

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática



Introducción a la Programación con Processing

Modesto Castrillón Santana

Índice general

1. Introducción a Processing	1
1.1. Introducción	1
1.2. Instalación y entorno de programación	1
1.3. Programación en modo básico	3
1.3.1. Dibujo de primitivas básicas	3
1.3.2. Variables	7
1.3.3. Repeticiones	8
1.3.4. Dibujar un tablero de ajedrez	8
1.4. Programación en modo continuo	11
1.4.1. El bucle infinito de ejecución	11
1.4.2. Imágenes	16
1.4.3. Control	17
1.4.4. Interacción	24
1.4.4.1. Teclado	24
1.4.4.2. Ratón	26
1.4.5. Creando un Paint	28
1.5. Avanzado	30
1.6. Tipos de datos	33
1.7. Applets	35
1.8. Utilidades gráficas	35
1.9. Referencias y fuentes	41

Capítulo 1

Introducción a Processing

1.1. Introducción

Processing es un proyecto de código abierto con fines creativos que se basa en el lenguaje Java, se concibe como un cuaderno de dibujo para artistas informáticos, programadores y diseñadores, siendo muy utilizado en dichas comunidades por su facilidad de uso. La facilidad sintáctica de Java, y la enorme comunidad existente, sirven de gran apoyo, ofreciendo un conjunto de herramientas de código abierto para la creación de aplicaciones creativas. Su diseño pretende facilitar la programación que integre imágenes, animación, sonido e interacción.

Es utilizado por estudiantes, artistas, diseñadores, investigadores y aficionados tanto para el aprendizaje, como la creación de prototipos y la producción audiovisual. De esta forma cubre necesidades no sólo para enseñar los fundamentos de programación, sino también como cuaderno de prototipos software, o herramienta de producción profesional. Processing está disponible en el siguiente enlace [Processing](http://processing.org/)¹, si bien en la sección de referencias se relacionan otros recursos con ejemplos y demos.

1.2. Instalación y entorno de programación

La instalación requiere realizar la descarga a través de processing.org, y descomprimir. Una vez realizados los preparativos, tras lanzar el entorno se te presenta la interfaz, si no hubieras activado la interfaz en español, y quisieras cambiarla, prueba a a través del menú con *File* → *Preferences*. No olvides que es posible acceder a los ejemplos a través de la barra de menú *Archivo* → *Ejemplos*.

El entorno de desarrollo de Processing (PDE), ver figura 1.1, consiste en un

¹<http://processing.org/>

simple editor de texto para escribir código, un área de mensajes, una consola de texto, fichas de gestión de archivos, una barra de herramientas con botones para las acciones comunes, y una serie de menús. Cuando los programas se ejecutan, se abren en una nueva ventana llamada la ventana de visualización (*display window*).

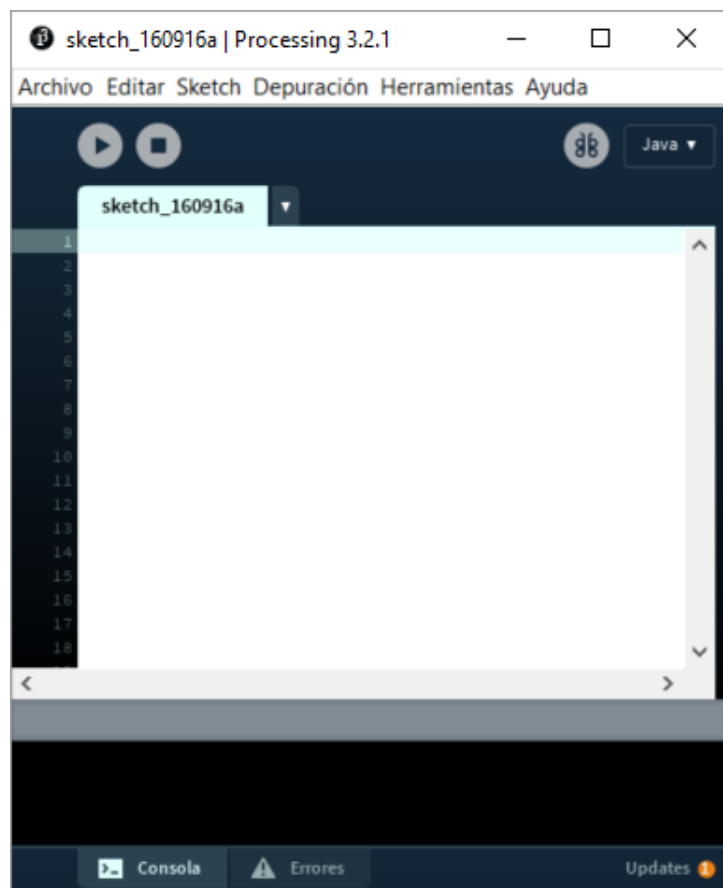


Figura 1.1: Imagen del entorno de programación, PDE, de Processing.



Cada pieza de software escrito con Processing se denomina boceto o *sketch*. Se escribe a través del editor de texto, disponiendo de las funciones típicas para cortar y pegar, así como las de búsqueda y reemplazo de texto.

El área de mensajes, en la parte inferior, ofrece información de la salida de texto del programa en ejecución, al hacer uso de las funciones *print()* y *println()*, además de mensajes de error, tanto en ejecución como durante la edición. Las utilidades para la depuración integradas en el entorno están disponibles desde la versión 2.0b7.

Los botones de la barra de herramientas permiten ejecutar y detener programas, ver Tabla 1.1. Comandos adicionales se encuentran dentro de la barra de

menú: *Archivo, Editar, Sketch, Depuración, Herramientas, Ayuda*. Los submenús son sensibles al contexto, lo que significa que sólo los elementos pertinentes a la labor que se está llevando a cabo están disponibles.

Tabla 1.1: Resumen de comandos habituales

	Ejecutar	Compila el código, abre una ventana de visualización, y ejecuta el programa
	Detener	Finaliza la ejecución de un programa

1.3. Programación en modo básico

Processing permite varios modos de programación: el modo básico, y el modo continuo. Mencionaremos brevemente ambos, para concluir con los *applets*.

El modo básico permite la elaboración de imágenes estáticas, es decir que no se modifican, además de poder ser útil para aprender los fundamentos de la programación. Líneas simples de código tienen una representación directa en la pantalla.

1.3.1. Dibujo de primitivas básicas

Un ejemplo mínimo de dibujo de una línea entre dos puntos de la pantalla.

Listado 1.1: Ejemplo de dibujo de una línea

```
line(0,0,10,10);
```

Se debe tener en cuenta que se emplea el sistema de coordenadas cartesiano, como es habitual, teniendo su origen en la esquina superior izquierda como se muestra en las figuras 1.2 y 1.3. Si se define una ventana de 320×240 píxeles, la coordenada $[0,0]$ se corresponde con la esquina superior izquierda como se ilustra en la figura 1.2.

Processing también puede simular el dibujo en tres dimensiones. En la superficie de la imagen, la coordenada z es cero, con valores z negativos moviéndose hacia atrás en el espacio. Cuando se realiza el dibujo en 3D simulado, la cámara se coloca en el centro de la pantalla.

Un siguiente paso lo damos al dibujar dos líneas modificando el color del pincel con la función *stroke*. En este [enlace](#) puedes practicar con el espacio de color RGB (rojo, verde y azul) modificando los valores de cada canal. RGB es el modelo de color por defecto que puede modificarse con *colormode()*.

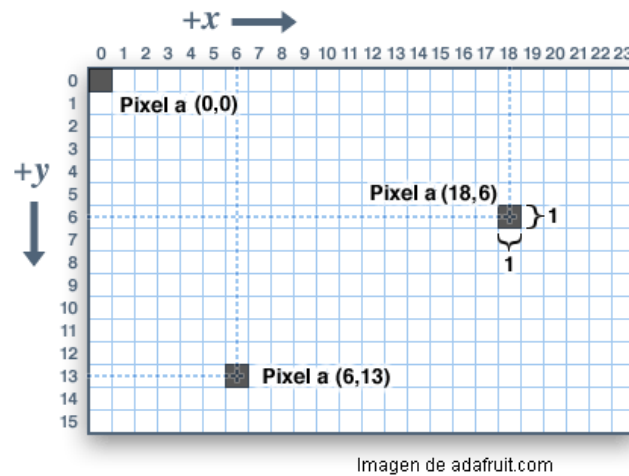


Figura 1.2: Sistema de coordenadas de la pantalla.

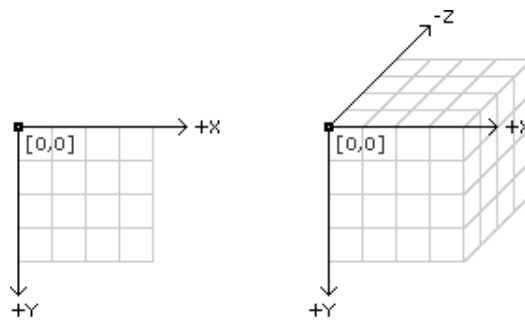


Figura 1.3: Sistema de coordenadas en 2 y 3 dimensiones.

Listado 1.2: Dibujo de dos líneas modificando el color

```
stroke(255,0,0); // Tripleta RGB
line(0,0,10,10);

stroke(0,255,0);
line(30,10,10,30);
```

Indicar que con el comando *stroke*, puede especificarse un único valor, entre 0 y 255, que se interpreta como tono de gris, p.e. *stroke(0)*; especifica el color negro, y *stroke(255)*; el blanco. Es también posible indicar la tripla RGB como un valor hexadecimal *stroke(#9ACD32)*;

También es posible establecer de forma arbitraria el tamaño de la ventana de visualización con el comando *size()*.

Listado 1.3: Una línea en una ventana mayor

```
size(640, 360);  
stroke(255,0,0);  
line(0,0,10,10);
```

Si no nos decidimos con el color, o simplemente queremos que cambie con cada ejecución, podemos escogerlo aleatorio haciendo uso de *random()*.

Listado 1.4: Dibujo de una línea con color aleatorio

```
size(640, 360);  
stroke(random(255),random(255),random(255));  
line(0,0,10,10);
```

Una vez que comprendemos el dibujo de líneas, para pintar un cuadrado podemos hacer uso de cuatro líneas bien colocadas. Sugiero deducir las coordenadas de los cuatro vértices antes en papel.

Listado 1.5: Dibujo de un cuadrado con cuatro líneas

```
background(128);  
  
size(400,400);  
  
strokeWeight(2); //Modifica el grosor del pincel  
line(30,30,30,60);  
line(30,60,60,60);  
line(60,60,60,30);  
line(60,30,30,30);
```

Realmente existen comandos que facilitan el dibujo de otras primitivas sencillas. Para conocer todas las primitivas 2D, ver 2D primitives en *Ayuda* → *Referencia*.

Listado 1.6: Dibujo de un cuadrado

```
stroke(255,255,255);  
rect(0,0,10,10);
```

El color de relleno se define con *fill()*, afectando a las primitivas a partir de ese momento. Las funciones *noFill* y *noStroke* cancelan respectivamente el relleno y el borde de las primitivas.

Listado 1.7: Dibujo de un cuadrado sólido

```
stroke(255,0,255);  
fill(232,123,87);  
rect(0,0,10,10);
```


En todos los comandos que define un color, la especificación de un único valor se interpreta como nivel de gris (0 negro, 255 blanco). Si se indicaran 4, el último define la transparencia, el canal alfa.

A modo de resumen, el siguiente código muestra el uso de varias primitivas 2D.

Listado 1.8: Primitivas 2D

```
size(450,450);

stroke(128);
fill(128);

ellipse(200,300,120,120); //Por defecto modo con coordenadas del centroy di'
    ametro

stroke(255,0,255);
noFill();
strokeWeight(2);
ellipse(400,300,60,60);

stroke(123, 0, 255);
strokeWeight(10);
ellipse(40,123,60,60);

stroke(0);
strokeWeight(1);
line(40,123,400,300);

triangle(10, 240, 50, 245, 24, 280);

fill(0);
rect(190,290,30,50);

stroke(255,0,0);
fill(255,0,0);
bezier(5,5,10,10,310,320,320,20);
```

Te propongo componer un dibujo combinando varias primitivas y colores (*rect*, *ellipse*, *line*, ...), ver por ejemplo el Mondrian de la figura 1.4.

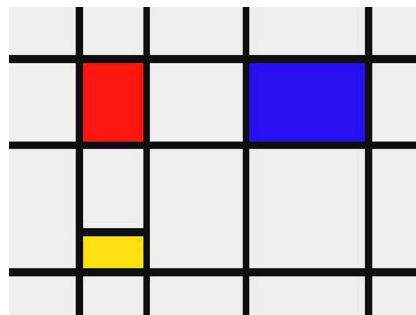


Figura 1.4: Piet Mondrian, *Composición con rojo, amarillo y azul* (1930).

1.3.2. Variables

Un paso más supone el uso de variables en el código. Para comenzar, ilustramos en uso de variables con aquellas presentes durante la ejecución, como son las dimensiones de la ventana que se almacenan respectivamente en las variables *width* y *height*, y que utilizamos en el siguiente código para pintar una estrella simple en el centro de la ventana, independientemente de las dimensiones que le fijemos con *size()*.

Listado 1.9: Dibujo de una estrella o asterisco

```
line ( width/2-10,height/2-10,width/2+10,height/2+10);  
line ( width/2+10,height/2-10,width/2-10,height/2+10);  
line ( width/2,height/2-10,width/2,height/2+10);  
line ( width/2+10,height/2,width/2-10,height/2);
```

Cada variable es básicamente un alias o símbolo que nos permite hacer uso de una zona de almacenamiento en memoria. Dado que recordar la dirección de memoria, un valor numérico, es engorroso, se hace uso de un nombre o identificador que permite darle mayor semántica a aquello que contiene la variable. En el caso de las variables del sistema mencionadas, *width* es una variable que justamente almacena el ancho de la ventana. Las variables se caracterizan por el nombre, el valor que contienen, su dirección, y el tipo de datos.

Modifica el ejemplo para pintar una estrella de mayor radio, es decir mayor que 10.

Una gran ventaja del uso de variables es que un programador puede definir y utilizar sus propias variables a conveniencia. Para ello en Processing es necesario declarar cualquier variable antes de utilizarla. El siguiente ejemplo utiliza **l** para establecer el tamaño de la estrella.

Listado 1.10: Dibujo de una estrella variable

```
int l=10;  
  
line ( width/2-l,height/2-l,width/2+l,height/2+l);  
line ( width/2+l,height/2-l,width/2-l,height/2+l);  
line ( width/2,height/2-l,width/2,height/2+l);  
line ( width/2+l,height/2,width/2-l,height/2);
```

En este nuevo ejemplo pintamos una línea y un círculo de un determinado radio, haciendo uso de la función *ellipse* definiendo previamente el color de relleno.

Listado 1.11: Dibujo de un círculo

```
int Radio = 50 ;  
  
size (500,500);
```

```
background(0);  
stroke(80);  
line(230,220,285,275);  
fill(150,0,0);  
ellipse(210,100,Radio, Radio);
```

1.3.3. Repeticiones

Pensemos en crear una ventana de dimensiones 800×800 en la que pintamos líneas verticales de arriba a abajo cada 100 píxeles. recordar que la coordenada y de a parte superior es 0, y la inferior es 800 o **height** si usamos la variable correspondiente.

Listado 1.12: Dibujo de varias líneas verticales

```
size(800,800);  
background(0);  
stroke(255);  
line(100,1,100,height);  
line(200,1,200,height);  
line(300,1,300,height);  
line(400,1,400,height);  
line(500,1,500,height);  
line(600,1,600,height);  
line(700,1,700,height);
```

Realmente las llamadas a la función *line* son todas muy similares, sólo varían las coordenadas *x* de los dos puntos. Los lenguajes de programación facilitan la especificación de llamadas repetidas por medio del uso de bucles. Una versión más compacta del dibujo de las líneas verticales. El bucle define una variable, **i**, a la que asignamos un valor inicial 100, un valor final, 700, y la forma en que se va modificando **i** con cada ejecución, en este caso añadiendo 100.

Listado 1.13: Dibujo de varias líneas verticales

```
size(800,800);  
background(0);  
stroke(255);  
for (int i=100;i<=700;i=i+100){  
  line(i,1,i,height);  
}
```

Particularmente útil cuando las repeticiones son cientos o miles. ¿Cómo lo modificarías para hacer además líneas horizontales?

1.3.4. Dibujar un tablero de ajedrez

Como tarea similar podemos plantear el dibujo de un tablero de ajedrez, que contiene 64 casillas como muestra la figura 1.5. Fijando el fondo a blanco, los

cuatro recuadros de la primera fila del tablero de ajedrez.

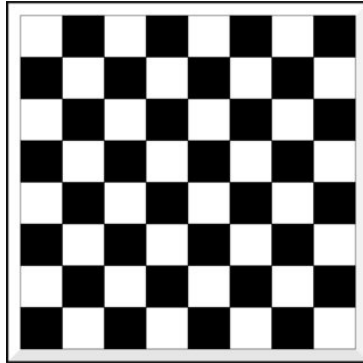


Figura 1.5: Rejilla estilo tablero de ajedrez.

Listado 1.14: Dibujo de una rejilla

```
size(800,800);  
background(255);  
fill(0);  
// Primera fila  
rect(0,0,100,100);  
rect(200,0,100,100);  
rect(400,0,100,100);  
rect(600,0,100,100);
```

Haciendo uso de un bucle *for* para pintar esas cuatro casillas negras

Listado 1.15: Dibujo de una rejilla con un bucle

```
size(800,800);  
background(255);  
fill(0);  
// Primera fila  
  
for (int i=0;i<=600;i=i+200){  
    rect(i,0,100,100);  
}
```

Pintamos cuatro filas, cada una con su bucle particular.

Listado 1.16: Dibujo de una rejilla con varios bucles

```
size(800,800);  
background(255);  
fill(0);  
// Primera fila  
  
for (int i=0;i<=600;i=i+200){  
    rect(i,0,100,100);
```

```

}
for (int i=0;i<=600;i=i+200){
    rect(i,200,100,100);
}
for (int i=0;i<=600;i=i+200){
    rect(i,400,100,100);
}
for (int i=0;i<=600;i=i+200){
    rect(i,600,100,100);
}

```

Realmente cada bucle es similar a los demás, sabiendo que es posible anidar bucles, con un bucle doble queda mucho más compacto.

Listado 1.17: Dibujo de una rejilla de ajedrez

```

size(800,800);
background(255);
fill(0);
// Primera fila

for (int j=0;j<=600;j=j+200){
    for (int i=0;i<=600;i=i+200){
        rect(i,j,100,100);
    }
}

```

Nos quedan las otras cuatro filas, que resultan de una leve variación ya que se alternan los tonos blancos y negros.

Listado 1.18: Dibujo de un tablero de ajedrez

```

size(800,800);
background(255);
fill(0);
// Primera fila

for (int j=0;j<=600;j=j+200){
    for (int i=0;i<=600;i=i+200){
        rect(i,j,100,100);
        rect(i+100,j+100,100,100);
    }
}

```

El modo básico permite componer una imagen, es el caso de los ejemplos previos, y del que se muestra a continuación. Es posible modificar los argumentos de las llamadas, eliminar comandos, o añadir otros, pero el resultado es siempre estático, no hay movimiento.

Listado 1.19: Modo básico

```

int Radio = 50;
void setup()
{

```

```

size(500,500);
background(0);
stroke(80);
line(230, 220, 285, 275);
fill(150,0,0);
ellipse(210, 100, Radio, Radio);
}

```

1.4. Programación en modo continuo

1.4.1. El bucle infinito de ejecución

El modo continuo, cuenta con dos métodos básicos:

- *setup()* que se ejecuta una vez cuando comience el programa
- *draw()* que por defecto se ejecuta de forma continua. Este método permite la escritura de funciones personalizadas, y hacer uso de la interacción.

El siguiente ejemplo usa ambos métodos, delimitando con llaves las instrucciones asociadas a cada uno de ellos. En concreto el método *setup* realiza la inicialización, en este caso fija las dimensiones de la ventana, siendo el método *draw* el encargado de dibujar. En este caso concreto el método *draw* pinta una línea con color y posición aleatoria. La gran diferencia del modo continuo es que las instrucciones contenidas en el método *draw* no se ejecutan una sola vez, sino que se llama a dicho método de forma reiterada cada segundo, por defecto 40, el resultado lo verás al probarlo.

Listado 1.20: Ejemplo de dibujo de líneas de colores y posición aleatoria

```

void setup()
{
  size(400, 400);
}

void draw()
{
  stroke(random(255), random(255), random(255));
  line(random(width), random(height), random(width), random(height));
}

```

Si sólo hacemos uso del método *setup*, el resultado es equivalente al modo básico, dado que dicho método se ejecuta una única vez.

Listado 1.21: Ejemplo básico

```

void setup()

```

```
{
  size (240,240);    // Dimensiones del lienzo
  background(128,128,128); // Color de lienzo en formato RGB
  noStroke();        // Sin borde para las figuras
  fill(0);           // Color de relleno de las figuras (0 es negro)
  rect(100, 100, 30, 30); // Esquina superior izquierda, ancho y alto
}
```

El siguiente ejemplo dibuja cuatro círculos en la pantalla y utiliza además una función propia llamada *circles()*. Observa el modo en que se define, y el uso de las llaves para delimitar las instrucciones contenidas en la función. En este caso concreto, el código de *draw()* sólo se ejecuta una vez, porque en *setup()* se llama a la función *noLoop()*, que cancela la repetición, resultando equivalente al modo básico.

Listado 1.22: Ejemplo de uso del método *draw*

```
void setup() {
  size(200, 200);
  noStroke();
  background(255);
  fill(0, 102, 153, 204);
  smooth();
  noLoop();
}

void draw() {
  circles(40, 80);
  circles(90, 70);
}

void circles(int x, int y) {
  ellipse(x, y, 50, 50);
  ellipse(x+20, y+20, 60, 60);
}
```

En general la ejecución reiterada tiene sentido cuando exista un cambio en aquello que dibujamos ya sea por movimiento, modificación del color, etc. Veamos un ejemplo de dibujado de líneas desde un punto fijo, con el otro extremo aleatorio, variando también el color de forma aleatoria.

Listado 1.23: Dibujo de líneas desde un punto

```
void setup() {
  size(400, 400);
  background(0);
}

void draw() {
  stroke(0, random(255), 0);
  line(50, 50, random(400), random(400));
}
```

En esta ocasión forzamos a que sean líneas aleatorias, pero verticales y blancas.

Listado 1.24: Dibujo de líneas blancas verticales

```
void setup() {
  size(400, 400);
}

void draw() {
  stroke(255);

  float dist_izq=random(400);
  line(dist_izq, 0, dist_izq, 399);
}

Variando el color

void setup() {
  size(400, 400);
}

void draw() {
  stroke(random(200,256),random(200,256),random(50,100)); // entre dos valores

  float dist_izq=random(400);
  line(dist_izq, 0, dist_izq, 399);
}
```

El color de fondo de la ventana se fija con la llamada al método *background*.

Listado 1.25: Modificando el color de fondo

```
La tasa de refresco

void setup() {
  size(400, 400);
  frameRate(4);
}

void draw() {
  background(random(255));
}
```

Nos puede ser útil si queremos forzar que se borre la pantalla antes de dibujar de nuevo.

Listado 1.26: Ejemplo de dibujo con borrado

```
void setup()
{
  size(400, 400);
}

void draw()
{
```



```
background(51); //Borra cada vez antes de pintar
stroke(random(255),random(255),random(255));
line(random(width),random(height),random(width),random(height));
}
```

El siguiente ejemplo introduce el uso del método *frameRate*, que fija el número de llamadas al método *draw* por segundo. En este caso dibuja líneas aleatorias, pero observa la tasa de refresco.

Listado 1.27: Ejemplo de dibujo de líneas a distinta frecuencia

```
void setup() {
  size(400, 400);
  frameRate(4);
}

void draw() {
  background(51);

  line(0, random(height), 90, random(height));
}
```

La tasa de refresco se controla a través de una serie de funciones, además de con *frameRate()*:

- *loop()*: Reestablece las llamadas a *draw()*, es decir el modo continuo.
- *noLoop()*: Detiene el modo continuo, no se realizan llamadas a *draw()*.
- *redraw()*: Llama a *draw()* una sólo vez.

El siguiente código varía la tasa de refresco en base al número de ejecuciones y el evento de pulsado de un botón del ratón.

Listado 1.28: Controlando la tasa de refresco

```
int frame = 0;
void setup() {
  size(100, 100);
  frameRate(30);
}
void draw()
{
  if (frame > 60)
  { // Si han pasado 60 ejecuciones (frames) desde el comienzo
    noLoop(); // para el programa
    background(0); // y lo pone todo a negro.
  }
  else
  { // En otro caso, pone el fondo
    background(204); // a un gris claro
    line(mouseX, 0, mouseX, 100); // dibuja
```

```

    line(0, mouseY, 100, mouseY);
    frame++;
  }
}
void mousePressed() {
  loop();
  frame = 0;
}

```

Un ejemplo de llamada a *redraw*.

Listado 1.29: Ejemplo con redraw

```

void setup()
{
  size(100, 100);
}
void draw() {
  background(204);
  line(mouseX, 0, mouseX, 100);
}
void mousePressed() {
  redraw(); // Llama a draw() una vez
}

```

Por defecto se establece el modo continuo con una frecuencia de 40 llamadas por segundo al método *draw()*, el siguiente código provoca el desplazamiento de un círculo por la ventana.

Listado 1.30: Desplazamiento del círculo

```

int Radio = 50;
int cuenta = 0;

void setup()
{
  size(500,500);
  background(0);
}

void draw()
{
  background(0);
  stroke(175,90);
  fill(175,40);
  println("Iteraciones: " + cuenta);
  ellipse(20+cuenta, height/2, Radio, Radio);
  cuenta++;
}

```

No olvidemos que además de dibujar, es posible escribir.

Listado 1.31: Hola mundo

```

void setup()

```

```
{
  background(128,128,128); // Color de lienzo en formato RGB
  noStroke();             // Sin borde para las figuras
  fill(0);                 // Color de relleno de las figuras (0 es negro)

  // Carga una fuente en particular
  textFont(createFont("Georgia",24));
  textAlign(CENTER, CENTER);

  // Escribe en la ventana
  text("'!Hola mundo!", 100,100);
}
```

1.4.2. Imágenes

La función *loadImage* facilita la carga de imágenes desde disco, que puede ser visualizada con *image*.

Listado 1.32: Carga de imagen

```
PImage img;

void setup(){
  size(600,400);

  img=loadImage("C:\\Users\\Modesto\\Documents\\basu.png");
}

void draw(){
  image(img,0,0);
}
```

Jugando con un desplazamiento aleatorio en la visualización de la imagen provocamos un efecto de tembleque.

Listado 1.33: Imagen que tiembla

```
PImage img;

void setup(){
  size(600,400);

  img=loadImage("C:\\Users\\Modesto\\Documents\\basu.png");
}

void draw(){
  image(img,random(10),random(10));
}
```

La función *tint* permite variar la intensidad de la visualización.

Listado 1.34: Variando la intensidad de la imagen

```

PImage img;

void setup() {
  size(600,400);

  img=loadImage("C:\\Users\\Modesto\\Documents\\basu.png");
}

void draw() {
  tint(mouseX/2);
  image(img,random(10),random(10));
}

```

Cargando varias imágenes de un ciclo, podemos almacenarlas en una lista o vector, y mostrarla de forma consecutiva, consiguiendo el efecto de animación.

Listado 1.35: Ciclo de animación

```

PImage [] img=new PImage [6];
int frame=0;

void setup() {
  size(600,400);

  img[0]=loadImage("C:\\Users\\Modesto\\Dropbox\\ExpertoVideoJuegos\\Ciclo1.png");
  img[1]=loadImage("C:\\Users\\Modesto\\Dropbox\\ExpertoVideoJuegos\\Ciclo2.png");
  img[2]=loadImage("C:\\Users\\Modesto\\Dropbox\\ExpertoVideoJuegos\\Ciclo3.png");
  img[3]=loadImage("C:\\Users\\Modesto\\Dropbox\\ExpertoVideoJuegos\\Ciclo4.png");
  img[4]=loadImage("C:\\Users\\Modesto\\Dropbox\\ExpertoVideoJuegos\\Ciclo5.png");
  img[5]=loadImage("C:\\Users\\Modesto\\Dropbox\\ExpertoVideoJuegos\\Ciclo6.png");

  frameRate(6);
}

void draw() {
  background(128);
  image(img[frame],0,0);

  frame=frame+1;
  if (frame==6){
    frame=0;
  }
}

```

1.4.3. Control

En un ejemplo previo desplazábamos un círculo por la ventana haciendo uso de una variable. Recuperamos el desplazamiento en horizontal.

Listado 1.36: Manejo de variables para dibujar circunferencias

```

int cir_x=0;

void setup() {
  size(400, 400);
  noStroke();
  fill(100,255,32); //Color de relleno
}

void draw() {
  background(127);
  ellipse(cir_x,50,50,50);

  cir_x=cir_x+1;//Modifica la coordenada x del centro de la figura
}

```

El siguiente ejemplo introduce el uso de condicionales, evitando que desaparezca la figura a la derecha de la ventana. Conociendo las dimensiones de la ventana, controlamos el valor de la posición, reiniciando el valor de la coordenada **x** del círculo cuando llega al borde derecho.

Listado 1.37: Controlando el regreso de la figura

```

int cir_x=0;

void setup() {
  size(400, 400);
  noStroke();
  fill(100,255,32);
}

void draw() {
  background(127);

  ellipse(cir_x,50,50,50);

  cir_x=cir_x+1;
  if (cir_x >= width)
  {
    cir_x=0;
  }
}

```

Hacerlo similar pero en vertical. Debemos tener en cuenta la coordenada **y** del objeto.

El mismo ejemplo pero con dos círculos a distintas velocidades.

Listado 1.38: Con dos círculos

```

float cir_rap_x = 0;
float cir_len_x = 0;

void setup() {
  size(400,400);
  noStroke();
}

```

```
}  
  
void draw() {  
  background(27,177,245);  
  
  fill(193,255,62);  
  ellipse(cir_len_x,50, 50, 50);  
  cir_len_x = cir_len_x + 1;  
  
  fill(255,72,0);  
  ellipse(cir_rap_x,50, 50, 50);  
  cir_rap_x = cir_rap_x + 5;  
  
  if(cir_len_x > 400) {  
    cir_len_x = 0;  
  }  
  if(cir_rap_x > 400) {  
    cir_rap_x = 0;  
  }  
}
```

Alterando la forma del círculo lento de forma aleatoria cuando se dan ciertas circunstancias.

Listado 1.39: Simulando un latido

```
float cir_rap_x = 0;  
float cir_len_x = 0;  
  
void setup() {  
  size(400,400);  
  noStroke();  
}  
  
void draw() {  
  background(27,177,245);  
  
  float cir_len_tam = 50;  
  
  if(random(10) > 9) {  
    cir_len_tam = 60;  
  }  
  
  fill(193,255,62);  
  ellipse(cir_len_x,50, cir_len_tam, cir_len_tam);  
  cir_len_x = cir_len_x + 1;  
  
  fill(255,72,0);  
  ellipse(cir_rap_x,50, 50, 50);  
  cir_rap_x = cir_rap_x + 5;  
  
  if(cir_len_x > 400) {  
    cir_len_x = 0;  
  }  
  if(cir_rap_x > 400) {  
    cir_rap_x = 0;  
  }  
}
```

Para no dar el efecto de pausa, modificamos el ciclo con una aparición más ajustada teniendo en cuenta el radio.

Listado 1.40: Aparición ajustada

```
int cir_x=0;

void setup() {
  size(400, 400);

  noStroke();

  fill(100,255,32);
}

void draw() {
  background(127);

  ellipse(cir_x,50,50,50);

  cir_x=cir_x+1;

  if (cir_x-50/2>width)
  {
    cir_x=25;
  }

  if (cir_x+50/2>=width)
  {
    ellipse(cir_x-width,50,50,50);
  }
}
```

La modificación de la coordenada *x* es siempre un incremento, podemos jugar a que haya un rebote al llegar al extremo, modificando el signo del movimiento, para que pueda ser incremento o decremento. De esta forma se consigue que el círculo rebote en los lados derecho e izquierdo.

Listado 1.41: Provocando el rebote

```
int pos=0;
int mov=1;

void setup(){
  size(400,400);
}

void draw(){
  background(128);
  ellipse(pos,50,30,30);
  pos=pos+mov;

  if (pos>=400 || pos<=0)
  {
```

```

    mov=-mov;
  }
}

```

Integramos un *jugador*, que se desplaza en vertical acompañando al ratón, y también puede alterar el movimiento del círculo cuando haya choque.

Listado 1.42: Frontón

```

int posX=0;
int posY=50;
int D=30;
int ancho=20;
int alto=50;
int mov=5;

void setup(){
  size(400,400);
}

void draw(){
  background(128);
  ellipse(posX,posY,D,D);

  //Donde se encuentra el jugador
  int jugx=width-50;
  int jugy=mouseY-30;
  rect(jugx,jugy,ancho,alto);

  posX=posX+mov;
  //verificando si hay choque
  if (posX>=400 || posX<=0 || (mov>0 && jugy<=posY+D/2 && posY-D/2<=jugy+alto &&
    jugx<=posX+D/2 && posX-D/2<=jugx+ancho))
  {
    mov=-mov;
  }
}

```

Añadimos un marcador contabilizando el número de veces que no controlamos la *pelota*. En el código introducimos una variable contador, inicialmente a cero, que modifica su valor cada vez que haya un choque con la pared de la derecha.

Listado 1.43: Integrando el marcador

```

int posX=0;
int posY=50;
int D=30;
int ancho=20;
int alto=50;
int mov=5;
int goles=0;

void setup(){
  size(400,400);
}

void draw(){

```



```

background(128);
ellipse(posX,posY,D,D);

//Donde se encuentra el jugador
int jugx=width-50;
int jugy=mouseY-30;
rect(jugx,jugy,ancho,alto);

posX=posX+mov;
//verificando si hay choque
if (posX>=400 || posX<=0 || (mov>0 && jugy<=posY+D/2 && posY-D/2<=jugy+alto &&
    jugx<=posX+D/2 && posX-D/2<=jugx+ancho))
{
    mov=-mov;
    //Si choca con la derecha, es gol
    if (posX>=400)
    {
        goles=goles+1;
    }
}
text("Goles "+goles, width/2-30, 20);
}

```

Sería también posible mostrar un mensaje cada vez que haya un gol, es decir, cuando se toque la pared derecha. Mostrarlo únicamente al modificar el tanteo, hará que casi no se vea, es por ello que lo hacemos creando un contador que se decrementa cada vez que se muestra el mensaje.

Listado 1.44: Celebrando el gol

```

int posX=0;
int posY=50;
int D=30;
int ancho=20;
int alto=50;
int mov=5;
int goles=0;
int muestragol=0;

void setup(){
    size(400,400);
}

void draw(){
    background(128);
    ellipse(posX,posY,D,D);

    //Donde se encuentra el jugador
    int jugx=width-50;
    int jugy=mouseY-30;
    rect(jugx,jugy,ancho,alto);

    posX=posX+mov;
    //verificando si hay choque
    if (posX>=400 || posX<=0 || (mov>0 && jugy<=posY+D/2 && posY-D/2<=jugy+alto &&
        jugx<=posX+D/2 && posX-D/2<=jugx+ancho))
    {
        mov=-mov;

```

```

    // Si choca con la derecha, es gol
    if (posX >= 400)
    {
        goles = goles + 1;
        muestragol = 40;
    }
}
text("Goles " + goles, width/2 - 30, 20);
if (muestragol > 0)
{
    text("GOOOOL", width/2 - 30, height - 50);
    muestragol = muestragol - 1;
}
}

```

Introducir el movimiento en los dos ejes requiere modificar la coordenada y de la pelota. El siguiente ejemplo lo integra, manteniendo el rebote en las paredes, ahora también inferior y superior.

Listado 1.45: Rebote de la bola

```

float cir_x = 300;
float cir_y = 20;
// desplazamiento
float mov_x = 2;
float mov_y = -2;

void setup() {
    size(400, 200);
    stroke(214, 13, 255);
    strokeWeight(7);
}

void draw() {
    background(33, 234, 115);
    ellipse(cir_x, cir_y, 40, 40);
    cir_x = cir_x + mov_x;
    cir_y = cir_y + mov_y;

    if(cir_x > width) {
        cir_x = width;
        mov_x = -mov_x;
        println("derecha");
    }
    if(cir_y > height) {
        cir_y = height;
        mov_y = -mov_y;
        println("abajo");
    }
    if(cir_x < 0) {
        cir_x = 0;
        mov_x = -mov_x;
        println("izquierda");
    }
    if(cir_y < 0) {
        cir_y = 0;
        mov_y = -mov_y;
        println("arriba");
    }
}
}

```

Crear un juego Pong, ver figura 1.6.

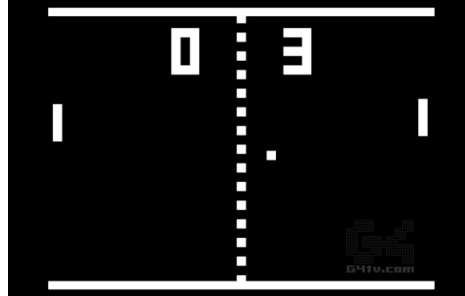


Figura 1.6: Interfaz clásica de Pong.

1.4.4. Interacción

En el modo continuo será frecuente la interacción con los usuario vía teclado o ratón.

1.4.4.1. Teclado

Las acciones de teclado puede utilizarse como evento en sí o para recoger los detalles de la tecla pulsada. Por ejemplo la variable booleana *keyPressed* se activa si una tecla es presionada, en cualquier otro caso es falsa. Además, dicha variable estará activa mientras mantengamos la tecla permanezca.

El siguiente código hace uso de dicha variable para desplazar una línea.

Listado 1.46: Ejemplo básico

```
int x = 20;
void setup() {
  size(100, 100);
  smooth();
  strokeWeight(4);
}
void draw() {
  background(204);
  if (keyPressed == true)
  {
    x++;    // incrementamos x
  }
  line(x, 20, x-60, 80);
}
```

Intenta modificarlo para dibujar dos líneas diagonales que se desplazan.

También es posible recuperar la tecla pulsada. La variable *key* es de tipo *char* y almacena el valor de la tecla que ha sido presionada recientemente.

Este ejemplo muestra la tecla pulsada.

Listado 1.47: Muestra la tecla

```
void setup() {  
  size(100, 100);  
}  
  
void draw() {  
  background(0);  
  text(key, 28, 75);  
}
```

Por supuesto, cada letra tiene un valor numérico en la Tabla ASCII, que también podemos mostrar

Listado 1.48: El código ASCII de la tecla

```
void setup() {  
  size(100, 100);  
}  
  
void draw() {  
  background(200);  
  if (keyPressed == true)  
  {  
    int x = key;  
    text(key+" ASCII "+x, 20, 20 );  
  }  
}
```

Una particularidad interesante es poder identificar las teclas especiales como por ejemplo: flechas, *Shift*, *Backspace*, tabulador y otras más. Para ello, lo primero que debemos hacer es comprobar si se trata de una de estas teclas comprobando el valor de la variable `key == CODED`. El siguiente código utilizando las teclas de las flechas cambie la posición de una figura dentro del lienzo.

Listado 1.49: Ejemplo básico

```
color y = 35;  
void setup() {  
  size(100, 100);  
}  
void draw() {  
  background(204);  
  line(10, 50, 90, 50);  
  if (key == CODED)  
  {  
    if (keyCode == UP)  
    {  
      y = 20;  
    }  
    else  
      if (keyCode == DOWN)
```

```
    {  
      y = 50;  
    }  
  }  
  else  
  {  
    y = 35;  
  }  
  rect(25, y, 50, 30);  
}
```

Existe la posibilidad de detectar eventos individualizados, es decir el pulsado de una tecla en concreto. El siguiente ejemplo realiza una acción al pulsar la tecla t.

Listado 1.50: Ejemplo básico evento de teclado

```
boolean drawT = false;  
  
void setup() {  
  size(100, 100);  
  noStroke();  
}  
  
void draw() {  
  background(204);  
  if (drawT == true)  
  {  
    rect(20, 20, 60, 20);  
    rect(39, 40, 22, 45);  
  }  
}  
void keyPressed()  
{  
  if ((key == 'T') || (key == 't'))  
  {  
    drawT = true;  
  }  
}  
  
void keyReleased() {  
  drawT = false;  
}
```

Crea un ejemplo que modifique el color del fondo pulsando cada vocal.

1.4.4.2. Ratón

En ejemplos previos hemos visto que es posible acceder a las coordenadas del ratón, simplemente haciendo uso de las variables adecuadas desde el método *draw()* como en el código siguiente.

Listado 1.51: Ejemplo de uso de las coordenadas del ratón

```
void setup() {  
  size(200, 200);  
  rectMode(CENTER);  
  noStroke();  
  fill(0, 102, 153, 204);  
}  
  
void draw() {  
  background(255);  
  rect(width-mouseX, height-mouseY, 50, 50);  
  rect(mouseX, mouseY, 50, 50);  
}
```

También es posible usar el evento, es decir, la función que asociada con el mismo, como en este ejemplo que cambia el color del fondo.

Listado 1.52: Ejemplo de cambio del tono de fondo

```
float gray = 0;  
void setup() {  
  size(100, 100);  
}  
void draw() {  
  background(gray);  
}  
void mousePressed() {  
  gray += 20;  
}
```

El siguiente ejemplo usa el evento de pulsado para pasar del modo básico al continuo.

Listado 1.53: Paso a modo continuo

```
void setup()  
{  
  size(200, 200);  
  stroke(255);  
  noLoop();  
}  
  
float y = 100;  
void draw()  
{  
  background(0);  
  line(0, y, width, y);  
  line(0, y+1, width, y+1);  
  y = y - 1;  
  if (y < 0) { y = height; }  
}  
  
void mousePressed()  
{  
  loop();  
}
```

Como puedes ver *mousePressed()* es una función enlazada a la acción de pulsar un botón del ratón. Siendo posible controlar también cuando se suelta, se mueve, o si se arrastra el ratón con un botón pulsado.

Listado 1.54: Ejemplo evento de arrastre

```
int dragX, dragY, moveX, moveY;
void setup() {
  size(100, 100);
  smooth();
  noStroke();
}
void draw() {
  background(204);
  fill(0);
  ellipse(dragX, dragY, 33, 33); // Círculo negro
  fill(153);
  ellipse(moveX, moveY, 33, 33); // Círculo gris
}
void mouseMoved() { // Mueve el gris
  moveX = mouseX;
  moveY = mouseY;
}
void mouseDragged() { // Mueve el negro
  dragX = mouseX;
  dragY = mouseY;
}
```

Mueve un objeto reflejando el movimiento del ratón.

1.4.5. Creando un Paint

Como ya sabemos, las coordenadas de ratón se almacenan en las variables *mouseX* y *mouseY*.

Listado 1.55: Coordenadas del ratón

```
void setup() {
  size(640, 360);
  noStroke();
}

void draw() {
  background(51);
  ellipse(mouseX, mouseY, 66, 66);
}
```

Ambas son variables conteniendo las posiciones actuales del ratón, las previas están disponibles en *pmouseX* y *pmouseY*. Podemos utilizar las coordenadas del ratón para pintar a mano alzada.

Listado 1.56: Pintado con el «pincel»

```
void setup() {  
    size(400,400);  
  
    background(128);  
}  
  
void draw() {  
    point(mouseX, mouseY);  
}
```

Es posible restringir el pintado a sólo cuando se pulse un botón del ratón, empleando una estructura condicional.

Listado 1.57: Pintado cuando se pulsa

```
void setup() {  
    size(400,400);  
  
    background(128);  
}  
  
void draw() {  
  
    if (mousePressed == true) {  
        point(mouseX, mouseY);  
    }  
}
```

Utilizando el teclado, en concreto las teclas del cursor arriba y abajo, para cambiar grosor del pincel (y pintamos círculo), y cualquier otra tecla para alterar el color. Se identifica primero si se ha pulsado una tecla, y luego la tecla en concreto pulsada.

Listado 1.58: Grosor y color del pincel con las teclas

```
int grosor=1;  
int R=0,G=0,B=0;  
  
void setup() {  
    size(400,400);  
    background(128);  
}  
  
void draw() {  
  
    if (mousePressed == true) {  
        point(mouseX, mouseY);  
    }  
  
    if (keyPressed == true) {  
        if (keyCode == UP) {  
            grosor = grosor+1;  
            strokeWeight(grosor);  
        }  
        else
```



```

    {
      if (keyCode == DOWN) {
        if (grosor > 1){
          grosor = grosor - 1;
          strokeWeight(grosor);
        }
      }
      else
      {
        R=(int)random(255);
        G=(int)random(255);
        B=(int)random(255);
      }
    }
  }

  //Muestra del pincel
  noStroke();
  fill(128);
  rect(4,4,grosor+2,grosor+2);
  fill(R,G,B);
  ellipse(5+grosor/2,5+grosor/2,grosor,grosor);

  stroke(R,G,B);
}

```

Que el radio del pincel dependa del valor de x o y .

Listado 1.59: Radio del pincel dependiente de ratón

```

void setup() {
  size(640, 360);
  noStroke();
  rectMode(CENTER);
}

void draw() {
  background(51);
  fill(255, 204);
  rect(mouseX, height/2, mouseY/2+10, mouseY/2+10);
  fill(255, 204);
  int inversaX = width-mouseX;
  int inversaY = height-mouseY;
  rect(inversaX, height/2, (inversaY/2)+10, (inversaY/2)+10);
}

```

Sugerir ejemplos dentro de *Archivo* → *Ejemplos*, seleccionando por ejemplo *Topics* → *Motion* → *Bounce*, *Basics* → *Input* → *Storinginput*, etc.

1.5. Avanzado

Las contribuciones o bibliotecas se pueden añadir a través de *Herramientas* → *Añadir herramientas* → *Libraries* Por ejemplo añadir *Video* y *Sound* para los ejemplos de dichas características.

Listado 1.60: Captura de cámara

```
import processing.video.*;

Capture video;

void setup(){
    size(320,240);

    video = new Capture(this,320,240);

    // Lanza la captura
    video.start();

    background(0);
}

void draw(){
    if (video.available()){
        video.read();
    }
    image(video,0,0);
}
```

Listado 1.61: Captura de cámara mostrando la mitad superior umbralizada

```
import processing.video.*;

Capture video;

void setup(){
    size(320,240);

    video = new Capture(this,320,240);

    // Lanza la captura
    video.start();

    background(0);
}

void draw(){
    if (video.available())
    {
        video.read();

        //Tamaño de la imagen
        int dimension = video.width * video.height;

        //Carga p'ixeles
        video.loadPixels();
        //Umbraliza la parte superior de la imagen
        for (int i=1;i<dimension/2;i++)
        {
            float suma=red(video.pixels[i])+green(video.pixels[i])+red(video.pixels[i]);

            //umbraliza al valor intermedio
            if (suma<255*1.5)
            {
```

```

        video.pixels[i]=color(0, 0, 0);
    }
    else
    {
        video.pixels[i]=color(255, 255, 255);
    }
}
// Actualiza p\ixeles
video.updatePixels();
}
image( video ,0 ,0) ;
}

```

Reproduciendo un sonido, puedes encontrar wavs [aquí](#).

Listado 1.62: Reproduciendo sonido

```

import processing.sound.*;

int pos=0;
int mov=5;

SoundFile sonido;

void setup(){
    size(400,400);

    sonido = new SoundFile(this,"E:/Intro Programacion con Processing Experto/Bass
    -Drum-1.wav");
}

void draw(){
    background(128);
    ellipse(pos,30,30,30);

    pos=pos+mov;

    if (pos>=400 || pos<=0){
        mov=-mov;
        sonido.play ( ) ;
    }
}

```

Solventar la latencia del sonido con un hilo y el método *thread*.

Listado 1.63: Latencia del sonido

```

import processing.sound.*;

int pos=0;
int mov=5;

SoundFile sonido;

void setup(){
    size(400,400);

    sonido = new SoundFile(this,"E:/Intro Programacion con Processing Experto/Bass
    -Drum-1.wav");
}

```

```
}  
  
void draw(){  
    background(128);  
    ellipse(pos,30,30,30);  
  
    pos=pos+mov;  
  
    if (pos>=400 || pos<=0){  
        mov=-mov;  
        thread ("Suenar");  
    }  
}  
  
void Suenar( ) {  
    sonido.play ( ) ;  
}
```

1.6. Tipos de datos

Processing está basado en Java, por lo que puedes utilizar cualquier característica de dicho lenguaje en Processing. Recordamos algunos aspectos básicos.

En Processing las variables se deben declarar explícitamente y asignar las variables antes de llamar o de realizar una operación sobre ellas. Para declarar una variable utilice una sintaxis similar a los ejemplos a continuación:

Listado 1.64: Ejemplos de tipos de variables

```
// Cadenas  
String myName = "supermanoeuvre";  
String mySentence = " was born in ";  
String myBirthYear = "2006";  
  
// Concatenar  
String NewSentence = myName + mySentence + myBirthYear;  
System.out.println(NewSentence);  
  
// Enteros  
int myInteger = 1;  
int myNumber = 50000;  
int myAge = -48;  
  
// Reales  
float myFloat = 9.5435;  
float timeCount = 343.2;  
  
// Booleanos // True o False  
boolean mySwitch = true;  
boolean mySwitch2 = false;
```

Ejemplos de accesos a vectores.

Listado 1.65: Uso de vectores

```
%\begin{lstlisting}[style=C++]
// Lista de Cadenas
String [] myShoppingList = new String [3];
myShoppingList[0] = "bananas";
myShoppingList[1] = "coffee";
myShoppingList[2] = "tuxedo";

// Lista de enteros
int [] myNumbers = new int [4];
myNumbers[0] = 498;
myNumbers[1] = 23;
myNumbers[2] = 467;
myNumbers[3] = 324;

// printamos un dato de la lista
println( myNumbers[2] );

int a = myNumbers[0] + myNumbers[3];
println( a );
```

Processing incluye la clase *ArrayList* de Java, que no requiere conocer su tamaño desde el inicio. De esta forma se facilita añadir objetos a la lista, ya que el tamaño de la lista aumenta o decrece de forma automática.

Listado 1.66: Uso del tipo ArrayList

```
ArrayList lista = new ArrayList();
int i = 0;

while (i<4){
    lista.add(i+3);
    i=i+1;
}

println("\nLos datos son: \n");
Iterator iter = lista.iterator();
while(iter.hasNext()){
    println(iter.next());
}

ArrayList myVectorList ;
myVectorList = new ArrayList();

// Asignamos objetos
myVectorList.add( new PVector(51,25,84) );
myVectorList.add( new PVector(98,3,54) );

// o //
PVector myDirection = new PVector(98,3,54);
myVectorList.add( myDirection );

// Bucle para acceder a objetos usando ArrayList.size() y ArrayList.get()
for(int i = 0; i < myVectorList.size(); i++){
    PVector V = (PVector) myVectorList.get(i); // ojo con el cast (PVector)
    println(V);
}
```

1.7. Applets

Una vez finalizado el proceso de prueba de un boceto, es posible exportarlo generando un ejecutable. Para ello mediante la opción *File* → *Export Application*, habiendo escogido el modo Javascript en el botón de la parte superior derecha, se genera el correspondiente applet con su fichero html para su carga con un navegador. Puedes probar con cualquiera de los ejemplos.

1.8. Utilidades gráficas

Processing incorpora funciones que hacen más fácil el manejo de objetos gráficos, como por ejemplo las transformaciones: traslación, rotación y escalado. Recuerda sin embargo que estas utilidades no puedes emplearlas para obtener los objetivos de la práctica, pero siempre podrás conocerlas para elementos de la interfaz o un futuro.

Para ilustrar la traslación, en las imágenes mostramos un cuadrado antes y después de trasladarlo, como se ve en la figura 1.7.

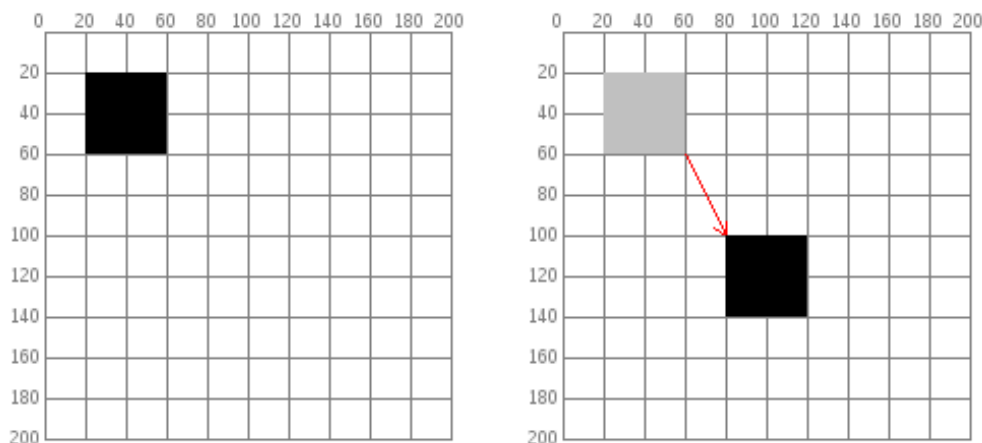


Figura 1.7: Cuadrado antes y tras trasladar.

Si desea mover el cuadrado 60 unidades a la derecha y 80 unidades hacia abajo, puedes cambiar las coordenadas mediante la suma a los puntos iniciales: $rect(20 + 60, 20 + 80, 40, 40)$ y el cuadrado aparecerá en un lugar diferente.

La traslación puede realizarse desplazando el cuadrado un número determinado de coordenadas, pero también obtenemos un efecto similar si movemos el sistema de coordenadas.

Lo importante a notar en el diagrama anterior es que, el rectángulo no se ha movido de su posición, su esquina superior izquierda se encuentra todavía

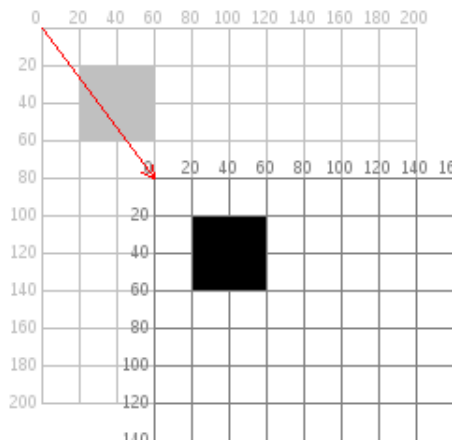


Figura 1.8: Traslación del sistema de referencia

en (20,20), por el contrario, el sistema de coordenadas se modifica. Incluimos el código que realiza el movimiento del cuadrado de las dos formas.

Listado 1.67: Traslación del sistema de referencia

```
void setup()
{
  size(200, 200);
  background(255);
  noStroke();
  // Dibuja el primer objeto
  fill(192);
  rect(20, 20, 40, 40);
  // Dibuja otro desplazado
  fill(255, 0, 0, 128);
  rect(20 + 60, 20 + 80, 40, 40);

  // Este nuevo objeto tiene las mismas coordenadas que el primero pero antes
  // hemos movido el sistema de referencia
  fill(0, 0, 255, 128);
  pushMatrix(); // Salva el sistema de coordenadas actual
  translate(60, 80);
  rect(20, 20, 40, 40);
  popMatrix(); // vuelve al sistema de coordenadas original
}
```

Modificar el sistema de coordenadas puede ser engorroso para un ejemplo tan sencillo como el cuadrado. Pero tomemos un ejemplo donde *translate()* puede hacer la vida algo más fácil. El siguiente código dibuja una hilera de casas usando los dos esquemas. Se utiliza un bucle que llama a la función llamada *house()*, que tiene la ubicación X e Y de la esquina superior izquierda de la casa como sus parámetros.

Listado 1.68: Dibujo de varias casas

```

void setup()
{
  size(400, 400);
  background(255);
  for (int i = 10; i < 350; i = i + 50)
  {
    house(i, 20);
  }
}

void house(int x, int y)
{
  triangle(x + 15, y, x, y + 15, x + 30, y + 15);
  rect(x, y + 15, 30, 30);
  rect(x + 12, y + 30, 10, 15);
}

*****

void setup()
{
  size(400, 400);
  background(255);
  for (int i = 10; i < 350; i = i + 50)
  {
    house(i, 20);
  }
}

void house(int x, int y)
{
  pushMatrix();
  translate(x, y);
  triangle(15, 0, 0, 15, 30, 15);
  rect(0, 15, 30, 30);
  rect(12, 30, 10, 15);
  popMatrix();
}

```

También se pueden aplicar otras transformaciones como la rotación con la función *rotate()*. Esta función tiene un argumento, que es el número de radianes que desea girar. En Processing, todas las funciones que tienen que ver con medir ángulos de rotación en radianes en lugar de grados. Cuando se habla de ángulos en grados, se dice que un círculo completo son 360° . Cuando se habla de los ángulos en radianes, se dice que un círculo completo tiene 2π radianes. Aquí hay un diagrama de cómo Processing mide los ángulos en grados (negro) y radianes (rojo).

Como la mayoría de la gente piensa en grados, Processing tiene una función integrada *radians()* que toma un número de grados como su argumento y lo convierte para ti. También tiene la función de *degrees()* que convierte radianes a grados. Teniendo en cuenta estos antecedentes, vamos a intentar girar un cuadrado de 45 grados en sentido horario.

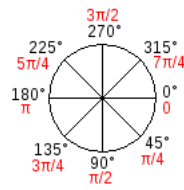


Figura 1.9: Ángulos de rotación en grados y radianes

Listado 1.69: Rotación

```
void setup()
{
  size(200, 200);
  background(255);
  smooth();
  fill(192);
  noStroke();
  rect(40, 40, 40, 40);

  pushMatrix();
  rotate(radians(45));
  fill(0);
  rect(40, 40, 40, 40);
  popMatrix();
}
```

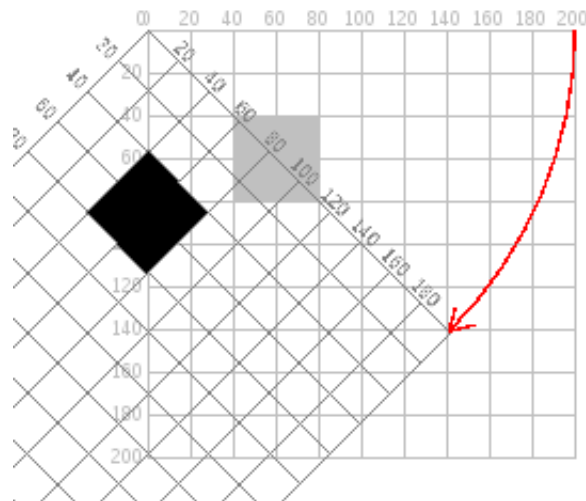


Figura 1.10: Rotación del cuadrado

La forma correcta de hacer girar el cuadrado es:

1. Trasladar el origen del sistema de coordenadas de (0, 0) a donde desea que la parte superior izquierda que desee del cuadrado.
2. Gire la red de $\pi/4$ radianes (45°)
3. Dibuja el cuadrado en el origen.

Listado 1.70: Transformación

```
void setup()
{
  size(200, 200);
  background(255);
  smooth();
  fill(192);
  noStroke();
  rect(40, 40, 40, 40);

  pushMatrix();
  // mueve el origen al punto pivote
  translate(40, 40);

  // rota sobre ese punto pivote
  rotate(radians(45));

  // y dibuja el cuadrado
  fill(0);
  rect(0, 0, 40, 40);
  popMatrix(); // luego reestablece el grid
}
```

Por último el escalado como en el siguiente ejemplo modifica el aspecto del cuadrado.

Listado 1.71: Escalado

```
void setup()
{
  size(200,200);
  background(255);

  stroke(128);
  rect(20, 20, 40, 40);

  stroke(0);
  pushMatrix();
  scale(2.0);
  rect(20, 20, 40, 40);
  popMatrix();
}
```

En primer lugar, parece que el cuadrado se ha movido, pero nada de eso. Su esquina superior izquierda se encuentra todavía en la posición (20, 20). También se puede ver que las líneas son más gruesas. Eso no es una ilusión óptica, las

líneas son en realidad dos veces más gruesas, porque el sistema de coordenadas se ha reducido al doble de su tamaño.

Al hacer múltiples transformaciones, el orden es importante. Una rotación seguida de una traslación seguido de una escala que no dará los mismos resultados que una trasladar seguido de una rotación por una escala.

Listado 1.72: Encadenando transformaciones

```
void setup()
{
  size(200, 200);
  background(255);
  smooth();
  line(0, 0, 200, 0); // dibuja bordes de la imagen
  line(0, 0, 0, 200);

  pushMatrix();
  fill(255, 0, 0); // cuadrado rojo
  rotate(radians(30));
  translate(70, 70);
  scale(2.0);
  rect(0, 0, 20, 20);
  popMatrix();

  pushMatrix();
  fill(255); // cuadrado blanco
  translate(70, 70);
  rotate(radians(30));
  scale(2.0);
  rect(0, 0, 20, 20);
  popMatrix();
}
```

Cada vez que haces una rotación, traslación, o escalado, la información necesaria para realizar la transformación se va acumulando en una matriz. Esta matriz contiene toda la información necesaria para hacer cualquier serie de transformaciones. Y esa es la razón por la que se usan las funciones *pushMatrix()* y *popMatrix()*.

Las funciones nos permiten manejar una pila de sistemas de referencia, *pushMatrix()* pone el estado actual del sistema de coordenadas en la parte superior de un área de memoria, y *popMatrix()* que extrae el estado de nuevo hacia fuera. El ejemplo anterior utiliza *pushMatrix()* y *popMatrix()* para asegurarse de que el sistema de coordenadas estaba *limpio* antes de cada parte del dibujo. En todos los otros ejemplos, las llamadas a esas dos funciones no eran realmente necesarias, pero no hace daño salvar y restaurar el estado del grid.

En Processing, el sistema de coordenadas se restaura a su estado original (de origen en la parte superior izquierda de la ventana, sin rotación ni escalado) cada vez que la función *draw()* se ejecuta.

Para trabajar en tres dimensiones basta con pasar tres argumentos a las funciones de transformación. Para las rotaciones haremos uso de las funciones *rotateX()*,

rotateY(), o *rotateZ()*.

1.9. Referencias y fuentes

Algunas fuentes de información:

- [Processing](#) y [Processing reference](#)
- [Processing exhibition archives](#)
- [Programming Interactivity](#), Joshua Noble, 2nd edition, O'Reilly, 2012. Libro que cubre tanto Processing como otros entornos openFrameworks como Arduino. [Enlace desde la red de la universidad.](#)
- [An Introduction To Programming With Processing](#)
- [Fun programming](#)
- [openProcessing share your sketches](#)
- [OpenProcessing games](#)
- [Creative Applications Net](#)