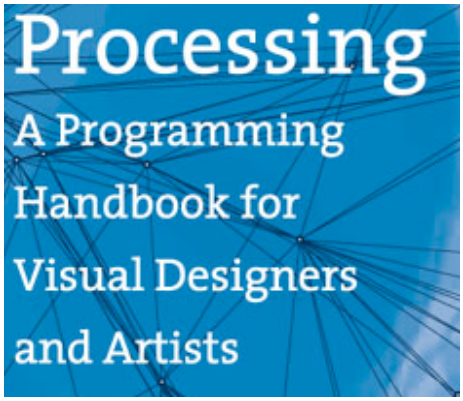


Nota del autor:

Este manual **no** es una traducción directa del libro en español, sino que, apoyándome en los sketches que se encuentran disponibles en:

http://www.processing.org/learning/books/

y en la información recogida en el I WORKSHOP INTERACTION NEW MEDIA, celebrado en el MUA (nov 2008, Alicante) he realizado una introducción a los conceptos y funciones de Processing.
Sobre todo, lo que más me ha interesado es qué puede hacer un estudiante, sin mucha base de programación, con este programa.
Espero que alguien se anime, y haga algo similar con otros programas como PureData o Arduino.

Ejemplo 0-04:

```
int x = 0; //
int y = 55; //

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

void draw() {
  background(204);
  line(x, y, x+20, y-40);                  // línea izquierda
  line(x+10, y, x+30, y-40);               // línea central
  line(x+20, y, x+40, y-40);               // línea derecha
  x = x + 1;                               // súmale 1 al valor de la X cada vez que se hace
el bucle
  if (x > 100) {                               // Si el valor de X supera el valor 100 (ancho
pantalla), entonces…
    x = -40;                               // …asigna el valor –40 al valor de X
  }
}
```

En este ejemplo, se usa un condicional (**if**), si el valor de X, pasa del valor 100 (que en este caso, es el límite de anchura de la pantalla), entonces, se le asigna el valor X= - 40, de esta forma, las líneas se vuelven a dibujar al principio de la pantalla.

También se puede ver como al final de dibujar las líneas, al valor X se le suma 1, de esta forma, se van dibujando las líneas a lo largo del eje X, ya que el **draw**() lo lee constantemente el programa.

Ejemplo 0-05:

```
void setup() {
  size(100, 100);
}

void draw() {
  background(204);
  float x = mouseX;                    // Asigno la coordenada X del ratón a la variable X
  float y = mouseY;                    // Asigno la coordenada Y del ratón a la variable Y
  line(x, y, x+20, y-40);
  line(x+10, y, x+30, y-40);
  line(x+20, y, x+40, y-40);
}
```

En este ejemplo, defino la variable (**float** = decimal) dentro del **draw** ( ) para que constantemente se vaya redefiniendo dentro del bucle que hace el programa. Si estas variables las hubiera definido en el **setup** ( ), el programa las identificaría con la posición del ratón en el primer instante de arrancar el programa y las líneas no se moverían.

Ejemplo 0-06:

```
void setup() {
  size(100, 100);
  noLoop();     // con esta orden, el draw no se redibujará en bucle constantemente
}         // si coloco la orden loop( ) o redraw ( ) en el draw, se volverá a dibujar el bucle

void draw() {
  diagonals(40, 90);
  diagonals(60, 62);
  diagonals(20, 40);
}

void diagonals(int x, int y) {
  line(x, y, x+20, y-40);
  line(x+10, y, x+30, y-40);
  line(x+20, y, x+40, y-40);
}
```

Cuando usamos la orden **noLoop** ( ), **no** podemos utilizar las ordenes de **mousePressed**( ) o **keyPressed**( ) directamente . Ver ayuda de Processing de esta orden.

En este ejemplo, podemos ver como al principio del draw( ), define unas diagonales con unas variables int (entre paréntesis). Diagonals no es una función de Processing, si no que es una función que estamos definiendo nosotros mismos, es por ello que luego necesitamos decirle al programa qué hacer cuando lee el comando de **diagonals**. Por esto, lo definimos con el **void diagonals** ( ) en el que se dibujan unas líneas (esta ya si que es una orden interna de Processing). Para ejecutar el comando de **line**, el programa necesita 4 parámetros, los dos primeros son las coordenadas iniciales de la línea en el eje X e Y respectivamente y los dos últimos, son las coordenadas finales de la línea en el eje X e Y respectivamente.

Ejemplo 0-07 :

```
int num = 20;
int[] dx = new int[num];                // declaramos una array de las coordenadas X
int[] dy = new int[num];                // declaramos una array de las coordenadas Y

void setup() {
  size(100, 100);
  for (int i=0; i<num; i++) {   //   para un intervalo entre i = 0 e i < variable num, suma 1 al valor de i
    dx[i] = i*5;                // si cumple esas condiciones, redimensiona el valor del array  dx [i]
    dy[i] = 12 + (i*6);         // si cumple esas condiciones, redimensiona el valor del array dy [i]
  }
}
void draw() {
  background(204);
  for (int i=0; i<num; i++) {   //   para un intervalo entre i = 0 e i < variable num, suma 1 al valor de i
    dx[i] = dx[i] + 1;          // si cumple esas condiciones, redimensiona el valor del array  dx [i]
    if (dx[i] > 100) {          // si el valor de dx [i] > de 100, entonces...
      dx[i] = -100;             // ... redimensiona el valor de dx [i] = -100
    }
    diagonals(dx[i], dy[i]);// ...dentro del intervalo i = 0 e i  < variable num, dibuja diagonales con las
arrays de las componentes X e Y ya redimensionadas
  }
}

void diagonals(int x, int y) {
  line(x, y, x+20, y-40);
  line(x+10, y, x+30, y-40);
  line(x+20, y, x+40, y-40);
}
```

En este ejemplo, y para no tener que programar manualmente 20 grupos de líneas (que requieren 20 variables en el eje X y 20 en el eje Y), podemos crear una **array**.

Una array puede almacenar **una lista de datos** como si fuera una variable simple. Con el comando (**for**), se programa un bucle cíclico que implica a todos los valores de la array.

La estructura de **for**, siempre funciona dentro de un intervalo que determinamos nosotros. Si la variable que metemos dentro de la estructura del **for** cumple con ese

intervalo, entonces cada vez que hace esa iteración, modificamos a esa variable con un redimensionado determinado por nosotros. En este caso **i++**, es lo mismo que escribir:

$$i = i + 1$$

Ejemplo 0-08 :

En este ejemplo, se introduce el concepto de clase (**class**). Una clase es una estructura de programación con las características esenciales para que el programa realice una determinada acción y pueden ser lo simple o complejo que queramos. Es como crear una plantilla (bloque de autocad) para que en cuando en el programa principal la hagamos llamar, no tengamos que escribir todo el código de esa clase otra vez. Es una manera de organizar el código y evitar errores, ya que cualquier modificación que hagamos desde el código principal de esa clase, no implicará modificaciones en el código específico de la clase, es decir, esa instrucción modificará automáticamente los parámetros que le digamos.

Las clases se definen en un archivo fuera del archivo principal para tener organizado el código. Este archivo se guarda donde esté el archivo principal con el nombre exacto de la clase. Por eso al abrir el archivo principal nos aparecerá una segunda pestaña en la interfaz con el nombre de la clase. En el ejemplo, por su sencillez, todo el código aparece en el mismo archivo, aunque esta práctica no es muy recomendable para después encontrar errores.

ESTRUCTURA 1: ELEMENTOS DEL CÓDIGO

// introduce un comentario, todo lo escrito en esa línea, después de las dos barras, no será leído por el programa.

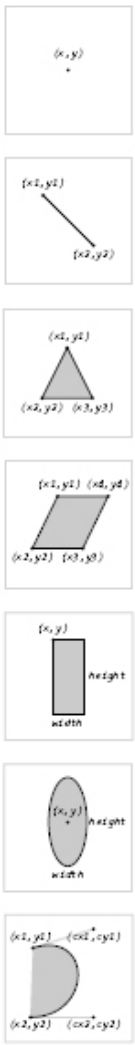
/*.....*/ Todo lo escrito dentro de esta estructura, no será leído por el programa. Se usa para hacer comentarios largos, como los de la licencia y similares.

; Es la orden para indicarle al programa que ha terminado una orden. En casi todos los comandos, es obligatorio escribirlo al final. Es un error muy común el olvidarse de escribirlo.

printLn() Sirve para que el programa escriba "algo" en la consola inferior. Se usa mucho para ver que está ocurriendo en Processing, ya que el programa hace operaciones que el ojo humano no puede percibir.

FORMA 1: COORDENADAS, PRIMITIVAS

Podemos dibujar una línea si dibujamos una serie de puntos seguidos...interesante para crear algún efecto gráfico.
Comandos para crear figuras:



Es muy importante el orden en el que se escriban los comandos, ya que el programa lee el código de arriba a abajo.
El **background** varía entre 0 (negro) y 255 (blanco). Si no se define en el código el background, se pondrá el valor 204 (gris claro) por defecto.
La función **fill** ( ) colorea las figuras o el interior de las formas y la función **stroke** ( ) colorea el borde o contorno de las figuras.
Una vez definido el color de las figuras, hasta que no se defina otro color, esté permanecerá fijo.
Se puede definir tanto **fill** () como el **stroke** () con un segundo valor, controlando la transparencia (**alpha**). El valor de alpha puede variar de 0 a 255.
Igualmente, existen las funciones **noFill** () y **noStroke** () para no colorear las formas o para no colorear el borde de las formas respectivamente.

smooth (); dibuja las formas sin pixelizar.

noSmooth (); dibuja las formas pixelizadas.

strokeWeight (); controla el ancho o grosor de las líneas o lo que sea.

strokeCap (); controla la forma como se dibuja el final de las líneas. Puede ser, ROUND (redondo), SQUARE (cuadrado) o PROJECT (mezcla de ambos).

strokeJoin (); controla el chaflán de las líneas. Puede ser, BEVEL (chaflán en líneas diagonales rectas), MITER (chaflán = 0) o ROUND (chaflán redondo con una curva).

: **ellipseMode** (CENTER ó RADIUS ó CORNER);

rectMode (CENTER ó RADIUS ó CORNERS);

DATOS 1 : VARIABLES

int	variable numérica de números enteros.
float	variable numérica de números decimales.
boolean	variable de verdadero o falso (como un enchufe)
char	variable alfabética (de 1 sólo carácter).
string	variable alfabética (de varios caracteres o un texto entero).
color	variable de color
byte	variable rara (ver ayuda de processing).

MATEMÁTICAS 1 : ARITMÉTICA, FUNCIONES

% Es un operador que calcula que queda como resto cuando un elemento lo dividimos por otro. Realmente lo que calcula es el **módulo** cuando dividimos el número delante del % entre el número que está detrás. Este operador se suele usar cuando una variable se incrementa constantemente (0, 1, 2,...), si le aplicamos este operador, el valor del módulo transformará este incremento de la variable.
La combinación de dos variables **int**, da como resultado una variable **int**.
La combinación de dos variables **float**, da como resultado una variable **float**.
La combinación de una variable **int** y una variable **float**, da como resultado una variable **float**.

_Atajos

x++	x = x + 1
x--	x = x – 1
x += 5	x = x + 5
x -= 5	x = x – 5
x *= 2	x = x * 2
x /= 2	x = x / 2
x = -x	x = x * (-1)

_Redondeando números:

La función **ceil** () redondea a un valor int, lo que haya dentro del paréntesis, siempre hacia arriba...por ejemplo:

```
int w = ceil (2.3);    //asigna el valor 2 a la variable w
```

La función **floor** () redondea a un valor int, lo que haya dentro del paréntesis, siempre hacia abajo...por ejemplo:

```
int w = floor (2.9);    //asigna el valor 2 a la variable w
```

La función round () redondea a un valor int, lo que haya dentro del paréntesis. Si sobrepasa el valor .5, se redondea al valor superior...por ejemplo:

```
int w = round (3.5);    //asigna el valor 4 a la variable w
```

La función **min** () asigna el valor más pequeño de un intervalo.
La función **max** () asigna el valor más grande de un intervalo.

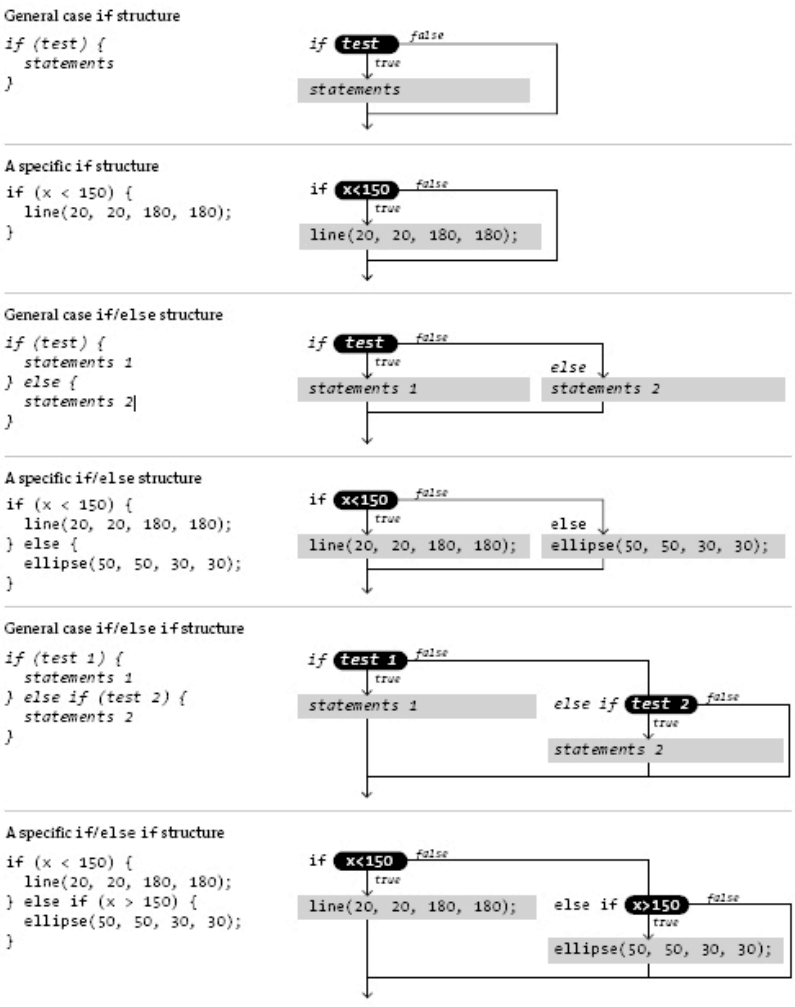
CONTROL 1: DECISIONES

El operador **!** es el opuesto de **--**. Determina cuándo dos variables no son equivalentes.

_Condicionales:

```
if (...escribo las condiciones...) {
        ...defino qué debe hacer al cumplir las condiciones...
        }
```

Esquema de los diferentes tipos de condicionales que podemos programar



else se usa en las estructuras condicionales. Significa **"en caso contrario"**...

Las estructuras condicionales se pueden encerrar dentro de otras como en una secuencia de carpetas del ordenador. Ver ejemplo 5-08 y 5-09.

_Operadores Lógicos:

&&	Significa "y"
||	Significa "o"
!	Significa "no"

CONTROL 2: REPETICIÓN

Estructura de Iteración:

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

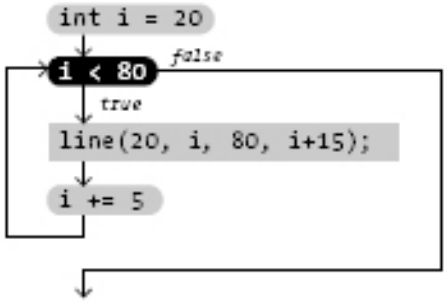
"Dentro de un intervalo en el que i va desde el valor 20 a i < 150 y donde al valor i se le va sumando 10 en cada iteración, dibuja una línea con las siguientes condiciones".

General case for structure

```
for (init; test; update) {
  statements
}
```

A specific for structure

```
for (int i = 20; i < 80; i += 5) {
  line(20, i, 80, i+15);
}
```



_Nested iteration (anidamiento de la iteración):

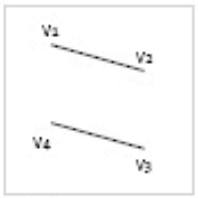
Se pueden crear estructuras de iteración en varias dimensiones. (Ver ejemplo 6-09)

FORMAS 2: VÉRTICES

Con la orden **beginShape** (), puedo empezar a dibujar los vértices de la figura que quiero dibujar. Cuando termino de definir los vértices, acabo con la orden de **endShape** (). Es muy importante el orden en el que voy definiendo los vértices de mi figura.

beginShape (**POINTS**) para dibujar sólo puntos
beginShape (**LINES**) para dibujar sólo líneas

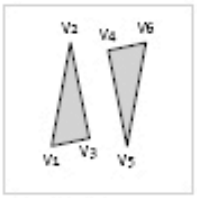
POINTS



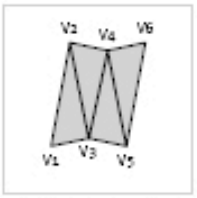
LINES

LINES

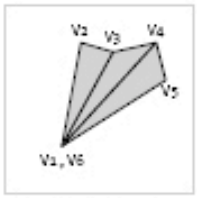
POINTS, LINES
The same data can be interpreted as a sequence of points or lines. The spatial order of the points affects what is drawn when using LINES.



TRIANGLES

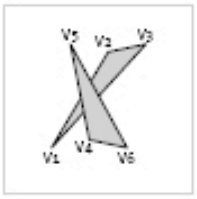


TRIANGLE_STRIP

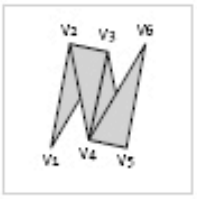


TRIANGLE_FAN

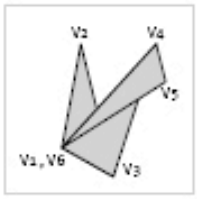
TRIANGLES, TRIANGLE_FAN, TRIANGLE_STRIP
Groups of three vertices are drawn as individual triangles or a connected group.



TRIANGLES

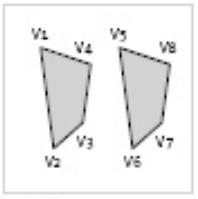


TRIANGLE_STRIP

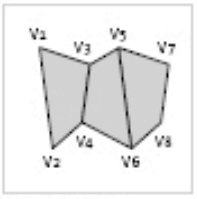


TRIANGLE_FAN

Unexpected results occur if the defined order is not followed.

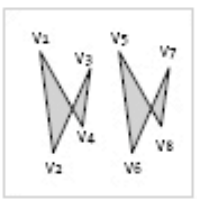


QUADS

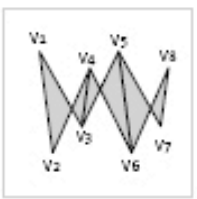


QUAD_STRIP

QUADS, QUAD_STRIP
Groups of four vertices are drawn as individual quads or a connected group. The spatial order determines whether a quad or a "bow" is drawn. Note that the order is reversed for QUADS and QUAD_STRIP.

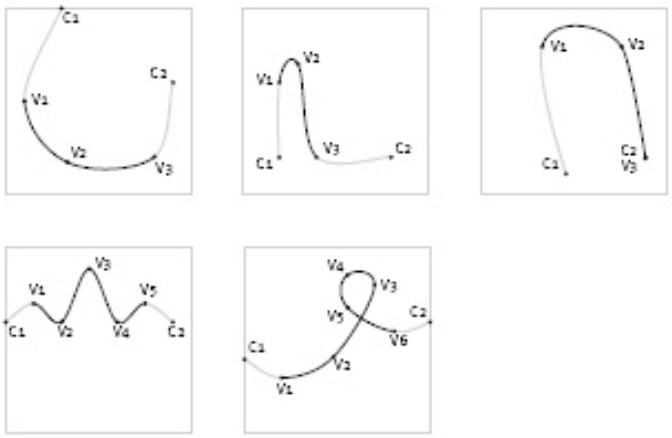


QUADS

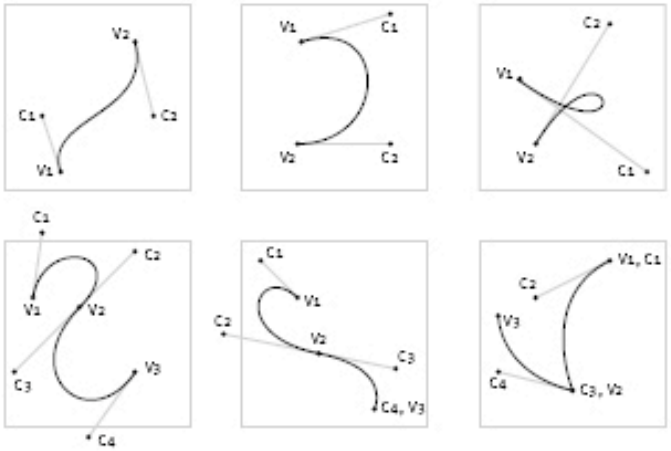


QUAD_STRIP

Todos estos comandos funcionan muy bien para dibujar líneas rectas. Para dibujar formas curvas, usaremos las funciones **curveVertex** () y **bezierVertex** () dentro de las funciones **beginShape** () y **endShape** ().



Curve vertices
The `curveVertex()` function defines coordinates that are connected with curved shapes. The first and last points are control points that define the shape of the curve at the end and beginning.



Bézier vertices
Bézier curves are defined by vertex points and control points used as parameters to the `bezierVertex()` function. The control points define the shape of the curves that are drawn between the vertex points.

Para dibujar estas curvas con precisión, es recomendable dibujar primero las curvas en un programa de dibujo vectorial como Illustrator y luego copiar las coordenadas en Processing. También se podrían importar esas coordenadas de las curvas desde un archivo. Processing incorpora una librería para leer archivos SVG.

MATEMÁTICAS 2: CURVAS

sq () operador para elevar al cuadrado un número.
sqrt () operador para calcular la raíz cuadrada de un número.
pow (num, exponent) el **num** es para multiplicar y el **exponent** es el número de de veces que hay que hacer la multiplicación.

```
float a = pow (3, 4);   //asigna 81.0 al valor de a, ya que equivale a 3*3*3*3
```

_Normalizando.Mapeando:

Normalmente convertimos los valores numéricos a un rango entre 0.0 y 1.0 para trabajar con ellos de una forma unitaria. Para ello utilizamos el comando:

norm (variable, valor más pequeño del rango a convertir, valor más grande del rango a convertir)

```
float y = norm (102.0, 0.0, 255.0);          //asigna el valor 0.4 a la variable x  (102/ 255 = 0.4)
```

lerp es una abreviatura para indicar (interpolación lineal). Ver ejemplo 8-06.

La función **map** () se usa para convertir directamente una variable de un rango numérico dado a otro buscado.

map (variable, low range1, high range1, low range2, high range2);

COLOR 1: COLOR SEGÚN NÚMEROS

background (r,g,b);
fill (r,g,b);
fill (r,g,b,alpha);
stroke (r,g,b);
stroke (r,g,b,alpha);

Para elegir color, ir al menú **Tools** y luego **Color Selector** y ahí podemos copiar las coordenadas de los RGB o lo que queramos.
Podemos conseguir colores al superponer otros colores con una trasparencia determinada. Ver ejemplo 9-09.

_Variable del color:

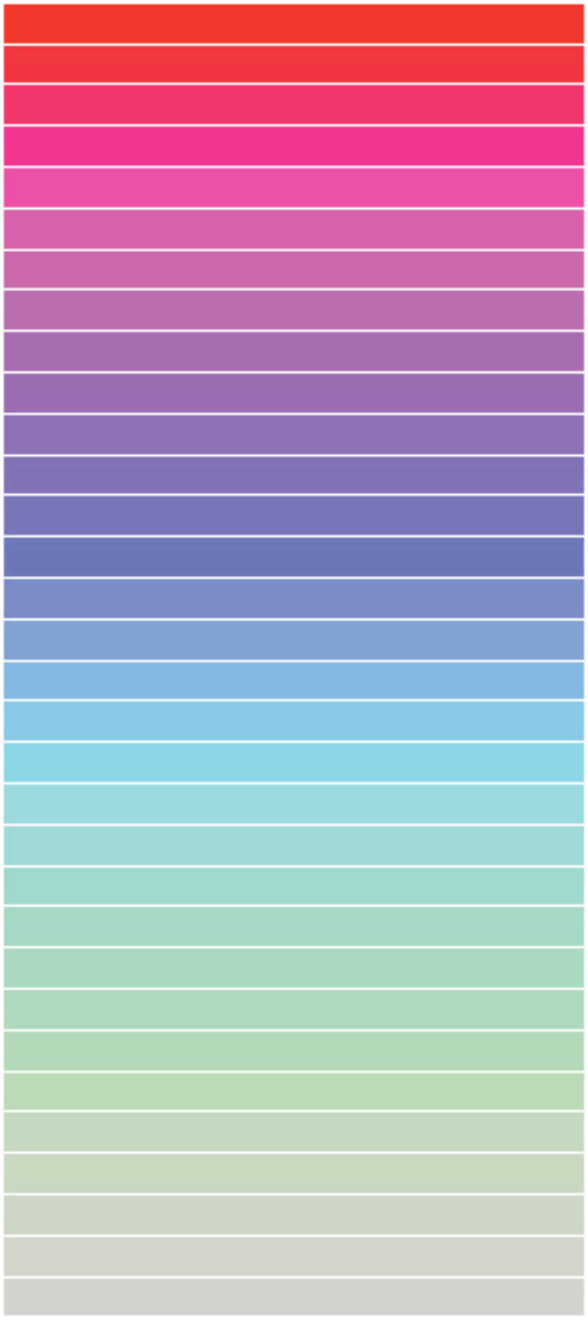
color (gris);
color (gris, alpha);
color (r, g, b);
color (r, g, b, alpha);

Podemos definir varias variables de color y luego llamarlas cuando nos sea necesario para funciones como **background** (), **fill** () o **stroke** ().

_Modo HSB:

H Hue. Parámetro para definir qué color.
S Saturación. Parámetro para definir la pureza del color.
B Brillo. Parámetro para definir la relación del color entre la luz y la oscuridad

TABLA DE CORRESPONDENCIA DE COLORES ENTRE LOS DIFERENTES MODELOS.



RGB			HSB			HEX
255	0	0	360	100	100	#FF0000
252	9	45	351	96	99	#FC0A2E
249	16	85	342	93	98	#F91157
249	23	126	332	90	98	#F9187E
246	31	160	323	87	97	#F720A4
244	38	192	314	84	96	#F427C4
244	45	226	304	81	96	#F42EE7
226	51	237	295	78	95	#E235F2
196	58	237	285	75	95	#C43CF2
171	67	234	276	71	94	#AB45EF
148	73	232	267	68	93	#944BED
126	81	232	257	65	93	#7E53ED
108	87	229	248	62	92	#6C59EA
95	95	227	239	59	91	#5F61E8
102	122	227	229	56	91	#667DE8
107	145	224	220	53	90	#6B94E5
114	168	224	210	50	90	#72ACE5
122	186	221	201	46	89	#7ABEE2
127	200	219	192	43	88	#7FCDE0
134	216	219	182	40	88	#86DDE0
139	216	207	173	37	87	#8BDDD4
144	214	195	164	34	86	#90DBC7
151	214	185	154	31	86	#97DBBD
156	211	177	145	28	85	#9CD8B5
162	211	172	135	25	85	#A2D8B0
169	209	169	126	21	84	#A9D6AD
175	206	169	117	18	83	#AFD3AD
185	206	175	107	15	83	#BAD3B3
192	204	180	98	12	82	#C1D1B8
197	201	183	89	9	81	#C5CEBB
202	201	190	79	6	81	#CACEC2
202	200	193	70	3	80	#CACCC5

La función para definir qué modelo de color voy a usar, utilizaré el comando:

colorMode (modo de color); //RGB o HSB

colorMode (modo de color, rango); //en este caso, el rango de una escala de grises

colorMode (modo de color, rango1,rango2, rango3);

//en este caso: rojo, verde, azul si es modo RGB ó color, saturación, brillo si es HSB

IMAGEN 1: VISUALIZACIÓN, TINTAR

Processing puede cargar GIF, JPEG y PNG.
Los JPEG no admiten transparencia, los PNG y los GIF si que la admiten, aunque si trabajamos con modificación de la transparencia, es mucho mejor trabajar con PNG.
La declaración de imágenes en processing se realiza con el comando: **PImage** ()
Para cargar la imagen en el programa se realiza con el comando:

loadImage () dentro del paréntesis se introduce el nombre de la foto "entre comillas"

Las imágenes que queramos usar se deberán guardar en una carpeta llamada **DATA** que estará junto a la ubicación del archivo principal de Processing.
Para visualizar la imagen en la ventana de Processing, usaremos el comando:

image (nombre de la variable de la foto, x, y)
image (nombre de la variable de la foto, x, y, anchura, altura)

_Color de la imagen, Transparencia:

tint (escala de grises);
tint (escala de grises, alpha);
tint (escala de grises, r, g, b);
tint (escala de grises, r, g, b, alpha);
tint (color);

El comando **tint** () se puede modificar durante el código con el comando **noTint**().

DATOS 2: TEXTO

La variable **char** almacena un caracter. El caracter asignado siempre se escribe entre comillas simples.

char a = **'n'**;

La variable **String** almacena palabras o textos. Las palabras o textos siempre se escriben entre comillas dobles.

String a = **"body"**;

Si vamos a usar un texto grande como un libro, será más práctico cargar ese texto como un archivo externo y operar con él en Processing.

DATOS 3: CONVERSIÓN, OBJETOS

Es muy usual en Processing, realizar conversiones de un tipo de variables en otro tipo de variables. Las variables **int**, **float**, **boolean** y **char** son variables primitivas, porque sólo almacenan un único dato. Las variables **String**, **PImage** y **PFont** son diferentes, ya que lo que almacenan son objetos, objetos que están compuestos por algunas variables primitivas y en su interior existen funciones que trabajan con esas variables. Por ejemplo, un objeto **String** almacena una array de caracteres y tiene funciones que devuelven el número de caracteres o el carácter de una determinada posición.
Los objetos se identifican visualmente de las variables primitivas, porque los objetos se declaran con la primera letra en mayúsculas.

_Conversión de variables:

boolean ();	convierte el número 0 en false y el resto de números en true.
byte ();	convierte otras variables en una representación de byte.
char ();	convierte otras variables en una representación de carácter.
float ();	convierte otras variables en una representación numérica decimal.
int ();	convierte otras variables en una representación numérica entera.
str ();	convierte otras variables en una representación de texto.
nf ();	nos da más control cuando convertimos un **int** o un **float** en un String de texto.

Las variables creadas con **PImage**, **PFont** y **String** son **objetos**.
Las variables sin objetos se llaman **campos** y las funciones sin objetos se llaman **métodos**.
Los campos y los métodos se acceden a través del operador **dot** (punto), después de escribir el nombre del campo o método. Ver ejemplo 12-08 y siguientes. Ejemplo:

```
int w = img.width;
```

```
println (s1.length());
```
imprime la longitud del string del texto en número

```
println (s1.startsWith("S"));
```
detecta si la variable de texto empieza con el texto usado como parámetro.

```
println (s1.endsWith("Five"));
```
detecta si la variable de texto termina con el texto usado como parámetro.

```
println (s.charAt(2));
```
imprime el tercer parámetro de la palabra analizada. El programa empieza a contar en 0.

```
char [] c = s.toCharArray();
```
crea una array de los caracteres contenidos en una string. El programa empieza a contar en 0.

El método .**substring** () crea una nueva string que es una parte de la string original. Cuando usamos un parámetro, la string se lee desde la posición dada hasta el final de la string. Cuando la usamos con dos parámetros, este método nos crea la nueva string entre los dos parámetros dados.

```
String s = "Giallo";
println(s.substring (1, 4));          //imprime "ial"
```

El método .**toLowerCase** () crea una copia de la string con los caracteres en minúsculas.
El método .**toUpperCase** () crea una copia de la string con los caracteres en mayúsculas.
El método .**equals** () se usa para determinar si dos arrays contienen los mismos caracteres.

TIPOGRAFÍA 1: VISUALIZACIÓN

_Cargar fuentes, dibujando texto:

Para crear una fuente, hay que ir al menú Tools, y "Create Font" y ahí elegimos la fuente que queremos insertar en nuestro trabajo. Luego tenemos que crear una fuente con el comando **PFont** ("nombre de la fuente");.
Para cargar una fuente, hay que usar el comando **loadFont** ();.
textFont (); sirve para asignar la fuente que esté asignada en ese momento.
text (); sirve para dibujar los caracteres en la ventana de salida.

text (variable, posición X, posición Y);
text (variable string, posición X, posición Y, anchura, altura);

En el primer caso de declaración del comando, la variable puede ser string, char, int o float. En el segundo caso, sólo puede ser una variable string.
La posición X y la posición Y son las coordenadas del punto de inserción de la esquina inferior izquierda.
La función **fill** () controla el color y la transparencia del texto. Este comando hay que escribirlo antes de la función **text** ().
Al texto no le afecta la función **stroke** ().

textSize (nuevo tamaño de la fuente);

textLeading (parámetro de espaciado entre líneas del texto);

textAlign (MODE); sirve para centrar el texto en la ventana de salida. Los modos pueden ser LEFT, CENTER ó RIGHT.
textWidth (); calcula la anchura de cualquier carácter o string de texto.

MATEMÁTICAS 3: TRIGONOMETRÍA

_Ángulos, ondas:

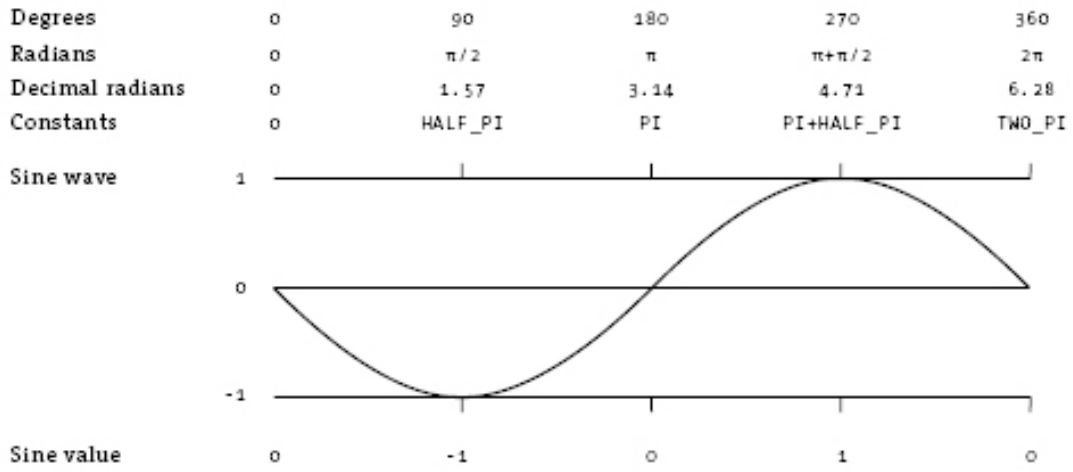
Processing toma como ángulos **positivos**, los ángulos que van en sentido **horario**, y como **negativos**, los que van en sentido **antihorario**.
Processing trabaja con ángulos en radianes, pero para nosotros es mucho más fácil e intuitivo movernos con grados. Con el comando radians () puedo convertir ángulos en

grados, en ángulos en radianes, del mismo modo con degrees () puedo convertir ángulos en radianes, en ángulos en grados.

sin (ángulo);
cos (ángulo);

El parámetro del ángulo siempre se especifica en radianes y devuelve un valor siempre entre –1 y 1. Pero podemos introducirlo como grados de esta manera:

sin (**radians** (ángulo));
cos (**radians** (ángulo));



Como el valor de sin () son números entre –1 y 1, es muy fácil controlar una composición, sólo con multiplicar por otro número, generaremos un intervalo controlado a nuestro antojo. Ver ejemplo 14-07.
Para simplificar el dibujo de arcos, Processing incluye la función:

arc (x origen del centro del arco, y origen del centro del arco, anchura, altura, ángulo donde empieza a dibujar, ángulo donde termina de dibujar);

En realidad, el arco se dibuja sobre el borde exterior de la **elipse** definida por (x0, y0, anchura, altura).

MATEMÁTICAS 4: RANDOM

La función **random** () se usa para crear valores aleatorios dentro de un rango.

random (valor aleatorio máximo);
random (valor aleatorio mínimo, valor aleatorio máximo);

Cuando alguno de esos datos aleatorios supera el valor máximo, se le asigna un 0.
Cuando usamos **random** () con dos parámetros, tenemos más control sobre los valores aleatorios. Ver ejemplo 15-04 y 15-06.

randomSeed (valor);

El valor debe de ser una variable int. Crea una secuencia de números aleatorios enteros. En el ejemplo 15-07 siempre genera el mismo valor aleatorio???.
La función **noise** () es una forma más controlada de generar valores aleatorios. Funciona en las 3 dimensiones. La función **noiseSeed** () funciona como randomSeed().
La función **noise** () permite la posibilidad de crear texturas en 2D y 3D, por ejemplo, combinándose con la función **sin** (). Ver ejemplo 15_09.
La función **abs** () devuelve el valor absoluto del número que tengamos dentro del paréntesis.

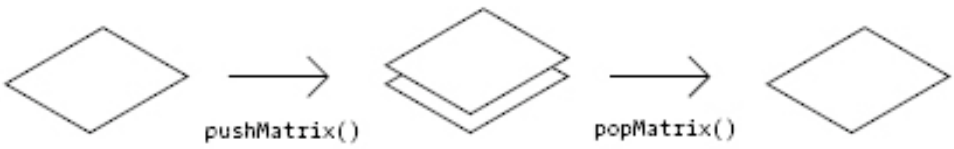
TRANSFORMACIÓN 1: TRANSLACIÓN, MATRICES

translate (x, y); mueve el origen de coordenadas de la esquina superior izquierda de la pantalla a una nueva localización dada.
Sólo lo programado después de esta orden se verán afectados por este cambio de origen de coordenadas.
Esta función es aditiva, se pueden sumar o restar varios cambios de origen de coordenadas.

pushMatrix (); permite modificar la matriz de processing para realizar diferentes funciones con la creación de una nueva matriz momentánea.

popMatrix (); vuelve a la matriz original de processing y sigue realizando funciones. Ambas funciones sólo se pueden utilizar juntas, como haciendo un paréntesis en la lectura del código.

La creación de esta matriz momentánea, sólo afectará a las funciones que se programen entre el pushMatrix () y el popMatrix().



TRANSFORMACIÓN 2: ROTACIÓN, ESCALA

rotate (ángulo); rota el sistema de coordenadas para poder dibujar en cualquier ángulo de la pantalla. El ángulo se le introduce en radianes. Podemos usar **radians** () para pasarle el ángulo en grados y que lo convierta el programa en radianes.
Ángulo positivo sentido horario
Ángulo negativo sentido antihorario
Es una función acumulativa, se pueden acumular giros de diferentes ángulos.

scale (nuevo tamaño); **scale** (tamaño X, tamaño Y);
escala la figura que quiero con ese valor de escalado, que puede ser en las dos dimensiones.

En el ejemplo 17-04, se aprecia como el grosor del borde también se ve afectado por la función **scale** (). Para evitar eso, debemos usar la función **strokeWeight** () y dividirla por el factor de escalado de la función **scale** ().
La función **scale** () es acumulativa, y se pueden acumular varios factores de escalado.

_Combinando transformaciones:

Hay dos maneras de pensar acerca de las transformaciones y no da lo mismo según como se programe, como se puede ver en estos ejemplos:

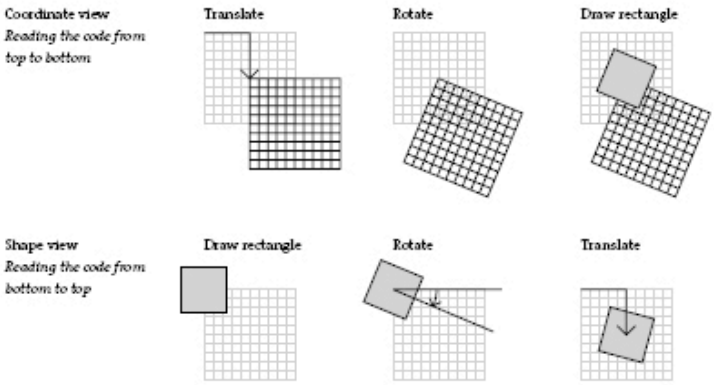
```
translate(width/2, height/2);
rotate(PI/8);
rect(-25, -25, 50, 50);
```

17-07

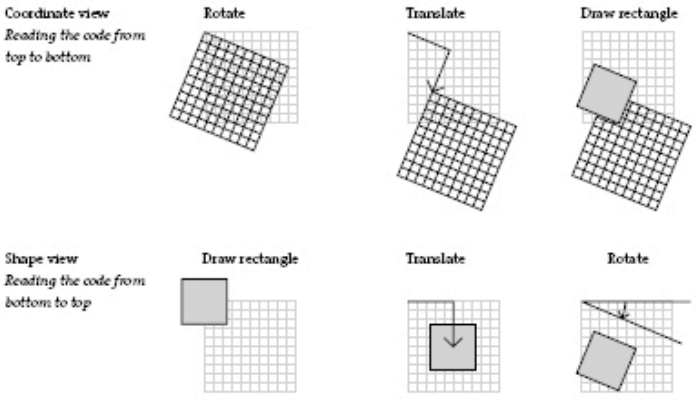
```
rotate(PI/8);
translate(width/2, height/2);
rect(-25, -25, 50, 50);
```

17-08

Code 17-07 analyzed from two perspectives



Code 17-08 analyzed from two perspectives



Transformation combinations
The order in which transformations occur in a program affects how they combine. For example, a `rotate()` *after a* `translate()` *will have a different effect than the reverse. These diagrams present two ways to think about the transformations in codes 17-06 and 17-07.*

SÍNTESIS 1: FORMA Y CÓDIGO

ENTREVISTAS 1: IMPRIMIR
_Fractal. Invaders, Substrate (entrevista con Jared Tarbell)
www.complexification.net
_Shape of Song (entrevista con Martín Wattenberg)
www.turbulence.org/Works/song
_The Objectivity Engine (entrevista con James Paterson)
www.presstube.com
_RandomFont Beowolf (entrevista con Erik van Blokland)
www.letterror.com/foundry/beowolf

ESTRUCTURA 2: CONTINUO

Los programas que se ejecutan continuamente deben incluir la función **draw** (). El código dentro del bloque del **draw** () se ejecuta en bucle hasta que apretemos el botón de stop de la ventana de Processing. Normalmente, los programas sólo tienen un bloque de código en **draw** ().
Por defecto, se dibujan 60 frames por segundo. Con la función **frameRate** () podemos controlar el número **máximo** de frames por segundo que aparecen en la ventana de salida. Nos es conveniente abusar de esta función, porque puede bloquear el equipo.
La función **frameCount** () contiene el número de frames proyectados en la ventana de salida desde que el programa ha empezado a ejecutarse hasta que se detiene con el stop.
En el ejemplo 20-03 se puede observar que hace el programa durante el draw ().

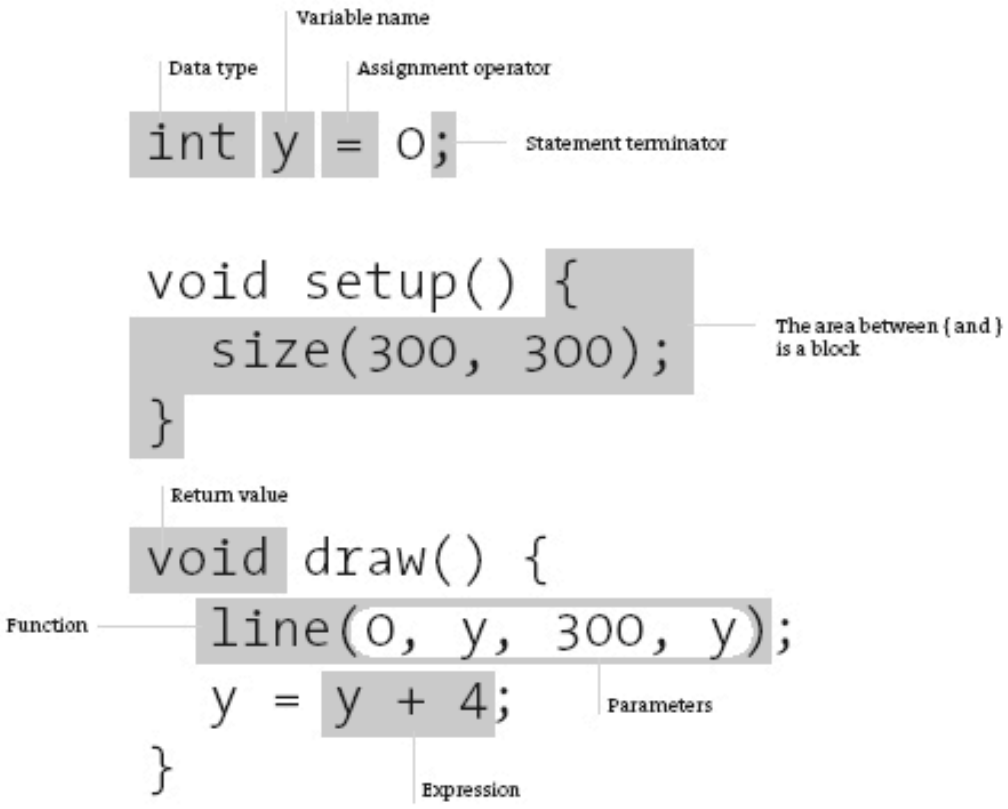

```
float y = 0.0;

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

When this code runs, the variables are replaced with their current values and the statements are run in this order:

```
float y = 0.0
frameRate(30)                    ..... Enter draw( )
line(0, 0.0, 100, 0.0)
y = 0.5
frameRate(30)                    ..... Enter draw( ) for the second time
line(0, 0.5, 100, 0.5)
y = 1.0
frameRate(30)                    ..... Enter draw( ) for the third time
line(0, 1.0, 100, 1.0)
y = 1.5
Etc...
```

En este ejemplo, cuando el programa entra en el **draw** (), el valor de Y es igual a 0. Ejecuta **frameRate** () y la función **line** () y redimensiona el valor de Y sumándole 0.5 y vuelve al inicio del **draw**. Continuará así hasta que paremos el programa.
La variable Y la hemos declarado fuera del **draw** (), pero se redimensionará constantemente porque dentro del draw () hemos escrito la última línea en la que estamos ordenando al programa a redimensionar la variable al acabar todas las operaciones que están dentro del **draw** ().
Como el background () no se refresca en cada ciclo del draw (), las líneas irán acumulándose y dejando negra la pantalla. Si queremos que se refresque, sólo tenemos que incorporar el background al principio del draw (). Ver ejemplo 20-03.

_Anatomía de Processing:



Anatomy of a program 2
Each program can have only one `setup()` and one `draw()`. When the program starts, the code outside of `setup()` and `draw()` is run. Next, the code inside the `setup()` block is run once. After that, the code inside the `draw()` block is run continuously until the program is stopped. Because the variable y is declared outside of `setup()` and `draw()`, it's a global variable and can be accessed and assigned anywhere within the program.

_Controlando el flujo:

Algunas funciones sólo se deben leer una vez y otras cada frame. La función **setup** () la lee el programa antes que el **draw** (). Cuando ordenamos al programa que lea el código, lo primero que va a leer es todas las funciones y declaraciones de variables que se encuentran fuera de setup () y fuera de draw (). Después, el programa lee el código que hay en el **setup** () y por último, lee lo que hay dentro del **draw** ().
Cuando usamos la función **noLoop** (), detenemos el bucle de lectura del **draw** ().
Cuando usamos el setup () y el draw () en un programa, es necesario pensar sobre donde vamos a declarar y asignar las variables que nos van a hacer falta.
La localización de la declaración de una variable, determina su alcance. Por eso es importante, el localizar donde sí y donde no se puede modificar una variable. De esta forma evitaremos muchos errores al leer el código. Ver ejemplo 20-13.

ESTRUCTURA 3: FUNCIONES

Processing además de usar sus funciones propias como line (), background (), size (),...nos da la posibilidad de personalizar y crear nuestras propias funciones. Estas funciones personalizadas suelen ser siempre rutinas para ejecutar otras funciones.
Podemos imaginarnos que una función es como una caja con mecanismos en su interior para operar con datos.

_Creando funciones:

En el ejemplo 21-03, nos hemos creado una función llamada void **eye** (int x, int y); en la que hemos definido el código necesario para dibujar un ojo. Ahora, cada vez que queramos dibujar otro ojo, sólo deberemos escribir **eye** () dentro del draw (), e introducir las coordenadas de posición global en la ventana de salida. De esta forma nos ahorramos escribir un montón de líneas de código con el consiguiente riesgo de errores sintácticos.
Para escribir una función, antes de todo, debemos de tener clara la utilidad de esa función: si es para dibujar una forma específica, para calcular un número, para filtrar una imagen,...Después de eso, debemos ver con qué clase de variables debemos de trabajar para programar esa función (int, float,...). Ver ejemplo de la construcción de la función para dibujar una X.
Cuando usamos funciones que nos devuelven valores, es muy importante pensar en qué tipo de valores nos va a devolver cada función. Por ejemplo, la función **random** (), nos devuelve un valor float. Si asignamos ese valor a una función que se ejecuta con un valor entero, nos dará error. Ver ejemplo 21-17 y 21-18.
Las funciones no se limitan a devolvernos valores numéricos, sino que nos pueden devolver PImage (), String, boolean, ...
Para obtener el valor que nos devuelve nuestra función, reemplazaremos **void** por el valor que nos devuelve. Hay que incluir **return** dentro de la función para extraer el valor devuelto.

FORMA 3: PARÁMETROS, RECURSIVIDAD

_Forma parametrizada

El ejemplo anterior de la hoja (**leaf**) es un ejemplo de forma parametrizada. Modificando los diferentes parámetros de la hoja, la función **leaf** () arrojará diferentes formas.
La forma parametrizada pude convertirse en muy compleja cuando usamos diferentes funciones combinadas.
En el ejemplo 22-01, podemos ver como incluso el cargar una imagen puede ser de manera aleatoria. Véase la forma de cargar las imágenes.

_Recursividad

Un ejemplo común de recursividad lo podemos observar entre dos espejos, donde las reflexiones de las imágenes es infinita. En programación, la recursividad significa que la función se llama a sí misma sin tenerlo que hacer nosotros. Para evitar que este bucle se convierta en eterno, es necesario hacer algo para que la función salga de ese bucle.
Ver la complejidad a la que podemos llegar con el concepto de recursividad en el ejemplo 22-11, pag(231).

INPUT 1: RATÓN 1

mouseX	coordenada X del ratón, respecto del origen de coordenadas
mousey	coordenada Y del ratón, respecto del origen de coordenadas

Cuando un programa arranca, **mouseX** y **mouseY** tienen un valor de 0. Para invertir el valor del ratón, simplemente resta la posición del ratón al valor de la anchura (o altura si lo queremos en las dos dimensiones):

pmouseX	coordenada X del ratón en el frame anterior, respecto del origen de coordenadas
pmouseY	coordenada Y del ratón en el frame anterior, respecto del origen de coordenadas

Los valores de posición del ratón pueden trasladarse, rotar y escalar, usándolos como parámetros en las funciones de transformación.
En el ejemplo 23-10, se puede observar cómo para hacer que una forma gire 360º, es necesario convertir los valores de la posición del ratón en valores dentro del rango 0.0 y 2π. Este escalado de rangos de valores se realiza como ya hemos visto con la función **map** ().

map (**mouseX**, 0, **width**, 0, **TWO_PI**);

Usando las variables de posición del ratón y con una estructura condicional (if), permitimos al ratón seleccionar regiones de la pantalla.

Nota: Operador Lógico **&&** equivale a "y además..."

_Botones del ratón

Processing puede detectar que botón del ratón está presionado para iniciar la ejecución de una función determinada.

mouseButton ();
mousePressed (LEFT/CENTER/RIGHT);

La variable **mousePressed** () y **mouseButton** () se convierte en falsa en cuando dejamos de presionar el botón del ratón. Ver ejemplo 23-16, pag.239.

_Icono del cursor

El icono del cursor del ratón se puede esconder con la orden **noCursor** () para que no aparezca en nuestra ventana de salida
Añadiendo un parámetro a la función **cursor** (), podemos cambiar el icono del ratón en la ventana de salida. Las opciones para cambiar el icono son:

ARROW
CROSS
HAND
MOVE
TEXT
WAIT

DIBUJANDO 1: FORMAS ESTÁTICAS

Podemos dibujar capturando la posición del ratón desde líneas, puntos, funciones propias (previamente definidas)....incluso imágenes (ver pag. 248).

INPUT 2: TECLADO

Processing registra las teclas que se presionan y que tecla está presionada en un determinado momento. La variable boolean **keyPressed** () es cierta, si una tecla está presionada y falsa si no es así. Si incluimos esta variable en una estructura condicional (if), podemos programar que líneas de código se leerán si una tecla está presionada. La variable **keyPressed** () permanece como verdadera mientras la tecla esté pulsada y se convertirá en falsa sólo cuando se suelte esa tecla.
La variable de una tecla es un carácter (variable tipo **char** 'A') y almacena la tecla presionada más reciente. Esta variable sólo puede almacenar un valor cada vez. Véase el ejemplo 25-03 y 25-04 (pág.252).
La variable de una tecla se puede usar para determinar qué ejecutar cuando una tecla específica está presionada.
Incluso con un mapeado de los rangos, podemos hacer que una figura gire al presionar determinadas teclas.

_Teclas codificadas

Processing puede leer los valores de teclas especiales como las teclas de dirección (flechitas) y Alt, Control, Shit, Backspace, Tab, Enter, Return, Escape y Delete.
La variable **keyCode** () almacena esas teclas como constantes. Ver ejemplo 25-07.

INPUT 3: EVENTOS

Las funciones llamadas eventos alteran el flujo normal del programa cuando una acción como pulsar una tecla o mover el ratón cuando el programa lee el código. Pulsar una tecla o mover el ratón se almacenan hasta el final del draw (). Las funciones de mover el ratón o pulsar una tecla son leídas independientemente de lo que está ocurriendo en el resto del programa.

_Eventos del ratón

mousePressed ();
El código dentro de este bloque se ejecuta 1 vez cuando el botón del ratón está pulsado.

mouseReleased ();
El código dentro de este bloque se ejecuta 1 vez cuando el botón del ratón está sin pulsar.

mouseMoved ();
El código dentro de este bloque se ejecuta 1 vez cuando el ratón se mueve.

mouseDragged ();
El código dentro de este bloque se ejecuta 1 vez cuando el ratón se mueve y mientras el botón del ratón está presionado.

_Controlando el flujo

Cuando el programa lee el **draw** (), dibuja frames en la pantalla lo más rápido que puede. Con la función **frameRate** () controlamos el límite de frames que se dibujan por segundo, y la función **noLoop** () la usamos para detener el bucle del draw.
Si pausamos un programa con **noLoop** (), lo podemos volver a poner en marcha con la función **loop** ().
Sólo las funciones de **eventos** siguen ejecutándose cuando un programa se detiene con noLoop ().
La función **redraw** () ejecuta el código dentro del draw () una vez y después detiene la ejecución. Es muy útil cuando la ventana no necesita refrescarse constantemente. En el ejemplo 26-10 sólo se lee el código en el draw () cuando presionamos el botón del ratón.

INPUT 4: RATÓN II

La posición del ratón es un punto en la ventana de salida que se refresca en cada frame. Este punto puede modificarse y analizarse en relación con otros elementos y calcular nuevos valores. Es posible restringir el valor del ratón a un rango determinado, calcular la distancia entre el ratón y otra posición, interpolar entre dos valores, determinar la velocidad de movimiento del ratón y calcular el ángulo del ratón en relación con otra posición.

_Restringir

La función **constrain** () limita un número a un rango.

constrain (valor del número a limitar, rango mínimo, rango máximo);

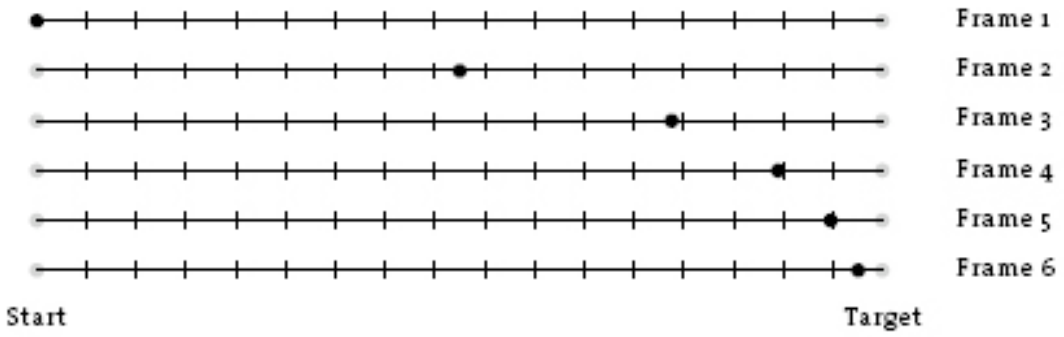
_Distancia

La función **dist** () calcula la distancia entre las coordenadas de dos puntos (punto1 y punto2) . Este valor puede usarse para determinar la distancia del cursor a un punto de la pantalla en la situación actual. Ver ejemplo 27-05.

dist (x1, y1, x2, y2);

_Easing

Easing o también llamado interpolación es una técnica para movimientos entre dos puntos. Moviendo una fracción de la distancia total en cada frame, una forma puede decelerar o acelerar cuando se aproxima a la localización de un objetivo. Este diagrama muestra que ocurre cuando un punto siempre se mueve la mitad de la distancia total entre su posición actual y el destino final.



En este ejemplo, cuando la forma se aproxima a la localización del objetivo, la distancia que se desplaza en cada frame decrece, por lo tanto, la forma reduce su velocidad. Ver ejemplo 27-06, pag (267).
En los dos ejemplos anteriores se continua haciendo el cálculo para la posición del círculo cada vez que éste alcanza su destino. Esto es ineficiente cuando tienes miles de círculos interpolando entre posiciones, esto haría ralentizarse mucho el programa. Para parar los cálculos cuando no son muy necesarios, hay que observar si la posición del objetivo y el destino coinciden y detener los cálculos.

En el siguiente ejemplo se introduce la función **abs** () para obtener el valor absoluto de un número. Esto es necesario porque el valor del factor de interpolación puede ser negativo o positivo, dependiendo si la posición está a la izquierda o a la derecha del objetivo. Usaremos una estructura condicional (**if**) para refrescar la posición sólo si no coinciden objetivo y destino. Ver ejemplo 27-08.

_Velocidad

Calcularemos la velocidad del ratón comparando su posición actual con su posición previa. Esto lo haremos usando la función **dist** () con mouseX, mouseY, pmouseX y pmouseY como parámetros. En el ejemplo 27-09 calculamos la velocidad del ratón y convertimos esa variable de velocidad en la variación del tamaño de la elipse.
En el ejemplo siguiente, se puede observar la velocidad instantánea del ratón. Los valores que obtenemos son extremos (saltan entre cero y el valor máximo en el siguiente frame). Con la ecuación de interpolación del código 27-06 podemos incrementar o decrecer la velocidad correctamente. Con este ejemplo, se demuestra cómo aplicar la ecuación de interpolación en este contexto. La barra superior es la velocidad instantánea y la barra inferior es la velocidad de interpolación.
_Orientación

La función **atan2** (Y, X)e usa para calcular el ángulo que forma cualquier punto con las coordenadas (0,0).
El parámetro Y es la coordenada Y del punto del que queremos encontrar su ángulo, y el parámetro x es la coordenada X de ese mismo punto. El valor del ángulo que obtenemos viene en radianes en el rango de π a -π. En el ejemplo siguiente, se ha usado la función **degrees** () para obtener el ángulo ya en grados.

Ángulo +	sentido horario
Ángulo -	sentido antihorario

Hay que tener cuidado que el orden de introducir los parámetros X e Y es el inverso que en las funciones que hemos ya visto anteriormente.

_INPUT 5: TIEMPO, FECHA

Processing puede leer el valor del reloj del ordenador. El segundo en curso lo lee con **second** () y devuelve valores entre 0 y 59. El minuto en curso lo lee en **minute** () y devuelve valores en 0 y 59. La hora en curso la lee en **hour** () y devuelve valores entre 0 y 23.
Si colocamos esas funciones en el draw, podemos leer el reloj constantemente.
Podemos crear un reloj en la pantalla usando **text** () para dibujar los números en la ventana de salida. Con la función **nf** () podemos espaciar los números de una forma equidistante entre la derecha y la izquierda. A los números del 1 al 9, se le añade un 0 delante para que siempre aparezcan números de dos cifras.

nf (intValue, digits);

intValue	tiene que ser o una variable entera o una array de variables enteras.
digits	número de dígitos a rellenar con ceros.

Además de leer el tiempo en curso, cada programa de Processing almacena el tiempo transcurrido desde que el programa se abrió. Ese tiempo se guarda en milisegundos. Ese número se obtiene con la función **millis** () y puede usarse para desencadenar eventos y calcular el paso del tiempo.
En el ejemplo 28-06, el dibujo de la línea se inicia cuando transcurren 3 segundos desde que se inicia el programa.
La función **millis** () devuelve una variable int, pero a veces la podemos usar para convertir una variable float representando el número de segundos transcurridos desde que el programa empezó a ejecutarse. El valor resultante puede usarse para controlar la secuencia de eventos en una animación.

_Fecha

La información relativa a la fecha es leída de una manera muy similar a la del tiempo:

day () devuelve valores entre 1 y 31
month () devuelve valores entre 1 y 12
year () devuelve valores de 4 dígitos
println (d + " " + m + " " + y);

MOVIMIENTO 1: LÍNEAS, CURVAS

_Controlando el movimiento

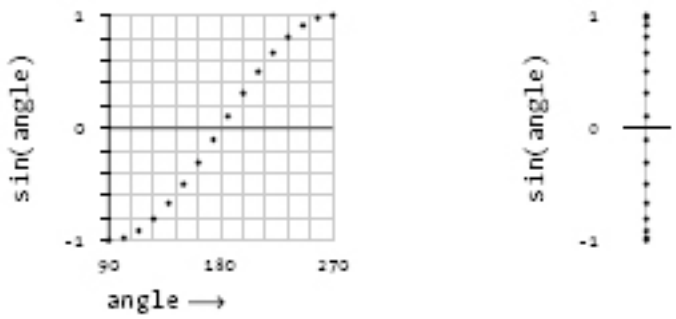
Para dotar de movimiento a una forma, usaremos una variable para cambiar sus atributos. Si queremos que la pantalla se refresque cada frame, deberemos incluir el **background** () al principio del draw. Con el **frameRate** () controlaremos el número de frames por segundo.
Para mover las imágenes de arriba abajo de una forma cíclica, necesitamos una variable que almacene la dirección del movimiento. En el ejemplo de la pelota, programamos la variable de dirección con valor 1 para cuando la pelota se mueve hacia daajo de la pantalla y con un valor –1 para cuando sube. Es decir, para cambiar la dirección cuando choca con los límites horizontales de la ventana de salida, sólo tenemos que cambiar el valor de la variable dirección.
Si queremos una forma que cambie su posición en relación también con los límites izquierdo y derecho de la ventana de salida, necesitamos una segunda variable para almacenar la variable X. Ver el ejemplo 31-03 pág.308.
También es posible cambiar no sólo la posición de una forma, si no también el valor del fondo de la ventana de salida, el grosor, color de relleno de la forma y el tamaño de la forma.

_Moviéndose a lo largo de curvas

Las curvas simples explicadas en Matemáticas 2 pueden servir como recorridos para formas en movimiento. Al igual que podemos dibujar la curva entera en un frame, podemos calcular cada paso de la curva en una secuencia de frames. Ver ejemplo 31-10, donde se introduce el concepto de aceleración en el código.

MOVIMIENTO 2: MÁQUINA, ORGANISMO

La función **sin** () se usa a menudo para producir un movimiento elegante. Puede generar una aceleración y deceleración mientras la forma se desplaza de un frame a otro.



Los valores de **sin** () se usan para crear el movimiento de la forma. El valor del ángulo se incrementa constantemente para producir cambios por **sin** () en el rango de –1 a 1. Los valores se multiplican por el valor del **radio** para magnificar los valores. Este resultado se asigna a la variable **yoofset** y se usará para determinar la coordenada Y de la elipse en la siguiente línea. Obsérvese como la pelota decelera cuando llega arriba o abajo y acelera cuando está por el medio de la pantalla.
Añadiendo valores de las funciones sin () y cos () podemos producir un movimiento más complejo.

_Movimiento Orgánico

Las funciones **sin** () y **cos** () se pueden usar para crear movimientos impredecibles cuando las empleamos con el **random** ().
La función **noise** () es otra herramienta para producir movimiento orgánico. Los números que devuelve la función **noise** () son fáciles de controlar, son una buena idea para generar una sutil irregularidad en el movimiento.
La función **noise** también se usa para generar texturas dinámicas. En el ejemplo 32-09(p325), los dos primeros parámetros se usan para producir las dos dimensiones de la textura y el tercer parámetro incrementa su valor en cada frame para variar la textura. Cambiando la variable de la **densidad**, incrementamos la resolución de la imagen y cambiando la variable **inc**, cambiamos la resolución de la textura.

VARIABLES 4: ARRAYS

En lenguaje de programación, una array es una lista de elementos almacenados bajo el mismo nombre. Las arrays pueden ser creadas para cualquier tipo de variable y cada elemento puede asignarse individualmente y leerse.
Los elementos de una array están numerados empezando por el 0. No tener en cuenta este detalle puede generar errores al principio de trabajar con arrays. La primera posición es la [0], y la segunda es la [1]...
Las arrays se declaran de una manera similar al resto de variables, pero las arrays se distinguen con **[ ]**. Cuando una array es declarada, el tipo de variable que almacena debe de ser específica.
Después de declarar la array, ésta debe de ser creada con el comando "**new**". Este paso adicional, reserva espacio en la memoria del ordenador para almacenar las variables de

la array. Después de ser creada la array, los valores pueden asignarse. Hay varias formas de declarar, crear y asignar arrays.
Para acceder a un elemento de la array, usaremos el nombre de la variable seguido de [ ] con el número de su posición dentro para que lo pueda leer.
El campo **lenght** almacena el número de elementos de la array. Este campo se almacena

con la array y puede accederse a él con el operador •

```
int [] data 3 = new int [127];
println (data3.length);                    //imprime "127" en la consola
```

Con una estructura condicional (**if**) podemos introducir variables dentro de la array, por ejemplo, podemos calcular una serie de números y entonces asignar cada valor para cada valor de la array.

_Almacenando la variable del ratón

Las arrays a menudo se usan para almacenar la variable del ratón. Las variables pmouseX y pmouseY almacenan las coordenadas del cursor del frame anterior. En cada frame, las variables pmouseX, pmouseY, mouseX y mouseY son actualizadas con nuevos datos y deshecha los anteriores. Crear una array es la manera más fácil de almacenar el historial de esas variables. En el ejemplo 33-14, los 100 valores más recientes de mouseY se almacenan en una array y se muestran en la pantalla como una línea de izquierda a derecha.

_Funciones de las array

Processing tiene un grupo de funciones para asistir en el manejo de las array de variable.
La función **append** () añade un elemento a la array, lo coloca en la última nueva posición de la array, y devuelve una nueva array con el nuevo elemento.
La función **shorten** () resta un elemento a la array, borrando el último elemento y devuelve una nueva array sin el nuevo elemento.
La función **expand** () incrementa el tamaño de la array. La podemos incrementar con un tamaño específico. Si no especificamos el nuevo tamaño, la longitud de la array se duplica. Se usa cuando una array necesita elementos extras.
Los valores de una array no pueden copiarse con los operadores de asignación, porque son objetos. La manera más fácil de copiar elementos de una array a otra es usar funciones especiales o copiar cada elemento individualmente con una estructura condicional. La función **arraycopy** () es la forma más eficiente para copiar el contenido íntegro de una array a otra. Los datos son copiados de la array escrita como primer parámetro hacia la array declarada en segundo lugar. Ambas arrays tienen que tener la misma longitud para que esta función se ejecute bien.
Cuando una array se usa como parámetro para otra función, la dirección (localización en la memoria) de la array se transfiere a la función con los datos actuales.
Cambiar una array con una función sin modificar los valores originales de la array, requiere líneas de código adicional.

_Arrays de dos dimensiones

Las variables pueden ser almacenadas en arrays con más de una dimensión. Una array de dos dimensiones es esencialmente una lista de arrays de una dimensión. Se debe declarar, crear y entonces los valores pueden asignarse como en una array de una dimensión.

IMAGEN 2: ANIMACIÓN

Ver presstube.com

_Imágenes en movimiento

Mover una imagen, en lugar de presentar una secuencia, es otro enfoque para animar imágenes.
Las imágenes pueden también animarse cambiando sus atributos de dibujo.

IMAGEN 3: PÍXELES

_Leyendo píxeles

La función **get** () puede leer el color de cualquier píxeles en la ventana de salida. Hay 3 versiones de esta función, una para cada uso:

get ()
get (x, y)
get (x, y, width, height)

Si la función **get** () se usa sin parámetros, una copia de la ventana de salida entera se devuelve como **PImage**.
La versión con dos parámetros devuelve el valor del color de un píxel con esas coordenadas x,y.
Un área rectangular de la ventana de salida es lo que devuelve la función si la usamos con los parámetros **width** y **height**. Si **get** () graba la ventana de salida entera o una sección de la ventana, los datos que devuelve, deben ser asignados como una variable PImage. Estas imágenes pueden ser redibujadas en la ventana en diferentes posiciones y modificando su tamaño.
set () cambia el color de un píxel o coloca una imagen directamente en la ventana de salida, determinados sus coordenadas X e Y y la imagen o el color.
Cuando usamos la función **get** con los parámetros (x,y) se devuelven valores que podrían asignarse como variación de la variable de color. Estos valores pueden usarse para colocar el color de otros píxeles o puede servir como parámetros para las funciones **fill** () o **stroke** ().
Los valores del ratón pueden usarse como parámetros para la función **get** (). Esto permite al cursor seleccionar colores de la ventana de salida.
La función **get** () puede usarse con una estructura condicional (if) para grabar píxeles o grupos de píxeles. En el siguiente ejemplo, los valores de cada fila de píxeles de la imagen se usan para los valores para las líneas de la derecha. Si ejecutamos este código y movemos el ratón arriba y abajo veremos la relación entre la imagen de la izquierda y las bandas de color de la derecha.

_Escribiendo píxeles

Los píxeles en la ventana de salida de Processing pueden ser definidos directamente con la función set (). Hay dos versiones de esta función, cada una con 3 parámetros.

set (x, y, color)
set (x, y, image)

Cuando el tercer parámetro es color, la función **set** () cambia el color de cualquier píxel en la ventana de salida. Cuando el tercer parámetro es una imagen, la función **set** () escribe una imagen en las coordenadas específicas (x,y).
La función **set** () puede escribir una imagen en la ventana de salida en cualquier coordenada. Usando **set** () para dibujar una imagen es más rápido que usando la función **image** () porque los píxeles se copian directamente. Sin embargo, las imágenes dibujadas con **set** () no pueden ser redimensionados o teñidos y las funciones de transformación no les afectan.

TIPOGRAFÍA 2: MOVIMIENTO

_Palabras en movimiento

Para que la tipografía se mueva, el programa tiene que ejecutarse en bucle en la función **draw** (). Usar la tipografía en el draw () requiere 3 pasos:
_1 tenemos que declarar una variable PFont fuera del setup () y del draw ().
_2 la fuente tiene que estar cargada en el setup (), con la orden **loadFont** ().
_3 la fuente puede ser usada para escribir caracteres dentro de la pantalla, colocándola dentro del draw () usando la función **text** ().
Usaremos la orden "**create Font**" dentro del menú **Tools**, para crear una fuente.

_Letras en movimiento

Animar letras individualmente ofrece más flexibilidad que mover palabras entera. Construyendo palabras letra a letra, cada una con un movimiento o velocidad, podemos transmitir un particular sentido o tono. Trabajar en esta idea, requiere más paciencia y a menudo programas largos, pero los resultados pueden ser más valiosos debido a sus grandes posibilidades.

TIPOGRAFÍA 3: RESPUESTA

La función **toCharArray** () se usa para extraer los caracteres individuales de una variable String y los coloca en una array de caracteres. El método **charAt** () es una alternativa para aislar las letras individuales de una string.

COLOR 2: COMPONENTES

Los colores se almacenan en Processing como números. Cada color se define por sus componentes. Cuando un color se define por sus valores RGB, estos valores almacenan los componentes rojo, verde y azul y un cuarto valor opcional que almacena el valor de la transparencia. De la misma manera funciona con el modelo de color HSB.

_Extrayendo el color

Las funciones **red** (), **green** () y **blue** () se usan para leer los componentes de un color.
La función alpha () lee el valor alpha del color (transparencia). Si no se define el valor de alpha, se asigna el valor 255 por defecto.
Las funciones **hue** (), **saturation** () y **brightness** () se usan para leer los componentes de un color en el modo de color HSB.
Los valores extraídos con las funciones **red** (), **green** () y **blue** () se pueden usar para muchas cosas. Por ejemplo, los números se pueden usar para controlar aspectos del movimiento o de el flujo del programa. En el ejemplo 38-09, el brillo de los píxeles de una imagen controlan la velocidad de 400 puntos a través de la pantalla. Cada punto se mueve de derecha a izquierda. El valor del píxel en la imagen con las mismas coordenadas como un punto es leído y se usa para definir la velocidad de movimiento de los puntos. Cada punto se mueve más despacio a través de las áreas oscuras y más rápido a través de las áreas más claras.
Cargar los colores de una imagen en una array abre muchas posibilidades. Los colores en la array pueden ser fácilmente reordenados. La función **sortColors** () toma una array de colores como un input, y los recoloca en orden de más oscuro a más claro y entonces, devuelve los colores ordenados, de 0 a 255.

IMAGEN 4: FILTRO, MEZCLA, COPIA, MÁSCARA

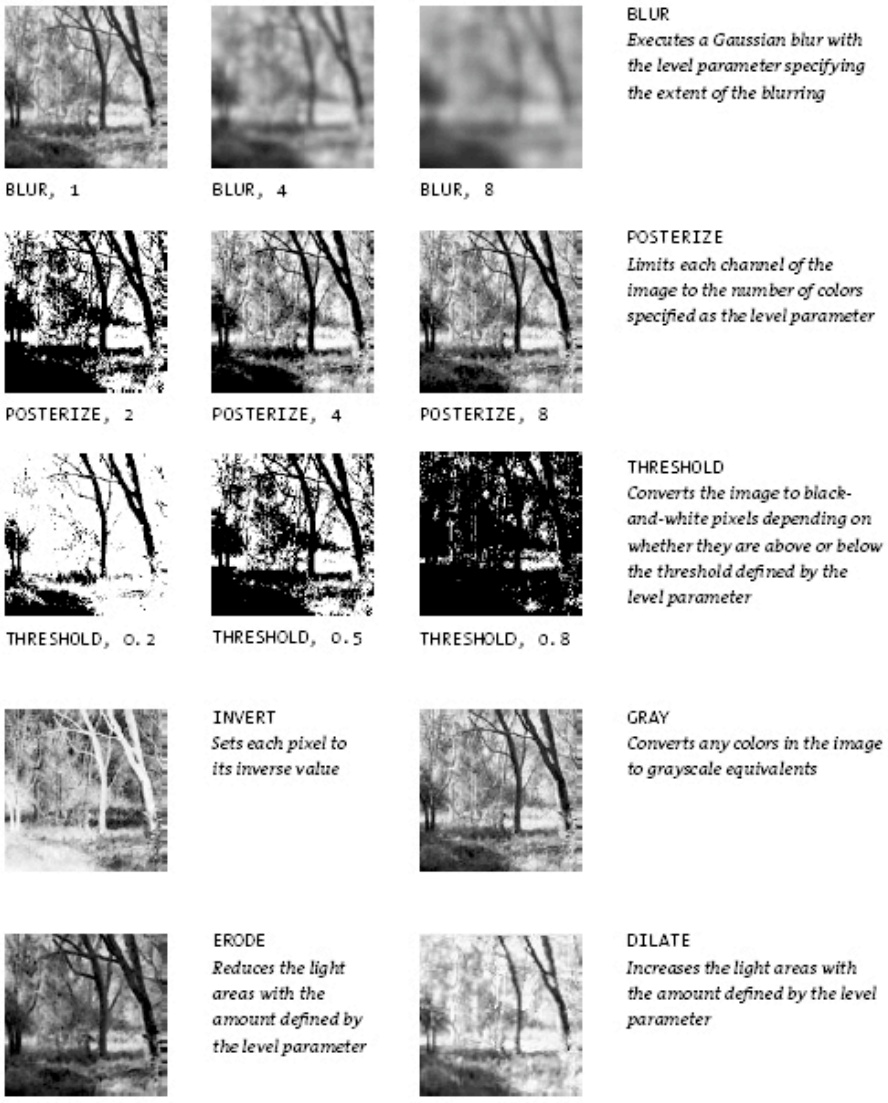
Las imágenes digitales tienen la posibilidad de ser fácilmente reconfiguradas y combinadas con otras imágenes. Cada píxel en una imagen digital es un grupo de números que pueden ser añadidos, multiplicados, se les puede hacer la media con los valores de otros píxeles.

__Filtrando, mezclando

Processing tiene funciones para filtrar y mezclar imágenes en la ventana de salida. Cada una de esas funciones opera transformando los valores de los píxeles de una imagen simple u operando con píxeles entre dos imágenes diferentes. La función filter () tiene dos prototipos:

filter (mode);
filter (mode, level);

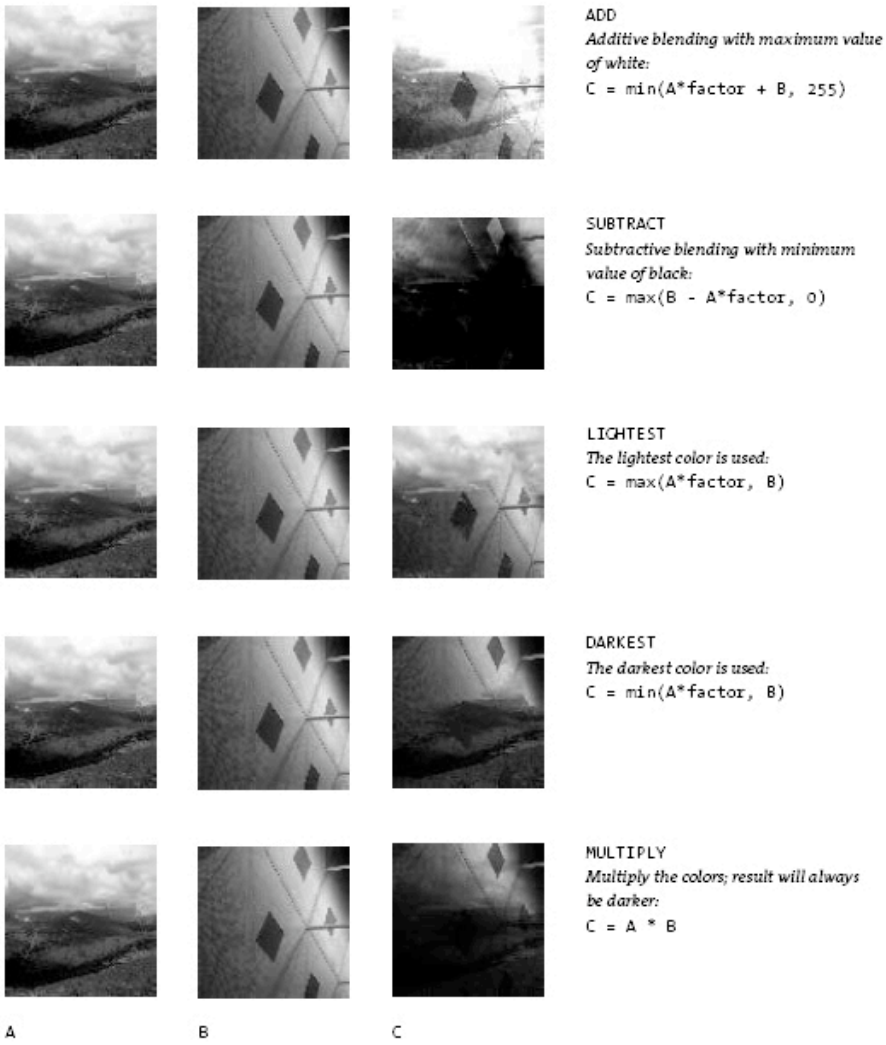
Existen 8 opciones para el parámetro (mode): THRESHOLD, GRAY, INVERT, POSTERIZE, BLUR, OPAQUE, ERODE y DILATE.
El modo THRESHOLD convierte cada píxel en un imagen en B/N basándose en si el valor introducido está por debajo del valor del parámetro (level).
En el siguiente ejemplo aplicamos el filtro THRESHOLD a una imagen con el parámetro (level) con un valor de 0.3, esto significa que los píxeles con una valor gris mayor del 30% del brillo máximo, serán dibujados blancos y los píxeles por debajo se dibujarán negros.



La función **filter** () sólo afecta cuando ya han sido dibujadas las imágenes. El parámetro **BLUR** ejecuta un desenfocado gaussiano con el parámetro (level) especificando el grado de desenfoque.
Cambiando el valor del parámetro de **filter** () con cada frame, podemos crear movimiento. Los efectos de la función **filter** () se actualizan cada vez que se ejecuta el bucle del draw (), pero incrementando o decreciendo el parámetro (level), los resultados de esta función son más o menos pronunciados. Este efecto se puede conseguir, introduciendo una variable float con una estructura condicional en el draw (), donde en cada bucle se aumenta y por lo tanto aumenta el desenfoque con el efecto **BLUR**.
La clase PImage () tiene el método filter () que puede aislar el filtro de una imagen específica.
La función **blend** () mezcla los píxeles de diferentes maneras dependiendo del parámetro del **MODE**. La función **blend** () tiene dos versiones diferentes.

blend (x, y, width, height, dx, dy, dwidth, dheight, mode);
blend (srcImg, x, y, width, height, dx, dy, dwidth, dheight, mode);

El parámetro **MODE** puede ser **BLEND**, **ADD**, **SUBTRACT**, **DARKEST**, **LIGHTEST**, **DIFFERENCE**, **EXCLUSION**, **MULTIPLY**, **SCREEN**, **OVERLAY**, **HARD_LIGHT**, **SOFT_LIGHT**, **DODGE** y **BURN**.



Los parámetros **x** e **y** son las coordenadas X e Y de la región a copiar.
Width y **height** son los parámetros para determinar la colocación de la región que vamos a copiar.
dx y **dy** son las coordenadas X e Y del área de destino.
dwidth y **dheight** son los parámetros para determinar la colocación de la región que hemos copiado.
Para mezclar ambas imágenes en la ventana de salida, la segunda imagen puede ser usada como parámetro **srgImg**. Si las regiones original y de destino tienen diferentes tamaños, los píxeles serán automáticamente redimensionados adecuadamente a la región objetivo específica. Se puede mezclar la imagen en la ventana de salida usando cualquier modo de los anteriores descritos.
Las variables PImage () tienen un método **blend** () que puede ser usado para mezclar una imagen o dos juntas sin afectar a la ventana de salida.
La función **blendColor** () se usa para mezclar los valores de los colores de forma individual.

blendColor (c1, c2, mode);

c1 y **c2** son parámetros de los valores de color para crear un nuevo color mezclado de ambos. Las opciones del parámetro **mode** son las mismas que las opciones para la función **blend** ().

_Copiando píxeles

La función **copy** () tiene dos versiones, cada una de ellas con un largo número de parámetros:

copy (x, y, width, height, dx, dy, dwidth, dheight, mode);
copy (srcImg, x, y, width, height, dx, dy, dwidth, dheight);

Si las regiones origen y destino tienen diferente tamaño, los píxeles automáticamente se redimensionarán adecuadamente a la anchura y altura de destino. Los otros parámetros son los mismos descritos para la función **blend** ().
La función **copy** () difiere de las anteriormente expuestas **get** () y **set** () porque ambas consiguen los píxeles de una localización y los transforman en otra.

_Enmascarando

El método **mask** () para las variables PImage transforma el valor de la transparencia de una imagen basándose en los contenidos de otra imagen. La imagen máscara contendrá sólo valores de escala de grises y pueden convertirse con la función **filter** (). Las áreas más claras de la máscara, permiten ver la imagen original, y las áreas más oscuras tapan la visión de la imagen original.

IMAGEN 5: PROCESANDO IMAGEN

Procesar una imagen es en términos generales manipular o modificar imágenes con el propósito de corregir un defecto, improvisar un efecto estético o facilitar la comunicación.

_Píxeles

Una array de píxeles almacena el valor de color para cada píxel en la ventana de salida. La función **loadPixels** () debe ser declarada antes de usar la array de píxeles. Antes de que los píxeles hayan sido leídos o modificados, se deben actualizar usando la función **updatePixels**(). Al igual que **beginShape** () y **endShape** (), **loadPixels** () y **updatePixels** () deben de aparecer siempre juntos.

_La imagen es una variable

Los datos numéricos, sin embargo, no necesitan verse como colores, pueden usarse para generar movimiento o definir los vértices de una forma. Ver el ejemplo 40-15, donde se convierte el valor del color rojo de un píxel en una gráfica.
En el ejemplo 40-16, los valores del color azul de una línea de una imagen se transforman en una serie de líneas.

OUTPUT 2: IMÁGENES

Un ordenador dibuja en la ventana de salida, una imagen muchas veces cada segundo. La mayoría de los sistemas operativos proporcionan un método para capturar esas imágenes mientras el programa se está ejecutando.
Salvar imágenes de una aplicación de software puede ser muy útil como técnica de documentación o una forma de crear animaciones frame a frame.

_Salvando imágenes

La función **save** () guarda una imagen de la ventana de salida. Requiere un parámetro, un string (texto) que será el nombre del archivo de la imagen guardada.
Las imágenes se pueden salvar en una variedad de formatos dependiendo de la extensión usada en el parámetro de nombre del archivo. Si la extensión no es incluye en el nombre del archivo, la imagen se salvará como TIFF (.tif).
Sólo los elementos dibujados antes de la función **save** () serán incluidos en la imagen.
Cuando la función **save** () aparece en el draw, el archivo se rescribe constantemente en cada bucle del draw. Esta circunstancia se puede evitar si colocamos el **save** () con un evento como **mousePressed** () o **keyPressed** (). Porque estos eventos son siempre llamados cuando el draw se termina, la imagen guardada incluirá todo lo que aparezca dibujado cuando ocurre el evento.

_Guardando imágenes en secuencia

La función **saveFrame** () guarda una secuencia numerada de imágenes.

saveFrame ():
saveFrame ("filename-####.ext");

Si **saveFrame** () se usa sin un parámetro, se guardan los archivos como screen-0000.tif, screen-0001,...
El nombre del archivo –componente que puede ser cambiada por cualquier nombre – y su extensión (.ext) puede definirse como .tif, o .tga. La porción #### del nombre especifica el número de dígitos para ordenar la secuencia de imágenes. Cuando los archivos se guardan, los 4 # son reemplazados con el valor de la variable **frameCount**.
Si usamos la función **saveFrame** () dentro de una estructura condicional (if), permitimos que el programa guarde imágenes sólo si se cumplen ciertas condiciones. Por ejemplo, si queremos guardar una secuencia de imágenes de 200 frames después de pulsar el ratón, o si queremos guardar un frame y entonces saltar unos cuantos frames antes de guardar otro frame.

ESTRUCTURA 4: OBJETOS I

Las variables y funciones son los bloques que construyen un programa. Algunas funciones a menudo son usadas conjuntamente para trabajar con varias variables.
Una clase define un grupo de métodos (funciones) y campos (variables). Un objeto es un ejemplo simple de una clase.
Las variables son la forma más elemental para pensar acerca de reutilizar elementos con un programa. Las variables permiten a un valor simple aparecer muchas veces con un programa y ser fácilmente modificado. Las funciones abstrae una tarea específica y permite a los bloques de código ser reutilizados a través de un programa.

_Definiendo clases y objetos

Al definir una clase, estamos creando su propio tipo de variables. Cuando creamos una clase, lo primero que hay que pensar cuidadosamente es acerca de que queremos que haga el programa. Es muy normal hacer una lista de las variables que voy a necesitar y de qué tipo deben de ser.
El nombre de la clase debe ser elegido cuidadosamente. El nombre debe ser fácilmente recordable, haciendo referencia a la acción que realiza. Los nombres de las clases empezarán siempre en mayúsculas.
El constructor es un bloque de código que se activa cuando un objeto se crea. El constructor siempre tiene el mismo nombre que la clase y típicamente se usa para asignar valores al campo de los objetos.

_Arrays de objetos

Trabajar con arrays de objetos es similar que trabajar con arrays de otros tipo de variables. Como todas las arrays, una array de objetos se distinguen de los objetos simples con [ ]. Como cada elemento del array es un objeto, cada elemento de la array debe ser creado antes de poder acceder a ella. Los pasos para trabajar con una array de objetos son:

1_ Declarar la array
2_ Crear la array
3_ Crear cada objeto en la array

DIBUJANDO 2: FORMAS CINÉTICAS

Los instrumentos de dibujo pueden cambiar una forma dependiendo de los gestos hechos con la mano. Comparando las variables X e Y del ratón con el valor previo de la posición del ratón podemos determinar la dirección y velocidad del movimiento.
Los instrumentos de dibujo pueden seguir un ritmo o cumplir unas reglas independientes de los gestos del dibujo. Esta es una forma de dibujo colaborativo en el que poder controlar en el boceto algunos aspectos de la imagen y controlar otros con el programa.

_Dibujos activos

Los elementos individuales de dibujo con su propio comportamiento pueden producir dibujos con o sin inputs de una persona. Estos dibujos activos son como si un mapache tropezara con una cubeta de pintura y luego corriera sobre el pavimento. Aunque están creados por una serie de reglas predeterminadas y acciones, estos dibujos son parcialmente autónomos. Ver ejemplo 44-04.

OUTPUT 2: EXPORTAR ARCHIVO

Los archivos digitales en las computadoras no son tangibles como sus análogos en papel, y ellos no se pueden colocar en carpetas durante años acumulando polvo. Un archivo digital es una secuencia de bytes colocado en una localización del disco duro del ordenador.

INPUT 6: IMPORTAR ARCHIVO

Los archivos son la forma más fácil de almacenar y cargar datos, pero antes de cargar un archivo de datos en un programa, es esencial conocer cómo el archivo está formateado. Como en un archivo de texto, el control de los caracteres con el tabulador o nueva línea se usa para diferenciar y alinear las componentes de los datos. Separando los elementos individuales con un tabulador o un carácter de espacio y cada línea con un carácter nueva línea es una técnica normal de formateo.
Para cargar datos como archivos de texto, es común cargar datos de archivos XML. XML tiene una estructura de archivo basada en la información por etiquetas, muy similar a los archivos HMTL. Se define una estructura para ordenar los datos, que deja el contenido y las categorías de los elementos de los datos abiertos.
En una estructura XML, se designa desde un libro de almacenamiento de información, donde cada elemento debe tener una entrada para el título y el editor.
En una estructura XML designada para almacenar una lista de websites, cada elemento debe tener una entrada para el nombre de la website y la **URL** (Uniform Resource Locator, es decir localizador uniforme de recurso. Es una secuencia de caracteres, de acuerdo aun formato estándar, que se usa para nombrar recursos, como documentos e imágenes en Internet, por su localización.).

_Cargando números

La manera más fácil de captar datos externos en Processing es guardarlos en un archivo con formato TXT. El archivo puede cargarse y analizarse para extraer los elementos individuales de los datos. Un archivo TXT almacena sólo caracteres de texto, esto quiere decir que no se ha editado con negrita, cursiva ni colores.
Los números se almacenan en filas como caracteres. La forma más fácil de cargarlos en Processing es tratar los números temporalmente como una string antes de convertirlos en float o int variables. Un archivo contiene números que pueden ser cargados en Processing con la función **loadStrings** ().Esta función lee los contenidos de un archivo y crea una array de string de las líneas individuales –una array de elementos de cada línea del archivo.
La función **split** () se usa para dividir cada línea del archivo de texto en elementos aislados. Esta función rompe una string en piezas usando un carácter o string como divisor.

split (str, delim);

El parámetro **str** debe ser una string, pero el parámetro **delim** puede ser un char o una string y no aparece en lo que devuelve String [] array.
La función splitTokens () nos permite romper una string en uno o varios caracteres "señal".

splitTokens (str);
splitTokens (str, tokens);

El parámetro tokens es una string que contiene una lista de caracteres que se usan para separar la línea. Si el parámetro tokens no se usa, todos los espacios en blanco (space, tab, new line,...) se usan como límites.

_Cargando caracteres

Cargar números de un archivo es similar que cargar datos de texto. Los archivos normalmente contiene múltiples clases de datos, por eso, es importante conocer qué clase hay dentro del archivo y puede ser analizada dentro de unas variables apropiadas.
En el siguiente ejemplo, se carga un texto de un libro en el programa y éste cuenta el número de las palabras, escribiendo palabras de más de diez letras en la consola. Se usa una variable llamada WHITESPACE, una string que contiene el control más común de los caracteres creando un espacio en blanco en un archivo de texto.

INPUT 7: INTERFAZ

El primer paso para construir un elemento en la interfaz es hacer una forma con el consentimiento del ratón. Las dos formas que más fácilmente reconoce el ratón con sus límites son el círculo y el rectángulo.
La clase **OverCircle** tiene 4 campos: coordenada X, coordenada Y, diámetro y valor de gris. El método **update** () se ejecuta cuando el ratón está sobre el elemento, y el método **display** () dibuja el elemento en la pantalla. La posición y tamaño del círculo están definidos con el constructor, y el valor de gris por defecto es el negro (0).
La función **dist** () con el **update** () calcula la distancia del ratón al centro del círculo, si la distancia es más pequeña que el radio del círculo, al valor de gris se le asigna el color blanco (255).
Los campos y métodos de la clase **OverRect** son idénticos a los de **OverCircle**, pero el campo del tamaño ahora se define con la anchura y altura del rectángulo, en vez de con el diámetro del círculo. Las expresiones de relación dentro del **update** () pretende ver si la posición siguiente de los valores mouseX y mouseY es la del rectángulo.

ESTRUCTURAS 5: OBJETOS II

Cuando un programa empieza a crecer y las ideas se convierten más ambiciosos, los conceptos de los objetos adicionales programados y sus técnicas confieren mucha importancia a la organización del código.

_Múltiples constructores

Una clase puede tener constructores múltiples que asignan los campos de diferentes maneras. A veces es útil especificar cada especto de los datos de un objeto asignando parámetros en los campos, pero otras veces, es más apropiado definir un parámetro o unos cuantos.

_Herencia

Una clase puede estar definida usando otra clase como unos cimientos. En terminología de programación de clases, una clase puede heredar campos y métodos de otra clase. Un objeto que hereda de otra es llamada subclase, y el objeto del que se hereda, es llamado superclase. Cuando una clase se extiende a otra, todos los métodos y campos de la superclase son automáticamente incluidos en la subclase.

SIMULAR 1: BIOLOGÍA

Las simulaciones de los programas emplean toda la potencia de los ordenadores para modelar los aspectos del mundo, como el tiempo o los modelos de tráfico. Una tremenda cantidad de energía intelectual en el campo de los gráficos de ordenador ha sido dedicada para afinar la simulación de las texturas de la luz y los movimientos de los materiales físicos como la ropa o el pelo.

__Autómata celular

Un autómata celular (CA) es un sistema auto-operativo que comprende una parrilla de celdas y reglas que se ejecutan según el comportamiento en relación con su vecino.

_Agentes autónomos

Un agente autónomo es un sistema que percibe y actúa según su entorno y sus propias reglas. La gente, las arañas y las plantas son agentes autónomos. Cada agente usa inputs del entorno como básicas para sus acciones.

SIMULAR 2: FÍSICA

La simulación física es una técnica que crea relaciones entre los elementos del programa y las reglas del mundo físico. Ayuda a comprender a la gente lo que ve en la pantalla.

_Simulación de movimiento

Los fenómenos de simulación física en programación, requieren un modelo matemático. El primer ejemplo que genera movimiento (31-01, p.279) usa una variable llamada speed para crear movimiento. En cada frame de la animación, la variable **y** es actualizada por la variable speed:

$$y = y + speed$$

Usando el código, la posición del círculo definido por la variable Y es modificado por la misma cantidad en cada frame. El código no tiene en cuenta otras fuerzas que pudieran interactuar con el círculo. Por ejemplo, el círculo podría tener una gran masa, o la gravedad podría aplicar una gran fuerza, o podría moverse a través de una superficie áspera y la alta fricción hacerla avanzar más despacio. Estas fuerzas están siempre presentes en el mundo físico, pero ellas afectan a la simulación de ordenador sólo si estas fuerzas están incluidas como parte del diseño. Estas fuerzas necesitar ser calculadas en cada frame para ejercer su influencia.

La velocidad cambia la posición del elemento, y la aceleración cambia la velocidad. La velocidad define la variable speed y la dirección con un solo número.
La aceleración define el rango de cambio de la velocidad. Un valor de la aceleración más grande que cero significa que la velocidad se incrementará en cada frame, y un valor de la aceleración menor que cero, significa que la velocidad decrecerá en cada frame. Usando los valores de la velocidad y la aceleración controlaremos la posición de los elementos visuales y si cambia la dirección o aumenta o disminuye la velocidad. La posición de un objeto se actualiza en dos pasos:

velocidad = velocidad + aceleración
y = y + velocidad

En el ejemplo 50-01 se usa los valores de la velocidad y la aceleración como una variable simple speed. Como la aceleración tiene un valor de 0.01, la velocidad se incrementa, por lo tanto el movimiento del círculo será más rápido en cada frame.
En el ejemplo 50-02, el círculo continuamente va decelerando hasta que algunas veces se para y cambia de dirección. Esto sucede porque la aceleración negativa gradualmente decrece la velocidad hasta que llega a ser negativa.
La fricción es una fuerza que va en contra de la dirección del movimiento. La velocidad de arrastrar un libro a lo largo de una mesa está afectada por la fricción entre las dos superficies. En el programa, la fricción es un número entre 0.0 y 1.0 que reduce la velocidad. En el ejemplo 50-03, el valor de la fricción está multiplicado por el valor de la velocidad en cada frame para gradualmente reducir la distancia que se desplaza el círculo en cada frame. Cambiando el valor de la variable de la fricción en el código, afecta al movimiento de la pelota.
Las componentes de la dirección y de la velocidad pueden ser alteradas independientemente. Si revertimos la dirección de la velocidad, simulamos el rebote de la pelota. En el ejemplo 50-04 invertimos la velocidad cuando el borde del círculo toca el final de la ventana de salida. La aceleración de 0.1 simula la gravedad, y la fricción reduce gradualmente la velocidad y detiene el rebote temporalmente.

_Sistemas de partículas

Un sistema de partículas es una array de partículas que responden al entorno y a otras partículas, simulando y renderizando fenómenos como el fuego, el humo y el polvo. Las partículas están afectadas por fuerzas y son usadas típicamente para simular las leyes físicas para generar movimiento.
Describir una clase simple de partículas puede ayudar a gestionar la complejidad de un sistema de partículas. La clase **Particle** tiene campos para el radio y la gravedad, un par de campos para almacenar la posición y la velocidad. La gravedad actúa como una variable de aceleración. Los parámetros del constructor definen la posición inicial, velocidad y radio.
En el ejemplo 50-06 se muestra como usar la clase **Particle**. En este como en la mayoría de ejemplos que usan objetos, la variable objeto se declara fuera del **setup** () y del **draw** ().
La clase **Particle** está muy limitada, pero permite la extensión y creación de más comportamientos aplicables. La clase **GenParticle** extiende la clase **Particle**, ya que la partícula vuelve a su posición inicial cuando se mueve y sale de la ventana de salida. Esto permite un continuo flujo de partículas con un número fijo de objetos. En el

ejemplo 50-07, las dos variables **originX** y **originY** almacenan las coordenadas del origen y el método **regenerate** () reposiciona la partícula cuando sale de la ventana de salida y resetea su velocidad.
El objeto **GenParticle** se usa de la misma forma que en el caso del objeto **Particle**, pero el método **regenerate** () necesita estar ejecutándose y al final eliminar el flujo de partículas.
La clase **LimitedParticle** extiende la clase **Particle** para cambiar la dirección de la velocidad cuando una partícula golpea el fondo de la ventana de salida. Esto introduce fricción y el movimiento de cada partícula se reduce en cada frame.
La clase **LimitedParticle** se usa en el ejemplo 50-10 para crear una pantalla llena de pequeñas bolas rebotando. Cada una empieza con una velocidad diferente, pero ellas se van parando y algunas veces llegan a detenerse en el fondo de la pantalla.
Las partículas en los ejemplos anteriores son dibujadas como círculos para hacer el código más sencillo de leer. Pero las partículas pueden ser dibujadas como cualquier forma. La clase **ArrowParticle** usa los campos y los métodos de su superclase para controlar la velocidad y dirección de la partícula, pero añade código para calcular un ángulo y dibujar una forma de flecha. La función **atan2** () se usa para determinar el ángulo actual de la flecha. Este valor se usa para definir el valor de la rotación. La flecha se posiciona horizontalmente, pero la rotación cambia la posición hacia arriba o hacia abajo.
Cada flecha está asignada con un valor random dentro de un rango. El rango de la velocidad en X, va de 1.0 a 8.0 y el rango de la velocidad en Y, va de –5 a –1. En cada frame, la fuerza de la gravedad se aplica a cada partícula y el ángulo de cada flecha desciende hasta el suelo y algunas incluso desaparecen en el fondo de la pantalla.

_Muelles

Un muelle es un dispositivo elástico, que vuelve a su forma original después de estirarse o comprimirse. La fuerza de un muelle es inversamente proporcional a cuanto se encoge, esta es la ley de Hooke (f = -k*x).

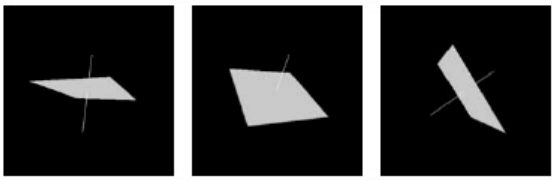
EXTENSIÓN 2: 3D

Antes de empezar a dibujar en 3D en Processing, es necesario decirle al programa con que motor de render va a trabajar. El motor de render por defecto en Processing es el P3D. Para usar el P3D hay que especificarlo como tercer parámetro en la función size ().

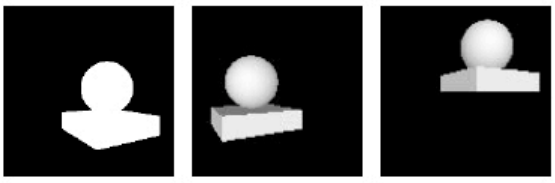
```
size (600, 600, P3D);
```

El motor de render **OPENGL** permite al boceto usar la librería **Opengl**, diseñada por artistas y diseñadores gráficos. Funciona mucho mejor y más rápido que el **P3D**.
Para importar una librería, seleccionamos "**Import Library**" del menú **Sketch** y se añadirá esta línea al principio de la ventana de salida:

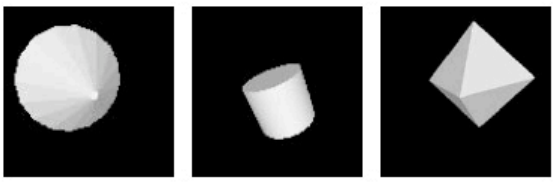
```
import processing.opengl.*;
```



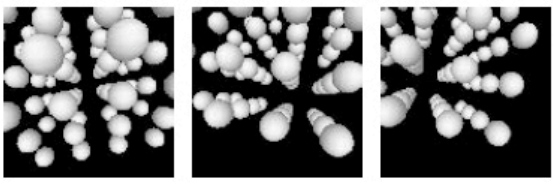
Example 1
The mouseX and mouseY values determine the rotation around the x-axis and y-axis.



Example 2
Draw a box and sphere. The objects move with the cursor. A mouse click turns the lights on.



Example 3
Shapes are constructed from triangles. The parameters for this shape can transform it into a cylinder, cone, pyramid, and many shapes in between.



Example 4
The geometry on screen is exported as a DXF file.

Example 5
Load a pre-constructed OBJ file and the mouse moves it from left to right.

Example 6
The mouse moves the camera position.

Example 7
The mouse position controls the specular quality of the sphere's material.

Example 8
Many types of lights are simulated. As the box moves with the cursor, it catches light from different sources.

Example 9
Textures are applied to geometry.

Con Processing se pueden exportar trabajos en formato DXF y al mismo tiempo importar archivos en formato OBJ (típicos archivos de programas 3D). Para hacerlo, sólo tenemos que llamar a las librerías externas, o bajarnos dichas librerías de la página oficial de Processing.

www.processing.org/reference/libraries

Con la función **ambientLight** () se especifican las luces ambientales que interactuarán con el ambiente del color de la forma dibujada. El ambiente de color de la forma dibujada se especifica con la función **ambient** (), función que toma los mismos parámetros de las funciones **fill** () y **stroke** (). Un material con el color ambiente blanco (255, 255, 255) reflejará toda la luz ambiente que le llega. Una superficie con un color ambiente verde oscuro (0, 128, 0) reflejará la mitad de la luz verde que recibe, pero nada de la roja o la azul.
La textura de los materiales es una importante componente de cualquier 3D. Processing permite colocar imágenes para ser mapeadas como texturas en las caras de los objetos. Las texturas se deforman según se deforman los objetos. En cada cara podemos tener una imagen mapeada y los vértices de la cara necesitan tener coordenadas 2D para la textura.

Estas coordenadas le indican al renderizador como tiene que encajar las imágenes para ocupar las caras. Las texturas son mapeadas para geometrizarlas usando una versión de la función **vertex** () con dos parámetros adicionales (**u**, **v**). Estos dos valores son las coordenadas X, Y de la imagen de la textura y son usadas para mapear los vértices de posición para que queden emparejados.

Ejemplo 6: Manipulación de la cámara

La posición y orientación de la cámara se define con la función **camera** (). Tiene 9 parámetros, ordenados en grupos de 3, para controlar la posición de la cámara, para localizar la situación de la cámara y para la orientación.

Ejemplo 7: Material

La función **lightSpecular** () define el color especular de las luces. La cualidad especular de un luz interactúa con las cualidades especulares del material, definido con la función **specular** (). La función **specular** () define el color especular de los materiales, que

establece el color de los aspectos más destacados. En este ejemplo, los parámetros de **specular** () cambian dependiendo de mouseX.

Ejemplo 8: Iluminando

Las funciones que crean cada tipo de luz, tienen diferentes parámetros, porque cada luz es única. La función **pointLight** () tiene 6 parámetros. Los 3 primeros definen el color de y los 3 siguientes definen la posición de la luz. La función **directionalLight** () también tiene 6 parámetros, los 3 primeros definen el color y los 3 siguientes, dónde está apuntando la dirección de la luz.
La función **spotLight** () es la más complicada con 11 parámetros para definir el color, posición, dirección, ángulo y concentración.

Ejemplo 9: Mapeando texturas

Este ejemplo muestra como aplicar una textura a una superficie plana, y como aplicar una textura a una serie de superficies planas que crean una forma curva. La función **texture** () define la textura que está aplicada a través de la función **vertex** (). Una versión de la función **vertex** () usando 5 parámetros, definimos con los 3 primeros las coordenadas (XYZ) y con los 2 último definimos las coordenadas (XY) de la textura de la imagen que mapeamos para hacerla en 3D. Los valores de seno y coseno que definen la geometría cuando la textura se aplica, están predefinidos en el setup () así que no se tendrá que recalcular en cada bucle del draw ().

EXTENSIÓN 3: VISIÓN

Ejemplo 1: Detectando el movimiento

Los movimientos de la gente (o de objetos), con una cámara de vídeo, pueden ser detectados y cuantificados, usando un sencillo método llamado frame diferenciado. En esta técnica, cada píxel en un vídeo (F1) es comparado con su correspondiente píxel en la secuencia siguiente (F2). La diferencia de color, o brillo entre los dos píxeles es la medida de cantidad de movimiento en esa situación concreta.

Ejemplo 2: Detectando la presencia

Una técnica llamada “background substraction” posibilita el detectar la presencia de gente o de otros objetos en una escena y distinguir que píxeles pertenecen a ellos y cuales no. Esta técnica funciona comparando cada fotograma del vídeo con una imagen almacenada de la escena del fondo de pantalla, detectando en un instante, cuando la escena está vacía. Esta técnica funciona bien en entornos heterogéneos, pero es muy sensible a los cambios de luces y depende de que los objetos tengan un suficiente contraste sobre la imagen del fondo de pantalla.

Ejemplo 3: Detectando a través del umbral de brillo

Con la ayuda de un control de iluminación (como la retroiluminación) y / o tratamientos de superficie (tales como pinturas de alto contraste), es posible garantizar que los objetos son considerablemente más oscuro o más ligero que su entorno.

En tales casos, los objetos de estudio pueden distinguirse basándonos en su propio brillo. Para ello, cada píxel de vídeo del brillo se compara con un valor-umbral y en consecuencia, se etiquetan como primer o segundo plano.

Example 1. *Detects motion by comparing each video frame to the previous frame. The change is visualized and is calculated as a number.*

Example 2. *Detects the presence of someone or something in front of the camera by comparing each video frame with a previously saved frame. The change is visualized and is calculated as a number.*

Example 3. *Distinguishes the silhouette of people or objects in each video frame by comparing each pixel to a threshold value. The circle is filled with white when it is within the silhouette.*

Example 4. *Tracks the brightest object in each video frame by calculating the brightest pixel. The light from the flashlight is the brightest element in the frame; therefore, the circle follows it.*

Un rudimentario sistema para el seguimiento de objeto, ideal para el seguimiento de la ubicación de un único punto de iluminación (por ejemplo, una luz de flash), se basa en localizar la situación de los píxeles más brillantes en cada nuevo fotograma de vídeo.

En este algoritmo, el brillo de cada píxel en el fotograma de vídeo entrante se compara con el valor del píxel más brillante encontrado antes en el frame anterior.
Si un píxel es más brillante que el valor del píxel más brillantes encontrado anteriormente, entonces, la ubicación y el brillo de ese píxel se almacenan. Después de todo, los píxeles son examinados y, a continuación, la ubicación más brillante en el fotograma de vídeo es conocida.
Esta técnica se basa en asumir, que sólo hay un objeto de interés. Con simples modificaciones, podemos igualmente localizar el píxel más oscuro de la escena, o localizar los objetos de color diferente.
Por supuesto, existen muchas más técnicas de software, de todos los niveles de sofisticación, para detectar, reconocer e interactuar con personas y otros objetos de interés.
De cada uno de los algoritmos de seguimiento que se ha descrito anteriormente, por ejemplo, se pueden encontrar versiones más elaboradas que modifican sus diversas limitaciones.
Otros algoritmos de fácil implementación pueden calcular determinadas características de seguimiento de un objeto, como por ejemplo su área, su centro de masas, orientación angular, compacidad, borde de los píxeles, contorno y características tales como esquinas y cavidades. Por otra parte, algunos de los algoritmos más difíciles de aplicar, que representan la vanguardia de la investigación de visión por ordenador hoy en día, son capaces (dentro de ciertos límites) de reconocer una única persona, captar la orientación de la mirada de una persona, o realizar una identificación facial correctamente. Ver el trabajo de Daniel Huber (HIPR).

_Código

Un vídeo puede ser capturado con Processing de cámaras USB, IEEE 1394 cámara, o tarjetas de vídeo. Los ejemplos que se explican a continuación asumen que ya tienes una cámara trabajando con Processing. Antes de probar estos ejemplos, conecta la cámara a tu ordenador y prueba como trabaja.

EXTENSIÓN 4: TRABAJAR CON LA RED

Las redes son formas complejas de organización. Estas formas complejas se pondrán asociar en entidades discretas o nodos, estos nodos que permiten conectarse a otros nodos y, de hecho, a otras redes.
Las redes que existen en el mundo tienen una gran variedad de formas y, aún más, contextos: políticos, sociales, biológicos, y de otra índole.
Si bien los artistas han utilizado las redes de muchas maneras -a partir de las redes postales para difundir su trabajo en las redes informales de colaboradores artísticos y en grandes movimientos estéticos, en esta sección se examinan específicamente un único ejemplo de tecnología de red, **Internet**, y cómo los artistas han incorporado esta la tecnología en su trabajo.
Hay dos tendencias generales: hacer arte donde Internet se utiliza como una herramienta para la rápida y fácil difusión de la obra, y hacer arte, donde Internet es el medio de la propia obra.
Estas dos tendencias no son mutuamente excluyentes. Algunos de los trabajos más interesantes en línea, mezclan técnicas de las dos tendencias en nuevas y emocionantes maneras que superan los problemas de cualquier técnica.

Ejemplo 1: Cliente Web

La librería **Net** incluida en Processing lee y crea clases tanto para servidores y clientes.
En fin para obtener una página de la Web, un primer paso es crear un cliente y conectarse a la dirección del servidor remoto. Usando una simple técnica de llamada y respuesta, el cliente pide al archivo, y el archivo es devuelto por el servidor.
Esta técnica de llamada y respuesta se define con un protocolo llamado Hypertext Transfer Protocol (**HTTP**). El HTTP se compone de un puñado de simples comandos que se utilizan para describir el estado del servidor y el cliente, petición archivos y para enviar datos al servidor si es necesario.
El comando HTTP más básico es **GET**. Este comando es similar a rellenar un formulario de solicitud de libro en una biblioteca: el cliente pide una archivo por su nombre, el servidor "recibe", que esa petición y lo envía el archivo al cliente.
HTTP también incluye una serie de códigos de respuesta para indicar que el archivo fue encontrado con éxito, o para indicar si se detectaron errores fueron (por ejemplo, si el archivo que pidió no existe).
El comando **GET / HTTP/1.0 \ n** significa que el cliente está solicitando el archivo por defecto en la raíz del directorio web (/) y que el cliente es capaz de comunicarse mediante HTTP la versión 1.0. El rastrero **\n** es el carácter de nueva línea, o el equivalente a golpear la tecla de enter. Si el archivo existe por defecto, el servidor lo transmite al cliente.
La unión de la dirección **IP** y el **número de puerto** (por ejemplo: 123.45.67.89:80) se llama un **socket**. El socket es lo fundamentar para la creación de redes.

Ejemplo 2: Lienzo de dibujo compartido

Usando la biblioteca **Net** de Processing, es posible crear un servidor simple. El ejemplo muestra un servidor que comparte un lienzo de dibujo entre los dos equipos. Con el fin de abrir un conexión de **socket**, un servidor debe seleccionar un puerto en el que estará en escucha para los clientes entrantes y a través de ese puerto comunicarse.
Aunque cualquier número de puerto puede ser utilizado, es mejor practicar para evitar el uso de números de puerto ya asignados a otras aplicaciones de red y protocolos. Una vez el **socket** está establecido, un cliente puede conectar con el servidor y enviar o recibir comandos y datos.

Vinculados con este servidor, la clase **ProcessingClient** se instancia especificando un dirección remota y el número de puerto, donde la conexión **socket** debe hacerse.
Una vez la conexión se realiza, el cliente puede leer (o escribir) los datos al servidor. Debido a los clientes y los servidores son las dos caras de la misma moneda, el código de los ejemplos son casi idénticos para ambos. Para este ejemplo, los coordenadas actuales y anteriores del ratón son enviadas entre el cliente y el servidor varias veces por segundo.

Ejemplo 3: Yahoo! Búsqueda SDK

A medida que Internet evoluciona desde un relativamente simple red de archivos y servidores a un red de datos que se convierte en más personalizada, la capacidad de los

clientes Web de seleccionar determinadas fuentes de datos en la red es cada vez más precisa.
Si consideramos la diferencia entre navegar en una web de metereología para ver la temperatura actual, frente a un ping de servicio Web con un código postal, y teniendo en ese servidor respuesta con un único número de referencia a la temperatura Fahrenheit para que el código postal.
Para el programador Web, esta evolución es bienvenida, ya que simplifica en gran medida el hecho de ir a buscar y analizar datos en línea por la disociación de los datos de la inmensidad innecesaria existente en el archivo de texto HTML que los rodea en cualquier página Web típica.
Uno de esos servicios Web es la Yahoo! motor de búsqueda. Usando de Yahoo, es posible realizar las consultas de una forma automatizada. (Esta técnica de utilizar un cliente HTTP para enviar y recibir búsquedas de las preguntas, que luego debe ser despojado de HTML y analizada como variables.) Yahoo esencialmente, es como una caja negra de conexión de la Web.
Esto ejemplo se utiliza el SDK para conectarse a la página principal del servidor de Yahoo y la buscando "processing.org." Por defecto, devuelve los 20 primeros resultados, pero ese número se puede ajustar. Para cada resultad, el título de la página web y su URL se imprimen a la consola.
Ver el ejemplo a fondo.

Ejemplo 4: Cliente **Carnivore**

Si en un nivel más profundo examinamos los flujos de las redes de datos, la librería **Carnivore** de Processing permite al programador ejecutar un analizador de paquetes en el entorno de Processing.
Un analizador de paquetes es cualquier aplicación que es capaz de escuchar indiscriminadamente el tráfico de datos que viaja a través de una red local (**LAN**), incluso si el tráfico no va dirigido al ordenador donde se ejecuta el análisis.
Si bien esto puede sonar poco ortodoxo y, de hecho, una máquina ejecutando un análisis es descrito como estar en "Modo promiscuo", los analizadores de paquetes son tecnologías que están omnipresentes en Internet. Los administradores de sistemas usan analizadores de paqueteas para solucionar problemas en las redes.
Todos los Macintosh navegan con el analizador de paquetes **tcpdump** preinstalado, mientras que Windows y los usuarios de Linux tienen un surtido de analizadores libres (incluido el tcpdump y sus variantes) para elegir.
La librería **Carnivore** de Processing simplifica el hecho de analizar paquetes, haciendo en tiempo real, el tráfico de vigilancia simple y fácil de aplicar para cualquier artista que lo desee.
Los paquetes capturados a través de **Carnivore** se pueden visualizar en forma de mapa, analizados por palabras clave, o simplemente para cualquier tipo de algoritmo que requiera un flujo constante de eventos aleatorios para que se empiecen a ejecutar.
Carnivore es un buen avance para las discusiones planteadas aquí sobre el fin de las redes de ordenadores ,los protocolos de Internet.
Un protocolo es un estándar tecnológico. Los protocolos de Internet son una serie de documentos que describen cómo poner en práctica las tecnologías estándar de Internet, tales como el enrutamiento de datos, handshaking entre dos máquinas,...
Dos protocolos ya han sido objeto de debate - **HTML**, que es el lenguaje de hipertexto protocolo para la disposición y el diseño, y **HTTP**, que es el protocolo para acceder a la

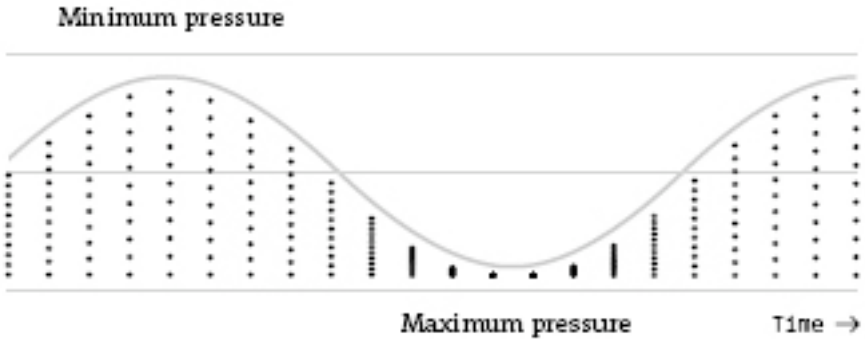
web accediendo a archivos - pero hay algunos otros protocolos vale la pena discutir en este contexto.

EXTENSIÓN 5: SONIDO

La historia de la música es, en muchos sentidos, la historia de la tecnología. De la evolución en la redacción y trascripción de la música (notación) para el diseño de espacios para el la ejecución de la música (la acústica) para la creación de instrumentos musicales, compositores y los músicos se han valido de los avances en la comprensión humana para perfeccionar y avanzar en sus profesiones.

_Sonido e informática musical

En pocas palabras, podemos definir el sonido como vibraciones que viajan a través de un medio (normalmente aire) que podemos percibir a través de nuestro sentido del oído. El sonido se propaga como una onda longitudinal que comprime y descomprime alternativamente las moléculas en la materia (por ejemplo, aire) a través de la cual se desplaza. Como resultado de ello, podemos representar un sonido como un área de presión durante el tiempo.



EXTENSIÓN 6: IMPRIMIR

Las tecnologías digitales han generado muchos cambios en el campo de la impresión dentro de las artes. La introducción de las impresoras láser y ordenadores personales en las oficinas de diseño a mediados de 1980, fue un catalizador durante años de experimentación e innovación en diseño de tipografía, diseño e impresión.
Los artistas han producido impresiones desde software desde mediados de los ‘60, pero esas técnicas se han mejorado desde 1990.
Las innovaciones digitales han permitido dar una mayor vida a sus trabajos, que las fotografías impresas desde una película. La reciente avalancha de cámaras digitales ha producido otro cambio. Amateurs y profesionales están dejando de lado a los laboratorios e imprimen sus imágenes en su casa.

Ejemplo 1: Renderizar en PDF

Cuando usamos PDF como el tercer parámetro para la función **size** (), el programa renderiza en un archivo PDF en lugar de dibujarar a la ventana de visualización. El

nombre del archivo se establece con el nombre del cuarto parámetro de **size** () y el archivo se guarda en la carpeta de **sketches**.
La mayoría de los **sketches** se puede renderizar en formato PDF con sólo añadir los dos parámetros para **size** () y seleccionando **Sketch** -> **Import Library** -> **PDF**. Una vez que hacemos esto, ya no veremos la imagen en la ventana de salida, ya que se está ejecutando, pero es posible crear un archivo PDF con un tamaño mucho más grande de la ventana de salida.

Ejemplo 2: Renderizar la ventana de salida, exportar en PDF

En este ejemplo, se guarda un archivo PDF al mismo tiempo que se dibuja en la pantalla. La función **beginRecord** () abre un nuevo archivo, y todas las funciones que se dibujaron se grabaron es este archivo, así como en la ventana de visualización.
La función **endRecord** () detiene el proceso de grabación y cierra el archivo. La función **beginRecord** () requiere de dos parámetros, el primero se usa para renderizar (en este ejemplo, en PDF), y el segundo es el nombre del archivo.

Ejemplo 3: Guardar un frame de un programa continuo

Este ejemplo guarda un archivo en PDF cada vez que se presiona el ratón. La variable boolean **saveOneFrame** se establece como **true** cuando se presiona un botón del ratón, iniciando así la función **beginRecord** () que se ejecute la próxima vez que empiece el draw (). Al final del draw, la función **endRecord** () se ejecuta y la variable se establece en **false**, por lo que otro archivo no se guardará mientras dibujo el siguiente frame. Cada archivo PDF está numerado con el frame actual (el número de frames transcurridos desde que se inició el programa).

Ejemplo 4: Acumular muchos frames dentro de un archivo PDF

Este ejemplo guarda varios frames dibujados en la pantalla en un solo archivo PDF. El archivo se abre cuando la tecla **B** está presionada, y todo lo dibujado en los siguientes frames se guarda en el archivo, hasta que la tecla **E** se presiona.
La función **background** () se ejecuta después de **beginRecord** () para refrescar el fondo de la pantalla cuando el documento PDF está en la ventana de visualización. Este ejemplo dibuja sólo una nueva línea en el archivo PDF cada frame, pero es posible escribir miles de líneas de cada frame. Sin embargo, cuando los archivos de vectores son muy grandes, los ordenadores pueden tener dificultades para abrir e imprimir.

Ejemplo 5: Guardar una imagen TIFF de alta resolución desde la ventana de visulización

Este ejemplo crea un archivo TIFF más grande que la pantalla y lo dibuja directamente sin tener que dibujarlo en la pantalla. La función **createGraphics** () crea un objeto de la clase **PGraphics** (PGraphics es el contexto principal de gráficos y renderización de Processing).
El método **beginDraw** () es necesario para preparar al programa para dibujar, entonces cada función de dibujo posterior se escribe dentro del gran objeto raster. Los métodos **endDraw** () y **save** () son necesarios para completar el archivo y guardarlo en el ordenador para más tarde ser usado en un programa diferente, como Photoshop o GIMP.

Ejemplo 6: Escalar un segmento en una imagen

Este ejemplo guarda una serie de imágenes con la resolución de la pantalla. de una sola imagen ampliada a cualquier dimensión. Estas imágenes puede ser manipuladas como un mosaico en un editor de imágenes, como Photoshop para crear una única imagen de alta resolución.