = 10.0;     //Coordenada inicial Y
float finalX = 70.0;      //Coordenada final X
float finalY = 80.0;      //Coordenada final Y
float distX;              //Distancia con el Eje X
float distY;              //Distancia con el Eje Y
float exponente = 0.5;    //Determina la Curva
float x = 0.0;            //Coordenada X actual
float y = 0.0;            //Coordenada Y actual
float step = 0.01;        //Tamaño de cada movimiento
float pct = 0.0;          //Porcentaje recorrido (de 0.0 a 1.0)
void setup() {
  size(100, 100);
  noStroke();
  smooth();
  distX = finalX - inicioX;
  distY = finalY - inicioY;
}
void draw() {
  fill(0, 2);
  rect(0, 0, width, height);
  pct += step;
  if (pct < 1.0) {
    x = inicioX + (pct * distX);
    y = inicioY + (pow(pct, exponente) * distY);
  }
}
```

```

    }
    fill(255);
    ellipse(x, y, 20, 20);
}

```

Todos los tipos de curvas vistos en las unidades previas pueden utilizarse para escalar figuras o modificar su posición. También, una vez que el programa ejecuta un camino curvo, puede calcular y generar otro.

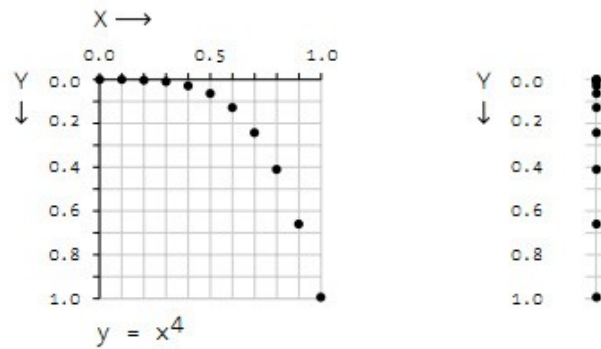


```

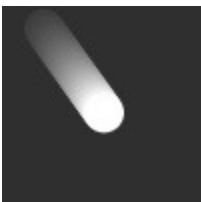
float inicioX = 20.0;           //Coordenada inicial X
float inicioY = 10.0;          //Coordenada inicial Y
float finalX = 70.0;           //Coordenada final X
float finalY = 80.0;           //Coordenada final Y
float distX;                    //Distancia con el Eje X
float distY;                    //Distancia con el Eje Y
float exponente = 0.5;         //Determina la Curva
float x = 0.0;                 //Coordenada X actual
float y = 0.0;                 //Coordenada Y actual
float step = 0.01;             //Tamaño de cada movimiento
float pct = 0.0;               //Porcentaje recorrido (de 0.0 a 1.0)
int direccion = 1;
void setup() {
    size(100, 100);
    noStroke();
    smooth();
    distX = finalX - inicioX;
    distY = finalY - inicioY;
}
void draw() {
    fill(0, 2);
    rect(0, 0, width, height);
    pct += step * direccion;
    if ((pct > 1.0) || (pct < 0.0)) {
        direccion = direccion * -1;
    }
    if (direccion == 1) {
        x = inicioX + (pct * distX);
        float e = pow(pct, exponente);
        y = inicioY + (e * distY);
    } else {
        x = inicioX + (pct * distX);
        float e = pow(1.0-pct, exponente*2);
        y = inicioY + (e * -distY) + distY;
    }
    fill(255);
    ellipse(x, y, 20, 20);
}

```

Ya que la figura realiza un trayecto curvo, su velocidad varía. De esta forma, dicho trayecto puede usarse para controlar la velocidad visual de un elemento. Los resultados que devuelven las funciones matemáticas, no deben usarse necesariamente en las dos dimensiones de la ventana de representación. Supongamos que omitimos la posición en x, de esta manera utilizaríamos la coordenada-y como controlador de la velocidad de la figura, a pesar de que el trayecto sea una recta.



El siguiente ejemplo muestra como utilizar una curva como regulador de la velocidad. La elipse comienza muy lento, e inmediatamente comienza a acelerarse. La variable exponente es la encargada de regular la curva, por lo tanto, la velocidad está ligada íntimamente a ella. Por hacer click con el mouse se selecciona un nuevo punto.



```

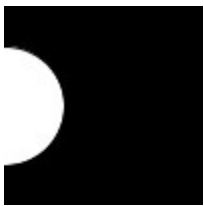
float inicioX = 20.0;           //Coordenada inicial X
float inicioY = 10.0;          //Coordenada inicial Y
float finalX = 70.0;           //Coordenada final X
float finalY = 80.0;           //Coordenada final Y
float distX;                     //Distancia con el Eje X
float distY;                     //Distancia con el Eje Y
float exponente = 0.5;          //Determina la Curva
float x = 0.0;                  //Coordenada X actual
float y = 0.0;                  //Coordenada Y actual
float step = 0.01;              //Tamaño de cada movimiento
float pct = 0.0;                //Porcentaje recorrido (de 0.0 a 1.0)
void setup() {
    size(100, 100);
    noStroke();
    smooth();
    distX = finalX - inicioX;
    distY = finalY - inicioY;
}
void draw() {
    fill(0, 2);
    rect(0, 0, width, height);
    if (pct < 1.0) {
        pct = pct + step;
        float rate = pow(pct, exponente);
        x = inicioX + (rate * distX);
        y = inicioY + (rate * distY);
    }
    fill(255);
    ellipse(x, y, 20, 20);
}
void mousePressed() {
    pct = 0.0;
    inicioX = x;
    inicioY = y;
    distX = mouseX - x;
    distY = mouseY - y;
}

```

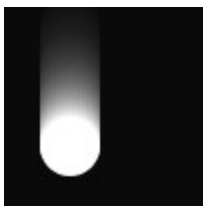
-Movimiento a través de la Transformación

Las funciones de transformación pueden usarse también en movimiento por cambiar los parámetros que reciben las funciones `translate()`, `rotate()` y `scale()`. Antes de utilizar las funciones de

transformación, es importante tener conocimientos plenos en ellas, y saber principalmente que estas cambian las propiedades del bloque draw(). Si ejecutamos la función translate(5, 0), todo lo que se incluya en el bloque draw() comenzará corrido 5 píxeles en el eje x. No solo eso, aumentará de 5 en 5 en cada repetición. De la misma forma, si incluimos la función en el bloque setup(), esta no tendrá efecto.



```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
  translate(50, 0);      //No surge efecto
}
void draw() {
  background(0);
  ellipse(0, 50, 60, 60);
}
```



```
float y = 50.0;
float vel = 1.0;
float radio = 15.0;
void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}
void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  translate(0, y); //Establece la coordenada-y del círculo
  ellipse(33, 0, radio, radio);
  y += vel;
  if (y > height+radio) {
    y = -radio;
  }
}
```



```
float y = 50.0;
float vel = 1.0;
float radio = 15.0;
void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}
void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  pushMatrix();
  translate(0, y);
  //Afectado por el primer translate()
  ellipse(33, 0, radio, radio);
  translate(0, y);
  // Afectado por el primer y el segundo translate()
  ellipse(66, 0, radio, radio);
  popMatrix();
  //No afectado por ningún translate()
```

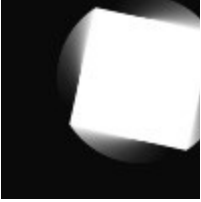


```

        ellipse(99, 50, radio, radio);
        y = y + vel;
        if (y > height+radio) {
            y = -radio;
        }
    }

float angulo = 0.0;
void setup() {
    size(100, 100);
    smooth();
    noStroke();
}
void draw() {
    fill(0, 12);
    rect(0, 0, width, height);
    fill(255);
    angulo = angulo + 0.02;
    translate(70, 40);
    rotate(angulo);
        rect(-30, -30, 60, 60);
}

```



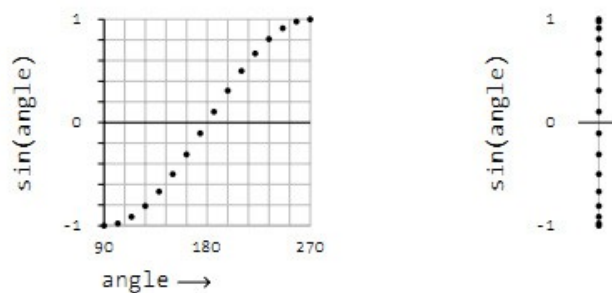
Las funciones `translate()`, `rotate()` y `scale()` pueden ser utilizadas para generar movimientos, pero usarlas todas juntas resulta intrincado. Para tener un mayor control de dichas propiedades se recomienda una profunda lectura de las unidades dedicadas a transformaciones vistas anteriormente, y recurrir al uso de `pushMatrix()` y `popMatrix()`.

Movimiento: Mecánicos y Orgánicos

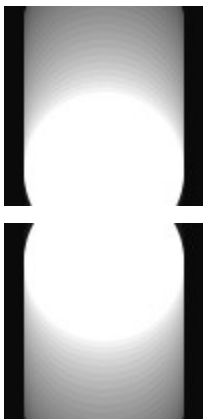
Los programadores suelen determinar como quieren que sus programas se muevan. Cosas como esas suelen transmitir y comunicar diversas cosas, por lo tanto no es correcto librarlo al azar. Solemos clasificar a las cosas entre lo que posee una clase de movimiento y lo que no lo posee. Hacemos distensiones entre lo animado y lo inanimado. El movimiento en un programa puede ser mas ventajoso aún, ya que podría intentarse hacer una fiel copia a la realidad o ignorar es por completo. Los programas hacen posible incontables tipos de movimientos diferentes, pero el foco está puesto en dos de ellos: los mecánicos y los orgánicos.

-Movimiento Mecánico

La función `sin()` suele usarse con mucha frecuencia a la hora de producir un movimiento elegante. Esto se debe a su facilidad para producir aceleraciones y desaceleraciones con un único valor. La onda seno, tomando los valores de las dos dimensiones, producirá un efecto un tanto extraño. Sin embargo, si tomamos únicamente los valores de la coordenada-y, solo conseguiríamos los valores de incremento y decrecimiento. En las siguientes imágenes, la de la izquierda representa la evolución de la onda seno a lo largo de las dos coordenadas. En cambio, la de la derecha, omite las coordenadas del eje x, mostrando solo los puntos que recorre el eje y.



Los valores de `sin()` son utilizados para crear el movimiento de una figura en el siguiente ejemplo. La variable `angulo` incrementa constantemente y produce el cambio en `sin()` en un rango de -1 a 1. Los valores son multiplicados por `radio` para hacer que estos sean mayores. El resultado es asignado a la variable `yoffset`, y esta luego es usada para determinar la posición de la elipse en la coordenada-y.



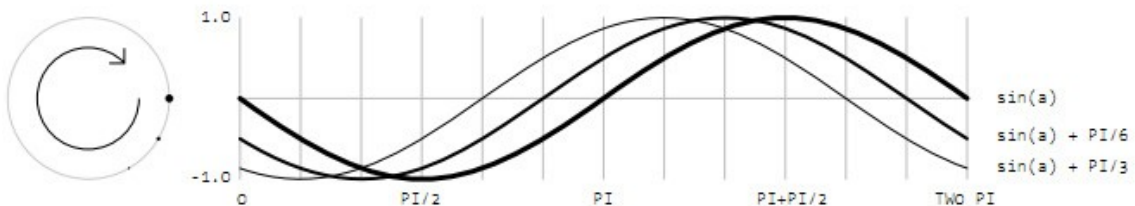
```
float angulo = 0.0;           //Ángulo Actual
float vel = 0.1;             //Velocidad del Movimiento
float radio = 40.0;         //Rango del Movimiento
void setup() {
  size(100, 100);
  noStroke();
  smooth();
}
void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  angulo += vel;
  float sinval = sin(angulo);
  float yoffset = sinval * radio;
  ellipse(50, 50 + yoffset, 80, 80);
}
```

Agregando valores de `sin()` y `cos()` se producirán movimientos mucho más complejos. En el siguiente ejemplo, un pequeño punto se mueve de forma circular usando los valores de `sin()` y `cos()`. Un gran punto utiliza los mismos valores pero agrega unos nuevos valores de `sin()` y `cos()`.

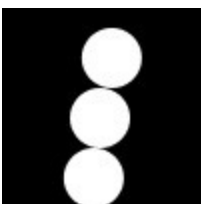


```
float angulo = 0.0; //Ángulo Actual
float vel = 0.05; //Velocidad del Movimiento
float radio = 30.0; //Rango del Movimiento
float sx = 2.0;
float sy = 2.0;
void setup() {
  size(100, 100);
  noStroke();
  smooth();
}
void draw() {
  fill(0, 4);
  rect(0, 0, width, height);
  angulo += vel; //Actualiza ángulo
  float sinval = sin(angulo);
  float cosval = cos(angulo);
  //Establece la posición del círculo pequeño
  float x = 50 + (cosval * radio);
  float y = 50 + (sinval * radio);
  fill(255);
  ellipse(x, y, 2, 2); //Dibuja el pequeño círculo
  //Establece la posición del círculo grande
  //basándose en la posición del pequeño
  float x2 = x + cos(angulo * sx) * radio/2;
  float y2 = y + sin(angulo * sy) * radio/2;
  ellipse(x2, y2, 6, 6); //Dibujó el círculo grande
}
```

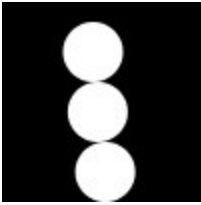
La fase de una función seno es una iteración a través de sus posibles valores. El *desplazamiento de fase* de una señal en relación a otra es sencillamente un retraso en tiempo expresado en grados de ángulo, en el cual un círculo completo (360 grados) es igual a un ciclo de la señal. Si se desplaza el ángulo utilizado para generar una animación, la animación resultante se encontrará desfasada.



Los siguientes dos ejemplos cambian la coordenada-x y el diámetro de los círculos para demostrar el desplazamiento de fase. En el primer ejemplo, cada círculo tiene el mismo movimiento horizontal, pero desfasado en el tiempo. En el segundo, cada círculo mantiene la misma posición y varían su tamaño, pero el rango de crecimiento está desfasado en el tiempo.



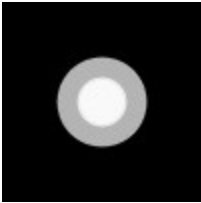
```
float angulo = 0.0;
float vel = 0.1;
void setup() {
  size(100, 100);
  noStroke();
  smooth();
}
```



```

}
void draw() {
  background(0);
  angulo = angulo + vel;
  ellipse(50 + (sin(angulo + PI) * 5), 25, 30, 30);
  ellipse(50 + (sin(angulo + HALF_PI) * 5), 55, 30, 30);
  ellipse(50 + (sin(angulo + QUARTER_PI) * 5), 85, 30,
  30);
}

```



```

float angulo = 0.0; //Ángulo que cambia
float vel = 0.05; //Velocidad de Crecimiento
void setup() {
  size(100, 100);
  noStroke();
  smooth();
  fill(255, 180);
}

```



```

void draw() {
  background(0);
  circuloFase(0.0);
  circuloFase(QUARTER_PI);
  circuloFase(HALF_PI);
  angulo += vel;
}
void circuloFase(float fase) {
  float diametro = 65 + (sin(angulo + fase) * 45);
  ellipse(50, 50, diametro, diametro);
}

```

-Movimiento Orgánico

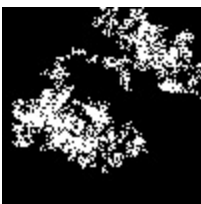
Entre los ejemplos de movimientos orgánicos se incluyen la caída de una hoja, la caminata de un insecto, el vuelo de un ave, una persona respirando, un río que fluye o la humareda que sube. Uno de los recursos con los que cuentan los programas es la utilización de valores aleatorios.



```

float x = 50.0; //Coordenada-X
float y = 80.0; //Coordenada-Y
void setup() {
  size(100, 100);
  randomSeed(0); //Fuerza los mismos valores aleatorios
  background(0);
  stroke(255);
}

```

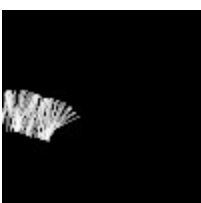


```

void draw() {
  x += random(-2, 2); //Asigna una nueva coordenada-x
  y += random(-2, 2); //Asigna una nueva coordenada-y
  point(x, y);
}

```

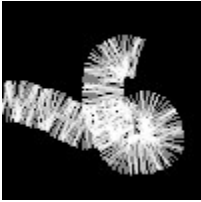
Las funciones `sin()` y `cos()` pueden utilizarse para generar movimientos impredecibles al emplearse con la función `random()`. El siguiente ejemplo presenta una línea, y en cada cuadro su dirección cambia entre pequeños valores, de -0.3 a 0.3.



```

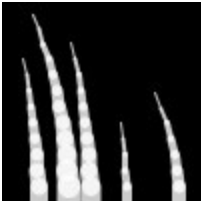
float x = 0.0; //Coordenada-X
float y = 50.0; //Coordenada-Y
float angulo = 0.0; //Dirección del Movimiento
float vel = 0.5; //Velocidad del Movimiento
void setup() {

```



```
size(100, 100);
background(0);
stroke(255, 130);
randomSeed(121);           //Fuerza los mismos valores
                             //aleatorios
}
void draw() {
  angulo += random(-0.3, 0.3);
  x += cos(angulo) * vel;   //Actualiza X
  y += sin(angulo) * vel;   //Actualiza Y
  translate(x, y);
  rotate(angulo);
  line(0, -10, 0, 10);
}
```

El siguiente ejemplo es una versión animada de un ejemplo visto unidades anteriores. Aquí la variable `angulo` para la función `hoja()` cambia constantemente.



```
float inc = 0.0;
void setup() {
  size(100, 100);
  stroke(255, 204);
  smooth();
}
void draw() {
  background(0);
  inc += 0.01;
  float angulo = sin(inc)/10.0 + sin(inc*1.2)/20.0;
  hoja(18, 9, angulo/1.3);
  hoja(33, 12, angulo);
  hoja(44, 10, angulo/1.3);
  hoja(62, 5, angulo);
  hoja(88, 7, angulo*2);
}
void hoja(int x, int units, float angulo) {
  pushMatrix();
  translate(x, 100);
  for (int i = units; i > 0; i--) {
    strokeWeight(i);
    line(0, 0, 0, -8);
    translate(0, -8);
    rotate(angulo);
  }
  popMatrix();
}
```

La función `noise()` es otro de los grandes recursos para producir movimientos orgánicos. Ya que los números regresados por `noise()` son fáciles de controlar, son muy útiles a la hora de producir irregularidades. El siguiente ejemplo dibuja dos líneas y la pantalla, y la posición de sus puntas finales está determinada por la función `noise()`.



```
float inc1 = 0.1;
float n1 = 0.0;
float inc2 = 0.09;
float n2 = 0.0;
void setup() {
  size(100, 100);
  stroke(255);
  strokeWeight(20);
}
```



```
        smooth();
    }
    void draw() {
        background(0);
        float y1 = (noise(n1) - 0.5) * 30.0; //De -15 a 15
        float y2 = (noise(n2) - 0.5) * 30.0; //De -15 a 15
        line(0, 50, 40, 50 + y1);
        line(100, 50, 60, 50 + y2);
        n1 += inc1;
        n2 += inc2;
    }
}
```

La función `noise()` también puede ser utilizada para generar texturas dinámicas. En el siguiente ejemplo, los primeros dos parámetros determinan una textura en dos dimensiones, y el tercer parámetro su incremento en cada cuadro.



```
float inc = 0.06;
int densidad = 4;
float znoise = 0.0;
void setup() {
    size(100, 100);
    noStroke();
}
void draw() {
    float xnoise = 0.0;
    float ynoise = 0.0;
    for (int y = 0; y < height; y += densidad) {
        for (int x = 0; x < width; x += densidad) {
            float n = noise(xnoise, ynoise, znoise) *
                256;
            fill(n);
            rect(y, x, densidad, densidad);
            xnoise += inc;
        }
        xnoise = 0;
        ynoise += inc;
    }
    znoise += inc;
}
}
```

Unidad 29

Datos: Arrays

Elementos que se introducen en esta Unidad:

Array, [] (array access), new, Array.length, append(), shorten(), expand(), arraycopy()

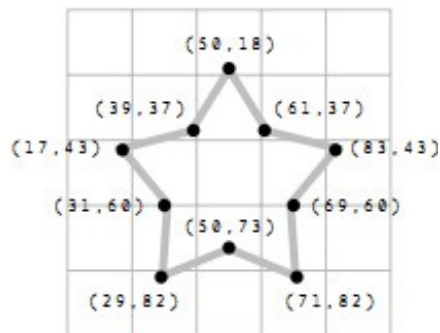
El término *array* se refiere a una estructura de grupo o a la imposición de un número. En programación, un array es un conjunto de datos almacenados bajo el mismo nombre. Los arrays pueden ser creados para almacenar un determinado tipo de datos, y cada uno de sus elementos pueden ser almacenados y leídos individualmente.

Cinco valores del tipo entero (1919, 1940, 1975, 1976, 1990) pueden ser almacenados en un array del tipo entero (*int*). Por ejemplo, si llamáramos al array "dates", guardaríamos los valores en la siguiente secuencia:

dates	1919	1940	1975	1976	1990
	[0]	[1]	[2]	[3]	[4]

La posición en la que se encuentran los datos en un array son numerados desde el número cero. Esto muchas veces produce errores, ya que por costumbre uno asocia la primera posición con el número uno. El primer elemento se encuentra en la posición cero [0], el segundo en la posición uno [1], y así.

Los arrays puede utilizarse para programar de un modo mucho más sencillo. Mientras no sea necesario utilizarlos, pueden emplearse para administrar estructuras. Por ejemplo, establecemos una serie de puntos que serán almacenados en el array, los mismos serán utilizados para dibujar una estrella.



El anterior ejemplo demuestra los beneficios del empleo de los arrays. La estrella tiene 10 vértices, compuesto de 2 puntos cada uno. Si utilizáramos variables requeriríamos de 20 de ellas. En cambio, con un array por eje, solo necesitaríamos 2.

Variables Separadas

```
int x0 = 50;
int y0 = 18;
int x1 = 61;
int y1 = 37;
int x2 = 83;
int y2 = 43;
int x3 = 69;
int y3 = 60;
int x4 = 71;
int y4 = 82;
int x5 = 50;
int y5 = 73;
int x6 = 29;
int y6 = 82;
int x7 = 31;
int y7 = 60;
int x8 = 17;
int y8 = 43;
int x9 = 39;
int y9 = 37;
```

Un array por cada punto

```
int[] p0 = { 50, 18 };
int[] p1 = { 61, 37 };
int[] p2 = { 83, 43 };
int[] p3 = { 69, 60 };
int[] p4 = { 71, 82 };
int[] p5 = { 50, 73 };
int[] p6 = { 29, 82 };
int[] p7 = { 31, 60 };
int[] p8 = { 17, 43 };
int[] p9 = { 39, 37 };
```

Un array por eje

```
int[] x = { 50, 61, 83, 69, 71, 50, 29, 31, 17, 39 };
int[] y = { 18, 37, 43, 60, 82, 73, 82, 60, 43, 37 };
```

El siguiente ejemplo muestra la utilización de un array en un programa. Los datos que almacena el array son leídos con una estructura FOR. La sintaxis será discutida en las páginas siguientes.



```
int[] x = { 50, 61, 83, 69, 71, 50, 29, 31, 17, 39 };
int[] y = { 18, 37, 43, 60, 82, 73, 82, 60, 43, 37 };
beginShape();
//Lee un elemento del array con cada ciclo del for()
for (int i = 0; i < x.length; i++) {
    vertex(x[i], y[i]);
}
endShape(CLOSE);
```

-Usando Arrays

Los arrays son declarados muy similar a otra clase de datos, pero se distinguen por la implementación de corchetes , [y]. Cuando un array es declarado, el tipo de datos que puede almacenar debe ser especificado también. Una vez que dicho array sea declarado y su tipo sea especificado, este debe ser creado con la palabra-clave new. Es paso adicional se utiliza para ahorrar espacio en la memoria de la computadora. Luego de que el array es creado, ya está listo para que se le asignen valores. Existen, además, diferentes formas de declarar, crear y asignar. En los siguientes ejemplo, se discuten las diferentes maneras, siempre asignando los valores 19, 40, 75, 76, y 90. Particularmente notar la relación entre los diferentes métodos y la estructura setup().

```
int[] data; //Declarar
void setup() {
    size(100, 100);
    data = new int[5]; //Crear
    data[0] = 19; //Asignar
    data[1] = 40;
    data[2] = 75;
    data[3] = 76;
    data[4] = 90;
}
```



```

int[] data = new int[5];      //Declarar, crear
void setup() {
    size(100, 100);
    data[0] = 19;            //Asignar
    data[1] = 40;
    data[2] = 75;
    data[3] = 76;
    data[4] = 90;
}

int[] data = { 19, 40, 75, 76, 90 };    //Declarar, crear y asignar
void setup() {
    size(100, 100);
}

```

Los anteriores tres ejemplos asumen que trabajaremos con los bloques `setup()` y `draw()`. Sin embargo, eso no es necesario, pueden usarse cualquiera de los 3 métodos anteriores sin dichas estructuras.

```

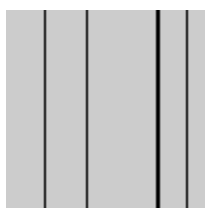
int[] data;                //Declarar
data = new int[5];        //Crear
data[0] = 19;             //Asignar
data[1] = 40;
data[2] = 75;
data[3] = 76;
data[4] = 90;

int[] data = new int[5];  //Declarar, crear
data[0] = 19;            //Asignar
data[1] = 40;
data[2] = 75;
data[3] = 76;
data[4] = 90;

int[] data = { 19, 40, 75, 76, 90 };    //Declarar, crear, asignar

```

El paso de declarar, crear y asignar permite a un array ser leído. Para acceder al elemento de un array es necesario usar el nombre de la variable seguido de los corchetes, y dentro de estos últimos el número de posición del elemento.



```

int[] data = { 19, 40, 75, 76, 90 };
line(data[0], 0, data[0], 100);
line(data[1], 0, data[1], 100);
line(data[2], 0, data[2], 100);
line(data[3], 0, data[3], 100);
line(data[4], 0, data[4], 100);

```

Recordar que el primer elemento se encuentra en la posición 0. Si el programa trata de acceder a un valor que excede las dimensiones del array, el programa se tendrá y terminará por devolver un *ArrayIndexOutOfBoundsException*.

```

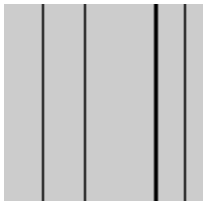
int[] data = { 19, 40, 75, 76, 90 };
println(data[0]); //Imprime 19 en la consola
println(data[2]); //Imprime 75 en la consola
println(data[5]); // ERROR! El último elemento del array está en la posición 4

```

La propiedad `length` almacena el número de elementos en un array. Se puede acceder a esta propiedad con el nombre del array seguido de un punto y dicha propiedad. El siguiente ejemplo demuestra como utilizarlo:

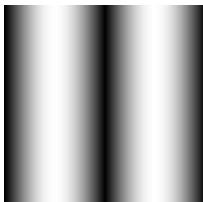
```
int[] data1 = { 19, 40, 75, 76, 90 };
int[] data2 = { 19, 40 };
int[] data3 = new int[127];
println(data1.length);      //Imprime "5" en la consola
println(data2.length);      //Imprime "2" en la consola
println(data3.length);      //Imprime "127" en la consola
```

Usualmente se utiliza una estructura FOR para acceder a los elementos del array, especialmente si es un array con muchos elementos:



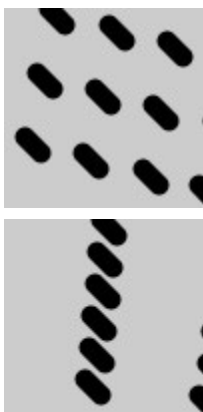
```
int[] data = { 19, 40, 75, 76, 90 };
for (int i = 0; i < data.length; i++) {
    line(data[i], 0, data[i], 100);
}
```

Una estructura FOR puede ser usada para agregar datos a un array. Por ejemplo, puede hacerse un calculo de valores y luego ser asignados como elementos de un array. El siguiente ejemplo almacena los valores devueltos por la función `sin()` en un array.



```
float[] sineWave = new float[width];
for (int i = 0; i < width; i++) {
    //Asigna valores al array devueltos de la función sin()
    float r = map(i, 0, width, 0, TWO_PI);
    sineWave[i] = abs(sin(r));
}
for (int i = 0; i < sineWave.length; i++) {
    //Establece el contorno leído del array
    stroke(sineWave[i] * 255);
    line(i, 0, i, height);
}
```

Almacenar las coordenadas de muchos elementos es otra forma de hacer un programa más fácil de leer y más manejable. En el siguiente ejemplo, el array `x[]` almacena la coordenada-x, y el array `vel[]` almacena el rango correspondiente a la misma. Escribir este programa sin la implementación de arrays requeriría 24 variables independientes.



```
int numLines = 12;
float[] x = new float[numLines];
float[] vel = new float[numLines];
float offset = 8; //Establece el espacio entre las líneas
void setup() {
    size(100, 100);
    smooth();
    strokeWeight(10);
    for (int i = 0; i < numLines; i++) {
        x[i] = i; //Establece posición inicial
        vel[i] = 0.1 + (i / offset);
        //Establece la velocidad inicial
    }
}
void draw() {
    background(204);
    for (int i = 0; i < x.length; i++) {
```

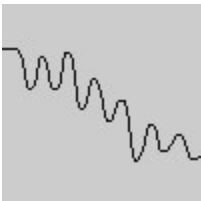
```

x[i] += vel[i]; //Actualiza posición
if (x[i] > (width + offset)) {
    //Si se va por la derecha
    x[i] = -offset * 2;
    //regresa por la izquierda
}
float y = i * offset; //Establece la coordenada-y
line(x[i], y, x[i]+offset, y+offset);
//Dibuja la línea
}
}

```

-Almacenando datos de Mouse

Una implementación frecuente de los arrays es el almacenamiento de los datos del mouse. Las variables pmouseX y pmouseY almacenan la posición anterior a la actual del mouse. Sin embargo, no hay una manera de acceder a esos datos que no sea inmediatamente. Con cada cuadro que pasa, mouseX y mouseY (y a sí mismo pmouseX y pmouseY) son reemplazados por nuevos valores, y los antiguos valores son descartados. Crear un array es la mejor manera de no perder esos valores y utilizarlos para algo. En el siguiente ejemplo, los últimos y mas recientes 100 valores de mouseY son almacenados y mostrados en pantalla como una línea, de izquierda a derecha.



```

int[] y;
void setup() {
    size(100, 100);
    y = new int[width];
}
void draw() {
    background(204);
    //Desplaza los valores a la derecha
    for (int i = y.length-1; i > 0; i--) {
        y[i] = y[i-1];
    }
    //Agrega nuevos valores al comienzo
    y[0] = constrain(mouseY, 0, height-1);
    //Muestra cada par de valores como una línea
    for (int i = 1; i < y.length; i++) {
        line(i, y[i], i-1, y[i-1]);
    }
}
}

```

Aplicando este mismo código simultáneamente a mouseX y mouseY:



```

int num = 50;
int[] x = new int[num];
int[] y = new int[num];
void setup() {
    size(100, 100);
    noStroke();
    smooth();
    fill(255, 102);
}
void draw() {
    background(0);
    //Desplaza los valores a la derecha
    for (int i = num-1; i > 0; i--) {
        x[i] = x[i-1];
        y[i] = y[i-1];
    }
}
}

```

```

    }
    //Agrega nuevos valores al inicio del array
    x[0] = mouseX;
    y[0] = mouseY;
    //Dibuja los círculos
    for (int i = 0; i < num; i++) {
        ellipse(x[i], y[i], i/2.0, i/2.0);
    }
}

```

El siguiente ejemplo produce los mismos resultados, pero utiliza un método mucho más óptimo. En lugar de ordenar los elementos del array en cada fotograma, el programa escribe los nuevos datos en la posición siguiente del array. Los elementos del array permanecen en la misma posición una vez que están escritos, pero se leen en un orden diferente en cada fotograma. La lectura comienza en el lugar de los elementos más antiguos y continúa hasta el final del array. Al final del array, el operador % se utiliza para volver de nuevo al principio. Esta técnica es especialmente útil con matrices muy grandes, para evitar la copia innecesaria de datos que pueden ralentizar el programa.

```

int num = 50;
int[] x = new int[num];
int[] y = new int[num];
int indexPosition = 0;

void setup() {
    size(100, 100);
    noStroke();
    smooth();
    fill(255, 102);
}

void draw() {
    background(0);
    x[indexPosition] = mouseX;
    y[indexPosition] = mouseY;
    //Ciclo entre 0 y el número de elementos
    indexPosition = (indexPosition + 1) % num;
    for (int i = 0; i < num; i++) {
        //Establece la posición del array a ser leída
        int pos = (indexPosition + i) % num;
        float radius = (num-i) / 2.0;
        ellipse(x[pos], y[pos], radius, radius);
    }
}

```

-Funciones del Array

Processing provee de un grupo de funciones para facilitar el trabajo con arrays. En estas páginas solo se explican cuatro de ellas, pero la información completa y detallada está disponible en línea, en www.processing.org/reference.

La función `append()` expande un array en un nuevo elemento, agrega los datos a la nueva posición y regresa el nuevo array:

```

String[] trees = { "ceniza", "roble" };
append(trees, "miel");           //INCORRECTO! No cambia el array
print(trees);                    //Imprime "ceniza roble"
println();
trees = append(trees, "miel");    //Agrega "miel" al final

```

```

print(trees); //Imprime "ceniza roble miel"
println();
//Agrega "corteza" al final del array y crea un nuevo array
//para almacenar los cambios
String[] moretrees = append(trees, "beech");
print(moretrees); //Imprime "ceniza roble miel corteza"

```

La función `shorten()` achica un array en un elemento al remover el último de ellos y regresar el array disminuido:

```

String[] trees = { "avellana", "coco", "higo"};
trees = shorten(trees); //Remueve el último elemento del Array
print(trees); //Imprime "avellana coco"
println();
trees = shorten(trees); //Remueve el último elemento del Array
print(trees); //Imprime "avellana"

```

La función `expand()` incrementa el tamaño de un array. Puede expandir el array en un tamaño específico, o incluso doblar su tamaño. Si el array necesita incluir una cantidad muy grande de nuevos elementos, es mucho más rápido y eficiente utilizar `expand()` en lugar de `append()`:

```

int[] x = new int[100]; //Array que almacena coordenadas de x
int count; //Almacena el número de la posición del array
void setup() {
    size(100, 100);
}
void draw() {
    x[count] = mouseX; //Asigna una nueva coordenada-x al array
    count++; //Incrementa el contador
    if (count == x.length) { //Si el array está completo
        x = expand(x); //doblar el tamaño del array
        println(x.length); //Escribir el nuevo tamaño en la consola
    }
}

```

Los valores de los arrays no pueden ser copiados con el operador porque ellos son objetos. El método clásico para copiar elementos de un array es utilizando una función especial para copiar el array entero y luego buscar el elemento deseado con un ciclo FOR. La función `arrayCopy()` es la manera más eficiente para copiar un array. Los datos son copiados del array utilizado como primer parámetro al array utilizado como segundo parámetro:

```

String[] norte = { "OH", "IN", "MI" };
String[] sur = { "GA", "FL", "NC" };
arraycopy(norte, sur); //Copia del array norte al array sur
print(sur); //Imprime "OH IN MI"
println();
String[] este = { "MA", "NY", "RI" };
String[] oeste = new String[este.length]; //Crea un nuevo array
arraycopy(este, oeste); //Copia del array este al array oeste
print(west); //Imprime "MA NY ir"

```

Nuevas funciones pueden ser escritas para perfeccionar el trabajo en array, pero hay que ser consciente de que actúan diferente si utilizan datos `int` o datos `char`. Cuando un array es usado como parámetro de una función, la dirección del array es transferida dentro de la función en lugar de los datos actuales. No se crea un nuevo array, y los cambios hechos en la función afectan solo al array usado como parámetro:

```

float[] data = { 19.0, 40.0, 75.0, 76.0, 90.0 };

void setup() {
    halve(data);
    println(data[0]);      //Imprime "9.5"
    println(data[1]);      //Imprime "20.0"
    println(data[2]);      //Imprime "37.5"
    println(data[3]);      //Imprime "38.0"
    println(data[4]);      //Imprime "45.0"
}

void halve(float[] d) {
    for (int i = 0; i < d.length; i++) {    //Para cada elemento del array
        d[i] = d[i] / 2.0;                  //dividir el valor por 2
    }
}

```

Cambiar el array en una función sin modificar los valores originales requiere algunas líneas más de código:

```

float[] data = { 19.0, 40.0, 75.0, 76.0, 90.0 };
float[] halfData;

void setup() {
    halfData = halve(data);                //Ejecuta la función
    println(data[0] + ", " + halfData[0]); //Imprime "19.0, 9.5"
    println(data[1] + ", " + halfData[1]); //Imprime "40.0, 20.0"
    println(data[2] + ", " + halfData[2]); //Imprime "75.0, 37.5"
    println(data[3] + ", " + halfData[3]); //Imprime "76.0, 38.0"
    println(data[4] + ", " + halfData[4]); //Imprime "90.0, 45.0"
}

float[] halve(float[] d) {
    float[] numbers = new float[d.length]; //Crea un nuevo array
    arraycopy(d, numbers);
    for (int i = 0; i < numbers.length; i++) { //Para cada elemento
        numbers[i] = numbers[i] / 2;          //divide el valor por 2
    }
    return numbers;                          //Regresa el nuevo array
}

```

-Arrays Bidimensionales

Los datos pueden ser almacenados y obtenidos de arrays con más de una dimensión. Usando la tabla al inicio de esta unidad, cambiaremos el array de una dimensión a dos dimensiones:

points	[0]	50	61	83	69	71	50	29	31	17	39
	[1]	18	37	43	60	82	73	82	60	43	37
		[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Los arrays en 2D son, esencialmente, una lista de arrays en 1D. Estos deben ser declarados, luego creador y recién después se pueden asignar valores. La siguiente sintaxis convierte este array a código:

```

int[][] points = { {50,18}, {61,37}, {83,43}, {69,60}, {71,82},
                  {50,73}, {29,82}, {31,60}, {17,43}, {39,37} };
println(points[4][0]); //Imprime 71
println(points[4][1]); //Imprime 82
println(points[4][2]); //ERROR! Este elemento está fuera del array
println(points[0][0]); //Imprime 50

```

```
println(points[9][1]);          //Imprime 37
```

Este programa muestra como utilizar todo esto en simultáneo:



```
int[][] points = { {50,18}, {61,37}, {83,43}, {69,60},  
                  {71,82}, {50,73}, {29,82}, {31,60},  
                  {17,43}, {39,37} };  
  
void setup() {  
  size(100, 100);  
  fill(0);  
  smooth();  
}  
  
void draw() {  
  background(204);  
  translate(mouseX - 50, mouseY - 50);  
  beginShape();  
  for (int i = 0; i < points.length; i++) {  
    vertex(points[i][0], points[i][1]);  
  }  
  endShape();  
}
```

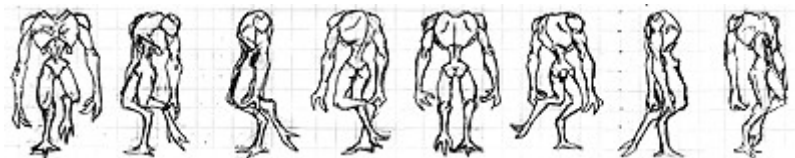
Es posible continuar y hacer arrays en 3D o 4D por sobre-explotar este recurso. Sin embargo, no es recomendable ya que los arrays multidimensionales suelen ser sumamente confusos. En casos en los que se requiera usar, se recomienda implementar arrays en 1D o 2D, no más que eso.

Imagen: Animación

La animación se produce cuando una serie de imágenes, cada una ligeramente diferente, se presentan en una sucesión rápida. Un medio diverso, con un siglo de historia, la animación ha progresado desde los experimentos iniciales de Winsor McCay a las innovaciones comerciales y realista de las primeras producciones de Walt Disney Studio, con películas experimentales por animadores como Lotte Reiniger y Whitney James a mediados del siglo XX. El alto volumen de los efectos especiales de animación en películas del tipo *live-action*, y la avalancha de películas infantiles de dibujos animados, están cambiando el papel de la animación en la cultura popular.

-Imágenes Secuenciales

Antes de que una serie de imágenes sea presentada como una secuencia, debemos cargarlas. La variable del tipo imagen debe ser declarada fuera del bloque `setup()` y `draw()`, y luego sus valores deben ser asignados en el `setup()`. Las siguientes imágenes pertenecen al usuario Eriance de DeviantArt (www.eriance.deviantart.com):



Y luego el código los dibuja de manera secuencial en orden numérico:



```
int numFrames = 8;      //Número de cuadros por animar
int frame = 0;         //Cuadro para mostrar al inicio
PImage[] images = new PImage[numFrames]; //Array de Imágenes
void setup() {
    size(100, 156);
    frameRate(10);
    //10 cuadros por segundo, en este caso (máximo 30)
    images[0] = loadImage("ani-000.gif");
    images[1] = loadImage("ani-001.gif");
    images[2] = loadImage("ani-002.gif");
    images[3] = loadImage("ani-003.gif");
    images[4] = loadImage("ani-004.gif");
    images[5] = loadImage("ani-005.gif");
    images[6] = loadImage("ani-006.gif");
    images[7] = loadImage("ani-007.gif");
}
void draw() {
    frame++;
    if (frame == numFrames) {
        frame = 0;
    }
    image(images[frame], 0, 0);
}
}
```

El siguiente ejemplo muestra una alternativa al código anterior, utilizando una estructura FOR para optimizar el proceso. Esas líneas de código pueden cargar entre 1 y 999 imágenes por cambiar el valor de la variable `numFrame`. La función `nf()` es usada para dar formato al nombre de la imagen a cargar. El operador `%` es usado para controlar el ciclo:


```

int numFrames = 8; //Número de cuadros por animar
PImage[] images = new PImage[numFrames]; //Array de Imágenes

void setup() {
  size(100, 156);
  frameRate(10); //10 cuadros por segundo, en este caso (máximo 30)
  //Automatiza la carga de imágenes. Números menores a 100
  //necesitan un cero extra.
  for (int i = 0; i < images.length; i++) {
    //Construye el nombre de la imagen a cargar
    String imageName = "ani-" + nf(i, 3) + ".gif";
    images[i] = loadImage(imageName);
  }
}

void draw() {
  //Calcula el cuadro para mostrar, usa % para generar el ciclo de cuadros
  int frame = frameCount % numFrames;
  image(images[frame], 0, 0);
}

```

Las imágenes pueden mostrarse en un orden aleatorio, lo cual agrega cierto interés al programa. Lo fundamental es reemplazar el tiempo por intervalos irregulares, en un orden aleatorio con tiempo aleatorio.

```

int numFrames = 8; //Número de cuadros por animar
PImage[] images = new PImage[numFrames];

void setup() {
  size(100, 100);
  for (int i = 0; i < images.length; i++) {
    String imageName = "ani-" + nf(i, 3) + ".gif";
    images[i] = loadImage(imageName);
  }
}

void draw() {
  int frame = int(random(0, numFrames)); //Cuadro por mostrar
  image(images[frame], 0, 0);
  frameRate(random(1, 60.0));
}

```

Hay muchas maneras de controlar la velocidad con la que se ejecuta una animación. La función `frameRate()` es una de las maneras más sencillas. El lugar donde se debe ubicar la función `frameRate()` es en el bloque `setup()`.

Si se desea que otros elementos se muevan de forma secuencial pero independientemente del movimiento principal, se debe establecer un temporalizador y avanzar al siguiente cuadro solo cuando el valor del temporalizador aumente como la variable predefinida. En el siguiente ejemplo, la animación que ocurre en la mitad superior es controlada por la función `frameRate()`. La animación en la mitad inferior es actualizada solo dos veces por segundo.



```

int numFrames = 8; //Número de cuadros para animar
int topFrame = 0; //El cuadro que se muestra arriba
int bottomFrame = 0; //El cuadro que se muestra abajo
PImage[] images = new PImage[numFrames];
int lastTime = 0;
void setup() {
  size(100, 100);
  frameRate(30);
}

```



```

for (int i = 0; i < images.length; i++) {
    String imageName = "ani-" + nf(i, 3) + ".gif";
    images[i] = loadImage(imageName);
}
}
void draw() {
    topFrame = (topFrame + 1) % numFrames;
    image(images[topFrame], 0, 0);
    if ((millis() - lastTime) > 500) {
        bottomFrame = (bottomFrame + 1) % numFrames;
        lastTime = millis();
    }
    image(images[bottomFrame], 0, 50);
}
}

```

-Imágenes en Movimiento

Mover una imagen, en lugar de presentar una secuencia, es otro enfoque de la animación de imágenes. Las mismas técnicas para la creación de movimientos han sido ya presentadas en unidades anteriores. El ejemplo siguiente mueve una imagen de izquierda a derecha, volviendo a la izquierda cuando se esté fuera del borde de la pantalla:



```

PImage img;
float x;
void setup() {
    size(100, 100);
    img = loadImage("mov-001.gif");
}
void draw() {
    background(255);
    x += 0.5;
    if (x > width) {
        x = -width;
    }
    image(img, x, 0);
}
}

```

Las funciones de transformación también pueden aplicarse a las imágenes. Estas pueden ser trasladadas, rotadas y escaladas, para producir la ilusión de movimiento en el tiempo. En el siguiente ejemplo, las imagen gira alrededor del centro de la ventana de representación:



```

PImage img;
float angulo;
void setup() {
    size(100, 100);
    img = loadImage("mov-002.gif");
}
void draw() {
    background(204);
    angulo += 0.01;
    translate(50, 50);
    rotate(angulo);
    image(img, -100, -100);
}
}

```

Las imágenes también pueden ser animadas por cambiar sus propiedades de dibujos. En este ejemplo, la opacidad de la imagen oscila produciendo transparencia.



```
PImage img;
float opacidad = 0;      //Establece la opacidad inicial
void setup() {
  size(100, 100);
  img = loadImage("mov-003.gif");
}
void draw() {
  background(0);
  if (opacity < 255) {   //Cuando sea menos que el máximo,
    opacidad += 0.5;     //Incrementa la opacidad
  }
  tint(255, opacidad);
  image(img, 0, 0);
}
```

Unidad 31

Imagen: Píxeles

Elementos que se introducen en esta Unidad:

get(), set()

En la primer unidad que trata el tema de imágenes, se define a una imagen como una cuadrícula rectangular de píxeles en la que cada elemento es un número que especifica un color. Debido a que la pantalla en sí es una imagen, los píxeles individuales se definen también como números. Por lo tanto, los valores de color de los píxeles individuales se pueden leer y cambiar.

-Leyendo Píxeles

Cuando un programa de Processing se ejecuta, lo primero que inicia es la ventana de representación, establecida con los valores que se incluyen en `size()`. El programa gana el control de dicha área y establece el color de cada píxel.

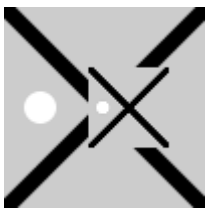
La función `get()` puede leer los datos de cualquier píxel de la pantalla. Hay tres versiones de dicha función, una para cada uso:

```
get()
get(x, y)
get(x, y, ancho, alto)
```

Si `get()` es usado sin parámetros, este hará una impresión completa de pantalla, y la regresará como una `PImage`. La versión con dos parámetros regresa el color de un píxel particular, localizado en la posición `x` e `y`, respectivamente. Un área rectangular es regresada si se utilizan los parámetros para `ancho` y `alto`. Si `get()` se utiliza sin parámetros, la captura que devuelve debe ser guardada en una variable `PImage`. Luego, dicha captura puede utilizarse como una imagen cualquiera.



```
strokeWeight(8);
line(0, 0, width, height);
line(0, height, width, 0);
PImage cross = get(); //Obtiene una captura entera
image(cross, 0, 50); //Dibuja la imagen en una nueva posición
```



```
smooth();
strokeWeight(8);
line(0, 0, width, height);
line(0, height, width, 0);
noStroke();
ellipse(18, 50, 16, 16);
PImage cross = get();
//Obtiene una captura entera
image(cross, 42, 30, 40, 40);
//Cambia el tamaño de la captura a 40x40
```



```
strokeWeight(8);
line(0, 0, width, height);
line(0, height, width, 0);
PImage slice = get(0, 0, 20, 100); //Obtiene una captura entera
set(18, 0, slice);
set(50, 0, slice);
```

La función `get()` obtiene los píxeles actuales mostrados en pantalla. Eso quiere decir que no solo puede ser una captura de formas del programa, sino también de una imagen que esté cargada.



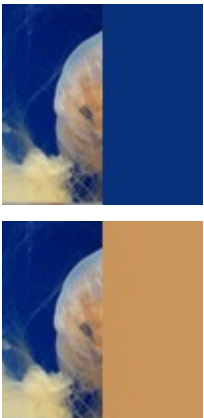
```
PImage medusa;
medusa = loadImage("medusa.jpg");
image(medusa, 0, 0);
PImage crop = get(); //Obtiene una captura entera
image(crop, 0, 50); //Dibuja la imagen en la nueva posición
```

Cuando es usado con los parámetros `x` e `y`, la función `get()` regresa un valor que debe ser asignado a una variable del tipo `color`. Ese valor puede ser utilizado para establecer el color de borde o de relleno de una figura. En el siguiente ejemplo, el color obtenido es usado como relleno de un rectángulo:



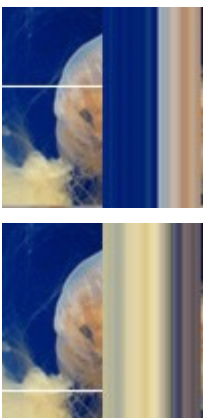
```
PImage medusa;
medusa = loadImage("medusa.jpg");
noStroke();
image(medusa, 0, 0);
color c = get(90, 80);
//Obtiene el color del Píxel en el (90, 80)
fill(c);
rect(20, 30, 40, 40);
```

Los valores devueltos por el mouse pueden ser usados como parámetros de `get()`. Esto le permite al cursor seleccionar colores de la ventana de representación:



```
PImage medusa;
void setup() {
  size(100, 100);
  noStroke();
  medusa = loadImage("medusa.jpg");
}
void draw() {
  image(medusa, 0, 0);
  color c = get(mouseX, mouseY);
  fill(c);
  rect(50, 0, 50, 100);
}
```

La función `get()` puede ser utilizada con un ciclo `FOR` para hacer poder identificar varios colores o un grupo de píxeles.



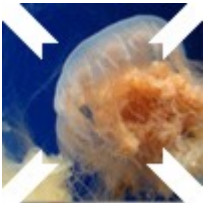
```
PImage medusa;
int y = 0;
void setup() {
  size(100, 100);
  medusa = loadImage("medusa.jpg");
}
void draw() {
  image(medusa, 0, 0);
  y = constrain(mouseY, 0, 99);
  for (int i = 0; i < 49; i++) {
    color c = get(i, y);
    stroke(c);
  }
}
```

```

        line(i+50, 0, i+50, 100);
    }
    stroke(255);
    line(0, y, 49, y);
}

```

Cada `PImage` tiene su propia función `get()` para conseguir píxeles de la imagen. Esto permite a los píxeles ser obtenidos de la imagen, independientemente de los píxeles que se muestran en pantalla. Ya que `PImage` es un objeto, se puede acceder a su función `get()` con el operador del punto, como se muestra a continuación:



```

PImage medusa;
medusa = loadImage("medusa.jpg");
stroke(255);
strokeWeight(12);
image(medusa, 0, 0);
line(0, 0, width, height);
line(0, height, width, 0);
PImage medusaCrop = medusa.get(20, 20, 60, 60);
image(medusaCrop, 20, 20);

```

-Escribiendo Píxeles

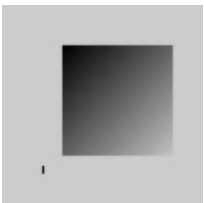
Los píxeles en la ventana de presentación de Processing pueden ser escritos directamente con la función `set()`. Hay dos versiones de esta función, ambas con tres parámetros:

```

set(x, y, color)
set(x, y, imagen)

```

Cuando el tercer parámetro es un `color`, `set()` cambia el color de un píxel de la ventana de representación. Cuando el tercer parámetro es una imagen, `set()` escribe una imagen en la posición de `x` e `y`.

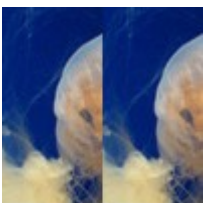


```

color negro = color(0);
set(20, 80, negro);
set(20, 81, negro);
set(20, 82, negro);
set(20, 83, negro);
for (int i = 0; i < 55; i++) {
    for (int j = 0; j < 55; j++) {
        color c = color((i+j) * 1.8);
        set(30+i, 20+j, c);
    }
}

```

La función `set()` puede escribir una imagen para mostrar en la ventana de representación. Usar `set()` es mucho más veloz que utilizar `image()`, ya que los píxeles son copiados directamente. Sin embargo, las imágenes dibujadas con `set()` no pueden ser escaladas ni pintadas, y no son afectadas por las funciones de transformación.



```

PImage medusa;
void setup() {
    size(100, 100);
    medusa = loadImage("medusa.jpg");
}
void draw() {

```



```
int x = constrain(mouseX, 0, 50);  
set(x, 0, medusa);  
}
```

Cada variable del tipo `PImage` tiene su propia función `set()`, para escribir los píxeles directamente a la imagen. Esto permite a los píxeles ser escritos en una imagen independientemente de los píxeles que se muestran en la ventana de representación. Ya que `PImage` es una imagen, la función `set()` se ejecuta con el nombre de la imagen y el operador de punto.



```
PImage medusa;  
medusa = loadImage("medusa.jpg");  
background(0);  
color blanco = color(255);  
medusa.set(0, 50, blanco);  
medusa.set(1, 50, blanco);  
medusa.set(2, 50, blanco);  
medusa.set(3, 50, blanco);  
image(medusa, 20, 0);
```

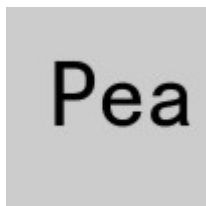
Tipografía: Movimiento

A pesar del potencial para el tipo de cinética en el cine, la tipografía animada no comenzó hasta la década de 1950, con el título de la película del trabajo de Saul Bass, Binder Maurice, y Pablo Ferro. Estos diseñadores y sus compañeros, lograron efectos de calidad muy alta con sus secuencias de títulos cinéticos en películas como *North by Northwest* (1959), *Dr. No* (1962), y *Bullitt* (1968). Entonces, se exploró el poder evocador de la cinética en forma de las letras para establecer un estado de ánimo y expresar niveles adicionales de significado en relación con el lenguaje escrito. En los años siguientes el diseño de títulos de película ha madurado y ha sido aumentado por la tipografía experimental para la televisión y el Internet.

-Palabras en Movimiento

Para que la tipografía se mueva, es necesario que el programa se ejecute continuamente. Por lo tanto, es necesario utilizar una estructura `draw()`. Usar la tipografía con `draw()` requiere tres pasos. Primero, la variable `PFont` debe ser declarada fuera de los bloques `setup()` y `draw()`. Luego, la fuente debe ser cargada y establecida en el bloque `setup()`. Finalmente, la fuente puede ser usada en el bloque `draw()` a través de la función `text()`.

El siguiente ejemplo utiliza una fuente llamada KaiTi. Para probar dicho ejemplo es necesario crear tu propia fuente con la herramienta "Create Font" y luego cambiar el nombre del parámetro de `loadFont()` por el indicado:



```
PFont font;
String s = "Pea";
void setup() {
  size(100, 100);
  font = loadFont("KaiTi-48.vlw");
  textFont(font);
  fill(0);
}
void draw() {
  background(204);
  text(s, 22, 60);
}
```

Para ponerla en movimiento, simplemente dibujaremos un cambio de posición en cada cuadro. Las palabras pueden moverse en un orden en un orden regular si sus valores son cambiados en cada cuadro, y también puede moverse sin un orden aparente por recibir valores aleatorios.



```
PFont font;
float x1 = 0;
float x2 = 100;
void setup() {
  size(100, 100);
  font = loadFont("KaiTi-48.vlw");
  textFont(font);
  fill(0);
}
void draw() {
  background(204);
  text("Derecha", x1, 50);
  text("Izquierda", x2, 100);
  x1 += 1.0;
  if (x1 > 150) {
    x1 = -200;
  }
}
```

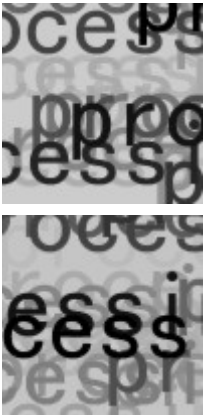


```

    }
    x2 -= 0.8;
    if (x2 < -200) {
        x2 = 150;
    }
}

PFont font;
void setup() {
    size(100, 100);
    font = loadFont("KaiTi-48.vlw");
    textFont(font);
    noStroke();
}
void draw() {
    fill(204, 24);
    rect(0, 0, width, height);
    fill(0);
    text("processing", random(-100, 100), random(-20, 120));
}

```



La tipografía no necesita moverse en orden para cambiar en el tiempo. Muchos de los parámetros de transformaciones, como la opacidad, puede utilizarse en el texto. Es posible cambiando el valor de una variable en el draw().

```

PFont font;
int opacidad = 0;
int direccion = 1;
void setup() {
    size(100, 100);
    font = loadFont("KaiTi-28.vlw");
    textFont(font);
}
void draw() {
    background(204);
    opacidad += 2 * direccion;
    if (( opacidad < 0) || ( opacidad > 255)) {
        direccion = -direccion;
    }
    fill(0, opacidad);
    text("fundido", 4, 60);
}

```

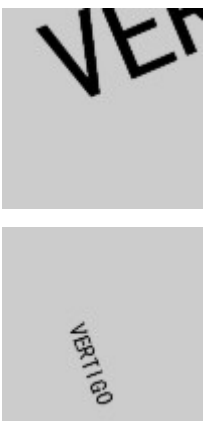


Al aplicar las funciones de translate(), rotate() y scale() también pueden producirse movimientos:

```

PFont font;
String s = "VERTIGO";
float angulo = 0.0;
void setup() {
    size(100, 100);
    font = loadFont("KaiTi-48.vlw");
    textFont(font, 24);
    fill(0);
}
void draw() {
    background(204);
    angulo += 0.02;
    pushMatrix();

```



```

        translate(33, 50);
        scale((cos(angulo/4.0) + 1.2) * 2.0);
        rotate(angulo);
        text(s, 0, 0);
        popMatrix();
    }

```

Otra técnica de representación, llamada presentación rápida de visualización serial (RSVP), muestra palabras en la ventana de forma secuencial y provee distintas formas de pensar como leerlo.



```

PFont font;
String[] palabras = { "Tres", "golpes", "y", "estas",
                    "fuera", " "};
int cualPalabra = 0;
void setup() {
    size(100, 100);
    font = loadFont("KaiTi-28.vlw");
    textFont(font);
    textAlign(CENTER);
    frameRate(4);
}
void draw() {
    background(204);
    cualPalabra++;
    if (cualPalabra == palabras.length) {
        cualPalabra = 0;
    }
    text(palabras[cualPalabra], width/2, 55);
}

```

-Letras en Movimiento

Individualmente, las letras ofrecen mayor flexibilidad a la hora de animar que las palabras. Construir palabras letra por letra, y darle un movimiento individual a cada una de ellas, puede convertirse en una tarea muy ardua. Trabajar de esta manera requiere más paciencia y dedicación que programas de otras características, pero los resultados son altamente satisfactorios.



```

//El tamaño de cada letra oscila su tamaño
//de izquierda a derecha
PFont font;
String s = "AREA";
float angulo = 0.0;
void setup() {
    size(100, 100);
    font = loadFont("KaiTi-48.vlw");
    textFont(font);
    fill(0);
}
void draw() {
    background(204);
    angulo += 0.1;
    for (int i = 0; i < s.length(); i++) {
        float c = sin( angulo + i/PI);
        textSize((c + 1.0) * 32 + 10);
        text(s.charAt(i), i*26, 60);
    }
}

```

h

o

hola

```
//Cada letra entra desde abajo en secuencia
//y se detiene cuando la acción es concluída
PFont font;
String palabra = "hola";
char[] letras;
float[] y; //Coordenada-Y para cada letra
int letraActual = 0; //Letra actual en movimiento
void setup() {
    size(100, 100);
    font = loadFont("KaiTi-20.vlw");
    textFont(font);
    letras = palabra.toCharArray();
    y = new float[letras.length];
    for (int i = 0; i < letras.length; i++) {
        y[i] = 130; //Posición fuera de la pantalla
    }
    fill(0);
}
void draw() {
    background(204);
    if (y[letraActual] > 35) {
        y[letraActual] -= 3; //Mueve la letra actual
        //hacia arriba
    }
    else {
        if (letraActual < letras.length-1) {
            letraActual++; //Cambia a la siguiente letra
        }
    }
    //Calcula el valor de x para centrar el texto
    float x = (width - textWidth(palabra)) / 2;
    for (int i = 0; i < letras.length; i++) {
        text(letras[i], x, y[i]);
        x += textWidth(letras[i]);
    }
}
```

Tipografía: Respuesta


Muchas personas pasan horas al día escribiendo cartas en las computadoras, pero esta acción se ve muy limitada. ¿Qué características se podría agregar a un editor de texto para que sea más sensible a la mecanógrafa? Por ejemplo, la velocidad de la escritura podría disminuir el tamaño de las letras, o una larga pausa en la escritura podría añadir muchos espacios, imitando así una pausa de una persona mientras habla. ¿Qué pasa si el teclado puede registrar la dificultad con que una persona escribe (la forma de quien toca una nota suave del piano, cuando se pulsa una tecla con suavidad) y automáticamente puede asignar atributos como la *cursiva para las expresar suavidad* y **negrita para expresas fuerza**? Estas analogías se sugieren cómo conservadoras del software actual que trata a la tipografía y mecanografía.

-Palabras responsivas

A los elementos tipográficos se le pueden asignar los comportamientos que definen una personalidad en relación con el mouse o el teclado. Una palabra puede expresar la agresión moviéndose rápidamente hacia el cursor, o uno que se aleja poco a poco puede expresar timidez.



```
//La palabra "evitar" se aleja del mouse porque su
//posición se establece como la inversa del cursor
PFont f;
void setup() {
  size(100, 100);
  f = loadFont("KaiTi-20.vlw");
  textFont(f);
  textAlign(CENTER);
  fill(0);
}
void draw() {
  background(204);
  text("evitar", width-mouseX, height-mouseY);
}
```

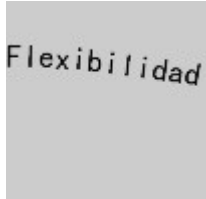



```
//La palabra "risa" vibra cuando el cursor se le pone encima
PFont f;
float x = 33; //Coordenada-X del texto
float y = 60; //Coordenada-Y del texto
void setup() {
  size(100, 100);
  f = loadFont("KaiTi-20.vlw");
  textFont(f);
  noStroke();
}
void draw() {
  fill(204, 120);
  rect(0, 0, width, height);
  fill(0);
  //Si el cursor está sobre el texto, cambia su posición
  if ((mouseX >= x) && (mouseX <= x+55) &&
      (mouseY >= y-24) && (mouseY <= y)) {
    x += random(-5, 5);
    y += random(-5, 5);
  }
  text("risa", x, y);
}
```



-Letras responsivas

Separar una palabra en las letras que lo componen crea más opciones para determinar su respuesta a la del mouse o el teclado. Las letras independientes tienen la capacidad de responder de una manera diferente y contribuyen a la respuesta total de la palabra. Los dos ejemplos siguientes muestran esta técnica. El método `toCharArray()` se utiliza para extraer los caracteres individuales de una variable `String`. El método `charAt()` es una forma alternativa para aislar las letras individuales dentro de un `String`.



```
//La posición horizontal del mouse determina la rotación
//del ángulo. El ángulo aumenta con cada letra, lo que
//permite hacer una curva tipográfica.
String palabra = "Flexibilidad";
PFont f;
char[] letras;
void setup() {
    size(100, 100);
    f = loadFont("KaiTi-16.vlw");
    textFont(f);
    letras = palabra.toCharArray();
    fill(0);
}
void draw() {
    background(204);
    pushMatrix();
    translate(0, 33);
    for (int i = 0; i < letras.length; i++) {
        float angulo = map(mouseX, 0, width, 0, PI/8);
        rotate(angulo);
        text(letras[i], 0, 0);
        //Compensación por el ancho de la letra actual
        translate(textWidth(letras[i]), 0);
    }
    popMatrix();
}
```



```
//Calcula el tamaño de cada letra basada en la
//posición del cursor. Las letras son mas grandes
//Cuando el cursor está cerca.
String palabra = "BULTO";
char[] letras;
float totalOffset = 0;
PFont font;
void setup() {
    size(100, 100);
    font = loadFont("KaiTi-20.vlw");
    textFont(font);
    letras = palabra.toCharArray();
    textAlign(CENTER);
    fill(0);
}
void draw() {
    background(204);
    translate((width - totalOffset) / 2, 0);
    totalOffset = 0;
    float primerAncho = (width / letras.length) / 4.0;
    translate(primerAncho, 0);
    for (int i = 0; i < letras.length; i++) {
        float distancia = abs(totalOffset - mouseX);
        distancia = constrain(distancia, 24, 60);
```

```
        textSize(84 - distancia);
        text(letras[i], 0, height - 2);
        float letrasAncho = textWidth(letras[i]);
        if (i != letras.length-1) {
            totalOffset = totalOffset + letrasAncho;
            translate(letrasAncho, 0);
        }
    }
}
```

Unidad 34

Color: Componentes





















Elementos que se introducen en esta Unidad:

red(), blue(), green(), alpha(), hue(), saturation(), brightness()

Los colores son almacenados en el software como números. Cada color se define por los elementos que lo componen. Cuando el color es definido por los valores RGB, hay tres números que almacenan los componentes de rojo, verde y azul, y un cuarto número opcional que almacena un valor de transparencia. Cuando se trabaja con los valores HSB, los tres números almacenan el tono, saturación, y los valores de brillo y un cuarto indica la transparencia. El color visible es una combinación de estos componentes. El ajuste de las propiedades de colores individuales en forma aislada de los demás es una técnica útil para cambiar dinámicamente un valor de un solo color o la paleta entera de un programa.

-Extrayendo Color

En Processing, una variable del tipo *color* simplemente almacena un valor individual de un componente del color. Este valor combina los componentes rojo, verde, azul y alfa. Técnicamente, este valor es un *int* (entero) y puede utilizarse dinámicamente como se ha visto en unidades anteriores. La variable *color* almacena valores entre 0 y 255 que luego puede utilizarse como componentes de color. La siguiente tabla muestra una abstracción de lo dicho:

rojo	verde	azul	alfa		color
				→	
64	124	188	255		
				→	
151	186	66	255		
				→	
214	124	43	255		
				→	
214	124	43	126		

Las funciones `red()`, `green()` y `blue()`, son usadas para leer los componentes de color. La función `red()` extrae el componente rojo. La función `green()` extrae el componente verde. La función `blue()` extrae el componente azul.

```
color c1 = color(0, 126, 255);           //Crea un nuevo color
float r = red(c1);                      //Asigna 0.0 a r
float g = green(c1);                    //Asigna 126.0 a g
float b = blue(c1);                     //Asigna 255.0 a b
println(r + ", " + g + ", " + b);       //Imprime "0.0, 126.0, 255.0 "
color c2 = color(102);                  //Crea un valor de gris
float r2 = red(c2);                     //Asigna 102.0 a r2
float g2 = green(c2);                   //Asigna 102.0 a g2
float b2 = blue(c2);                    //Asigna 102.0 a b2
println(r2 + ", " + g2 + ", " + b2);    //Imprime "102.0, 102.0, 102.0"
```

La función `alpha()` extrae el componente alfa de un color. Recordar agregar un cuarto valor a `color()`, de lo contrario, alfa será siempre 255 por defecto.

```

color c = color(0, 51, 102);           //Crea un nuevo color
color g = color(0, 126, 255, 220);    //Crea un nuevo color
float a = alpha(c);                   //Asigna 255.0 a a
float b = alpha(g);                   //Asigna 220.0 a b
println(a + ", " + b);                //Imprime "255.0, 220.0"

```

Las funciones `hue()`, `saturation()` y `brightness()` funcionan como `red()`, `green()` y `blue()` pero en el modo HSB de color. Respectivamente, extraen el valor de tono, saturación y brillo. Recordar cambiar de modo color a modo HSB antes de utilizar dichas funciones. Sin embargo, hay veces que se desea obtener alguno de esos valores en el modo por defecto, y esto es posible en Processing.

```

colorMode(HSB, 360, 100, 100);        //Establece el modo de color como HSB
color c = color(210, 100, 40);        //Crea un nuevo color
float h = hue(c);                     //Asigna 210.0 a h
float s = saturation(c);              //Asigna 100.0 a s
float b = brightness(c);             //Asigna 40.0 a b
println(h + ", " + s + ", " + b);    //Imprime "210.0, 100.0, 40.0"

```

```

color c = color(217, 41, 117);        //Crea un nuevo color
float r = red(c);                     //Asigna 217.0 a r
float h = hue(c);                     //Asigna 236.64774 a h
println(r + ", " + h);                //Imprime "217.0, 236.64774"

```

Los valores de estas funciones son escalas basadas en el modo de color. Si el rango de color es cambiado con `colorMode()`, el valor regresado será escalado con el nuevo rango.

```

colorMode(RGB, 1.0);                 //Establece el modo de color como RGB
color c = color(0.2, 0.8, 1.0);      //Crea un nuevo color
float r = red(c);                    //Asigna 0.2 a r
float h = hue(c);                    //Asigna 0.5416667 a h
println(r + ", " + h);                //Asigna "0.2, 0.5416667"

```

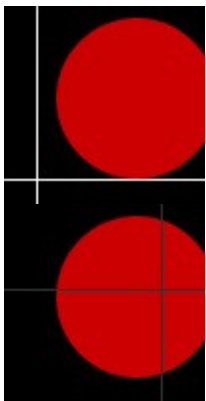
Los valores regresados por dichas funciones son todos valores del tipo `float` (decimal), por lo tanto dará un error si se trata de asignar a una variable tipo `int`. Siempre pueden utilizarse las funciones de conversión si se lo desea.

```

color c = color(118, 22, 24);        //Crea un nuevo color
int r1 = red(c);                     //ERROR! red() regresa un valor decimal
float r2 = red(c);                   //Asigna 118.0 a r2
int r3 = int(red(c));                 //Asigna 118 a r3

```

Como se discutió en las unidades de imágenes, estas funciones se utilizan para leer píxeles individualmente en la pantalla. En el siguiente ejemplo, la función `get()` es utilizada para acceder al color actual de donde se encuentra la posición del cursor.



```

//Establece el color de las líneas cuando el
//Componente rojo esta sobre el cursor
void setup() {
    size(100, 100);
    smooth();
    fill(204, 0, 0);
}
void draw() {
    background(0);
    noStroke();
    ellipse(66, 46, 80, 80);
}

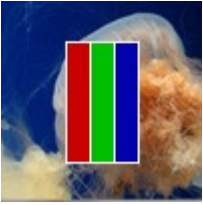
```



```

    color c = get(mouseX, mouseY);
    float r = red(c);          //Extrae el componente rojo
    stroke(255-r);            //Establece el contorno
    line(mouseX, 0, mouseX, height);
    line(0, mouseY, width, mouseY);
}

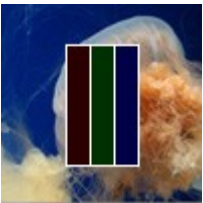
```



```

//Simula la composición cromática de un píxel en un
//panel plano
PImage medusa;
void setup() {
    size(100, 100);
    medusa = loadImage("medusa.jpg");
    stroke(255);
}

```

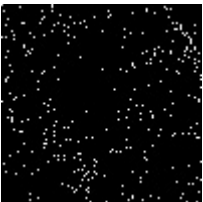


```

void draw() {
    background(medusa);
    color c = get(mouseX, mouseY);
    float r = red(c);          //Extrae Rojo
    float g = green(c);       //Extrae Verde
    float b = blue(c);        //Extrae Azul
    fill(r, 0, 0);
    rect(32, 20, 12, 60);     //Componente Rojo
    fill(0, g, 0);
    rect(44, 20, 12, 60);     //Componente Verde
    fill(0, 0, b);
    rect(56, 20, 12, 60);     //Componente Azul
}

```

Las funciones `red()`, `green()` y `blue()` pueden ser utilizadas de diferentes maneras. Por ejemplo, los valores que devuelven pueden utilizarse para controlar el movimiento de las formas y figuras. En el siguiente ejemplo, se utiliza el valor de brillo de una imagen para controlar la velocidad de 400 puntos en pantalla.



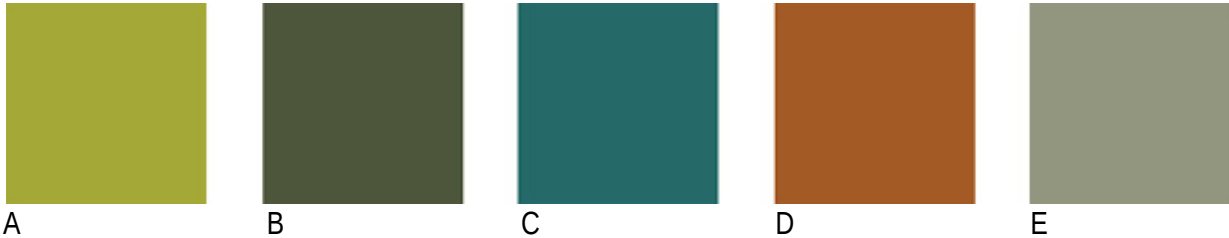
```

int num = 400;
float[] x = new float[num];
float[] y = new float[num];
PImage img;
void setup() {
    size(100, 100);
    img = loadImage("medusa.jpg");
    for (int i = 0; i < num; i++) {
        x[i] = random(width);
        y[i] = random(height);
    }
    stroke(255);
}
void draw() {
    background(0);
    for (int i = 0; i < num; i++) {
        color c = img.get(int(x[i]), int(y[i]));
        float b = brightness(c) / 255.0;
        float vel = pow(b, 2) + 0.05;
        x[i] += vel;
        if (x[i] > width) {
            x[i] = 0;
            y[i] = random(height);
        }
        point(x[i], y[i]);
    }
}
}

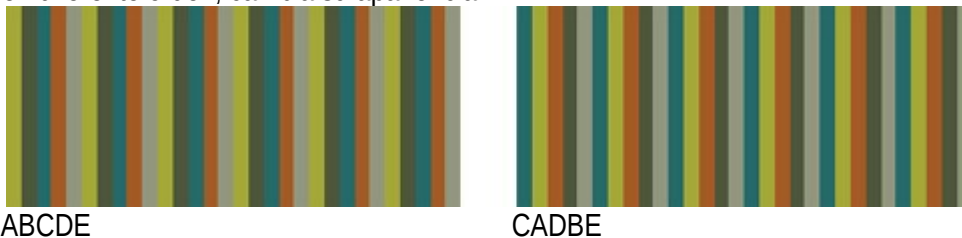
```

-Paletas de color Dinámico

Uno de los conceptos más importantes en el trabajo con el color es la relatividad. Cuando un color se coloca al lado de otro, ambos parecen cambiar. Si el color tiene el mismo aspecto en una yuxtaposición, a menudo deben ser diferentes (definidos con diferentes números). Esto es importante para considerar cuando se trabaja con el color en el software, ya que los elementos son a menudo el movimiento y el cambio de colores. Por ejemplo, la colocación de estos cinco colores...



...en diferente orden, cambia su apariencia:



En el siguiente ejemplo, los colores almacenados en una variable `oliva` y `gris` siempre son los mismos, mientras que los valores de `amarillo` y `naranja` se cambian dinámicamente en relación a la posición `mouseY`.



```
color oliva, gris;
void setup() {
  size(100, 100);
  colorMode(HSB, 360, 100, 100, 100);
  noStroke();
  smooth();
  oliva = color(75, 61, 59);
  gris = color(30, 17, 42);
}
void draw() {
  float y = mouseY / float(height);
  background(gris);
  fill(oliva);
  quad(70 + y*6, 0, 100, 0, 100, 100, 30 - y*6, 100);
  color amarillo = color(48 + y*20, 100, 88 - y*20);
  fill(amarillo);
  ellipse(50, 45 + y*10, 60, 60);
  color naranja = color(29, 100, 83 - y*10);
  fill(naranja);
  ellipse(54, 42 + y*16, 24, 24);
}
```

Una buena técnica para crear paletas de colores es utilizando imágenes y extraer sus colores con la función `get()`. Dependiendo de los objetivos, se puede cargar una imagen fotográfica o una que se ha construido píxel por píxel. Una imagen de cualquier dimensión puede ser cargada y usado como una paleta de colores. A

veces es conveniente utilizar pocos colores.

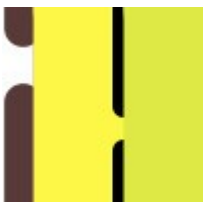


```
PImage img;
void setup() {
  size(100, 100);
  smooth();
  frameRate(0.5);
  img = loadImage("medusa.jpg");
}
void draw() {
  background(0);
  for (int x = 0; x < img.width; x++) {
    for (int y = 0; y < img.height; y++) {
      float xpos1 = random(x*10);
      float xpos2 = width - random(y*10);
      color c = img.get(x, y);
      stroke(c);
      line(xpos1, 0, xpos2, height);
    }
  }
}
```



```
PImage img;
void setup() {
  size(100, 100);
  noStroke();
  img = loadImage("medusa.jpg");
}
void draw() {
  int ix = int(random(img.width));
  int iy = int(random(img.height));
  color c = img.get(ix, iy);
  fill(c, 102);
  int xgrid = int(random(-2, 5)) * 25;
  int ygrid = int(random(-2, 5)) * 25;
  rect(xgrid, ygrid, 40, 40);
}
```

Al cargar los colores de la imagen en un array se abren más posibilidades. Una vez que los colores se encuentran en un array, pueden ser fácilmente reorganizados o cambiados. En el siguiente ejemplo, los valores de color de la imagen se cargan de forma secuencial en un array y luego son reordenados de acuerdo con su brillo. Una función, que llamaremos `tipoColor()`, toma una variedad de colores como entrada, los pone en orden de oscuro a claro, y luego regresa los colores ordenados. Ya que cuenta de 0 a 255, pone todos los colores con el valor actual del array sin ordenar el nuevo array.



```
PImage img;
color[] imageColors;
void setup() {
  size(100, 100);
  frameRate(0.5);
  smooth();
  noFill();
  img = loadImage("colores.jpg");
  imageColors = new color[img.width*img.height];
  for (int y = 0; y < img.height; y++) {
    for (int x = 0; x < img.width; x++) {
      imageColors[y*img.height + x] =
        img.get(x, y);
    }
  }
}
```

```

        }
    }
    imageColors = tipoColor(imageColors);
}
void draw() {
    background(255);
    for (int x = 10; x < width; x += 10) {
        int r = int(random(imageColors.length));
        float thick = ((r) / 4.0) + 1.0;
        stroke(imageColors[r]);
        strokeWeight(thick);
        line(x, height, x, height-r+thick);
        line(x, 0, x, height-r-thick);
    }
}
color[] tipoColor(color[] colors) {
    color[] sorted = new color[colors.length];
    int num = 0;
    for (int i = 0; i <= 255; i++) {
        for (int j = 0; j < colors.length; j++) {
            if (int(brightness(colors[j])) == i) {
                sorted[num] = colors[j];
                num++;
            }
        }
    }
    return sorted;
}
}

```

Imagen: Filtro, Mezcla, Copia, Máscara

Elementos que se introducen en esta Unidad:

`filter()`, `blend()`, `blendColor()`, `copy()`, `mask()`

Las imágenes digitales tienen el potencial notable para ser fácilmente re-configuradas y combinarse con otras imágenes. El software ahora simula un proceso de operaciones de consumo que antes eran la preocupación de un cuarto oscuro con luz y química. Cada píxel de una imagen digital es un conjunto de números que se pueden agregar o multiplicar. Algunos de estos cálculos se basan en la aritmética y otros utilizan las matemáticas más complejas de procesamiento de señales, pero los resultados visuales son lo más importantes. Los programas de software tales como el GNU Image Manipulation Program (GIMP) y Adobe Photoshop, han hecho posible la realización de muchos de los cálculos más comunes y útiles, sin pensar en las matemáticas detrás de los efectos. Estos programas permiten a los usuarios realizar fácilmente las operaciones técnicas, tales como la conversión de imágenes de colores RGB a escala de grises, aumentando el contraste de una imagen, produciendo un balance de color u otra clase de ajustes. Un filtro puede desenfocar una imagen, imitar la solarización, o simular los efectos de acuarela. Las acciones de filtrado de ficción, mezcla, y la copia se puede controlar fácilmente con el código para producir cambios notables. Estas técnicas pueden ser demasiado lenta para su uso en animación en tiempo real.

-Filtrando y Mezclando

Processing provee de funciones para filtrar y mezclar imágenes mostradas en la ventana de representación. Estas operan por alterar los píxeles de una imagen o utilizar como parámetro dos de estas. La función `filter()` posee dos versiones:

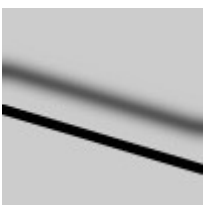
```
filter(modo)
filter(modo, nivel)
```

Existen ocho opciones diferentes para *modo*: THRESHOLD, GRAY, INVERT, POSTERIZE, BLUR, OPAQUE, ERODE, o DILATE. Algunos de estos modos requieren que se les establezca un *nivel*. El siguiente ejemplo aplica el filtro THRESHOLD a una imagen, con un nivel de 0.7.



```
PImage img = loadImage("medusa.jpg");
image(img, 0, 0);
filter(THRESHOLD, 0.7);
```

La función `filter()` afecta solo a lo que ya ha sido dibujado. Por ejemplo, si un programa dibuja dos líneas, y un desenfoque es creado luego de que una línea es dibujada, afectará únicamente a la primera.



```
smooth();
strokeWeight(5);
noFill();
line(0, 30, 100, 60);
filter(BLUR, 3);
line(0, 50, 100, 80);
```

Al cambiar el parámetro de *nivel* en cada cuadro se puede producir movimiento. Para esto, haremos uso de un bloque `draw()` que actualice el nivel de desenfoque de un filtro.



```
float fuzzy = 0.0;
void setup() {
  size(100, 100);
  smooth();
  strokeWeight(5);
  noFill();
}
void draw() {
  background(204);
  if (fuzzy < 16.0) {
    fuzzy += 0.05;
  }
  line(0, 30, 100, 60);
  filter(BLUR, fuzzy);
  line(0, 50, 100, 80);
}
```

Cualquier variable `PImage` también posee su propio método `filter()`. Esto permite trabajar con las imágenes de forma individual. Para esto, utilizamos el operador punto.



```
PImage img = loadImage("medusa.jpg");
image(img, 0, 0);
img.filter(INVERT);
image(img, 50, 0);
```

La función `blend()` mezcla los píxeles de diferentes maneras, dependiendo la clase de parámetros que reciba. Tiene dos versiones diferentes:

```
blend(x, y, ancho, alto, dx, dy, dancho, dalto, modo)
blend(img, x, y, ancho, alto, dx, dy, dancho, dalto, modo)
```

El parámetro `modo` puede ser: `BLEND`, `ADD`, `SUBTRACT`, `DARKEST`, `LIGHTEST`, `DIFFERENCE`, `EXCLUSION`, `MULTIPLY`, `SCREEN`, `OVERLAY`, `HARD_LIGHT`, `SOFT_LIGHT`, `DODGE`, y `BURN`. Los parámetros `x` e `y` corresponden a las coordenadas `x` e `y` de la ubicación a copiar. Los parámetros `ancho` y `alto`, establecen en tamaño del área a copiar. Los parámetros `dx` y `dy` son la coordenada `x` e `y` de destino. Los parámetros `dancho` y `dalto`, establecen el tamaño del área de destino. Para mezclar dos imágenes, lo idea es utilizar la segunda como parámetro de `img`.

En el siguiente ejemplo se muestra la utilización de `blend()` en modo `ADD`.



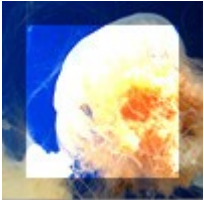
```
background(0);
stroke(153);
strokeWeight(24);
smooth();
line(44, 0, 24, 80);
line(0, 24, 80, 44);
blend(0, 0, 100, 100, 16, 0, 100, 100, ADD);
```

También se pueden mezclar imágenes importadas a la ventana de representación por utilizar la segunda versión de la función `blend()`. El siguiente ejemplo utiliza como *modo* a `DARKEST`.



```
PImage img = loadImage("medusa.jpg");
background(0);
stroke(255);
strokeWeight(24);
smooth();
line(44, 0, 24, 80);
line(0, 24, 80, 44);
blend(img, 0, 0, 100, 100, 0, 0, 100, 100, DARKEST);
```

La variable `PImage` tiene su método `blend()`. De esta forma, podemos mezclar dos imágenes.

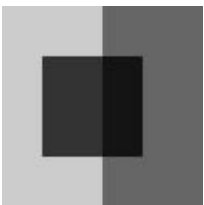


```
PImage img = loadImage("medusa.jpg");
PImage img2 = loadImage("medusa.jpg");
img.blend(img2, 12, 12, 76, 76, 12, 12, 76, 76, ADD);
image(img, 0, 0);
```

La función `blendColor()` es usada para mezclar los colores individualmente.

```
blendColor(c1, c2, modo)
```

Los parámetros `c1` y `c2`, son los valores para crear un nuevo color cuando estos se mezclen juntos. La opción de *modo* es lo mismo que en la función `blend()`.



```
color g1 = color(102); //Gris medio
color g2 = color(51); //Gris oscuro
color g3 = blendColor(g1, g2, MULTIPLY); //Crea Negro
noStroke();
fill(g1);
rect(50, 0, 50, 100); //Rectángulo de la Derecha
fill(g2);
rect(20, 25, 30, 50); //Rectángulo de la Izquierda
fill(g3);
rect(50, 25, 20, 50); //Rectángulo superpuesto
```

-Copiando Píxeles

La función `copy()` posee dos versiones, y cada una de ellas una gran cantidad de parámetros:

```
copy(x, y, ancho, alto, dx, dy, dancho, dalto)
copy(srcImg, x, y, ancho, alto, dx, dy, dancho, dalto)
```

La versión de `copy()` con ocho parámetros, copia los píxeles de una región de la ventana de representación, y los pega en otra región. La versión con nueve parámetros copia toda o una porción de una imagen, especificada con `srcImg`, y la pega en una región de la ventana de representación. Si el recurso y el destino son de diferentes tamaños, los píxeles son automáticamente escalados para alcanzar el ancho y el alto de destino. Los otros parámetros son los mismos que se describieron previamente en la función `blend()`. La función `copy()` difiere de las funciones `get()` y `set()`, vistas anteriormente, ya que esta puede obtener píxeles y restablecerles una posición.



```
PImage img = loadImage("medusa.jpg");
image(img, 0, 0);
copy(0, 0, 100, 50, 0, 50, 100, 50);
```



```
PImage img1, img2;
void setup() {
  size(100, 100);
  img1 = loadImage("bosque.jpg");
  img2 = loadImage("puerto.jpg");
}
void draw() {
  background(255);
  image(img1, 0, 0);
  int my = constrain(mouseY, 0, 67);
  copy(img2, 0, my, 100, 33, 0, my, 100, 33);
}
```

La variables PImage también posee un método `copy()`. Puede utilizarse para copiar porciones de la misma imagen, o de una imagen a otra.



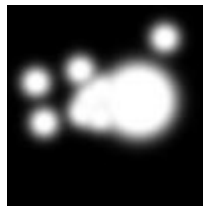
```
PImage img = loadImage("torre.jpg");
img.copy(50, 0, 50, 100, 0, 0, 50, 100);
image(img, 0, 0);
```

-Enmascarando

El método `mask()` de la clase PImage establece el valor de transparencia de una imagen basándose en otra imagen. La imagen máscara debe contener solo valores en escala de grises, y debe ser del mismo tamaño de la imagen a enmascarar. Si la imagen no es escala de grises, puede convertirse con la función `filter()`. Las áreas más claras muestran la imagen, mientras que las más oscuras la cubren. El siguiente ejemplo utiliza la función `mask()` con las siguientes imágenes:



puerto.jpg



mascara.jpg

La imagen resultante y el código para producirla:



```
background(255);
PImage img = loadImage("puerto.jpg");
PImage maskImg = loadImage("mascara.jpg");
img.mask(maskImg);
image(img, 0, 0);
```


Apéndice Filtros



BLUR, 1



BLUR, 4



BLUR, 8

BLUR

Ejecuda un desenfoue Gaussniano con la intensidad referida en el valor.



POSTERIZE, 2



POSTERIZE, 4



POSTERIZE, 8

POSTERIZE

Limita el valor de cada canal de color en la imagen por el número enviado como parámetro.



THRESHOLD, 0.2



THRESHOLD, 0.5



THRESHOLD, 0.8

THRESHOLD

Convierte la imagen en blanco y negro dependiendo el parametro que indica el umbral.



INVERT

Invierte el valor de color de los píxeles.



GRAY

Convierte a escala de grises



ERODE

Reduce la lar áreas luminosas dependiendo el parámetro.



DILATE

Aumenta la lar áreas luminosas dependiendo el parámetro.

Imagen: Procesamiento de Imagen

Elementos que se introducen en esta Unidad:

pixels[], loadPixels(), updatePixels(), createImage()

El procesamiento de imágenes es un término muy amplio y general para referirse a la manipulación y modificación de las mismas, o en su defecto, producir un resultado estético o mejorar la comunicación. Programas como GIMP o Adobe Photoshop provee al usuario de herramientas para manipular imágenes, incluyendo cambios de contraste o desenfoco.

En Processing, cada imagen es almacenada **como un array de una dimensión de colores**. Cuando imagen es mostrada en pantalla, cada elemento del array es dibujado como un píxel. Por lo tanto, al multiplicar el ancho por el alto de la imagen se puede determinar el número de píxeles empleados. Si una imagen es de 100 píxeles de ancho por 100 píxeles de alto, el array contendrá 10000 elementos. Si una imagen es de 200 píxeles de ancho por 2 píxeles de alto, el array tendrá 400 elementos. La primera posición del array es el primer píxel de la esquina superior izquierda, y la última es el último píxel de la esquina inferior derecha.

-Píxeles

El array pixels[] almacena el valor de color de cada píxel. La función loadPixels() debe ser llamada antes de pixels[]. Luego de que los píxeles son leídos o modificados, ellos deben actualizarse mediante la función updatePixels(). Como beginShape() y endShape(), loadPixels() y updatePixels(), siempre están juntas.



```
void setup() {
    size(100, 100);
}
void draw() {
    float gray = map(second(), 0, 59, 0, 255);
    color c = color(gray);
    int index = frameCount % (width*height);
    loadPixels();
    pixels[index] = c;
    updatePixels();
}
```

Las funciones loadPixels() y updatePixels() aseguran que el array pixels[] esté listo para ser modificado y actualizado. Hay que asegurarse, además, que esté declarado en torno a cualquier bloque, y solo utilizarlo cuando sea necesario, puesto su sobreutilización puede entorpecer el programa y hacerlo lento. Leer y escribir datos directamente con pixels[] es un método distinto que hacerlo con las funciones de get() y set(). La coordenada-x y la coordenada-y pueden ser mapeadas a una posición correspondiente del array, por multiplicar el valor de y por el ancho de la ventana de representación, y sumarle a este el valor de x. Para calcular, entonces, cualquier posición de píxeles en el array, debe utilizarse la siguiente ecuación: $(y*width)+x$

```
//Estas tres líneas de código equivalen a: set(25, 50, color(0))
loadPixels();
pixels[50*width + 25] = color(0);
updatePixels();
```

Para convertir los valores en la dirección opuesta, debe dividirse la posición de los píxeles en el array por el ancho de la ventana de representación, para obtener la coordenada-y, y obtener el módulo de la posición y el

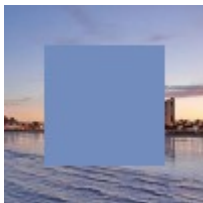
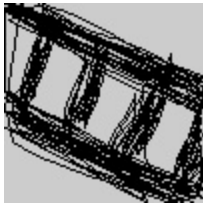
ancho de la ventana de representación, para obtener la coordenada-x.

```
//Estas tres líneas de código equivalen a: pixels[5075] = color(0)
int y = 5075 / width;
int x = 5075 % width;
set(x, y, color(0));
```

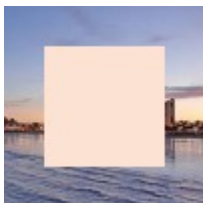
En programas que requieren manipular una gran cantidad de píxeles, la implementación de `pixels[]` es mucho más rápida que la de `get()` y `set()`. Los siguientes ejemplos han sido mostrados al explicarse `get()` y `set()` en unidades anteriores. No obstante, aquí se implementan con `pixels[]` para demostrar la velocidad con la que opera el programa.



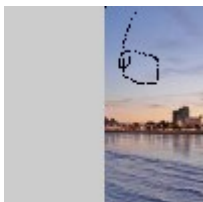
```
void setup() {
    size(100, 100);
}
void draw() {
    //Contrae para no exceder el límite del array
    int mx = constrain(mouseX, 0, 99);
    int my = constrain(mouseY, 0, 99);
    loadPixels();
    pixels[my*width + mx] = color(0);
    updatePixels();
}
```



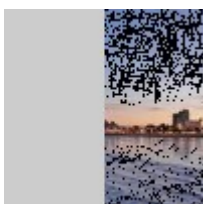
```
PImage arch;
void setup() {
    size(100, 100);
    noStroke();
    arch = loadImage("arch.jpg");
}
void draw() {
    background(arch);
    //Contrae para no exceder el límite del array
    int mx = constrain(mouseX, 0, 99);
    int my = constrain(mouseY, 0, 99);
    loadPixels();
    color c = pixels[my*width + mx];
    fill(c);
    rect(20, 20, 60, 60);
}
```



Cada imagen tiene su propio array `pixels[]`, al cual se puede acceder con el operador punto. Esto hace posible mostrar una imagen mientras se modifican los píxeles en otra imagen.



```
PImage arch;
void setup() {
    size(100, 100);
    arch = loadImage("puerto.jpg");
}
void draw() {
    background(204);
    int mx = constrain(mouseX, 0, 99);
    int my = constrain(mouseY, 0, 99);
    arch.loadPixels();
    arch.pixels[my*width + mx] = color(0);
}
```



```

        arch.updatePixels();
        image(arch, 50, 0);
    }

```

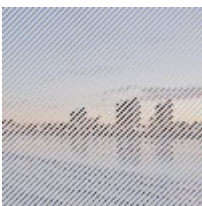
Al implementar el array `pixels[]`, en lugar de la función `image()` para mostrar una imagen en pantalla, este array provee de más control de la mencionada imagen. Simples operaciones matemáticas o pequeños cálculos, sumado a un ciclo FOR, revela el verdadero potencial de `pixels[]` como array.



```

PImage arch = loadImage("puerto.jpg");
int count = arch.width * arch.height;
arch.loadPixels();
loadPixels();
for (int i = 0; i < count; i += 2) {
    pixels[i] = arch.pixels[i];
}
updatePixels();

```



```

PImage arch = loadImage("puerto.jpg");
int count = arch.width * arch.height;
arch.loadPixels();
loadPixels();
for (int i = 0; i < count; i += 3) {
    pixels[i] = arch.pixels[i];
}
updatePixels();

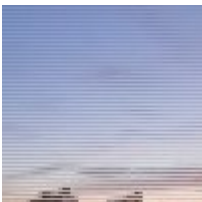
```



```

PImage arch = loadImage("puerto.jpg");
int count = arch.width * arch.height;
arch.loadPixels();
loadPixels();
for (int i = 0; i < count; i++) {
    pixels[i] = arch.pixels[count - i - 1];
}
updatePixels();

```



```

PImage arch = loadImage("puerto.jpg");
int count = arch.width * arch.height;
arch.loadPixels();
loadPixels();
for (int i = 0; i < count; i++) {
    pixels[i] = arch.pixels[i/2];
}
updatePixels();

```

-Componentes del Píxel

Las funciones `red()`, `green()` y `blue()` son utilizadas para leer valores individuales de los píxeles. Estos valores pueden ser modificados y devueltos al array `pixels[]`. En el siguiente ejemplo, se muestra como invertir los valores de color de una imagen.



```

PImage arch = loadImage("puerto.jpg");
background(arch);
loadPixels();
for (int i = 0; i < width*height; i++) {
    color p = pixels[i];           //Conseguir píxeles
    float r = 255 - red(p);        //Modificar valor rojo
    float g = 255 - green(p);      //Modificar valor de verde
    float b = 255 - blue(p);       //Modificar valor de azul
    pixels[i] = color(r, g, b);    //Asignar valor modificado
}

```

```

}
updatePixels();

```

Los valores del teclado y el mouse pueden ser utilizados en la manera en la que el array `pixels[]` modifica mientras el programa se está ejecutando. En el siguiente ejemplo, una imagen es convertida a escala de grises al incrementar sus componentes. Estos valores son incrementados con `mouseX`, haciendo la imagen mas luminosa.



```

PImage arch;
void setup() {
  size(100, 100);
  arch = loadImage("puerto.jpg");
}
void draw() {
  background(arch);
  loadPixels();
  for (int i = 0; i < width*height; i++) {
    color p = pixels[i]; //Leer color de la pantalla
    float r = red(p); //Modificar valor de rojo
    float g = green(p); //Modificar valor de verde
    float b = blue(p); //Modificar valor de azul
    float bw = (r + g + b) / 3.0;
    bw = constrain(bw + mouseX, 0, 255);
    pixels[i] = color(bw); //Asignar valor modificado
  }
  updatePixels();
  line(mouseX, 0, mouseX, height);
}

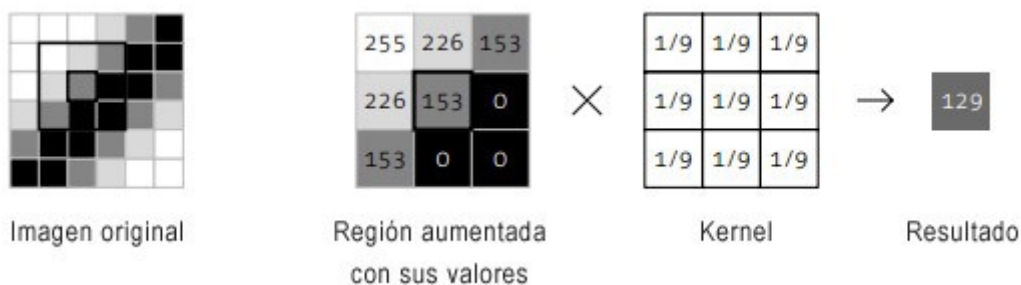
```

Estas funciones de modificación individual son muy útiles, pero también muy lentas. Cuando se requiere que se modifique de forma individual cientos o miles de veces mientras se ejecuta el programa, se recomienda utilizar una técnica conocida como *bit-shifting*.

-Convolución

Otra forma de modificar las imágenes es cambiar el valor de cada píxel en relación a sus píxeles vecinos. Una matrix de números, llamada kernel de convolución, es aplicada a cada píxel de la pantalla. Aplicando el kernel a cada píxel es llamado proceso de convolución. Este tipo de proceso matemático es tan eficiente que opera en programas de producción gráfica muy potentes como el Adobe Photoshop.

Como un ejemplo, tomaremos un píxel de una imagen a la que le aplicaremos el kernel. El píxel opera como centro de los píxeles que se encuentran a su alrededor. Cada píxel es multiplicado por el kernel y, luego estos valores se suman entre sí. El resultado es un píxel con un nuevo valor de color, producto de aplicar el kernel:



La primer expresión creada al aplicar el kernel es la siguiente:

$$\begin{aligned}
 &(255 * 0.111) + (226 * 0.111) + (153 * 0.111) + \\
 &(226 * 0.111) + (153 * 0.111) + (0 * 0.111) + \\
 &(153 * 0.111) + (0 * 0.111) + (0 * 0.111)
 \end{aligned}$$

Simplificado como:

$$\begin{aligned}
 &28.305 + 25.086 + 16.983 + \\
 &25.086 + 16.983 + 0.000 + \\
 &16.983 + 0.000 + 0.000
 \end{aligned}$$

Esto da como resultado 129,426. Pasado a entero es 129, y luego es convertido a valor de grises para el píxel. Para realizar una convolución en la imagen completa, es necesario aplicar el kernel en cada uno de los píxeles. Es sabido que un problema importante son los píxeles de las aristas o los bordes, donde algunos de los píxeles que los rodean escapan de la ventana de representación. En estos casos, el programa simplemente los ignora y aplica el kernel al resto de los valores.

Los patrones que se utilizan en los kernel crean diversos filtros para aplicar a la imagen. Si todos los valores del kernel son positivos, crea lo que suele denominarse un filtro pasa-bajos. Un filtro pasa-bajos, remueve áreas donde hay una extrema diferencia en los valores adyacentes de los píxeles. Al aplicarlo en una imagen completa, crea una especie de desenfoque. Utilizar, en el kernel, una mezcla de valores positivos y negativos crea un filtro pasa-altos. Un filtro pasa-altos, remueve áreas donde hay muy poca diferencia en los valores adyacentes de los píxeles. Esto produce una clase de enfoque.

El siguiente ejemplo, demuestra como usar una matrix kernel de 3 x 3, para transformar una imagen. Si se modifican los valores del array `kernel[][]`, se obtendrán distintos resultados. La función `createImage()` crea un píxel vacío. Dicha función requiere tres parámetros, para asignar el ancho, el alto y el formato de la imagen. El formato puede ser RGB (color completo) o ARGB (color completo con alfa). No es necesario utilizar `loadPixels()` inmediatamente después de utilizar `createImage()`.

```

float[][] kernel = { { -1, 0, 1 },
                    { -2, 0, 2 },
                    { -1, 0, 1 } };

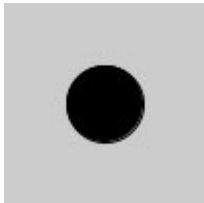
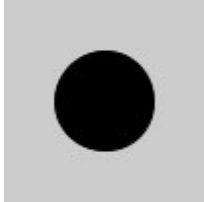
size(100, 100);
PImage img = loadImage("puerto.jpg"); //Carga de la imagen original
img.loadPixels();
//Crea una imagen opaca con el mismo tamaño que la original
PImage edgeImg = createImage(img.width, img.height, RGB);
//Bucle para recorrer cada píxel
for (int y = 1; y < img.height-1; y++) { //Ignora los bordes de arriba y de abajo
    for (int x = 1; x < img.width-1; x++) { //Ignora los bordes
        float sum = 0; //Kernel suma cada píxel
        for (int ky = -1; ky <= 1; ky++) {
            for (int kx = -1; kx <= 1; kx++) {
                //Calcula el píxel adyacente para este punto del kernel
                int pos = (y + ky)*width + (x + kx);
                //Las Imagenes en escala de grises, RGB, son idénticas
                float val = red(img.pixels[pos]);
                //Multiplica los píxeles adyacentes, basado en el kernel
                sum += kernel[ky+1][kx+1] * val;
            }
        }
        //Para cada píxel en la imagen, establecer la escala de grises
        //basada en la suma del kernel
        edgeImg.pixels[y*img.width + x] = color(sum);
    }
}
//Establece que hay cambios en edgeImg.pixels[]

```

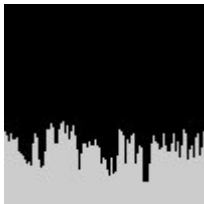
```
edgeImg.updatePixels();
image(edgeImg, 0, 0); //Dibuja la nueva imagen
```

-Imagen como Datos

Esta unidad ha introducido a las imágenes digitales como secuencias de números en una dimensión que definen colores. Los datos numéricos, sin embargo, pueden utilizarse para muchas otras cosas que solo mostrar colores. Los siguientes ejemplos muestran diversas formas de utilizar los datos de `pixels[]`.



```
//Convierte cada valor de píxel en diámetros de un círculo
PImage arch;
int index;
void setup() {
  size(100, 100);
  smooth();
  fill(0);
  arch = loadImage("puerto.jpg");
  arch.loadPixels();
}
void draw() {
  background(204);
  color c = arch.pixels[index]; //Obtiene píxel
  float r = red(c) / 3.0; //Obtiene valor rojo
  ellipse(width/2, height/2, r, r);
  index++;
  if (index == width*height) {
    index = 0; //Retorna al primer píxel
  }
}
```

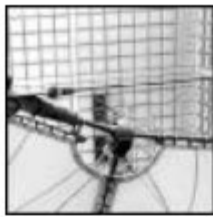


```
//Convierte los valores de rojo en la longitud de las líneas
PImage arch;
void setup() {
  size(100, 100);
  arch = loadImage("puerto.jpg");
  arch.loadPixels();
}
void draw() {
  background(204);
  int my = constrain(mouseY, 0, 99);
  for (int i = 0; i < arch.height; i++) {
    //Obtiene el píxel
    color c = arch.pixels[my*width + i];
    //Obtiene valor de rojo
    float r = red(c);
    line(i, 0, i, height/2 + r/6);
  }
}
```

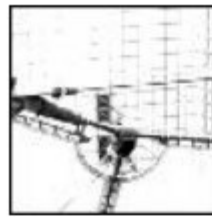


```
//Convierte los valor de azul de una imagen
//en coordenadas de una serie de líneas
PImage arch;
void setup() {
  size(100, 100);
  smooth();
  arch = loadImage("puerto.jpg");
  arch.loadPixels();
}
void draw() {
  background(204);
  int mx = constrain(mouseX, 0, arch.width-1);
  int offset = mx * arch.width;
  beginShape(LINES);
  for (int i = 0; i < arch.width; i += 2) {
    float r1 = blue(arch.pixels[offset + i]);
    float r2 = blue(arch.pixels[offset + i + 1]);
    float vx = map(r1, 0, 255, 0, height);
    float vy = map(r2, 0, 255, 0, height);
    vertex(vx, vy);
  }
  endShape();
}
```


Apéndice Convolución



```
.11 .11 .11
.11 .11 .11
.11 .11 .11
```



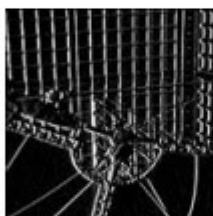
```
.11 .11 .11
.11 .66 .11
.11 .11 .11
```



```
-1 -1 -1
-1 8 -1
-1 -1 -1
```



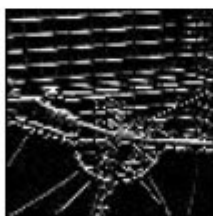
```
-1 -1 -1
-1 12 -1
-1 -1 -1
```



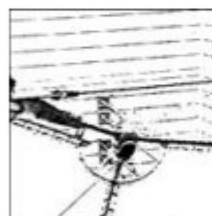
```
-1 0 1
-2 0 2
-1 0 1
```



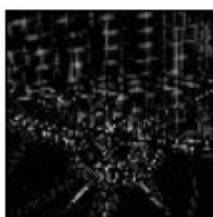
```
-2 0 1
-3 0 2
-2 0 1
```



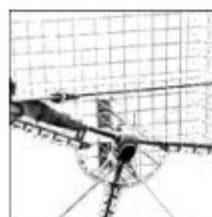
```
-1 -2 -1
0 0 0
1 2 1
```



```
-1 -2 -1
0 0 0
2 3 2
```



```
0 -1 0
-1 4 -1
0 -1 0
```



```
0 -1 0
-1 6 -1
0 -1 0
```

Datos de Salida: Imágenes

Elementos que se introducen en esta Unidad:

`save()`, `saveFrame()`

La pantalla de una computadora muestra una imagen en la pantalla muchas veces por segundo. La mayoría de los sistemas operativos permiten obtener una captura de la pantalla mientras se está ejecutando el sistema. Si se utiliza una Macintosh, debe oprimirse Comando-Shift-3. En caso de usarse Windows, debe oprimirse el botón Impresión de Pantalla (Impr Pant Pet Sis). Esta imagen es guardada en portapapeles, por lo tanto en cualquier editor de imágenes puede usarse la función Pegar (Control+V) para ver la correspondiente captura. Existen, además, software que facilitan el almacenamiento de estas impresiones. Incluso, lo interesante, es que permiten almacenarlas en forma de secuencia, cuadro-a-cuadro. Debe considerarse que 30 cuadros por segundo, tan solo 30 segundos de secuencia necesitan almacenar 900 cuadros. Si se utiliza en la web, podría hacerse muy lenta la descarga de la animación secuencial. Las opciones más comunes suelen ser técnicas de degrado de calidad de la imagen, para una más rápida visualización en la web.

-Guardando Imágenes

La función `save()` es empleada para guardar una imagen del programa. Requiere tan solo un parámetro, un `String` que se convierte en el nombre de la imagen.

```
save(nombredearchivo)
```

Las imágenes pueden ser almacenadas en una gran variedad de formatos, dependiendo de la extensión utilizada en el parámetro. Por ejemplo, si el parámetro `nombredearchivo` es `miArchivo.tif`, será una imagen de formato TIFF; en cambio, si es `miArchivo.tga`, será una imagen de formato TARGA. Si ninguna extensión es incluida, se almacenará como formato TIFF por defecto, y el `.tif` será añadido al nombre. Hay que asegurarse de poner el nombre entre comillas para diferenciar al parámetro como un `String`. La imagen será guardada en la actual carpeta del sketch.

```
line(0, 0, width, height);
line(width, 0, 0, height);
//Guarda un archivo TIFF llamado x.tif en la actual carpeta del sketch
save("x.tif");
```

Solo los elementos dibujados antes de la declaración de `save()` son guardados como una imagen. En el siguiente ejemplo, solo la primer línea es guardada en `linea.tif`.

```
line(0, 0, width, height);
//Guarda un archivo TIFF llamado "linea.tif" en la actual carpeta del sketch
save("linea.tif");
line(width, 0, 0, height);
```

Si la función `save()` aparece dentro de un bloque `draw()`, la imagen será almacenada cada vez que el bucle se ejecute. Esto quiere decir que se estará guardando constantemente la imagen mostrada en pantalla. Esto, como es sabido, no solo no es correcto sino que además entorpece el funcionamiento del programa. De entre las soluciones más simples que se pueden emplear, se encuentra utilizar la función `save()` en un evento. Lo más común es en un evento de `mousePressed()` o `keyPressed()`.

```

void setup() {
    size(100, 100);
}
void draw() {
    background(204);
    line(0, 0, mouseX, height);
    line(width, 0, 0, mouseY);
}
void mousePressed() {
    save("linea.tif");
}

```

-Guardando Imágenes Secuenciales

La función `saveFrame()` almacena una secuencia numerada de imágenes.

```

saveFrame()
saveFrame("nombredearchivo-####.ext")

```

Si `saveFrame()` se utiliza sin parámetro, las imágenes son guardadas como `screen-0000.tif`, `screen-0001.tif`, `screen-0002.tif`, etc. El componente `nombredearchivo-` puede ser cambiado por cualquier nombre se desee, y la extensión `ext` puede ser cambiada por cualquier extensión de imagen. Los `####` es la porción que especifica los dígitos con los que se guardan las imágenes. Si se utiliza `####`, ese valor es reemplazado por el valor que en ese momento tiene `frameCount`. Por ejemplo, el cuadro número 127 es llamado `nombredearchivo-0127.tif`, y el 1723 es `nombredearchivo-1723.tif`. Si se agrega un `#` extra, soportará 10000 o más.

```

//Guarda los primeros 50 cuadros
float x = 33;
float numFrames = 50;
void setup() {
    size(100, 100);
    smooth();
    noStroke();
}
void draw() {
    background(0);
    x += random(-2, 2);
    ellipse(x, 50, 40, 40);
    if (frameCount <= numFrames) {
        saveFrame("circulos-####.tif");
    }
}

```

Usando `saveFrame()` dentro de una estructura IF se le otorga mas control al programa. Por ejemplo, si se desea guardar 200 cuadros luego de oprimir el botón del mouse.

```

//Guarda 24 cuadros, de x-1000.tif a x-1023.tif
void draw() {
    background(204);
    line(mouseX, mouseY, pmouseX, pmouseY);
    if ((frameCount > 999) && (frameCount < 1024)) {
        saveFrame("x-####.tif");
    }
}

```

```

//Guarda un cuadro cada 5 cuadros (ej., x-0005.tif, x-0010.tif, x-0015.tif)
void draw() {

```

```
background(204);  
line(mouseX, mouseY, pmouseX, pmouseY);  
if ((frameCount % 5) == 0) {  
    saveFrame("x-####.tif");  
}  
}
```

Datos de Salida: Exportar Archivos

Elementos que se introducen en esta Unidad:

```
nf(), saveStrings(),
PrintWriter, createWriter(),
PrintWriter.flush(), PrintWriter.close(), exit()
```

Los archivos digitales en los ordenadores no son tangibles, como sus homónimos de papel, y no se acumulan en los gabinetes durante años acumulando polvo. Un archivo digital es una secuencia de bytes en una ubicación en el disco del ordenador. A pesar de la diversidad de los contenidos almacenados en archivos digitales, el material de cada uno es el mismo - una secuencia de 1s y 0s. Casi todas las tareas realizadas con computadoras implica trabajar con los archivos. Por ejemplo, antes de que un documento de texto se escribe, el editor de texto debe leer el archivo y un nuevo archivo debe ser creado para almacenar el contenido. Cuando la información se guarda, se le da un nombre para la futura recuperación del mismo. Todos los archivos del software tienen un formato, un convenio de orden de los datos, para que las aplicaciones de software sepan cómo interpretarlos cuando los leen de la memoria. Algunos formatos comunes incluyen TXT para archivos de texto sin formato, MP3 para almacenar sonido y EXE para los programas ejecutables en Windows. Formatos comunes para los datos de las imágenes son JPEG y GIF y formatos comunes para los documentos de texto son DOC y RTF. La formato XML se ha popularizado en los últimos años como un formato de datos de uso general que puede ser extendido para sostener los tipos específicos de datos en un formato fácil de leer archivo.

-Formateando Datos

Existen caracteres particulares que no poseen una visualización directa. Los más comunes son el espacio *tabular* y la *nueva línea*. Estos caracteres pueden representarse con `\t` y `\n`, respectivamente.

```
//Imprime "espacio tabular"
println("espacio\ttabular");
//Imprime cada palabra luego de \n en una nueva línea
//linea1
//linea2
//linea3
println("linea1\nlinea2\nlinea3");
```

Los datos, también, pueden ser formateados con funciones como `nf()`. Hay dos versiones de dicha función:

```
nf(intValor, digitos)
nf(floatValor, izquierdo, derecho)
```

El `intValor` es un valor tipo `int` a ser formateado, y los `digitos` es la cantidad total de dígitos que debe producir el formato. Por otro lado, en la segunda versión, el `floatValor` es el valor tipo `float` a ser formateado. El parámetro `izquierdo` establece la cantidad de dígitos a la izquierda del decimal, mientras que el `derecho` establece la cantidad total de dígitos a la derecha.

```
println(nf(200, 10));           //Imprime "0000000200"
println(nf(40, 5));           //Imprime"00040"
println(nf(90, 3));           //Imprime "090"
println(nf(200.94, 10, 4));   //Imprime "0000000200.9400"
println(nf(40.2, 5, 3));      //Imprime "00040.200"
println(nf(9.012, 0, 5));     //Imprime "9.01200"
```

-Exportando Archivos

Guardar archivos es una forma muy útil de poder mostrar datos de determinado tipo una vez que el programa

se detuvo. La función `saveStrings()` escribe un array de `Strings` a un archivo. El archivo es almacenado en la carpeta del sketch. Se puede acceder a esta simplemente utilizando la herramienta "Show Sketch Folder" en el menú "Sketch". En adición, la función `exit()` se utiliza para detener el programa.

```
int[] x = new int[0];
int[] y = new int[0];
void setup() {
    size(100, 100);
}
void draw() {
    background(204);
    stroke(0);
    noFill();
    beginShape();
    for (int i = 0; i < x.length; i++) {
        vertex(x[i], y[i]);
    }
    endShape();
    //Muestra el siguiente segmento de línea añadido
    if (x.length >= 1) {
        stroke(255);
        line(mouseX, mouseY, x[x.length-1], y[x.length-1]);
    }
}
void mousePressed() { //Click para agregar un segmento de línea
    x = append(x, mouseX);
    y = append(y, mouseY);
}
void keyPressed() { //Presiona una tecla para almacenar los datos
    String[] lines = new String[x.length];
    for (int i = 0; i < x.length; i++) {
        lines[i] = x[i] + "\t" + y[i];
    }
    saveStrings("lineas.txt", lines);
    exit(); //Detiene el programa
}
```

La clase `PrintWriter` provee de otra forma de exportar archivos. En lugar de escribir el archivo completamente una vez, como en `saveStrings()`, el método `createWriter()` abre un archivo para escribirlo y permite su escritura mientras se ejecuta el programa. Para poder conseguir que el archivo se guarde correctamente, es necesario valerse del método `flush()`, para escribir cualquier clase de dato al archivo. El método `close()` es igualmente necesario para finalizar la escritura de manera correcta.

```
PrintWriter output;
void setup() {
    size(100, 100);
    //Crea un nuevo archivo en el directorio del sketch
    output = createWriter("posiciones.txt");
    frameRate(12);
}
void draw() {
    if (mousePressed) {
        point(mouseX, mouseY);
        // Write the coordinate to a file with a
        // "\t" (TAB carácter) entre cada entrada
        output.println(mouseX + "\t" + mouseY);
    }
}
```

```

void keyPressed() { //Presiona una tecla para guardar los datos
    output.flush(); //Escribe los datos
    output.close(); //Finaliza el archivo
    exit(); //Detiene el programa
}

```

El siguiente ejemplo es una variación del anterior, pero utiliza las teclas Enter y Barra de Espacio para controlar la escritura de datos y cuando el archivo es cerrado:

```

PFont font;
String letras = "";
PrintWriter output;
void setup() {
    size(100, 100);
    fill(0);
    font = loadFont("Eureka-24.vlw");
    textFont(font);
    //Crea un nuevo archivo en el directorio del sketch
    output = createWriter("palabras.txt");
}
void draw() {
    background(204);
    text(letras, 5, 50);
}
void keyPressed() {
    if (key == ' ') { //Barra de Espacio presionada
        output.println(letras); //Escribe datos en palabras.txt
        letras = ""; //Limpia el String letras
    } else {
        letras = letras + key;
    }
    if (key == ENTER) {
        output.flush(); //Escribe los datos
        output.close(); //Finaliza el archivo
        exit(); //Detiene el programa
    }
}
}

```

Estructuras: Objetos

Elementos que se introducen en esta Unidad:

class, Object

Las variables y las funciones son los bloques constructores de la programación. Frecuentemente, muchas funciones son utilizadas juntas para relacionar variables. La programación orientada a objetos, la cual es posible con Processing, utiliza objetos y clases como bloques constructores. Una clase define un grupo de métodos (funciones) y de campos (variables). Un objeto es simplemente una instancia de la clase. Los campos de un objeto son, generalmente, accesibles solo vía sus propios métodos, lo cual permite ocultar su complejidad del programa general. Similar a lo que ocurre al manejar, un conductor no ve los procesos de ingeniería que ocurren en el motor, pero si puede ver la velocidad a la que va en el velocímetro. El mismo tipo de abstracción es utilizada en la programación orientada a objetos (POO).

-Programación Orientada a Objetos

Un programa modular se compone de módulos de código, encargados de una tarea específica. Las variables son la forma más básica en pensar en reutilizar elementos en un programa. Permiten que un valor pueda aparecer la cantidad de veces que sea necesario y que sea cambiado. Las funciones abstraen una tarea específica y permiten a los bloques de código se reutilizados. Generalmente, uno está conectado con lo que la función hace, y no como ella funciona. La programación orientada a objetos extiende la modularidad de utilizar variables y escribir funciones por relacionarlas entre ellas, al ser agrupadas juntas.

Es posible hacer una analogía entre los objetos del software y los *artefactos reales*. Para obtener una visión del mundo como pequeños programas orientados a objetos, se presenta la siguiente lista:

Nombre	Manzana
Campos	color, peso
Métodos	crecer(), caer(), pudrir()
Nombre	Mariposa
Campos	especie, genero
Métodos	agitarAlas(), aterrizar()
Nombre	Radio
Campos	frecuencia, volumen
Métodos	encender(), tono(), establecerVolumen()
Nombre	Auto
Campos	marca, modelo, color, año
Métodos	acelerar(), frenar(), girar()

Al extender el ejemplo de la manzana, se revelan posibilidades mas interesantes de la programación orientada a objetos. Para hacer un programa que simule la vida de una manzana, el método `crecer()` posiblemente reciba valores de entrada de temperatura y humedad. El método `crecer()` puede incrementar el campo `peso` del objeto, basándose en esos valores de entrada. El método `caer()` se encontraría revisando constantemente el campo `peso`, y cuando este pase cierto límite, la manzana caerá a la tierra. El método `pudrir()` comenzará a funcionar una vez que la manzana llegue al suelo. Este método podría hacer decrecer el campo `peso` y cambiar el `color` de la manzana.

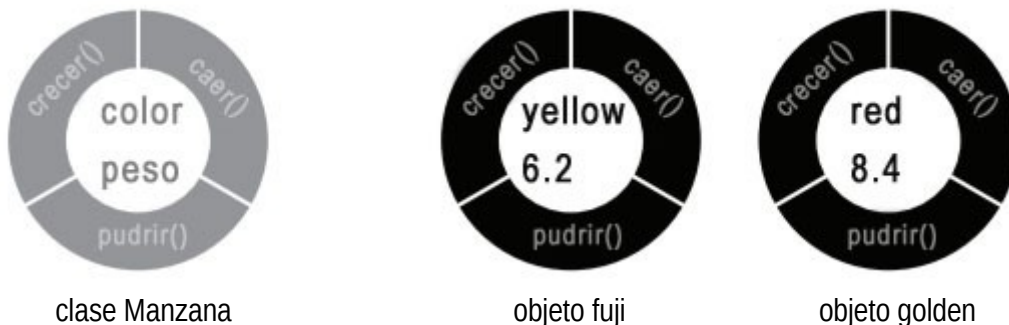
Como queda dicho en la introducción, los objetos son creados con las clases y las clases son un conjunto de métodos y campos. Los objetos de una clase deben tener un nombre propio, al igual que una variable. Por ejemplo, dos objetos llamados `fuji` y `golden` pertenecen a la clase `Manzana`.

Nombre fuji
Campos color: red
 peso: 6.2
Nombre golden
Campos color: yellow
 peso: 8.4

Dos populares estilos de representación de clases son las tablas y un diagrama circular inspirado en la biología de una célula. Cada estilo de representación muestra el nombre de la clase, los campos y los métodos. Es muy útil definir las características de cada clase en alguno de estos diagramas antes de comenzar con el código en sí.

<i>Manzana</i>	<i>fuji</i>	<i>golden</i>
<i>color</i> <i>peso</i>	<i>red</i> <i>6.2</i>	<i>yellow</i> <i>8.4</i>
<i>crecer()</i> <i>caer()</i> <i>podrir()</i>	<i>crecer()</i> <i>caer()</i> <i>podrir()</i>	<i>crecer()</i> <i>caer()</i> <i>podrir()</i>
clase Manzana	objeto fuji	objeto golden

El diagrama circular adquiere más bien un concepto de *encapsulación*, la idea de que los campos de un objeto no deberían ser accedidos desde afuera. Los métodos de un objeto actúan como un *buffer* entre el código fuera de la clase y los datos contenidos.



Los campos y métodos de un objetos son accesibles a través del operador *punto*, un *período*. Para obtener el calor de color del objeto `fuji`, la sintaxis correspondiente será `fuji.color`. Para ejecutar el método `crecer()` dentro de objeto `golden`, habrá que escribir `golden.crecer()`.

-Usando Clases y Objetos

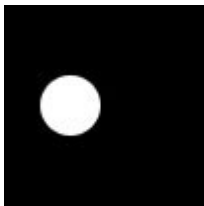
Definir una clase es crear tu propio tipo de datos. Pero no como los tipos de datos primitivos tales como `int`, `float` o `boolean`, sino más bien como los tipo `String`, `PImage` y `PFont`, por dicha razón siempre se escriben con letra capital (primera letra en mayúsculas). Cuando se crea una clase, lo primero es pensar cuidadosamente el código que llevará dentro, y no programar a la ligera. Lo común es hacer primero una lista de variables requeridas (las cuales pasaran a llamarse campos), y determinar que tipo de datos serán. En el primero de los ejemplos que se verán a continuación, un círculo es posicionado en la pantalla, pero el código es escrito de una forma distinta a como se venía viendo (es escrito en un ámbito de paradigma orientado a objetos). Es necesario dos campos que almacenen la posición del círculo, y un tercer campo que

controle su diámetro.

float x	Coordenada-x del círculo
float y	Coordenada-y del círculo
float diametro	Diámetro del círculo

El nombre de la clase debe ser cuidadosamente considerado. Este puede ser un nombre simbólico de lo que se esté dibujando en pantalla, o un nombre de las convenciones de variables. Sin embargo, siempre, deben ser escritos con la letra capital. Esto ayuda al programa a separar clases como `String` o `PImage` de los tipos primitivos como `int` o `boolean`. El nombre de la clase para el siguiente ejemplo será `Punto`, puesto que `Circulo` no hubiese sido pertinente al caso (ya que se esta dibujando un punto).

En principio, se dispondrá del programa escrito como programación estructurada, ya que siempre es recomendable escribirlo así y luego convertirlo en un objeto:

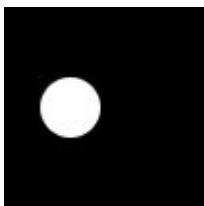


```
float x = 33;
float y = 50;
float diametro = 30;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}

void draw() {
  background(0);
  ellipse(x, y, diametro, diametro);
}
```

Para hacer a este código más útil y genérico, es necesario que la elipse tenga su propia clase. De esta forma, antes del bloque `setup()`, la primer línea de código crea al objeto `pt` de la clase `Punto`. El objeto `pt` es construido dentro del `setup()`. Luego, usando el operador punto, se le asignan los valores a la elipse dentro del `draw()`.



```
Punto pt;           //Declara el objeto

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  pt = new Punto(); //Construye el objeto
  pt.x = 33;        //Asigna 33 al campo x
  pt.y = 50;        //Asigna 50 al campo y
  pt.diametro = 30; //Asigna 30 al campo diametro
}

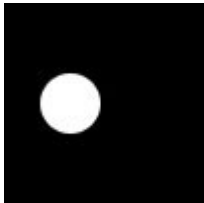
void draw() {
  background(0);
  ellipse(pt.x, pt.y, pt.diametro, pt.diametro);
}

class Punto {
  float x, y;           //Coordenada x e y
  float diametro;      //Diámetro del círculo
}
```

Ahora ya está lista la clase Punto. Aún así, todavía no está siendo implementada de manera útil. En el siguiente ejemplo, se le agregará un método a la clase Punto (es aquí donde la programación orientada a objetos comienza a tomar ventaja). El método `mostrar()` es añadido a la clase Punto. Para añadir métodos, necesario seguir la siguiente sintaxis:

```
void display()          Dibuja el círculo para mostrar en la ventana
```

En la última línea del código de la clase, se coloca el método `mostrar()`, el cual funciona como el operador punto anteriormente visto. Dentro del método, se coloca la función que dibuja el círculo. Es evidente notar que ahora los parámetros se escriben sin el operador punto, esto es porque la función `ellipse()` se encuentra dentro de la clase Punto, por lo que no es necesario hacer una concatenación.



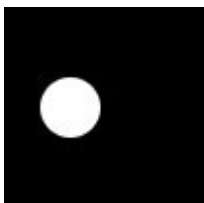
```
Punto pt;          //Declara el objeto

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  pt = new Punto();    //Construye el objeto
  pt.x = 33;
  pt.y = 50;
  pt.diametro = 30;
}

void draw() {
  background(0);
  pt.mostrar();
}

class Punto{
  float x, y, diametro;
  void mostrar() {
    ellipse(x, y, diametro, diametro);
  }
}
```

El siguiente ejemplo introduce un paso en la programación de objetos llamado *constructor*. Un constructor es un bloque de código que es activado cuando el objeto es creado. El constructor siempre lleva el mismo nombre que la clase, y es generalmente utilizado para asignar valores a los campos del objeto (si no existe el constructor, el valor de los campos será *cero*). El constructor es otro tipo de método, solamente que no es precedido por el `void`, puesto que no debe devolver ninguna clase de dato. Cuando el objeto `pt` es creado, los parámetros 33, 50 y 30 son asignados a las variables `xpos`, `ypos` y `dia`, respectivamente con el constructor. Con dicho bloque, los valores son asignados a los campos del objeto, `x`, `y` y `diametro`. Para que los campos sean accesibles con cada método del objeto, ellos deben ser declarados fuera del constructor.



```
Punto pt;          //Declara el objeto
void setup() {
  size(100, 100);
  smooth();
  noStroke();
  pt = new Punto(33, 50, 30); //Construye el objeto
}
void draw() {
  background(0);
}
```

```

        pt.mostrar();
    }

    class Punto {
        float x, y, diametro;
        Punto(float xpos, float ypos, float dia) {
            x = xpos;           //Asigna 33 a x
            y = ypos;           //Asigna 50 a y
            diametro = dia;     //Asigna 30 a diametro
        }
        void mostrar() {
            ellipse(x, y, diametro, diametro);
        }
    }
}

```

Así mismo, la clase `Punto` puede ser extendida por agregar más métodos que mejoren su utilización en la escena. El siguiente ejemplo extiende la funcionalidad de la clase por añadir movimiento constante de arriba a abajo. Para esto es necesario determinar dos nuevos campos. Un campo que regule la velocidad y otro la dirección del objeto. Los llamaremos `vel` y `dir`, respectivamente, para que sean cortos y figurativos. De esta forma, `vel` será de tipo `float` para tener un mayor rango de velocidad, y `dir` un `int` ya que solo necesitamos dos valores.

```

float vel          Distancia que se mueve cada cuadro
int dir            Dirección del movimiento (1 es abajo, -1 es arriba)

```

Para crear el movimiento deseado, se necesita actualizar la posición del objeto en cada cuadro. La dirección también tiene que cambiar al llegar a los bordes de la ventana de representación. Además, hay que considerar el tipo de valor a regresar. Como en este caso no se necesita regresar ningún valor, se antepone la palabra clave `void`.

```

void mover()      Actualiza la posición del círculo

```

El código dentro de los métodos `mover()` y `mostrar()` podría incluirse en tan solo un método. Pero, por cuestiones de claridad, en estos ejemplos se prefiere separarlos.

```

class Punto {
    float x, y;           //Coordenada-x, coordenada-y
    float diametro;      //Diámetro del círculo
    float vel;           //Distancia que se mueve cada cuadro
    int dir = 1;         //Dirección del movimiento (1 es abajo, -1 es arriba)

    //Constructor
    Punto(float xpos, float ypos, float dia, float sp) {
        x = xpos;
        y = ypos;
        diametro = dia;
        vel = sp;
    }

    void mover() {
        y += (vel * dir);
        if ((y > (height - diametro/2)) || (y < diametro/2)) {
            dir *= -1;
        }
    }
}

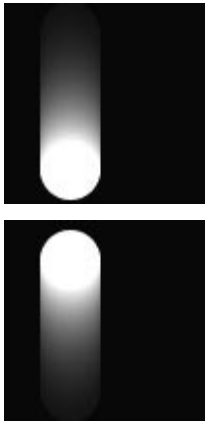
```

```

void mostrar() {
    ellipse(x, y, diametro, diametro);
}
}

```

Para guardar espacio y mantenerse enfocado en el programa en sí, los ejemplos que se muestran en el presente manual no vuelven a mostrar el código de la clase Punto, y simplemente pone un comentario como `//Insertar clase Punto`. Cuando usted vea un comentario como ese, copie y pegue el código de la clase correspondiente en lugar del comentario. El siguiente ejemplo muestra el resultado de utilizar una clase con un objeto en movimiento:



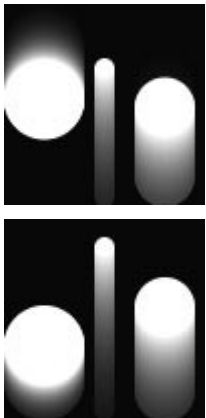
```

Punto pt;           //Declara el objeto
void setup() {
    size(100, 100);
    smooth();
    noStroke();
    pt = new Punto(33, 50, 30, 1.5); //Construye el objeto
}
void draw() {
    fill(0, 15);
    rect(0, 0, width, height);
    fill(255);
    pt.mover();
    pt.mostrar();
}
//Insertar clase Punto

```

Como con las funciones, una clase bien escrita pone el foco en el resultado y no en los detalles de ejecución. Los objetos deben construirse con el propósito de ser reutilizados. Luego de que un complejo programa es escrito y codificado como un objeto, puede ser utilizado posteriormente como herramienta para construir un nuevo programa. Por ejemplo, las funciones y las clases que vienen incluidas en Processing, son utilizadas en cualquier programa de autor como simples herramientas.

Como con los tipos de variables, los objetos adicionales son añadidos al programa por declarar más nombres. En el siguiente ejemplo se utilizan tres objetos, nombrados `pt1`, `pt2`, `pt3`, y cada uno tiene sus propios métodos.



```

Punto pt1, pt2, pt3; //Declara el objeto
void setup() {
    size(100, 100);
    smooth();
    noStroke();
    pt1 = new Punto(20, 50, 40, 0.5); //Construye pt1
    pt2 = new Punto(50, 50, 10, 2.0); //Construye pt2
    pt3 = new Punto(80, 50, 30, 1.5); //Construye pt3
}
void draw() {
    fill(0, 15);
    rect(0, 0, width, height);
    fill(255);
    pt1.mover();
    pt2.mover();
    pt3.mover();
    pt1.mostrar();
    pt2.mostrar ();
    pt3.mostrar ();
}
//Insertar clase Punto

```

Resulta muy complejo resumir los conceptos básicos y la correcta sintaxis de la programación orientada a objetos en tan solo un ejemplo. Por lo tanto, se añade una nueva clase llamada *Huevo*, para compararla con la clase *Punto*. La clase *Huevo* se crea con la necesidad de poder mostrar una figura con forma de huevo, que se tambalea de izquierda a derecha. La construcción inicia con la declaración de los campos y los métodos necesarios para la creación de la clase:

```
float x           Coordenada-x desde la mitad del huevo
float y           Coordenada-y desde abajo del huevo
float ladeo       Ángulo de compensación de izquierda a derecha
float angulo      Se utiliza para definir el ladeo
float escala      Alto del huevo
void tambalear() Mueve el huevo de un lado al otro
void mostrar()   Dibuja el huevo
```

Luego de que los requerimientos de la clase sean establecidos, se distribuyen de la misma forma que en la clase *Punto*. La clase *Huevo* inicia muy sencillamente, con las coordenadas *x* e *y*, y el método *mostrar()*. La clase es añadida a los bloques *setup()* y *draw()* para comprobar que funciona. La función *scale()* es añadida a *mostrar()* para decrecer el tamaño del objeto.

Cuando se compruebe que la primer parte del programa está funcionando a la perfección, el método *rotate()* y el campo *ladeo* son añadidos para cambiar el ángulo del objeto. Finalmente, el código es escrito para que el objeto se mueva. El campo *angulo* se utiliza para calcular la inclinación. De esta forma, el método *tambalear()* es añadido para incrementar el ángulo y calcular, de este modo, el ladeo mismo. La función *cos()* se agrega para acelerar o frenar el tambaleo de lado a lado. Luego de muchas pruebas, la clase *Huevo* final se muestra de la siguiente manera:

```
class Huevo {
  float x, y;           //Coordenada-x, coordenada-y
  float ladeo;         //Ángulo de compensación de izquierda a derecha
  float angulo;        //Se utiliza para definir el ladeo
  float escala;        //Alto del huevo

  Huevo(int xpos, int ypos, float l, float e) { //Constructor
    x = xpos;
    y = ypos;
    ladeo = l;
    escala = e / 100.0;
  }

  void tambalear() {
    ladeo = cos(angulo) / 8;
    angulo += 0.1;
  }

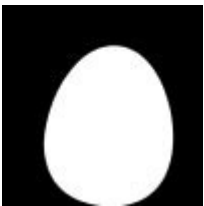
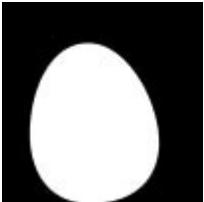
  void mostrar() {
    noStroke();
    fill(255);
    pushMatrix();
    translate(x, y);
    rotate(ladeo);
    scale(escala);
    beginShape();
    vertex(0, -100);
    bezierVertex(25, -100, 40, -65, 40, -40);
    bezierVertex(40, -15, 25, 0, 0, 0);
    bezierVertex(-25, 0, -40, -15, -40, -40);
  }
}
```

```

        bezierVertex(-40, -65, -25, -100, 0, -100);
        endShape();
        popMatrix();
    }
}

```

La clase `Huevo` se incluye en el programa de la misma forma que la clase `Punto`. Un objeto de tipo `Huevo`, llamado `humpty`, es creado fuera del `setup()` y del `draw()`. Dentro del `setup()`, el objeto `humpty` es construido. Dentro del `draw()`, las funciones `tambalear()` y `mostrar()` son ejecutadas, causando la actualización del objeto.



```

Huevo humpty; //Declara el objeto
void setup() {
    size(100, 100);
    smooth();
    //Valores de entrada: coordenada-x, coordenada-y, lado
    //y alto
    humpty = new Huevo(50, 100, PI/32, 80);
}
void draw() {
    background(0);
    humpty.tambalear();
    humpty.mostrar();
}
//Insertar clase Huevo

```

-Array de Objetos

Trabajar con array de objetos es similar a trabajar con array de otro tipo de datos. Como todos los array, un array de objetos se distingue por llevar corchetes `[]`. Como cada elemento del array es un objeto, cada elemento del array debe ser declarado antes de ser accedido. Los pasos básicos para trabajar con un array de objetos son los siguientes:

- 1- Declarar el array
- 2- Crear el array
- 3- Crear cada objeto en el array

Estos pasos son trasladados en el siguiente código:



```

int numPuntos = 6;
//Declarar y crear el Array
Punto[] puntos = new Punto[numPuntos];
void setup() {
    size(100, 100);
    smooth();
    noStroke();
    for (int i = 0; i < puntos.length; i++) {
        float x = 10 + i*16;
        float rate = 0.5 + i*0.05;
        //Crea cada objeto
        puntos[i] = new Punto(x, 50, 16, rate);
    }
}
void draw() {
    fill(0, 12);
    rect(0, 0, width, height);
    fill(255);
}

```

```

        for (int i = 0; i < puntos.length; i++) {
            puntos[i].mover();        //Mueve cada objeto
            puntos[i].mostrar();     //Muestra cada objeto
        }
    }
    //Insertar clase Punto

```

La clase `Anillo` se presenta como otro ejemplo de array de objetos. Esta clase define un círculo que aparece cuando se hace click con el mouse, y crece hasta un ancho de 400 píxeles, luego de esto deja de mostrarse. Cuando esta clase es añadida, un nuevo anillo aparece cada vez que el botón del mouse es oprimido. Estos son los campos y los métodos para hacer esta clase posible:

<code>float x</code>	Coordenada-x del anillo
<code>float y</code>	Coordenada-y del anillo
<code>float diametro</code>	Diámetro del anillo
<code>boolean on</code>	Muestra o no el anillo
<code>void crecer()</code>	Incrementa el diámetro si la variable <code>on</code> es un <code>true</code>
<code>void mostrar()</code>	Dibuja el anillo

La clase `Anillo` se programa como una simple clase. Esta clase no tiene constructor puesto que sus valores no se establecen hasta que se ejecuta el método `inicio()` es llamado desde el programa.

```

class Anillo{
    float x, y;           //Coordenada-x, coordenada-y
    float diametro;      //Diámetro del anillo
    boolean on = false;  //Muestra o no el anillo
    void inicio(float xpos, float ypos) {
        x = xpos;
        y = ypos;
        on = true;
        diametro = 1;
    }

    void crecer() {
        if (on == true) {
            diametro += 0.5;
            if (diametro > 400) {
                on = false;
            }
        }
    }

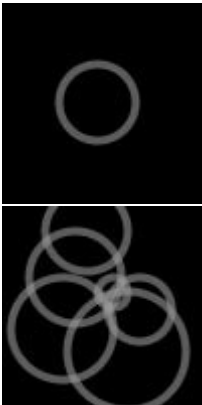
    void mostrar() {
        if (on == true) {
            noFill();
            strokeWeight(4);
            stroke(155, 153);
            ellipse(x, y, diametro, diametro);
        }
    }
}

```

En este programa, el array `anillos[]` es creado para sostener cincuenta objetos de clase `Anillo`. El espacio en memoria del array `anillos[]` y de los objetos de clase `Anillo` debe ser siempre almacenado en el bloque `setup()`. Cuando el botón del mouse es oprimido por primera vez, las variables de posición del objeto `Anillo` se establecen como la actual posición del cursor.

La variable de control `anilloActual`, es incrementada de a uno por vez, así la próxima vez los bloques

`draw()`, `crecer()` y `mostrar()` serán ejecutados por el primer elemento de `Anillo`. Cada vez que el botón del mouse es presionado, un nuevo anillo se muestra en pantalla. Cuando el límite de elementos es alcanzado, el programa salta al inicio de los mismos y asigna las primeras posiciones.



```
Anillo[] anillos;          //Declara el array
int numAnillos = 50;
int anilloActual = 0;
void setup() {
    size(100, 100);
    smooth();
    anillos = new Anillo[numAnillos]; //Crea el array
    for (int i = 0; i < numAnillos; i++) {
        anillos[i] = new Anillo();    //Crea cada objeto
    }
}
void draw() {
    background(0);
    for (int i = 0; i < numAnillos; i++) {
        anillos[i].crecer();
        anillos[i].mostrar();
    }
}
//Click para crear un nuevo anillo
void mousePressed() {
    anillos[anilloActual].inicio(mouseX, mouseY);
    anilloActual++;
    if (anilloActual >= numAnillos) {
        anilloActual = 0;
    }
}
//Insertar clase Anillo
```

De igual manera que los códigos de programación modular, estos pueden usarse de muchas maneras y combinarse de infinitas formas, dependiendo las necesidades de cada proyecto. Esta es una de las cosas más excitantes de programar con objetos.

-Múltiples Archivos

Los programas escritos hasta el momento se escribían como un solo gran conjunto de código. Ya que esta clase de programas pueden convertirse en largos paquetes de código, un simple archivo se convierte en un gran problema. Cuando un programa se convierte en una convergencia de cientos o miles de líneas, es conveniente que exista una posibilidad de separar sus partes en módulos. Processing permite manejar gran cantidad de archivos, y cada sketch puede estar compuesto por gran cantidad de sketches, que son mostrados en forma de pestañas.

```
Processing
File Edit Sketch Tools Help
[Run] [Stop] [New] [Open] [Save] [Undo] [Redo]
Ejemplo Punto [Run]

Punto pt;
void setup() {
  size(100, 100);
  smooth();
  noStroke();
  pt = new Punto(33, 50, 30, 1.5);
}
void draw() {
  fill(0, 15);
  rect(0, 0, width, height);
  fill(255);
  pt.mover();
  pt.mostrar();
}
```

```
Processing
File Edit Sketch Tools Help
[Run] [Stop] [New] [Open] [Save] [Undo] [Redo]
Ejemplo Punto [Run]

class Punto {
  float x, y;
  float diametro;
  float vel;
  int dir = 1;

  Punto(float xpos, float ypos, float dia, float sp) {
    x = xpos;
    y = ypos;
    diametro = dia;
    vel = sp;
  }

  void mostrar() {
    ellipse(x, y, diametro, diametro);
  }
}
```

En la esquina superior derecha, se encuentra un botón con el icono de una flecha que apunta, justamente, hacia la derecha. Se trata del un sub-menú destinado a el manejo de pestañas en Processing. Al oprimir dicho botón, se desplegará una lengüeta con unas pocas opciones. La opción *New Tab* (Nueva Pestaña) nos permite, justamente, crear una nueva pestaña para poder trabajar con código modular. Al oprimir la misma, se nos dará la opción de ponerle un nombre. Suele ser recomendable que el nombre tenga un sentido al contenido del código y no sea un simple acto azaroso. De esta forma, si el código contiene la clase *Punto*, es conveniente que la pestaña se llame "Punto". A continuación se le da al botón *OK*, y acto seguido tendremos la nueva pestaña lista para comenzar a añadirle el código que necesitamos. La pestaña se guardará como un sketch a parte dentro de la carpeta de producción.

Además, el menú de las pestañas permite otras operaciones muy útiles. La posibilidad de Renombrar una pestaña con la opción *Rename*; o de Eliminarla con la opción *Delete*. Agregado a esto, para cuando se tiene una gran cantidad de pestañas, se encuentra la opción *Previous Tab* (Pestaña Anterior) y *Next Tab* (Pestaña Siguiente), las cuales nos pueden ser muy útiles; o directamente nos aparece una lista con el nombre de las pestañas que estamos trabajando, basta con solo oprimir en la deseada.

Dibujos: Formas Cinéticas

Las técnicas de animación experimental de dibujo, pintura, y calado en cine, son todos predecesores de software basados en dibujos cinéticos. La inmediatez y la frescura de los cortos fílmicos, tales como el *Hen Hop* de Norman McLaren (1942), *Free Radicals* de Len Lye (1957), y *The Garden of Earthly Delights* de Stan Brakhage (1981) se debe a las extraordinarias cualidades del gesto físico, que el software, mucho después, hizo más accesible.

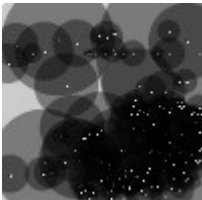
Las herramientas de software para animación amplían, aún más, las técnicas de cine, permitiendo al artista editar y animar los elementos de forma continua después de que han sido elaborado. En 1991, Scott Snibbe Sketch dio movimiento con software a las técnicas exploradas por McLaren, Lye, y Brakhage. La aplicación traduce el movimiento de la mano, al de los elementos visuales en la pantalla. Cada gesto crea una forma que se mueve en un bucle de un segundo. Las animaciones resultantes pueden ser capas para crear una obra de reminiscencias complejidad espacial y temporal, del estilo de Oskar Fischinger. Snibbe, ampliando este concepto con el teléfono de movimiento (1995), permitió a la gente trabajar simultáneamente en un espacio de dibujo compartido a través de Internet.

-Herramientas Activas

Muchas herramientas creadas con software pueden cambiar su forma mientras se encuentran en proceso de dibujo. Por comparar los valores de `mouseX` y `mouseY` con sus valores previos, podemos determinar la dirección y la velocidad con la que se hacen los trazados. En el siguiente ejemplo, la diferencia entre la actual y la anterior posición del cursor, determina el tamaño de una elipse que es dibujada en pantalla.



```
void setup() {
  size(100, 100);
  smooth();
}
void draw() {
  float s = dist(mouseX, mouseY, pmouseX, pmouseY) + 1;
  noStroke();
  fill(0, 102);
  ellipse(mouseX, mouseY, s, s);
  stroke(255);
  point(mouseX, mouseY);
}
```



Los instrumentos de dibujos con software pueden, además, seguir un ritmo o ser arbitrario por reglas independientes del dibujo manual. Esta es una forma de controlar el dibujo, más allá de las características independientes de cada uno. En los ejemplos que siguen, el dibujo solo obedece el punto de origen y de cierre dado por autor, lo que ocurre en medio es incontrolable por este último.



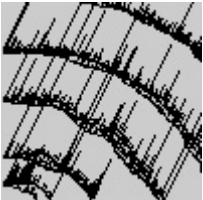
```
int angle = 0;
void setup() {
  size(100, 100);
  smooth();
  noStroke();
  fill(0, 102);
}
void draw() {
  //Dibuja solo cuando el mouse es presionado
  if (mousePressed == true) {
    angle += 10;
    float val = cos(radians(angle)) * 6.0;
```



```

        for (int a = 0; a < 360; a += 75) {
            float xoff = cos(radians(a)) * val;
            float yoff = sin(radians(a)) * val;
            fill(0);
            ellipse(mouseX + xoff, mouseY + yoff, val/2,
                val/2);
        }
        fill(255);
        ellipse(mouseX, mouseY, 2, 2);
    }
}

```



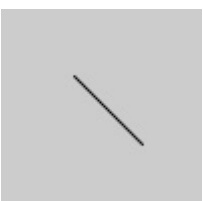
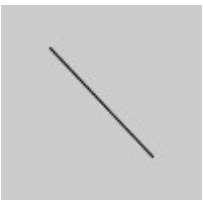
```

Espada diagonal;
void setup() {
    size(100, 100);
    diagonal = new Espada(30, 80);
}
void draw() {
    diagonal.crecer();
}
void mouseMoved() {
    diagonal.seed(mouseX, mouseY);
}
class Espada {
    float x, y;
    Espada(int xpos, int ypos) {
        x = xpos;
        y = ypos;
    }
    void seed(int xpos, int ypos) {
        x = xpos;
        y = ypos;
    }
    void crecer() {
        x += 0.5;
        y -= 1.0;
        point(x, y);
    }
}
}

```

-Dibujos Activos

Los elementos de dibujo individuales, con su correspondiente cuidado, pueden producir dibujos con o sin valores de entrada producidos por un usuario. Estos dibujos activos podrían traducirse como un gato o un mapache que cae en un bote de pintura, y deja sus huellas por todas partes, sin que nosotros podamos controlarlo. Creado por una serie de acciones predeterminadas, el dibujo activo puede ser parcialmente o totalmente autónomo.



```

float x1, y1, x2, y2;
void setup() {
    size(100, 100);
    smooth();
    x1 = width / 4.0;
    y1 = x1;
    x2 = width - x1;
    y2 = x2;
}
void draw() {
    background(204);
}

```

```

        x1 += random(-0.5, 0.5);
        y1 += random(-0.5, 0.5);
        x2 += random(-0.5, 0.5);
        y2 += random(-0.5, 0.5);
        line(x1, y1, x2, y2);
    }

```

Si varias de estas líneas se dibujan a la vez, el dibujo se degrada con el tiempo, ya que cada línea sigue a vagar de su posición original. En el siguiente ejemplo, el código anterior fue modificado para crear la clase `MueveLinea`. Quinientos de estos objetos `MueveLinea` llenan la ventana de representación. Cuando las líneas se dibujan en primera instancia, vibran, pero mantienen su forma. Con el tiempo, la imagen se degrada en un caos, ya que cada línea vaga través de la superficie de la ventana.



```

int numLineas = 500;
MueveLinea[] lines = new MueveLinea[numLineas];
int lineaActual = 0;
void setup() {
    size(100, 100);
    smooth();
    frameRate(30);
    for (int i = 0; i < numLineas; i++) {
        lines[i] = new MueveLinea();
    }
}
void draw() {
    background(204);
    for (int i = 0; i < lineaActual; i++) {
        lines[i].mostrar();
    }
}
void mouseDragged() {
    lines[lineaActual].setPosicion(mouseX, mouseY,
    pmouseX, pmouseY);
    if (lineaActual < numLineas - 1) {
        lineaActual++;
    }
}
class MueveLinea {
    float x1, y1, x2, y2;
    void setPosicion(int x, int y, int px, int py) {
        x1 = x;
        y1 = y;
        x2 = px;
        y2 = py;
    }
    void mostrar() {
        x1 += random(-0.1, 0.1);
        y1 += random(-0.1, 0.1);
        x2 += random(-0.1, 0.1);
        y2 += random(-0.1, 0.1);
        line(x1, y1, x2, y2);
    }
}
}

```

El siguiente ejemplo enseña una simple herramienta de animación que muestra un ciclo continuo de 12 imágenes. Cada imagen es mostrada por 100 milisegundos para crear la animación. Mientras cada imagen es creada, es posible dibujar en la misma por presionar y mover el mouse.



```
int cuadroActual = 0;
PImage[] cuadros = new PImage[12];
int tiempoFinal = 0;
void setup() {
  size(100, 100);
  strokeWeight(4);
  smooth();
  background(204);
  for (int i = 0; i < cuadros.length; i++) {
    cuadros[i] = get();    //Crea un cuadro en blanco
  }
}
void draw() {
  int tiempoActual = millis();
  if (tiempoActual > tiempoFinal+100) {
    siguienteCuadro();
    tiempoFinal = tiempoActual;
  }
  if (mousePressed == true) {
    line(pmouseX, pmouseY, mouseX, mouseY);
  }
}
void siguienteCuadro() {
  //Obtiene la ventana de representación
  cuadros[cuadroActual] = get();
  //Incrementa al siguiente cuadro
  cuadroActual++;
  if ( cuadroActual >= cuadros.length) {
    cuadroActual = 0;
  }
  image(cuadros[cuadroActual], 0, 0);
}
```

Extensión: Modos

Como parte de una implementación más elevada del software, se comienzan a incluir los Modos [a partir de Processing 1.5]. Esta funcionalidad alterna las formas de programar o de mostrar el resultado de los programas ejecutándose. Desde la interfaz, se provee de un simple botón en la esquina superior derecha:



Al oprimirlo se nos desplegará una pequeña lengüeta con opciones para alternar entre los distintos modos (pueden llegar a existir más modos en las nuevas versiones que en las antiguas, siempre es conveniente consultar la referencia). Por defecto, el botón y, por lo tanto, el modo, se encuentra en “Java” (llamado “Standard” en versiones previas a la 2.0).

También existe la posibilidad de añadir modos, de la misma forma que uno añade librerías (library) o herramientas (tool). Para esto solo hay que oprimir en el botón y, a continuación, la opción “Add Mode...”. El programa se encargará de buscar los modos disponibles para su sencilla descarga. Esta última opción solo es posible a partir de la versión 2.0.

-Java

Se trata del modo de trabajo que se encuentra por defecto en nuestro programa. Al iniciar el software por primera vez, y si este no tuvo ningún inconveniente, debería encontrarse en el modo “Standard”. Dicho modo, trabaja con el motor de Java para la pre-visualización y la exportación del sketch. Esto quiere decir que exportará el sketch en un Java Applet. Está pensado para la creación de aplicaciones de escritorio. Siempre se recomienda que se trabaje en este modo, más allá de que luego quiera pasarse a otro. Esto se debe a la facilidad y la precisión (puesto Processing es principalmente pensado para Java). Luego de esto, será cuestión de pasar de modo y adaptar lo que no funcione de manera óptima. En este modo no se pueden exportar los Applets para web, solo para aplicaciones de escritorio. En versiones anteriores a la 2.0, el modo Java era conocido como **Standard**.

-Android

El modo de trabajo “Android” consiste en, como bien indica su nombre, crear aplicaciones y programas para dispositivos Android (Sistema Operativo para móviles distribuido por Google). No obstante, para trabajar en Android es condición que se encuentre instalado en el ordenador el SDK de Android. Este se encuentra disponible de manera gratuita en <http://developer.android.com/sdk> y admite sistemas Linux, Macintosh y Windows. Sin embargo, la forma de trabajo no es exactamente la misma, por lo que podría haber inconvenientes o incompatibilidades en las aplicaciones. Un ejemplo común es que algunas galería no funcionen correctamente o simplemente se desestabilicen. Dado algunos cambios realizados por Google, las versiones 1.5.x de Processing permiten emplear el modo Android. Se recomienda descargar versiones a partir de la 2.0 si se desea desarrollar para estos sistemas.

Es un modo pensado básicamente para programadores en sistemas móviles. Se debe consultar, además, la versión de Processing en relación al SDK y en relación a la última versión de Android (es posible que Processing no soporte las más modernas, sin embargo, estas no se distribuyen al público en general). Ante cualquier inconveniente, consultar la referencia en el sitio web: <http://wiki.processing.org/w/Android>.

-Javascript

Con la creación de la librería para Javascript llamada Processing.js, nace la posibilidad de traducir los sketches en objeto de Canvas (tag de HTML5 que permite dibujar con Javascript). A partir de la versión 2.0, se incluye

el modo Javascript, el cual permite programar en el entorno de Processing y probar el código directamente en el navegador predeterminado (el cual debe soportar Canvas, se recomienda la última versión de Mozilla Firefox). Además de esto, al exportar el sketch, se creará el Applet en su versión web (Javascript), listo para usar en cualquier sitio HTML.

Debe tenerse en cuenta que existe la posibilidad de que algunas funciones no se encuentren del todo soportadas en el modo Javascript, o funcionen de una manera distinta. Ante cualquier inconveniente, consultar la referencia en el sitio web: <http://processingjs.org/reference/>.

A pesar de que las sintaxis de ambos entornos derivan del lenguaje de C/C++, el trabajo por detrás no es precisamente el mismo. Por lo tanto, hay que tener cierto cuidado a la hora de programar, ya que algunas estructuras pueden no comportarse como uno lo espera. Mas allá de eso, existen ciertos comandos especiales dedicados íntimamente al trabajo en el modo Javascript. El comando `@pjs`, permite la inserción de directivos que solo leerá el modo Javascript (la librería Processing.js). Por ende, debe ser incluido en un comentario y arriba de todo del sketch.

```
/* @pjs directivo="valor"; */
```

Cabe destacar que hay que tener cierto cuidado con la forma en la que se escriben los directivos. No es igual el tratamiento de una línea simple o multi-línea:

```
/* @pjs directivo_1="valor"; directivo_2="valor"; ... */  
/* @pjs directivo_1="valor";  
   directivo_2="valor";  
   ...  
*/
```

Si bien los directivos no son muchos, son útiles para controlar el flujo del sketch y prepararlo para la web. Una de las utilidades que posee es la precarga de ciertos elementos, por ejemplo, las imágenes. El directivo `preload` es utilizado para cargar las imágenes mientras se carga el sketch. Las imágenes deben encontrarse en la carpeta "data" y ser cargadas a través de la función `loadImage()` o `requestImage()`:

```
/* @pjs preload="miImagen.jpg"; */  
  
void setup(){  
    size(200,200);  
    noLoop();  
}  
void draw(){  
    background(255);  
    PImage im = loadImage("miImagen.jpg");  
    image(im, 50,50, 100,100);  
}
```

Otra utilidad es la precarga de las fuentes de texto, que últimamente cobran un impulso importante en los sitios web. Para esto se utiliza el directivo `font`, el cual se utiliza con la función `createFont()`. La carga es realizada a través de la regla `@font-face` de CSS.

```
/* @pjs font="Arial.ttf"; */  
  
void setup(){  
    size(200,200);  
    noLoop();  
    textFont(createFont("Arial",32));  
}  
void draw(){
```

```

    background(255);
    String t = "P.js";
    float tw = textWidth(t);
    fill(0);
    text(t, (width-tw)/2, (height+32)/2);
}

```

Ya que los sketch realizados en el modo Javascript están destinados a la web, se encuentran permanentemente ejecutándose. Muchas veces, en sketch complejos, esto puede ocupar mucha capacidad del ordenador (especialmente en ordenadores más antiguos). Por lo tanto, se encuentra disponible el directivo `pauseOnBlur`. El mismo puede recibir solo dos valores, `true` o `false`. Si el valor es `true`, cuando el usuario no tenga la ventana donde se encuentra el sketch puesta “en foco”, el mismo será “pausado”. Si es `false`, este continuará ejecutándose (esta opción es por defecto):

```
/* @pjs pauseOnBlur="true"; */
```

Por otra parte, también existe un directivo que controla los eventos del teclado. El directivo `globalKeyEvents` permite controlar si las teclas serán habilitadas para el sketch constantemente, incluso mientras navega por el mismo sitio, o si se prefiere solo habilitarla para cuando el usuario está en foco. Recibe solo dos valores, `true` o `false`. Si el valor es `true`, los eventos del teclado serán habilitados por mas que no se esté en foco en el sketch. En cambio, si es `false`, será deshabilitado (esta opción es por defecto):

```
/* @pjs globalKeyEvents="true"; */
```

Dependiendo de la configuración por defecto del navegador, los elementos de Canvas pueden o no tener aplicada la condición de suavizado en los bordes. Para eso se utiliza el directo `crisp`. Este directivo permite obligar a que esto si se cumpla en la representación de `line()`, el `triangle()`, y el `rect()` primitivas de dibujo. Esto permite líneas suaves y limpias. Por otro lado, es muy posible que su empleo necesite un incremento del trabajo del CPU. Recibe solo dos valores, `true` o `false`:

```
/* @pjs crisp="true"; */
```

Los directivos son útiles y están en constante expansión. Para evitar confusiones y ver algunos ejemplos de su funcionamiento, consultar la referencia: <http://processingjs.org/reference/>.

-Experimental

El modo Experimental nace con la fusión de dos proyectos realizados por Casey Reas y Ben Fry. Consiste en una especialización del software orientado a programadores con una demanda más precisa en la producción de programas. Entre otras cuestiones, posee una consola de errores, un debugger y diversas herramientas que lo acompañan (como un inspector de variables).

Este modo aún se encuentra en proceso de pruebas y puede ser muy inestable.

Unidad 42

Extensión: Figuras 3D

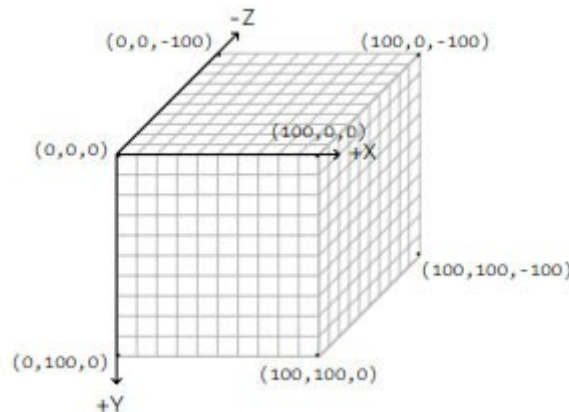
Elementos que se introducen en esta Unidad:

`size()`, `P2D`, `P3D`, `JAVA2D`, `OPENGL`,
`rotateX()`, `rotateY()`, `rotateZ()`, `lights()`, `camera()`, `texture()`, `box()`, `sphere()`,
`beginRaw()`, `endRaw()`

El software Processing admite la posibilidad de crear aplicaciones y programas que hagan su render en tres dimensiones. No obstante, no se recomienda que los programadores aborden esta temática de entrada, y traten de generar soluciones en dos dimensiones. Por cuestiones que conllevan a la creación de un manual simple, no abordaremos en la explicación de gráficos 3D (ya que la extensión sería de otro libro). Aún así, vemos conveniente explicar la base para sembrar en el lector una semilla de curiosidad.

-Forma 3D

Las formas 3D son posicionadas en una ventana de representación con **3 coordenadas**. En construcciones 2D, estas coordenadas son la coordenada-x y la coordenada-y, en 3D se suma la coordenada-z. Processing usa, como coordenada de origen, el $(0, 0, 0)$ en la esquina superior izquierda, y mientras las coordenadas x e y aumentan, la coordenada-z va decreciendo.



Las figuras son dibujadas agregando la tercer coordenada-z al sistema de coordenadas. En estos casos, las funciones `point()`, `line()` y `vertex()` funcionan exactamente igual en un entorno 3D que uno 2D, solo hay que agregar una tercer coordenada. Sin embargo, mientras las funciones `translate()` y `scale()` funcionan de la misma manera, la función `rotate()` sufre un cambio. Se utiliza una función de rotación para cada coordenada. Entonces, conseguimos las funciones `rotateX()`, `rotateY()` y `rotateZ()`. Cada uno rota alrededor del eje con el cual es nombrado. También, las funciones `pushMatrix()` y `popMatrix()` funcionan de la misma forma que en un entorno 2D.

Sabiendo esto, hay que entender un punto clave en el trabajo 3D. Antes de comenzar a incluir sintaxis con una tercer coordenada, es necesario decirle a Processing que motor render necesita utilizar (en este caso, uno 3D). Processing posee, en principio, cuatro diferentes motores de render:

JAVA2D: Motor por defecto de Processing para render en 2D.

P2D: Motor para render 2D mucho más rápido que JAVA2D, pero de menos calidad.

P3D: Motor para render 3D.

OPENGL: Motor para render 3D con aceleración de Hardware, basado en OpenGL.

En versiones anteriores a la 2.0, los modos P2D y P3D funcionaban independientes. Sin embargo, a partir de la 2.0, estos modos funcionan con la ingeniería de OpenGL. Esto produce que sean más eficientes, pero también que consuman un poco más de recursos.

Por lo tanto, existen dos motores de render para 3D: el P3D y el OPENGL. Para establecerlos como el motor de nuestra aplicación, simplemente hay que incluir dicha constante como un tercer parámetro de la función `size()`. Por ejemplo:

```
size(600, 600, P3D);
```

Suele recomendarse trabajar con P3D ya que es el motor más compatible con el software y no requiere de librerías externas. No obstante, para trabajos de mayor complejidad y elaboración, a veces se opta por utilizar el motor OPENGL. En versiones anteriores a la 2.0, requiere que se incluya en el sketch la librería que permite el funcionamiento de motor OpenGL. La siguiente línea de código debe ir arriba de todo en el programa:

```
import processing.opengl.*;
```

Para versiones posteriores a la 2.0, la librería ha sido eliminada y el motor OpenGL fue incluido en el núcleo del software. Por lo tanto, no es necesario importar la librería.

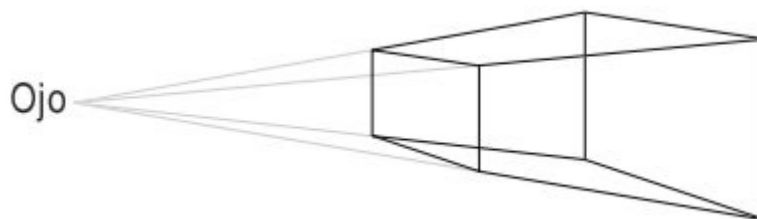
Luego puede hacerse el llamado correspondiente a la función `size()`:

```
size(600, 600, OPENGL);
```

A partir de esta declaración, es posible dibujar en 3D. Incluso es posible tener un resultado de salida de un componente 3D a través de las funciones `beginRaw()` y `endRaw()`. Los componentes de salida para 3D requieren un formato específico (ya que luego pueden usarse en programas de modelado 3D). Formatos como DXF y OBJ deben ser reconocidos a través de una librería dedicada. (más información en: <http://processing.org/reference/libraries/>)

-Cámara

Todos los modelos se componen, básicamente, de una figura en una escena y una cámara que lo observa. Processing, por lo tanto, ofrece un mapeo explícito de una cámara analógica, a través de OpenGL. En internet existe un gran número de documentación acerca de OpenGL, distribuido de manera gratuita. La cámara en Processing puede ser definida por unos pocos parámetros en cuestión: la distancia focal, y la lejanía y la cercanía de los planos. La cámara contiene el "Plano de Imagen", una captura "bidimensional", de lo que se observa.



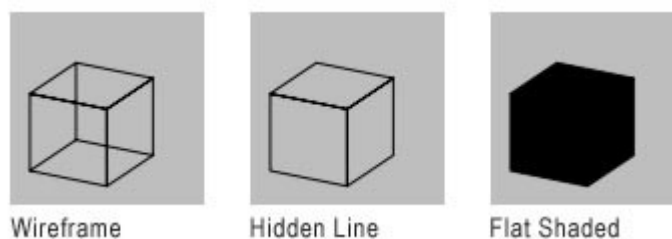
El render requiere tres transformaciones. La primera de ellas es llamada transformación de *vista*. Esta transformación posiciona y orienta a la cámara en el mundo. Al establecerse una transformación de vista (expresada como una matriz de 4 x 4), de manera implícita define que el "punto focal" es el origen. En Processing, una forma sencilla de establecer la transformación vista es con la función `camera()`. Por encima

de la transformación de vista se encuentra la transformación de *modelo*. Usualmente, estas son multiplicadas una con la otra y, de ese modo, consideradas como la transformación del *modelo-vista*. La posición de la transformación de modelo en la escena es relativo a la cámara. Finalmente, se plantea la transformación de *proyección*, la cual se basa en las características internas de la cámara.

-Materiales y Luz

Luego de que las formas están generadas y transformadas, es muy común que estas se conviertan en una imagen estática o en una animación. En trabajo de un render 3D es, principalmente, el modelado matemático, y la eficiente computación en la interacción entre luz y superficie. El modelo de *Ray-tracing* (Trazado con rayos) y tras técnicas avanzadas son las más básicas variantes en el render. *Ray-tracing* modela a partir de rayos de luz, emergentes de una fuente lumínica que rebota sobre las superficies de los objetos, hasta que golpea a la escena completa. Al ser un cálculo matemático, muchas veces tiene problemas en reproducir fenómenos naturales, como el "sangrado del color", cuando un color se refleja sobre otra superficie. Suele solucionarse con técnicas como el modelo de "iluminación global", que se aplica, no sólo por la luz que viene directamente de las fuentes de luz predefinidas, sino también la luz reflejada de las superficies regulares en una escena.

Existen tres métodos de render que no requieren cálculos de luz: *wireframe* (alambre), *hidden-line* (línea oculta) y *flat-shaded* (plano sombreado):

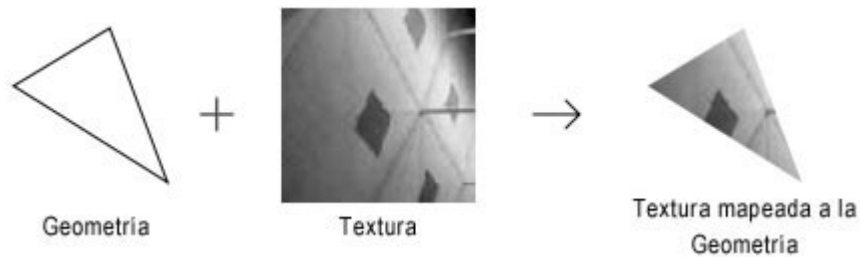


El wireframe es el método más simple para crear un render. Este simplemente muestra las líneas de los contornos de los polígonos en su color básico. Está implementado en Processing por dibujar con una función `stroke()` (contorno) y sin un `fill()` (relleno). El siguiente en complejidad es hidden-line. En este modelo solo los contornos son dibujados, pero se ocultan aquellos que son enmascarados por superficies. Processing no tiene una forma de generar esto directamente, no obstante, se obtiene el mismo resultado usando una función `stroke()` y una `fill()` cuyo color sea el mismo que el del fondo. El último modelo, flat-shaded, se genera de forma similar, solo que la función `fill()` posee un color distinto que el del fondo.

La simulación de luces y superficies, de manera correcta, producen un resultado más realista. La primera función lumínica que nos aporta Processing es la función `ambientLight()`. Se trata de una simulación de luz ambiente. Esta función interactúa con el ambiente de color del programa, el cual se establece con la función `ambient()`. Dicha función trabaja por conseguir los parámetros de `fill()` y `stroke()`.

La función `lightSpecular()` establece el color especular para las luces. La calidad especular del color interactúa con la calidad especular del material a través de la función `specular()`. La función `specular()` establece el color de los materiales, los cuales establecen el color de las luces. También se presentan funciones de luces de mayor complejidad. Entre ellas, `pointLight()`, la cual crea una luz puntual, `directionalLight()`, la cual crea una luz direccional, y `spotLight()` que crea una especie de luz "chispeante", como una gota. Reciben distintos parámetros porque cada luz es única. Todas las luces deben ser reseteadas al final del bloque `draw()`, puesto que necesitan ser re-calculadas.

La textura de los materiales son un elemento importante en el realismo de la escena 3D. Processing permite que las imágenes sean mapeadas a las caras de los objetos. La textura se deforma del mismo modo que el objeto se deforma. Para conseguir un mapeo, es necesario que la cara obtenga las coordenadas 2D de la textura.



Las texturas son mapeadas a las formas geométricas con una versión de la función `vertex()` con dos parámetros adicionales, conocido como `u` y `v`. Estos parámetros son las coordenada-`x` y la coordenada-`y` de la textura a ser mapeada. Además, es necesario establecer la textura a utilizar con la función `texture()`. Como la imagen esta cargada en el bloque `setup()`, no es re-calculada en cada ciclo del bloque `draw()`.

-Ejemplos

Ejemplo 1: Dibujando en 3D (Transformación)

`//Rotar un rectángulo alrededor de su eje x y su eje y`

```
void setup() {
  size(400, 400, P3D);
  fill(204);
}

void draw() {
  background(0);
  translate(width/2, height/2, -width);
  rotateY(map(mouseX, 0, width, -PI, PI));
  rotateX(map(mouseY, 0, height, -PI, PI));
  noStroke();
  rect(-200, -200, 400, 400);
  stroke(255);
  line(0, 0, -200, 0, 0, 200);
}
```

Ejemplo 2: Dibujando en 3D (Luces y Formas 3D)

`//Dibujar una esfera arriba de una caja y mover las coordenadas con el mouse`
`//Presionar el botón del mouse para encenderlos con una luz`

```
void setup() {
  size(400, 400, P3D);
}

void draw() {
  background(0);
  if (mousePressed == true) { //Si el mouse se oprime
    lights(); //encender la luz
  }
  noStroke();
  pushMatrix();
  translate(mouseX, mouseY, -500);
  rotateY(PI/6); //Rotar alrededor del eje y
}
```

```

    box(400, 100, 400);          //Dibujar caja
    pushMatrix();
    popMatrix();
    translate(0, -200, 0);      //Posición de la esfera
    sphere(150);                //Dibujar la esfera sobre la caja
    popMatrix();
}

```

Ejemplo 3: Construyendo Formas 3D

//Dibujar un cilindro centrado en el eje Y, que vaya desde y=0 a y=height.
//El radio de arriba puede ser diferente de el radio de abajo
//y el número de lados dibujados es variable

```

void setup() {
    size(400, 400, P3D);
}

void draw() {
    background(0);
    lights();
    translate(width/2, height/2);
    rotateY(map(mouseX, 0, width, 0, PI));
    rotateZ(map(mouseY, 0, height, 0, -PI));
    noStroke();
    fill(255, 255, 255);
    translate(0, -40, 0);
    dibujarCilindro(10, 180, 200, 16);          //Dibuja una mezcla entre
                                                //cilindro y cono
    //dibujarCilindro(70, 70, 120, 64);        //Dibuja un Cilindro
    //dibujarCilindro(0, 180, 200, 4);        //Dibuja una pirámide
}

void dibujarCilindro(float topRadius, float bottomRadius, float tall, int sides)
{
    float angle = 0;
    float angleIncrement = TWO_PI / sides;
    beginShape(QUAD_STRIP);
    for (int i = 0; i < sides + 1; ++i) {
        vertex(topRadius*cos(angle), 0, topRadius*sin(angle));
        vertex(bottomRadius*cos(angle), tall, bottomRadius*sin(angle));
        angle += angleIncrement;
    }
    endShape();
    //Si no es un cono, dibujar el circulo arriba
    if (topRadius != 0) {
        angle = 0;
        beginShape(TRIANGLE_FAN);
        //Punto Central
        vertex(0, 0, 0);
        for (int i = 0; i < sides + 1; i++) {
            vertex(topRadius * cos(angle), 0, topRadius *
                sin(angle));
            angle += angleIncrement;
        }
        endShape();
    }
    //Si no es un cono, dibujar el circulo abajo
    if (bottomRadius != 0) {
        angle = 0;

```



```

        beginShape(TRIANGLE_FAN);
        // Center point
        vertex(0, tall, 0);
        for (int i = 0; i < sides+1; i++) {
            vertex(bottomRadius * cos(angle), tall, bottomRadius *
                sin(angle));
            angle += angleIncrement;
        }
        endShape();
    }
}

```

Ejemplo 4: Exportar en DXF

```

//Exportar a un archivo DXF cuando la tecla R es presionada

import processing.dxf.*;      //Solo en versiones anteriores a la 2.0
boolean record = false;

void setup() {
    size(400, 400, P3D);
    noStroke();
    sphereDetail(12);
}

void draw() {
    if (record == true) {
        beginRaw(DXF, "output.dxf"); //Iniciar al acción de guardar
    }
    lights();
    background(0);
    translate(width/3, height/3, -200);
    rotateZ(map(mouseY, 0, height, 0, PI));
    rotateY(map(mouseX, 0, width, 0, HALF_PI));
    for (int y = -2; y < 2; y++) {
        for (int x = -2; x < 2; x++) {
            for (int z = -2; z < 2; z++) {
                pushMatrix();
                translate(120*x, 120*y, -120*z);
                sphere(30);
                popMatrix();
            }
        }
    }
    if (record == true) {
        endRaw();
        record = false;          //Detener la acción de guardar
    }
}

void keyPressed() {
    if (key == 'R' || key == 'r') { //Presionar R para guardar el archivo
        record = true;
    }
}

```

Ejemplo 5: Importar un archivo OBJ

```

//Importar y mostrar un archivo OBJ

```

```

import saito.objloader.*;
OBJModel model;

void setup() {
  size(400, 400, P3D);
  model = new OBJModel(this);
  model.load("chair.obj"); //Los modelos deben estar en la carpeta data
  model.drawMode(POLYGON);
  noStroke();
}

void draw() {
  background(0);
  lights();
  pushMatrix();
  translate(width/2, height, -width);
  rotateY(map(mouseX, 0, width, -PI, PI));
  rotateX(PI/4);
  scale(6.0);
  model.draw();
  popMatrix();
}

```

Ejemplo 6: Control de la Cámara

//La cámara se mueve en relación al mouse observando el mismo punto

```

void setup() {
  size(400, 400, P3D);
  fill(204);
}

void draw() {
  lights();
  background(0);
  //Cambia la altura de la cámara con mouseY
  camera(30.0, mouseY, 220.0, //ojoX, ojoY, ojoZ
        0.0, 0.0, 0.0, //centroX, centroY, centroZ
        0.0, 1.0, 0.0); //arribaX, arribaY, arribaZ
  noStroke();
  box(90);
  stroke(255);
  line(-100, 0, 0, 100, 0, 0);
  line(0, -100, 0, 0, 100, 0);
  line(0, 0, -100, 0, 0, 100);
}

```

Ejemplo 7: Material

//Varía la reflexión especular del material
//con la posición en X del mouse

```

void setup() {
  size(400, 400, P3D);
  noStroke();
  colorMode(RGB, 1);
  fill(0.4);
}

void draw() {

```

```

    background(0);
    translate(width/2, height/2);
    //Establece el color de la luz especular
    lightSpecular(1, 1, 1);
    directionalLight(0.8, 0.8, 0.8, 0, 0, -1);
    float s = mouseX / float(width);
    specular(s, s, s);
    sphere(100);
}

```

Ejemplo 8: Iluminación

//Dibujar un cubo con 3 diferentes tipos de luz

```

void setup() {
    size(400, 400, P3D);
    noStroke();
}

void draw() {
    background(0);
    translate(width/2, height/2);
    //Luz puntual naranja a la derecha
    pointLight(150, 100, 0,          //Color
              200, -150, 0);        //Posición
    //Luz direccional azul a la izquierda
    directionalLight(0, 102, 255,    //Color
                   1, 0, 0);        //La dirección en los ejes x, y, z
    //Luz chispeante amarilla al frente
    spotLight(255, 255, 109,        //Color
             0, 40, 200,            //Posición
             0, -0.5, -0.5,        //Dirección
             PI/2, 2);              //Ángulo
    rotateY(map(mouseX, 0, width, 0, PI));
    rotateX(map(mouseY, 0, height, 0, PI));
    box(200);
}

```

Ejemplo 9: Mapeo de la Textura

//Cargar una imagen y mapearla en un cilindro y en un paralelogramo

```

int tubeRes = 32;
float[] tubeX = new float[tubeRes];
float[] tubeY = new float[tubeRes];
PImage img;

void setup() {
    size(400, 400, P3D);
    img = loadImage("berlin-1.jpg");
    float angle = 270.0 / tubeRes;
    for (int i = 0; i < tubeRes; i++) {
        tubeX[i] = cos(radians(i * angle));
        tubeY[i] = sin(radians(i * angle));
    }
    noStroke();
}

void draw() {
    background(0);
}

```

```

translate(width/2, height/2);
rotateX(map(mouseY, 0, height, -PI, PI));
rotateY(map(mouseX, 0, width, -PI, PI));
beginShape(QUAD_STRIP);
texture(img);
for (int i = 0; i < tubeRes; i++) {
    float x = tubeX[i] * 100;
    float z = tubeY[i] * 100;
    float u = img.width / tubeRes * i;
    vertex(x, -100, z, u, 0);
    vertex(x, 100, z, u, img.height);
}
endShape();
beginShape(QUADS);
texture(img);
vertex(0, -100, 0, 0, 0);
vertex(100, -100, 0, 100, 0);
vertex(100, 100, 0, 100, 100);
vertex(0, 100, 0, 0, 100);
endShape();
}

```

Unidad 43

Extensión: XML

Elementos que se introducen en esta Unidad:

XML, loadXML(), getChildCount(), getChild(), getChildren(), getContent(), getInt(), getFloat(), getString(), getName()

El lenguaje de XML es un lenguaje de maquetado, desarrollado por la W3C para el intercambio de información. Si bien es muy utilizado en internet, no solo se utiliza en ese ambiente. Se encuentra presente en los RSS de los blogs y otros sitios. Por su flexibilidad en la creación de etiquetas (tags), es muy sencillo levantar su contenido y seleccionar que quiere levantarse. Por ejemplo, se puede seleccionar la lectura solo del título de un documento, o del contenido general.

-Objeto XML

A partir de la versión 2.0, el objeto XML pasa a formar parte de los objetos preestablecidos de Processing. La clase XML se declara como cualquier objeto. Para asignar un documento, se utiliza la función `loadXML()`. Esta conviene que se encuentre en el bloque `setup()`.

```
XML xml;
xml = loadXML("sitio.xml");
```

Los documentos XML deben encontrarse en la carpeta *data*. No obstante, es posible levantar documentos XML directamente desde la web.

-Funciones de XML

Para tener más control sobre el objeto XML, el software Processing ofrece una gran variedad de funciones. En estos casos, las funciones más clásicas son las que sirven para tener control sobre los *hijos* de un documento. Suelen utilizarse las siguientes funciones:

<code>getChildCount()</code>	Regresa el número de hijos de un documento
<code>getChild()</code>	Regresa solo un hijo
<code>getChildren()</code>	Regresa todos los hijos de un documento como un Array
<code>getContent()</code>	Regresa el contenido de un elemento

En orden de aplicar esto, a continuación se presenta un ejemplo donde se crea un simple lector de RSS. El lector levanta una URL donde se encuentra un XML, por lo tanto no es necesario tener el documento en la carpeta *data*. Luego, simplemente imprime los últimos títulos de las noticias en la consola (el sketch se mostrará vacío):

```
//Cargar XML
String url = "http://processing.org/updates.xml";
XML rss = loadXML(url);

//Conseguir el título de cada elemento
XML[] titleXMLElements = rss.getChildren("channel/item/title");
for (int i = 0; i < titleXMLElements.length; i++) {
    String title = titleXMLElements[i].getContent();
    println(i + ": " + title);
}
```

Existen, además, otras funciones un poco más dedicadas. Estas sirven para devolver diversos valores de los atributos del documento:

<code>getInt()</code>	Regresa un atributo entero de un elemento
<code>getFloat()</code>	Regresa un atributo decimal de un elemento
<code>getString()</code>	Regresa un atributo String de un elemento
<code>getName()</code>	Regresa el nombre de un elemento

-Otras Versiones

Ocurre una confusión al momento de declarar el objeto XML. En versiones anteriores a la 2.0, Processing utilizaba un elemento XML que derivaba de una librería. En versiones anteriores a la 1.5.1, la librería debía incluirse. Es recomendable actualizarse a la última versión, especialmente por la gran cantidad de ventajas que ofrece utilizar el objeto XML y no el elemento. Para el ejemplo anterior del lector de RSS, en versiones anteriores a la 2.0 debería funcionar así:

```
//Método para anterior a 2.0
//Cargar XML
String url = "http://processing.org/updates.xml";
XMLElement rss = new XMLElement(this, url);

//Conseguir el título de cada elemento
XMLElement[] titleXMLElements = rss.getChildren("channel/item/title");
for (int i = 0; i < titleXMLElements.length; i++) {
    String title = titleXMLElements[i].getContent();
    println(i + ": " + title);
}
```

Apartados: Documentación

-Apartado 1: Herramientas

El software de Processing permite la implementación de Herramientas (Tools) para facilitar el trabajo al usuario. Algunas de estas vienen instaladas en el propio programa, y puede accederse a ellas desde el menú *Tools*. Por defecto, se incluyen:

-*Auto Format (Auto-Formato)*: Intenta formatear el código para hacer un *layout* (diseño) más legible y humano. Se añaden o eliminan espacios y saltos de líneas para hacer un código mas "limpio". Para acceder rápidamente puede utilizarse su atajo en Windows: *Control + T*.

-*Archive Sketch (Archivar Sketch)*: Archiva una copia del sketch en un documento *.zip*. La copia se almacena en el mismo directorio que el sketch.

-*Color Selector (Selector de Color)*: Simple interfaz para seleccionar colores en RGB, HSB o hexadecimal.

-*Create Font (Crear Fuente)*: Convierte las fuentes de texto en el archivo de fuentes que utiliza Processing para trabajar y lo añade al directorio del sketch. Se abre un cuadro de diálogo que le dan opciones para configurar el tipo de letra, su tamaño, si se trata de anti-aliasing, y si todos los caracteres deben ser generados.

-*Fix Encoding & Reload (Arreglar Codificación y Recargar)*: Los sketches que contienen caracteres que no son código ASCII y fueron guardados en Processing 0140 o versiones anteriores, posiblemente se vean muy extraños cuando son abiertos. Caracteres como diéresis, o japoneses. Al ser abiertos en una nueva versión, la herramienta los mostrará como en las versiones antiguas, y al momento de guardar el sketch nuevamente, lo codificará con la nueva codificación UTF-8.

Además, existen herramientas creadas por usuarios que contribuyen al programa. Estas pueden descargarse de forma libre y gratuita desde el sitio de herramientas de Processing (<http://processing.org/reference/tools/>). Las herramientas de contribución deben ser descargadas por separado y colocadas dentro de la carpeta "Tools" del programa Processing (para encontrar la ubicación de Processing en el ordenador, abra la ventana de Preferencias de la aplicación de Processing y busque "Sketchbook location" en la parte superior). Se debe copiar la carpeta de la herramienta descargada dentro de la carpeta "Tools". Es posible que se necesite crear la carpeta "Tools" si es la primer herramienta que se instala.

En las versiones posteriores a la 2.0, existe una opción dentro del menú "Tools", la opción "Add Tools..." (Añadir Herramientas). A partir de esta opción, Processing busca nuevas herramientas aun no instaladas y las presenta en un entorno amigable para su instalación. Luego de esto solo hay que seleccionar la herramienta deseada y apretar el botón "Install".

-Apartado 2: Librerías

Para producciones que requieren mucho desarrollo, a veces es necesario implementar Librerías (Libraries) que potencien nuestros programas. No se trata de mucho mas que una gran cantidad de código. Para insertar alguna librería de las que vienen por defecto, se debe dirigir al menú "Sketch" y luego a "Import Library...". Aquí mismo aparecerán las librerías disponibles que vienen instaladas por defecto en Processing:

-*Video*: Interfaz de la librería GSVideo para utilizar una cámara, reproducción de películas y creación, también, de las mismas. Los usuarios de linux deberán instalar el gstreamer para utilizar esta librería. En versiones anteriores a la 2.0, la interfaz era del Quicktime de Apple.

-*Arduino*: Permite control directo de una placa Arduino a través de Processing.

-*Network*: Envía y recibe datos vía Internet a través de la creación de simples Clientes y Servidores.

-*Minim*: Utiliza la JavaSound API para proveer una fácil uso como librería de sonido. Un simple API que provee un gran manejo y flexibilidad al usuario que desea implementar sonido.

- Netscape.Javascript*: Métodos de comunicación entre Javascript y Java Applets exportados de Processing.
- Serial*: Soporte para el envío de data entre Processing y un hardware externo, vía comunicación serializada (RS-232).
- PDF Export*: Genera archivos PDF
- DXF Export*: Líneas y triángulos en los modos de render PD3 o OPENGL, pueden ser exportados como un archivo DXF.

Con las actualizaciones, existen librerías que han sido trasladadas a la base de datos y reemplazadas por funciones o automatizaciones. Estas son:

- Candy SVG Import*: Esta biblioteca se ha trasladado a la base de código de Processing en la versión 149. Para la carga de gráficos SVG, se recomienda utilizar las funciones `Pshape()`, `loadShape()` y `shape()`.
- XML Import*: Esta biblioteca se ha trasladado a la base de código de Processing en la versión 149. Para cargar datos XML, se recomienda utilizar la clase `XML` y la función `loadXML()`. En versiones anteriores a la 2.0, se deberá utilizar el elemento `XMLElement`.
- OpenGL*: Soporte para exportar sketch con aceleración en gráficos a través de OpenGL. Utiliza la librería JOGL. A partir de la versión 2.0 ha dejado de existir como librería para encontrarse en el núcleo del software. Por lo tanto, solo es necesario llamar a la librería en versiones anteriores a la 2.0.

Cabe destacar que cada librería tiene una documentación correspondiente con la cual se entiende su implementación y correcta sintaxis. La misma se encuentra en la referencia del sitio (<http://processing.org/reference/libraries/>). En dicho sitio, además, podremos encontrar uno de los puntos fuertes de Processing, las librerías creadas por usuarios contribuidores. Hay a disposición más de 100 librerías, todas con un atractivo diferente. Las librerías de contribución deben ser descargadas por separado y colocadas dentro de la carpeta "Libraries" de tu programa de Processing (para encontrar la ubicación de Processing en el ordenador, abra la ventana de Preferencias de la aplicación de Processing y busque "Sketchbook location" en la parte superior). Si todo es correcto, la librería debería aparecer en el menú "Sketch", "Import Library..."

En las versiones posteriores a la 2.0, existe una opción dentro del menú "Sketch", "Import Library...", la opción "Add Library..." (Añadir Librería). A partir de esta opción, Processing busca nuevas librerías aun no instaladas y las presenta en un entorno amigable para su instalación. Luego de esto solo hay que seleccionar la librería deseada y apretar el botón "Install".

-Apartado 3: Processing.js

Al incluirse la tag Canvas como un estandar de HTML5 (`<canvas></canvas>`) los programadores web son capaces de "dibujar" en internet con Javascript. Lo cierto es que no se tardó demasiado en notar que las sintaxis de Canvas es muy similar a la de Processing. John Resig (luego es desarrollado por Florian Jenett) crea una librería que "traduce" un sketch (.pde) de Processing al Javascript necesario para que funcione en Canvas. De esta forma, insertar un objeto Canvas en la web es mucho mas práctico y eficiente que un Java Applet.

Para comenzar a implementarlo, solo hay que dirigirse a <http://processingjs.org> y descargar la ultima versión de la librería, la cual tiene soporte para todos los navegadores web modernos. Luego de esto, solo es necesario tener un `loquesea.html` y un `loquesea.pde` (que será el propio sketch). De manera simple, en el `<head>` de tu `loquesea.html`, se linkea la librería de la siguiente manera:

```
<script src="processing.js"></script>
```

Siempre uno asegurándose que el `src` sea el directorio correcto. Posteriormente, en el `<body>`, se crea una etiqueta de Canvas de la siguiente manera:


```
<canvas data-processing-sources="loquesea.pde"></canvas>
```

Una vez colocado esto, el sketch de Processing debería estar funcionando sin ningún problema como un objeto de Canvas. Es posible que algunas funciones de Processing no funcionen correctamente o directamente no funcionen al pasarlas a Canvas (se está constantemente trabajando en ello). Ante cualquier duda, lo ideal es consultar la referencia de Processing.js: <http://processingjs.org/reference/>.

-Apartado 4: Wiring

Wiring es un entorno de programación de código libre para microcontroladores. Permite la escritura de programas multiplataforma para controlar diversos dispositivos conectados a una amplia gama de tableros de microcontrolador y así crear todo tipo de códigos creativos, objetos interactivos, espacios o experiencias físicas. El entorno está cuidadosamente creado pensando en diseñadores y artistas, y así fomentar una comunidad donde los principiantes, a través de expertos de todo el mundo, compartan las ideas, conocimientos y su experiencia colectiva. Más información en: <http://wiring.org.co/>

-Apartado 5: Arduino

Arduino es una plataforma de código abierto basada en prototipos de electrónica flexible y fácil de usar mediante hardware y software. Está pensado para los artistas, diseñadores, aficionados y cualquier persona interesada en la creación de objetos o entornos interactivos. El microcontrolador de la placa se programa utilizando el lenguaje de programación de Arduino (basado en Wiring) y el entorno de desarrollo Arduino (basado en Processing). Más información en: <http://arduino.cc/>

-Apartado 6: Fritzing

Fritzing es una iniciativa de código abierto para apoyar a diseñadores, artistas, investigadores y aficionados a trabajar de manera creativa con la electrónica interactiva. Basado en Processing y Arduino. Se trata del desarrollo de una herramienta que permite a los usuarios documentar sus prototipos, compartirlos con los demás, enseñar la electrónica en un salón de clases, y crear un diseño para la fabricación de PCB profesional. Más información en: <http://fritzing.org/>

-Apartado 7: Has Canvas

Algunos proyectos que derivan al comenzar a implementarse librerías como Processing.js, pretenden utilizar Processing en la web a través de Canvas. Sea el caso del sitio Has Canvas, uno de los más potentes. Al ingresar al sitio <http://hascanvas.com> se nos presenta una interfaz sencilla que nos permite escribir código de Processing directamente en el navegador de nuestro ordenador, y automáticamente dicho código es levantado con PHP y visualizado en forma de Canvas.

Solo con tener una cuenta en Has Canvas (a través de OpenID) se puede hacer un sketch, obtener una URL del mismo, compartirlo dentro de dicho sitio, y conseguir el recurso listo para incluir como objeto de Canvas en cualquier web.