Processing

Oriol Manau Galtés

PID_00216131

UOC
Universitat Oberta de Catalunya

Índice

Introducción

En esta actividad iniciaremos el trabajo con el lenguaje de programación Processing, un lenguaje que en los últimos años ha ido ganando adeptos sobre todo en el ámbito de las artes gráficas gracias a su simplicidad y al hecho de ser una alternativa real a los proyectos hechos, por ejemplo, con el Action Script.

Processing

Como hemos dicho, Processing es un lenguaje de programación orientado al trabajo con imágenes, animaciones y sonidos, pero también es un entorno de desarrollo que nos ofrece un conjunto completo de herramientas para poder llevar a cabo nuestros proyectos de una manera fácil y cómoda.

El lenguaje está basado en Java, aprovecha su estructura y su potencia pero lo simplifica para que sea muy sencillo trabajar y evita la necesidad de tener un conocimiento profundo del lenguaje para poder obtener buenos resultados. En el supuesto de que tengamos un conocimiento más profundo de Java, lo podremos aprovechar para sacar más provecho aún del lenguaje y aprovechar sus ventajas, a pesar de que no es necesario para poder obtener buenos resultados.

Al estar diseñado para ser sencillo, nos permite utilizarlo para introducirnos en la programación de una manera progresiva y obtener resultados muy rápidamente. Es por eso por lo que esta actividad seguirá una estructura de enseñanza basada en ejemplos que se irán completando poco a poco, añadiendo nuevos elementos y formas de organización del código para poder llegar al final con una buena base y poder afrontar cualquier proyecto.

Documentación oficial y otras referencias

1) Documentación oficial

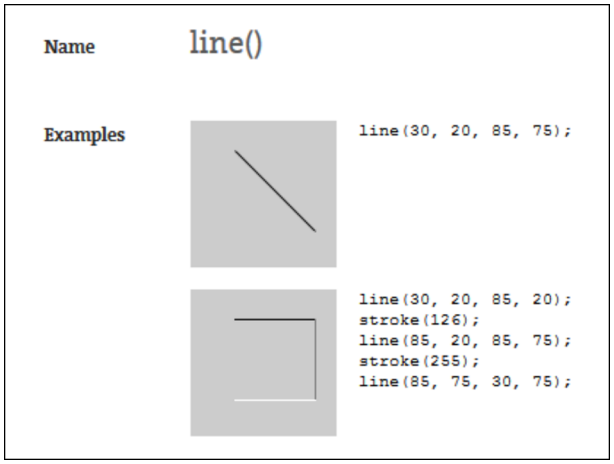
El primer lugar donde tenemos que buscar la información es en la web oficial, donde encontraréis la información más actualizada y una gran cantidad de referencias a ejemplos y otros lugares en los que podéis encontrar información extra.

Nota

Processing.org/

Uno de los apartados de la web más útiles es el de la referencia del lenguaje, puesto que aquí podremos obtener toda la información de cómo se debe trabajar con las diferentes opciones que nos ofrece el lenguaje. Por ejemplo, en la figura 1 podemos ver una parte de la referencia para la función *line* que nos permite dibujar líneas.

Figura 1. Referencia en línea



2) Documentación en Mosaic

Nota

Mosaic.uoc.edu

Aparte de la documentación oficial podéis encontrar mucha información en la revista Mosaic, donde cada vez hay más artículos de Processing excelentes para ver sus características concretas o cómo se han solucionado problemas diversos utilizando este lenguaje, por lo cual es muy recomendable echar un vistazo tanto para ver su funcionamiento como para obtener ideas de lo que podemos llegar a hacer.

A continuación tenéis una lista de algunos de los artículos disponibles que podéis consultar, pero hay muchos más:

- http://mosaic.uoc.edu/2012/04/30/introduccion-a-Processing
- http://mosaic.uoc.edu/2013/02/19/cursor-animado-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/02/27/caleidoscopio-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/02/27/sistema-de-particulas-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/02/27/colisiones-y-particulas-con-lenguaje-de-programacion-Processing
- http://mosaic.uoc.edu/2013/02/27/animacion-en-base-a-transformaciones-homogeneas-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/02/27/Processing-practico/

- http://mosaic.uoc.edu/2013/03/26/Processing-practico-parte-2-de-3/
- http://mosaic.uoc.edu/2013/04/24/Processing-practico-parte-3-de-3/
- http://mosaic.uoc.edu/2013/03/25/carga-de-un-modelo-3d-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/03/25/imagenes-y-luts-en-programacion-Processing/
- http://mosaic.uoc.edu/2013/03/25/efecto-dinamico-de-pointillism-en-imagenes-via-programacion-Processing/
- http://mosaic.uoc.edu/2013/03/25/sonido-con-lenguaje-de-programacion-Processing-primer-ejemplo/
- http://mosaic.uoc.edu/2013/03/25/sonido-con-lenguaje-de-programacion-Processing-segundo-ejemplo/
- http://mosaic.uoc.edu/2013/04/24/realidad-aumentada-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/04/24/deteccion-de-colisiones-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/04/24/acceso-a-webcam-y-luts-en-programacion-Processing/
- http://mosaic.uoc.edu/2013/04/24/acceso-a-webcam-y-procesado-en-programacion-Processing/
- http://mosaic.uoc.edu/2013/04/24/operaciones-con-imagenes-en-programacion-Processing/

3) Otras fuentes de información

En este caso nadie tendrá ninguna sorpresa si tomamos como tercera fuente de información Internet, donde podremos encontrar una gran cantidad de información sobre el lenguaje, como por ejemplo, el vídeo introductorio que podemos ver en este enlace, o libros disponibles en línea, como *Processing, un lenguaje al alcance de todos*.

Instalación

Lo primero que habrá que hacer será descargarnos el programa de la web oficial para poderlo instalar en nuestro ordenador. Para hacerlo, tenemos que seguir estos pasos:

Nota

Processing.org/

- Vamos al apartado de descargas, donde podemos decidir hacer una donación para apoyar el desarrollo de este proyecto de código abierto.

Figura 2. Página inicial de descarga

- Seleccionamos *download* y accederemos a la página donde podremos seleccionar el sistema operativo de nuestro ordenador y la versión de Processing que queremos descargar.

Figura 3. Página de descarga, selección del sistema operativo

- Una vez finalizada la descarga, solo habrá que descomprimir el fichero y aparecerá el directorio donde está el programa.

- Para ejecutarlo, solo habrá que entrar dentro del directorio creado y ejecutar el fichero *processing.exe* en el caso de utilizar el Windows o el equivalente de cada sistema operativo, y ya nos aparecerá el entorno de desarrollo que utilizaremos para hacer las actividades.

Una vez se haya conseguido ejecutar la aplicación, ya estamos preparados para elaborar nuestro primer programa.

1. Estructura básica de un programa

En este apartado analizaremos la estructura y los elementos básicos que tiene que haber en todos los programas para que puedan funcionar correctamente.

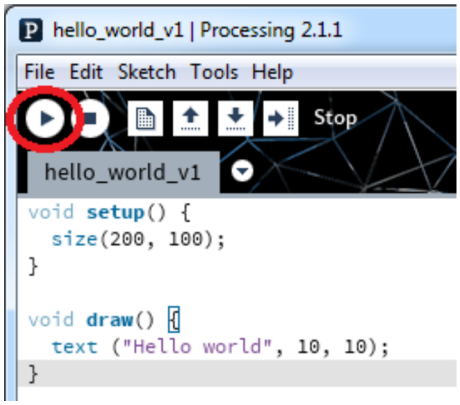
1.1. "Hola mundo"

Abrid el editor del Processing y escribid el código siguiente:

```
void setup() {
   size(200, 100);
}

void draw() {
   text ("Hello world", 10, 10);
}
```

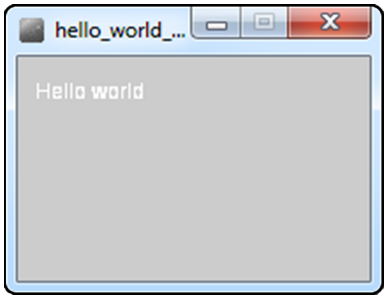
Una vez escrito, ejecutad el código pulsando el botón *play* de la interfaz.

Figura 4. Ejecución de un programa



Al hacerlo se abrirá una ventana nueva, donde podremos leer el texto "Hello world".

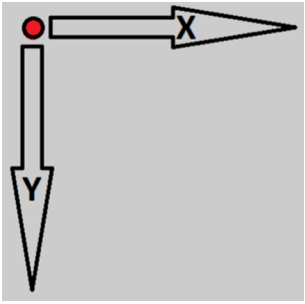
Figura 5. Ventana del programa



Veamos los elementos que forman el programa:

1) La función *setup* nos sirve para inicializar los diferentes parámetros que utilizaremos en el código y siempre es la primera función que se ejecuta. Dentro de esta función, la primera línea siempre es una llamada a la función *size(x, y)*, que es la función que definirá la medida de la ventana gráfica que aparecerá al ejecutar el código. Los parámetros que recibe la función indican la medida horizontal y vertical, respectivamente, y hay que tener en cuenta que la posición (0, 0) es la esquina superior izquierda. Por lo tanto, las coordenadas *x* van de izquierda a derecha y las coordenadas *y* van de arriba abajo.

Figura 6. Coordenadas de la pantalla



2) La función *draw* es donde definiremos los elementos que queremos que se pinten en la pantalla y desde donde se controlarán las diferentes acciones que irán pasando durante la ejecución del programa. En el ejemplo lo único que hacemos es llamar a la función *texto* e indicamos el texto que queremos escribir entre comillas dobles y la posición dentro de la pantalla donde queremos que aparezca.

1.2. Elementos básicos

1.2.1. Inserción de comentarios

Para poder entender mejor el código que hacemos es importante irlo comentando. Por ejemplo, podemos indicar cuál es el objetivo del código que escribimos, la funcionalidad de una variable que declaramos o los valores correctos que tendría que devolver una función determinada, entre muchísimas otras cosas.

Fijaos en este código:

```
/*
 * Hello world with comments
 * Version: 2
 */

// Main initialization function
```

```
void setup() {
   size(200, 100);  //Window size
}

void draw() {
   text ("Hello world", 10, 10);
}
```

Para añadir un comentario de varias líneas, utilizamos la combinación */* text del comentario */*. En el código podemos ver que aparte de indicar el principio y el final del comentario con /* i */ hemos añadido un * al principio de cada línea, esto es opcional pero se acostumbra a hacer, para seguir el estilo recomendado para Java, y es útil para tener claro que forma parte de un comentario.

Otra manera de añadir comentarios es con //. En este se interpreta que todo aquello que haya entre // y el final de la línea es un comentario. Por lo tanto, es útil para comentar una única línea o añadir algún comentario a continuación de alguna instrucción.

1.2.2. Declaración de variables

Las variables son contenedores de información que utilizaremos a lo largo de nuestro programa para referenciar datos, hacer cálculos, almacenar información...

Observad este código:

```
String message;

void setup() {
   size(200, 100);
   message = "Hello world";
}

void draw() {
   text (message, 10, 10);
}
```

En este caso hemos añadido una variable de tipo *string* que se llama *message*, y la hemos declarado al principio, justo antes de la función *setup*. Las variables siempre las definiremos de esta manera, indicando el tipo de dato que almacena (en este caso, una cadena de texto) y el nombre que usaremos para referenciarla.

String

Recordad que *string*, o cadena de caracteres, es una lista de caracteres, por ejemplo, una palabra o una frase.

Aparte de declarar la variable la hemos de inicializar, y esto lo hacemos dentro de la función *setup* e indicamos que la variable *message* será igual a un texto determinado. En este caso concreto en el que trabajamos con un texto utilizamos las "dobles comillas" para limitarlo.

Ahora ya solo nos falta utilizar la variable que hemos definido. Para hacerlo, escribimos el nombre de la variable donde querríamos que estuviera el valor, en este caso, dentro de la función *text*, y ya lo tenemos hecho. En el momento en que se ejecute esta función, al escribir el texto, mirará el valor de la variable para escribir el mensaje almacenado.

1.2.3. Función *setup*

La función *setup* se utiliza para inicializar las variables y el resto de parámetros que se usarán a lo largo del programa. Su característica principal es que solo se ejecuta una única vez al principio de todo, y es por este motivo por lo que la utilizamos para inicializar los valores.

Como hemos dicho anteriormente, la primera instrucción que se ha de ejecutar dentro de de esta función es para definir la dimensión de la ventana gráfica:

```
void setup() {
   size(200, 100);
   ....
   ....
}
```

A partir de aquí podemos continuar con el resto de inicializaciones que necesitemos. En general hay una serie de parámetros globales que acostumbramos a definir:

1) ***size (x, y)***: como hemos dicho, esta es la función que utilizamos para indicar las dimensiones de la ventana gráfica. En los ejemplos siempre hemos utilizado dos valores para indicar la medida, pero probad a sustituir la llamada a la función por este código:

```
size(displayWidth, displayHeight);
```

Como podéis ver, al ejecutar el programa la ventana es mucho más grande, y de hecho es de la misma medida que la imagen que muestra vuestro monitor. Esto es así porque *displayWidth* y *displayHeight* son unas variables propias de Processing que se inicializan automáticamente con la dimensión actual del monitor, y por lo tanto, son muy útiles cuando queremos que una aplicación se ejecute a pantalla completa.

2) *frameRate (x)*: es la frecuencia de refresco que utilizaremos para redibujar el contenido de la pantalla. Al hablar de la función *draw* profundizaremos más en este concepto.

3) *background (r, g, b)*: es la función que utilizaremos para definir el color del fondo de la pantalla; en este caso le pasamos el color utilizando la notación rojo, verde, azul. Probad a añadir este código y observad cómo el fondo de la ventana en lugar de gris es rojo.

```
background (255, 0, 0);
```

4) *stroke (r, g, b)*: similar a la función *background*, en este caso *stroke* nos permite definir el color por defecto que utilizaremos para dibujar los elementos gráficos; probad de añadir esta línea de código dentro de *setup*:

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

Y esta otra dentro de *draw*:

```
line (0, 0, 200, 100);
```

Recordad que podéis consultar más detalles y ejemplos de estas funciones en la referencia en línea de la web de Processing o desde el menú **help** del programa mismo, donde también encontraréis la referencia fuera de línea.

1.2.4. Función *draw*

Una vez finalizada la ejecución de la función *setup*, todos los programas hechos con Processing continúan con la función *draw*, y lo más importante de todo es que esta función se irá ejecutando continuamente hasta que finalice el programa.

Es decir, la función *draw* se estará ejecutando en un bucle infinito mientras nadie le diga que tiene que parar. Además, no lo hará de manera descontrolada, sino que se ejecutará tantas veces por segundo como lo hayamos definido con la función *frameRate (x)* que hemos comentado en el apartado anterior.

Dentro de esta función será donde definiremos todas las acciones por hacer, desde pintar los elementos de la pantalla hasta hacer todos los cálculos necesarios para continuar con la ejecución correcta del programa.

Repetición infinita

Es importante que nos quede clara la idea de repetición cuando hablamos de la función *draw*, puesto que la estructura general del programa depende directamente de ella y hace que a veces pueda ser un poco confuso intentar entender qué hace nuestro código.

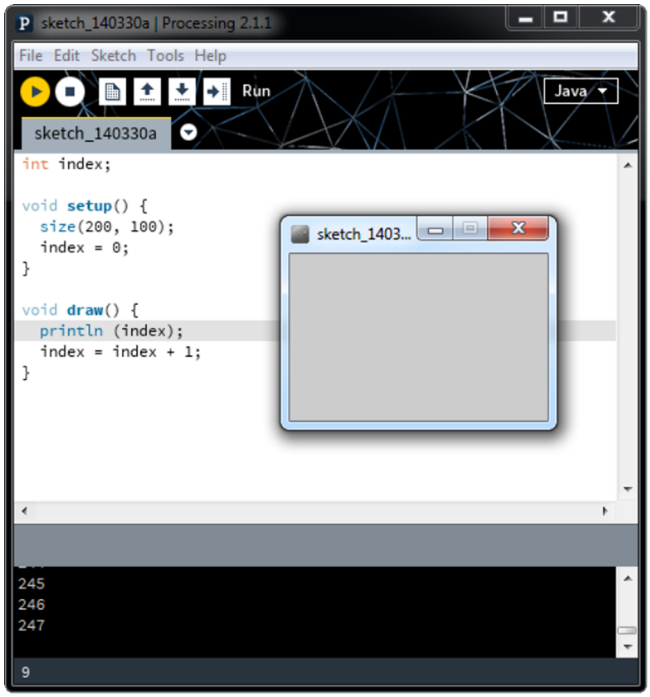
Supongamos este ejemplo:

```
int index;

void setup() {
   size(200, 100);
   index = 0;
}

void draw() {
   println (index);
   index = index + 1;
}
```

Podemos ver que hemos declarado una variable numérica *index* (lo veremos con más detalle en un capítulo posterior) y la inicializamos dentro de la función *setup*. A partir de este momento se empieza a ejecutar la función *draw* de manera continuada, sin dejar de repetirse, pero ¿cómo afecta esto a la variable *index*? y sobre todo, ¿qué valor tiene en cada momento?

Ejecutad el programa y veréis algo parecido a esto:

Figura 7. Ejecución del programa



Se abrirá una ventana gris, que es nuestro programa, y en la parte inferior del editor de código empezará a aparecer una secuencia numérica: en el caso de la imagen podemos ver 245, 246, 247. Por lo tanto, después de crear la variable e inicializarla a 0, empezamos a ejecutar sin cesar dos instrucciones:

- Escribimos el valor de la variable.
- Incrementamos el valor de la variable.

De momento no aparece la secuencia numérica en la ventana gráfica del programa; esto es así porque estamos utilizando la instrucción *println (index)*, que se encarga de escribir directamente el valor de la variable *index* en este espacio que tenemos debajo del editor del código. Esta función *println ()* es muy útil para ver el estado de las variables y enseñar mensajes que nos pueden servir mientras estamos haciendo el programa para comprobar que todo funciona bien o detectar posibles problemas.

Aprovechamos este mismo código y añadimos esta línea dentro de la función *setup*:

```
frameRate (1);
```

Ahora, al ejecutar el programa podréis ver que los valores van apareciendo cada 1 segundo, acabamos de definir el número de repeticiones por segundo en que queremos que se ejecute nuestra función *draw*. Comprobad cómo cambiando el valor se modifica la velocidad con que se repite la función, y por lo tanto, los números aparecen más o menos rápidamente.

Importancia del orden de los elementos

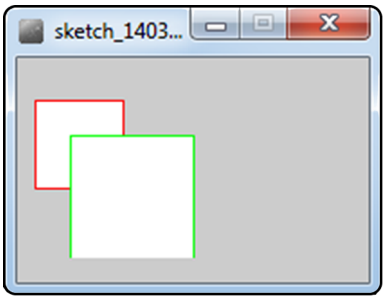
Al añadir los diferentes elementos gráficos que queremos pintar dentro de la función *draw* hay que tener en cuenta el orden utilizado, puesto que de ello dependerá el resultado final que obtengamos.

Utilizamos este código de ejemplo:

```
void setup() {
   size(200, 100);
}

void draw() {
   stroke (255, 0, 0);
   rect (10, 10, 50, 50);
   stroke (0, 255, 0);
   rect (30, 30, 70, 70);
}
```

Al ejecutarlo podéis ver este resultado:

Figura 8. Orden de pintar



Antes que nada dibujamos un cuadrado de color rojo y después uno de color verde, y a efectos prácticos, el último elemento que pintamos queda por encima de todo lo que ya estaba pintado con anterioridad. Esto es lo que se denomina *algoritmo del pintor*, puesto que un pintor primero dibuja los elementos más alejados y después va añadiendo el resto de los elementos más cercanos. Si modificamos el código e intercambiamos el orden de los cuadrados, el de color verde quedará por detrás.

```
void draw() {
   stroke (0, 255, 0);
   rect (30, 30, 70, 70);
   stroke (255, 0, 0);
   rect (10, 10, 50, 50);
}
```

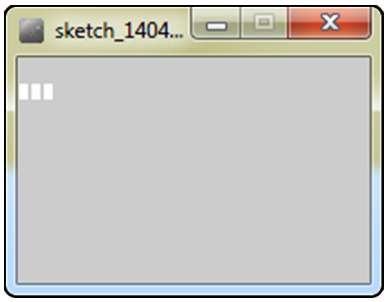
Es importante recordar esta manera de pintar los elementos, puesto que, cuando empiece a haber un número más importante, pueden aparecer efectos extraños si no lo tenemos en cuenta y planificamos correctamente el orden de pintar.

Ahora combinaremos estos últimos conceptos, el de repetición de la función *draw* y el del orden de pintar de los elementos; utilizaremos este código:

```
int index;

void setup() {
   size(200, 150);
   frameRate (1);
   index = 0;
}

void draw() {
   println (index);
   text (index, 0, 10);
   index = index + 1;
}
```

Es muy similar al que hemos visto anteriormente, pero hemos añadido la instrucción *text*, que nos permite escribir un mensaje en la ventana del programa, indicando el texto y la posición. Al ejecutar el programa veréis que tenemos un problema:

Figura 9. Texto gráfico borroso

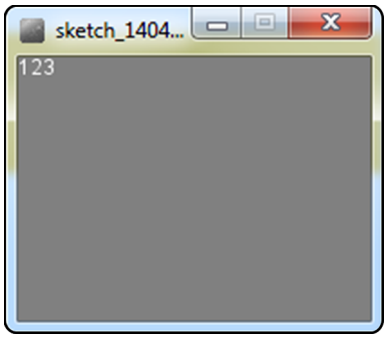


Nos aparece la ventana, pero en lugar de números tenemos una especie de mancha que no se puede leer correctamente, y para solucionarlo, haremos un pequeño cambio en la función *draw*:

```
void draw() {
   background (128, 128, 128);
   println (index);
   text (index, 0, 10);
   index = index + 1;
```

```
}
```

Figura 10. Texto gráfico visible



¿Por qué se pinta bien el texto ahora? Como hemos dicho, cuando pintamos algo lo estamos apilando por encima de lo que ya había, y en nuestro caso estábamos escribiendo un texto sobre otro, y otro, y otro... y por lo tanto, estábamos obteniendo el mismo efecto que si cogemos un trozo de papel y escribimos un texto sobre otro.

Al añadir la llamada *background* al principio de todo, hacemos que, justo al empezar a repintar nuestra ventana hagamos una repintada de todo el contenido con un color determinado. El efecto es el mismo que borrar nuestro papel antes de escribir nada más en él, y por lo tanto, el nuevo contenido se podrá ver perfectamente.

Añadimos esta nueva instrucción antes de escribir el texto, para poder modificar la medida:

```
textSize (30);
```

Al ejecutar el código veréis que habrá que modificar la posición donde pintamos el texto, porque al ser de mayor tamaño, se sale de la pantalla, pero a partir de ahora ya podremos controlar la medida para adaptarla a nuestras necesidades.

2. Tipo de datos básicos

En todos los programas necesitaremos guardar información en algún momento. Para hacerlo dispondremos de diferentes tipos de variables que nos permitirán guardarla y disponer de ella en cualquier momento. A continuación veremos las más básicas que utilizaremos en los próximos apartados.

2.1. Definición e inicialización de variables

Antes de poder guardar o utilizar una variable hay que definirla e inicializarla para mantener una estructura clara y que nos permita entender mejor el código. Lo haremos de la manera siguiente:

```
tipus_de_variable nom_de_variable;

void setup() {
   nom_de_variable = valor_inicial;
}

void draw() {
}
```

Al principio del código definiremos las variables que utilizaremos, indicando el tipo de dato que contendrá y el nombre que utilizaremos para hacer referencia a él.

Dentro de la función *setup* inicializaremos el valor. A pesar de que en la mayoría de casos no es estrictamente necesario, sí que es muy recomendable y nos puede ahorrar más de un problema.

2.2. Variables: booleanas

Las variables booleanas son aquellas que nos permiten indicar el concepto de cierto o falso, y por lo tanto, actúan como un interruptor que podremos utilizar para indicar si se ha cumplido alguna condición o si tenemos que hacer alguna tarea determinada.

```
boolean a;

void setup() {
   a = true;
   a = false;
}
```

En el código podemos ver cómo definimos la variable *a*, que después podemos inicializar con el valor cierto o falso.

Este tipo de variables se utilizan mucho con las estructuras de control, como por ejemplo, los bloques *if* que veremos más adelante.

2.3. Variables: números enteros

Los enteros nos permiten definir números sin decimales, como por ejemplo el índice que hemos utilizado en los ejemplos anteriores para hacer un contador.

```
int index;

void setup() {
   index = 0;
}
```

Naturalmente, una vez definida una variable numérica, podremos hacer cualquier tipo de operación matemática.

```
int index;
int doble;

void setup() {
   index = 0;
   doble = 0;
}

void draw() {
   println ("index: "+ index +" valor doble: " + doble);
   index = index + 1;
   doble = index * 2;
}
```

Por ejemplo, en este caso escribimos en la ventana de mensajes del programa el valor actual de la variable *índex* y su valor doble.

Fijaos cómo lo hemos hecho para combinar texto nuestro y diferentes variables en el mensaje que hemos escrito:

```
println ("index: "+ index +" valor doble: " + doble);
```

Cuando queremos escribir un texto nuestro, utilizamos las comillas dobles para delimitarlo, y cuando queremos escribir una variable, indicamos el nombre directamente, pero si queremos concatenar texto y variables, utilizamos la operación suma y construimos la frase que queremos enseñar.

2.4. Variables: números decimales

Los números con decimales los definiremos utilizando el tipo *float*, a pesar de que tenemos otras opciones, pero nos basaremos en este tipo, puesto que es el que utilizan las funciones de Processing cuando tienen que trabajar con decimales.

```
float decimal;

void setup() {
   decimal = 0.0;
}
```

Un detalle importante al trabajar con decimales es el de remarcarlo indicándolo explícitamente; en el caso de la inicialización, utilizamos el valor **0,0** para hacerlo evidente. Es importante para evitar problemas como los que genera el código siguiente:

```
int a;
int b;
float result;

void setup() {
   a = 11;
   b = 2;

   result = a / b;
   println ("a/b = " + result);

   result = 11 / 2;
   println ("a/2 = " + result);

   result = a / 2.0;
   println ("a/2 = " + result);
}
```

Al ejecutar el código veréis el resultado siguiente:

```
a/b = 5.0
a/2 = 5.0
a/2 = 5.5
```

Cuando tomamos dos variables enteras y las dividimos, el resultado que nos devuelve es un número entero, y por lo tanto, redondea el valor; este es el caso de las dos primeras operaciones.

En cambio, si indicamos que uno de los operandos es un decimal, ya nos devuelve un resultado con decimales, y en este caso lo hemos hecho dividiendo entre **2,0**.

Es importante recordarlo porque puede producir resultados extraños y a veces este tipo de problema es difícil de detectar.

2.5. Variables: conjuntos

Los conjuntos o *arrays* son una agrupación de datos que podremos almacenar y a los cuales podremos acceder desde una única variable. Al definir una variable del tipo *array*, deberemos decirle el tipo de información que contendrá, y lo hacemos de la manera siguiente:

```
int[] int_array;
```

En este ejemplo definimos un *array* de enteros, indicando el tipo de datos que contiene el *array* y a continuación añadiendo *[]* antes de escribir el nombre de la variable.

Podemos inicializarlos de dos maneras diferentes en función de si conocemos todos los datos que tienen que tener desde el principio o si las queremos ir añadiendo después.

2.5.1. Inicialización con todos los valores conocidos

En este caso la inicialización se tiene que hacer en el mismo momento de declarar la variable, y se indican todos los elementos que la forman separados por comas y entre corchetes.

```
int[] int_array = { 10, 20, 30 };


void setup() {
}
```

2.5.2. Inicialización sin valores

En este caso deberemos indicar cuántos elementos queremos que contenga el *array* como máximo.

```
int[] int_array;


void setup() {
   int_array = new int [3];
}
```

Utilizamos la palabra clave *new*, el tipo de datos que tendrá el *array* y el número de elementos entre corchetes. Con esto ya tenemos el *array* preparado para poder guardar datos, pero al contrario del caso anterior, de momento no tiene ninguno. Ahora veremos cómo podemos guardar y leer información.

2.5.3. Acceso a los datos del *array*

Independientemente de cómo hayamos inicializado el *array*, ahora ya lo podremos utilizar para guardar información y acceder a la esta cuando nos haga falta, y para hacerlo, debemos indicar qué elemento o posición queremos modificar o leer. Por ejemplo, si queremos guardar el valor 444 en la posición 1, lo hacemos de la manera siguiente:

```
int_array[1]=444;
```

Y si quisiéramos escribir este valor lo podríamos consultar haciendo:

```
println (int_array[1]);
```

La posición que estamos consultando o modificando es el índice del *array*, y nos indica la casilla donde hacemos estas operaciones. El índice de un *array* empieza por el valor 0 y va hasta el número de elementos que tiene el *array* menos 1.

Por ejemplo, en las inicializaciones que hemos visto definíamos un *array* de 3 posiciones; por lo tanto, su índice irá de 0 a 2.

```
int[] int_array = { 10, 20, 30 };


void setup() {
   println (int_array[0]);
   println (int_array[1]);
   println (int_array[2]);
}
```

Aquí podemos ver que para acceder a los 3 enteros que tenemos guardados utilizamos los índices 0, 1 y 2.

Para consultar el número de elementos que puede guardar un *array* podéis utilizar:

```
println (int_array.length);
```

Al añadir *.length* después de una variable *array*, nos devolverá esta información, que después podremos utilizar para acceder a los datos (veremos ejemplos más adelante).

2.6. Variables: cadenas de texto

Como hemos visto, cuando queremos escribir un texto utilizamos las comillas dobles para pasarlo a la función de pintar, pero si lo queremos guardar en una variable para utilizarlo en otro momento, tenemos que utilizar el tipo *String*. Como en el caso anterior, para inicializar una cadena de texto utilizaremos *new*, de forma que tendremos un código de este estilo:

```
String text;


void setup() {
   text = new String ("Hello");
   println (text);
}
```

Por lo tanto, primero definimos el nombre de la variable y después la inicializamos dentro de la función *setup*. Tened en cuenta que la inicialización solo hay que hacerla una vez, y por lo tanto, si queremos cambiar el texto no hay que volver a utilizar el formato *new String* (texto nuevo), sino que se puede hacer directamente:

```
void setup() {
   text = new String ("Hello");
   text = "bye!";
   println (text);
}
```

En este caso actualizamos el valor igualándolo directamente al nuevo texto limitado por comillas dobles.

Aparte de poder asignar un texto o leerlo, tenemos otras funcionalidades, como poder hacer comparaciones, recuperar una parte del texto, etc., pero esto lo iremos viendo en los ejemplos futuros.

2.7. Variables: color

El Processing es un lenguaje que tiene como objetivo el trabajo con gráficos, y por lo tanto, el color es un parámetro muy importante que nos interesará poder guardar para utilizarlo en nuestros programas.

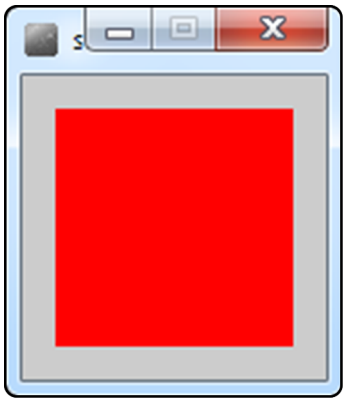
La manera más fácil de definir una variable de color es esta:

```
color c;


void setup() {
   c = color (255, 0, 0);
}
```

```
void draw() {
   background (c);
}
```

Fijaos que después de definir la variable utilizamos una función que también se llama *color(r, g, b)*, y a la cual pasamos los componentes rojo, verde y azul del color que estamos definiendo (valores entre 0 y 255), y a continuación ya podemos utilizar nuestra variable para definir el color de fondo, por ejemplo.

Figura 11. Variable color



Si en algún momento queréis consultar el valor del color y lo escribís directamente por pantalla, obtendréis un valor muy difícil de interpretar, pero hay una solución muy simple:

```
color c;

void setup() {
   c = color (255, 120, 60);

   println ("Direct value: " + c);
   println ("Hexadecimal value: " + hex(c));
}
```

Y la salida por pantalla es:

```
Direct value: -34756
Hexadecimal value: FFFF783C
```

Si enseñamos el valor directo es un número negativo, y es muy complicado de interpretar a qué color real hace referencia, pero si utilizamos la función *hex()* lo que hagamos es convertir este color a hexadecimal, y en este caso, la interpretación es mucho más sencilla, puesto que obtenemos un número que podemos descomponer en 4 bloques muy definidos AARRGGBB, que se corresponden con:

- **AA**: Valores entre 00 y FF (0 a 255 en decimal), definen el canal alfa, o lo que es el mismo, la transparencia del color.
- **RR**: Valores entre 00 y FF (0 a 255 en decimal), definen el canal rojo.
- **GG**: Valores entre 00 y FF (0 a 255 en decimal), definen el canal verde.
- **BB**: Valores entre 00 y FF (0 a 255 en decimal), definen el canal azul.

Por lo tanto, ahora sí que es más fácil poder interpretar este color y, por lo tanto, si en algún momento tenemos que comprobar alguno de estos, nos resultará más intuitivo hacerlo.

2.8. Ejemplo: "Hola mundo" utilizando variables

Ahora que ya conocemos las variables básicas, evolucionaremos el código que habíamos hecho inicialmente para añadir algunos elementos más interesantes. Empezaremos a partir de este punto:

```
void setup() {
   size(300, 200);
}

void draw() {
   textSize (30);
   text ("Hello world", 10, 30);
}
```

Lo primero que haremos será quizás lo más lógico, sustituir la cadena de texto por una variable *string*:

```
String message;

void setup() {
   size(300, 200);

   message = new String ("Hello world");
}

void draw() {
   textSize (30);
   text (message, 10, 30);
}
```

Naturalmente, al ejecutar el código obtenemos el mismo resultado, pero utilizar una variable nos puede ser muy útil cuando hemos de escribir el mismo texto en diferentes sitios del programa. En estos casos nos aseguramos de que si hay que cambar el texto en algún momento, solo habrá que modificarlo en

un único sitio (dentro de *setup*, donde lo hemos inicializado), y a partir de este momento, se verá reflejado el cambio en todos los lugares donde utilizábamos esta variable.

Ahora haremos otro cambio; mirad el código siguiente:

```
String message;
int size;

void setup() {
   size(300, 200);

   message = new String ("Hello world");
   size = 10;
}

void draw() {
   textSize (size);
   text (message, 10, 30);
}
```

Hemos hecho lo mismo que en el texto, pero con el tamaño. Esto nos permite hacer:

```
String message;
int size;

void setup() {
   size(300, 200);
   frameRate (1);

   message = new String ("Hello world");
   size = 10;
}

void draw() {
   textSize (size);
   text (message, 10, 30);
   size = size + 1;
}
```

Ahora el texto va creciendo poco a poco, pero hay un problema y no se ve bien, ¿verdad? ¿Recordáis cómo lo solucionábamos? Exacto, con la función *background*, y esta vez también usaremos una variable del tipo color:

```
String message;
int size;
```

```
color c;

void setup() {
   size(300, 200);
   frameRate (1);

   message = new String ("Hello world");
   size = 10;
   c = color (100, 100, 100);
}

void draw() {
   background (c);
   textSize (size);
   text (message, 10, 30);
   size = size + 1;
}
```

Ahora sí, ya lo podemos leer sin problemas y vemos cómo va creciendo. Naturalmente, también podríamos utilizar variables para definir la posición del texto, de forma que podríamos modificar la posición del mismo modo que modificamos el tamaño.

Para acabar este ejemplo, haremos que el mensaje vaya alternando entre dos textos diferentes, y por eso utilizaremos un *array* de textos:

```
String[] message;
int size;
color c;

void setup() {
   size(300, 200);
   frameRate (1);

   message = new String [2];
   message[0] = new String ("Hello");
   message[1] = new String ("world!");
   size = 10;
   c = color (100, 100, 100);
}

void draw() {
   background (c);
   textSize (size);
   text (message[(size % 2)], 10, 30);
   size = size + 1;
```

```
}
```

Miramos en detalle lo que hemos hecho:

- Definimos la variable *array*, que contiene información de tipo *string*.

- *message = new String [2]*: inicializamos el *array* y decimos que tendrá lugar para 2 elementos de tipo *string*.

- *message[0] = new String ("Hello")*: inicializamos cada una de las posiciones (en este caso el elemento 0) con un dato de tipo *string*. Aquí lo que puede sorprender es que volvemos a hacer *un new*String, pero recordad que es porque para crear una cadena de texto normal, también teníamos que hacerlo de esta manera. En el caso de un entero o un decimal, pondríamos directamente el valor. Repasad el ejemplo que está en el apartado de *arrays* para verlo más claro.

- Una vez tenemos los dos textos cargados en el *array*, uno en la posición 0 y otro en la posición 1, hay que buscar la manera de que se vayan alternando. En nuestro caso, aprovecharemos la variable *size*, que se va incrementando, y la función módulo (*%*). Al hacer el cálculo (*size % 2*) nos está devolviendo el valor 0 cuando el valor *size* es par o 1 cuando es impar, y por lo tanto, lo podemos aprovechar para seleccionar qué elemento del *array* enseñamos en cada momento.

2.9. Ejemplo: calculando el tiempo

Un dato que nos puede ser muy útil es el tiempo, puesto que puede ser vital para hacer una animación que sea correcta o simplemente para tener un control general del programa. Con este ejemplo, aparte de trabajar con variables y hacer algunos cálculos, también entenderemos mejor la función *draw* y el hecho de que se vaya ejecutando constantemente a lo largo del tiempo.

Empezaremos calculando cuánto tiempo pasa entre cada ejecución de la función *draw*.

```
int time_now;
int time_old;
int time_delta;

void setup() {
   size (400, 150);
   frameRate (20);

   time_now = 0;
   time_old = 0;
```

```
    time_delta = 0;
}

void draw() {

    time_now = millis();
    time_delta = time_now - time_old;
    time_old = time_now;

    background(0, 0, 0);

    textSize (30);
    text (time_delta, 50, 35);
    text ("ms", 130, 35);
}
```

Analizamos el código en detalle:

1) Declaramos e inicializamos 3 variables enteras que necesitaremos para hacer el cálculo.

2) Inicialmente fijamos que el programa funcione a 20 fps (planos por segundo); esto quiere decir que cada segundo redibujaremos la imagen 20 veces. Por lo tanto, el tiempo que pasa entre cada plano que pintamos es de1/20 = 0,05 s, o lo que es lo mismo, 50 ms.

3) Dentro de la función *draw* está el cálculo que realmente nos interesa, pero antes de verlo, comentaremos la función *millis()*. Esta función devuelve los milisegundos que han pasado desde que hemos ejecutado el programa, y la utilizaremos para calcular el tiempo que pasa entre dos ejecuciones de la función *draw*.

4) Ahora sí, veremos cómo utilizamos estas variables y qué cálculos hacemos. Para hacerlo, simularemos la ejecución del programa paso por paso, y por lo tanto, veremos cómo se van modificando los valores cada vez que la función *draw* se ejecuta:

a) Inicializaciones: aquí aún no se ha ejecutado la función *draw* e inicializamos las 3 variables a 0 (*time_now = 0*, *time_delta = 0*, *time_old = 0*).

b) *Draw* 1: primera ejecución de la función, y primera actualización de los valores de las variables. Suponemos que el tiempo que hemos tardado para inicializarlo todo y llegar hasta aquí ha sido de 10 ms, y por lo tanto *time_now = 10*. Esta variable la utilizaremos para indicar el tiempo total que hace que ejecutamos el programa desde el principio de todo.

A continuación actualizamos el valor de *time_delta*, que es el tiempo actual menos el tiempo en el que se ejecutó por última vez la función *draw*: es decir, *time_old*. Por lo tanto, el valor de *time_delta* es el que estamos buscando, es el tiempo que ha pasado entre dos planos de nuestro programa.

Para acabar con el ciclo, hacemos *time_old* = *time_now*, y lo hacemos porque tenemos que dejar el valor de *time_old* actualizado para que cuando se vuelva a ejecutar la función *draw* tenga el valor correcto.

c) *Draw* 2: la segunda ejecución de la función se tiene que hacer 50 ms más tarde (recordad que hemos definido 20 fps), por lo tanto, *time_now* será igual a 60 ms (los 10 ms que hemos tardado en inicializar el programa + 50 ms entre los dos planos).

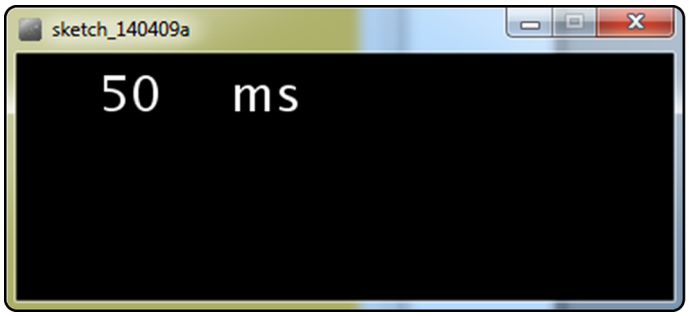
time_delta será igual a 60 – 10 = 50 ms, es decir, el tiempo actual menos el tiempo de la última ejecución de la función. Estos 50 ms ya son los que queríamos obtener.

Para acabar, *time_old = 60 ms*, para preparar la ejecución siguiente.

d) *Draw* n: a partir de aquí vamos repitiendo lo mismo para tener el valor actualizado en cada momento.

A pesar de que parece una manera complicada de hacer el cálculo, es la que nos ofrece una de las mejores soluciones, una vez tenemos claro el concepto de repetición.

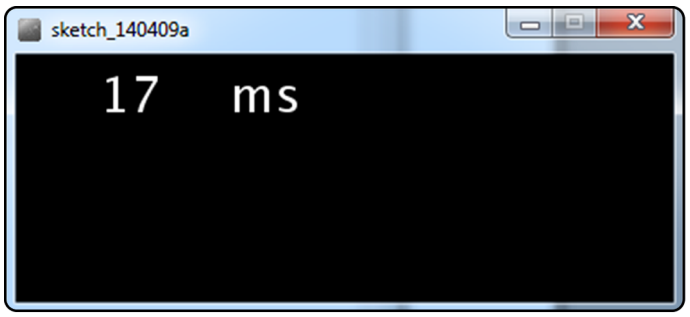
Figura 12. Calculando el tiempo entre planos



Ahora que tenemos el programa hecho, comentad la línea donde definimos los planos por segundo que queremos y volved a ejecutarlo:

```
//frameRate (20);
```

Figura 13. Tiempo mínimo entre planos



Al eliminar la línea que controla los planos por segundo, el programa intenta ejecutarlo lo máximo de rápido posible, de manera que el valor que obtenemos nos dice el tiempo mínimo entre planos que es capaz de generar nuestro ordenador.

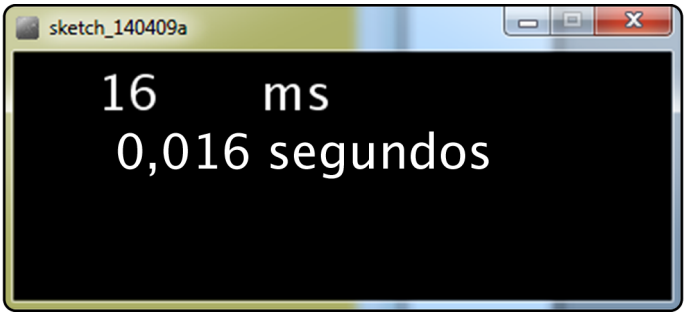
A partir de los milisegundos que hemos calculado, podemos obtener los segundos dividiendo entre 1.000, y lo hacemos con una nueva variable *sec*:

```
int time_now;
int time_old;
int time_delta;
float sec;

void setup() {
   size (400, 150);
   //frameRate (20);

   time_now = 0;
   time_old = 0;
   time_delta = 0;
   sec = 0;
}

void draw() {

   time_now = millis();
   time_delta = time_now - time_old;
   time_old = time_now;
   sec = time_delta / 1000.0;

   background(0, 0, 0);

   textSize (30);
   text (time_delta, 50, 35);
   text ("ms", 150, 35);
   text (sec, 50, 70);
   text ("segons", 150, 70);
```

```
}
```

Fijaos que en el código hemos dividido entre 1.000,**0**; ¿recordáis el motivo? Probad qué pasaría si dividiéramos entre 1.000.

Figura 14. Tiempo en segundos

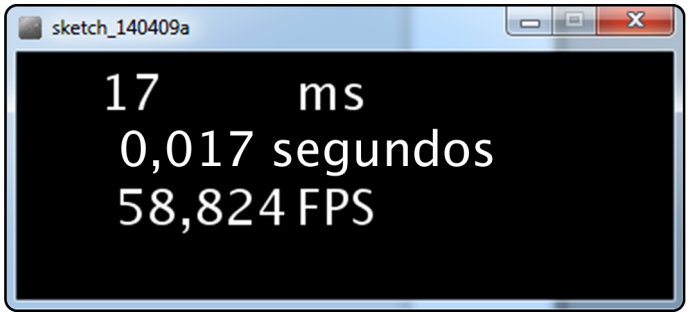


Para acabar, calcularemos los planos por segundo que estamos obteniendo. Para hacerlo solo tendremos que hacer 1/tiempo entre planos, pero igual que antes, lo haremos indicando 1,**0**/tiempo entre planos para no tener problemas.

```
int time_now;
int time_old;
int time_delta;
float sec;
float fps;

void setup() {
   size (400, 150);
   //frameRate (20);

   time_now = 0;
   time_old = 0;
   time_delta = 0;
   sec = 0;
   fps = 0;
}

void draw() {

   time_now = millis();
   time_delta = time_now - time_old;
   time_old = time_now;
   sec = time_delta / 1000.0;
   fps = 1.0 / sec;

   background(0, 0, 0);

   textSize (30);
```

```
    text (time_delta, 50, 35);
    text ("ms", 170, 35);
    text (sec, 50, 70);
    text ("segons", 170, 70);
    text (fps, 50, 105);
    text ("FPS", 170, 105);
}
```

Figura 15. Planos por segundo



Y aquí tenemos el resultado, no solamente sabemos cuánto tiempo tardamos en ejecutar la función *draw*, sino que además podremos utilizar esta información para calcular cómo va evolucionando el programa, las animaciones, las transiciones entre pantallas, etc.

3. Tipos de datos gráficos

En el apartado anterior hemos visto las variables básicas que se utilizan de manera general en cualquier tipo de programa, pero ahora es el momento de ver las herramientas que nos ofrece el Processing para crear contenidos gráficos, centrándonos en las primitivas y la gestión del color.

Para hacerlo nos basaremos en el código siguiente, que simplemente dibuja una ventana de 500 × 400 píxeles de color blanco:

```
void setup() {
   size(500, 400);
}

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

3.1. Formas básicas

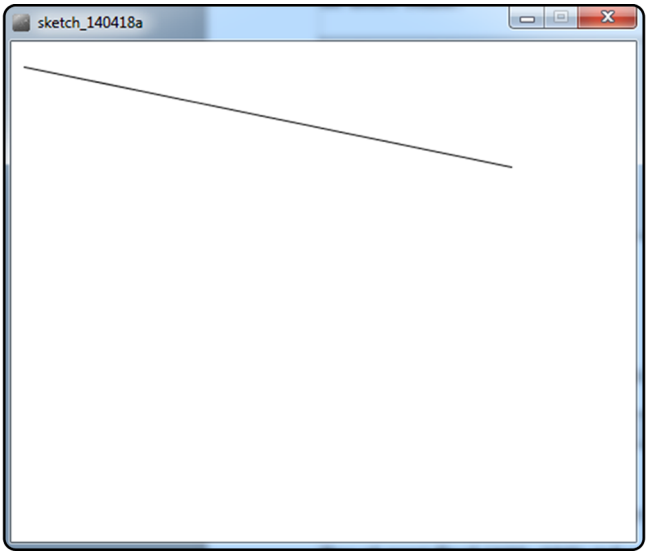
3.1.1. Líneas

Añadimos el código siguiente dentro de la función *draw*:

```
line (10, 20, 400, 100);
```

Al ejecutar el código podemos ver que nos aparece una línea dibujada desde el punto inicial (10, 20) hasta el punto final (400, 100), tal como hemos indicado en los parámetros que hemos pasado a la función *line*.

Figura 16. Dibujar líneas



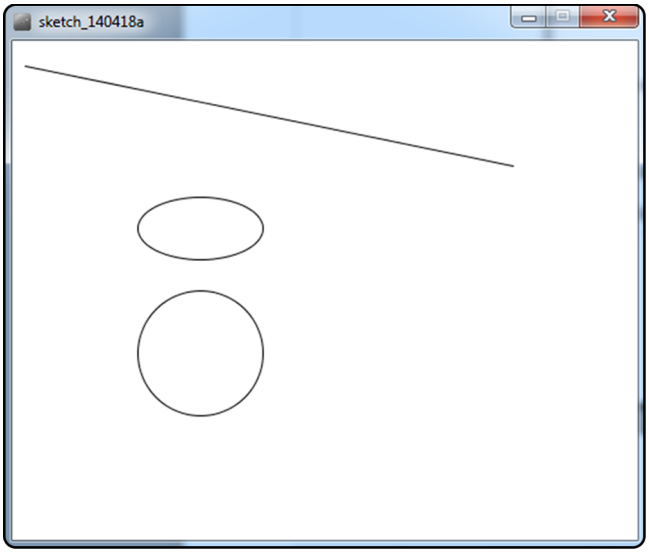
En este caso, y en general para cualquier elemento que dibujamos, lo más importante será tener clara la posición que tendrá que tener y cómo la controlaremos, puesto que el hecho de crear una línea, añadir una imagen, etc., no será ningún problema, gracias al hecho de que el Processing nos lo simplifica mucho, y por lo tanto, nuestra preocupación final será la de la composición de la pantalla que queremos mostrar.

3.1.2. Elipses

Añadimos las líneas a los códigos siguientes:

```
ellipse (150, 150, 100, 50);
ellipse (150, 250, 100, 100);
```

Figura 17. Dibujar elipses



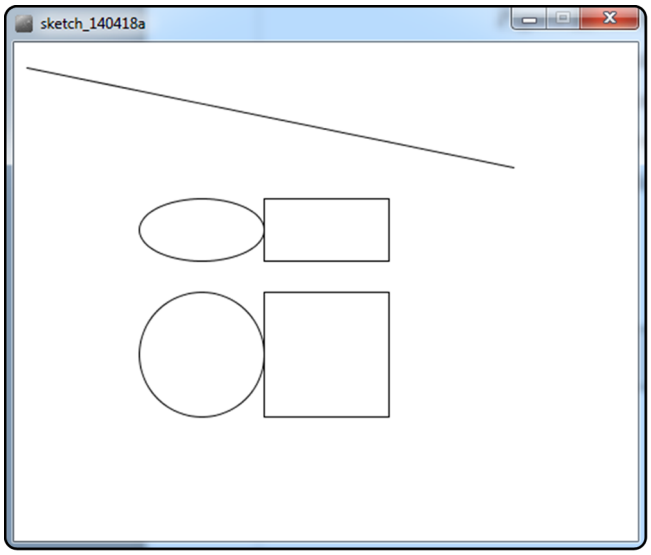
Igual que en el caso de la línea, la función *ellipse* necesita que le indiquemos 4 parámetros: los dos primeros hacen referencia a la posición del centro de la elipse, el tercero es la anchura y el cuarto la altura. Por lo tanto, para poder crear un círculo tendremos que definir los dos últimos parámetros con el mismo valor, tal como hemos hecho en la segunda línea de código que hemos añadido (valor 100).

3.1.3. Rectángulos

Añadimos las líneas a los códigos siguientes:

```
rect (200, 125, 100, 50);
rect (200, 200, 100, 100);
```

Figura 18. Dibujar rectángulos

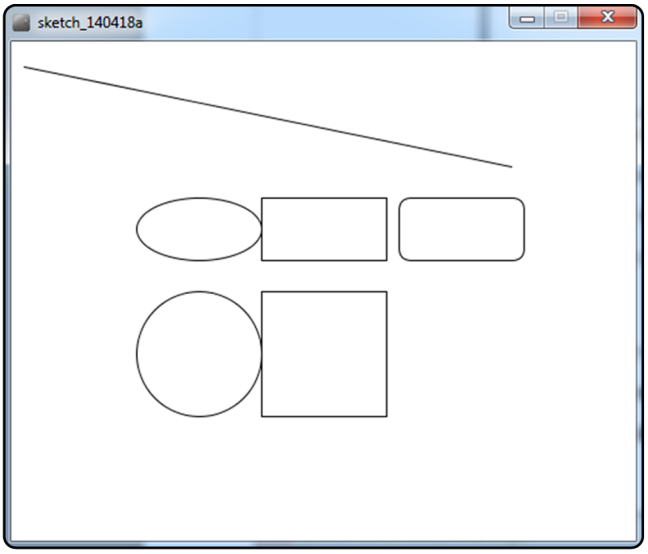


A diferencia de las elipses, en el caso de los rectángulos los dos primeros parámetros que indicamos indican la posición del vértice superior izquierda de la figura, y a continuación indicamos la anchura y la altura que tendrá. En el ejemplo podemos ver cómo hemos definido 2 rectángulos que podrían contener las elipses que habíamos definido antes.

La función *rect* tiene una particularidad que aún no habíamos visto en ningún otro caso, y es que le podemos indicar un número diferente de parámetros para conseguir diferentes resultados. En el ejemplo anterior le hemos dado cuatro parámetros, pero observad qué pasa si añadimos la llamada siguiente, con cinco parámetros:

```
rect (310, 125, 100, 50, 10);
```

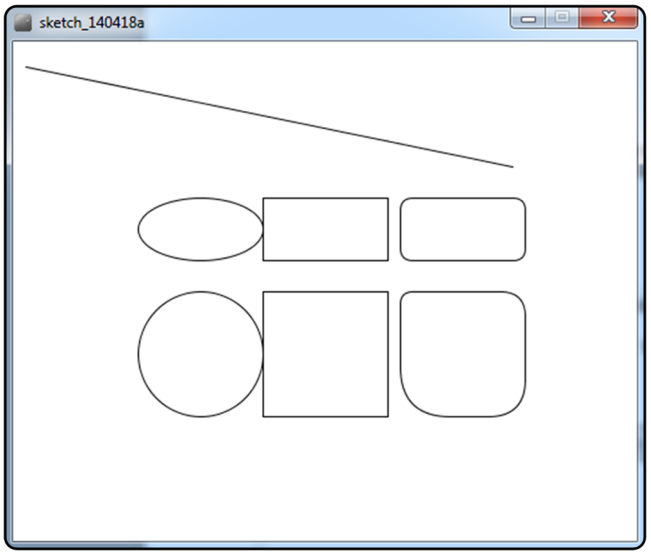
Figura 19. Dibujar rectángulos con opciones (1)



Con este parámetro extra conseguimos definir el radio que tendrán las cuatro esquinas del rectángulo, pero si en lugar de añadir un único parámetro añadimos cuatro más, podremos modificar cada radio por separado:

```
rect (310, 200, 100, 100, 10, 20, 30, 40);
```

Figura 20. Dibujar rectángulos con opciones (2)



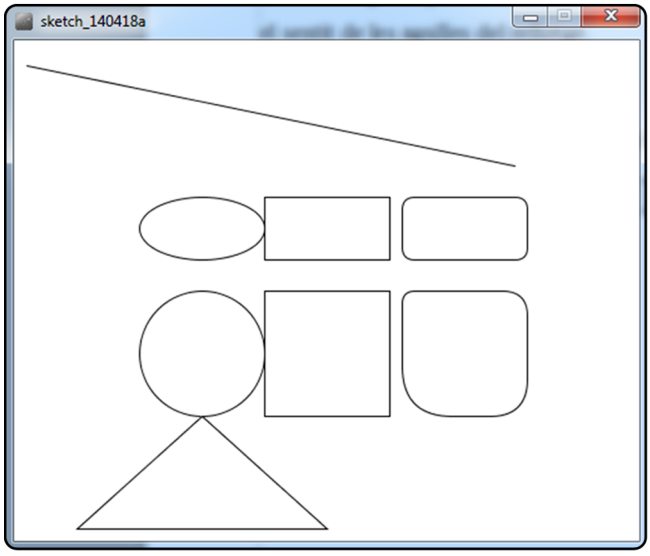
Fijaos que el primer radio hace referencia a la esquina superior izquierda, y que el resto se aplica en el sentido de las agujas del reloj.

3.1.4. Triángulos

Añadimos el código siguiente dentro de la función *draw*:

```
triangle (50, 390, 150, 300, 250, 390);
```

Figura 21. Dibujar triángulos



En el caso de los triángulos, siempre tendremos que indicar seis parámetros, que definirán la posición de los tres vértices de la figura. En nuestro caso son (50, 390), (150, 300) y (250, 390).

3.2. Gestión del color de las formas

Ahora que podemos dibujar figuras básicas, veremos cómo podemos indicar el color que queremos que tengan. Para hacerlo, nos basaremos en este código inicial:

```
color c_black;
color c_white;
color c_green;
color c_red;
color c_blue;

void setup() {
   size(500, 400);
   c_black = color (0, 0, 0);
   c_white = color (255, 255, 255);
   c_green = color (0, 255, 0);
   c_red = color (255, 0, 0);
   c_blue = color (0, 0, 255);
}

void draw() {
   background (c_black);
   rect (10, 10, 100, 100);
}
```

Para simplificar el uso de los colores, hemos creado 5 variables, que hemos inicializado con los colores blanco, negro, rojo, verde y azul; de esta manera no hará falta que introduzcamos los valores a mano cada vez que los tengamos que utilizar. Podéis ver cómo en el caso del color de fondo ya hemos utilizado la variable que indica el color negro en lugar de los valores (0, 0, 0).

También tenemos un cuadrado que por defecto se pintará de color blanco.

3.2.1. Relleno de las formas

Al dibujar una forma podemos definir el color con que pintaremos el interior o si, por el contrario, queremos que sea transparente. Para hacerlo, utilizaremos las funciones *fill* y *noFill*, tal como podemos ver en el código siguiente, que añadimos dentro de la función *draw*:

```
void draw() {
   background (c_black);
   fill (c_red);
   rect (10, 10, 100, 100);
}
```

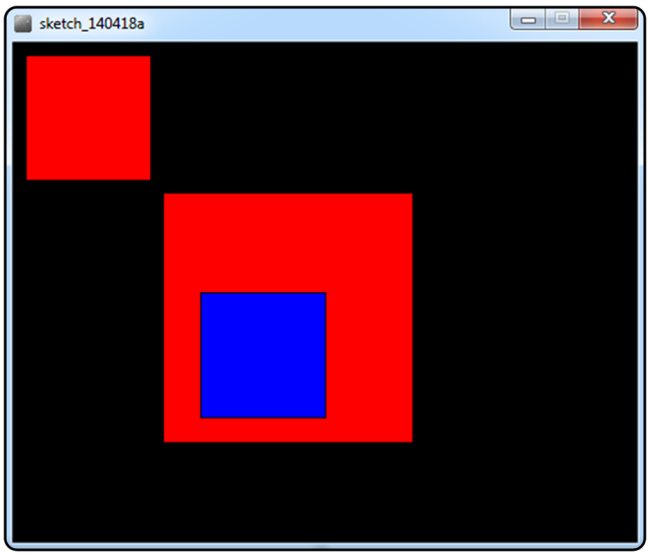
Figura 22. Función *fill*: cuadrado rojo

Ya lo tenemos, ahora el cuadrado es de color rojo; añadiremos alguno más:

```
void draw() {
   background (c_black);
   fill (c_red);
   rect (10, 10, 100, 100);
   rect (120, 120, 200, 200);
   fill (c_blue);
   rect (150, 200, 100, 100);
```

```
}
```

Figura 23. Función *fill*: orden de pintar



Ya hemos añadido un par de figuras; recordad que el último recuadro de color azul aparece encima del rojo porque lo hemos dibujado después y el Processing siempre añade los nuevos elementos por encima de los que ya había (el algoritmo del pintor).

Esta característica también es muy importante de tener en cuenta cuando trabajamos con funciones que modifican la manera de pintar los elementos, como es el caso de la función *fill*.

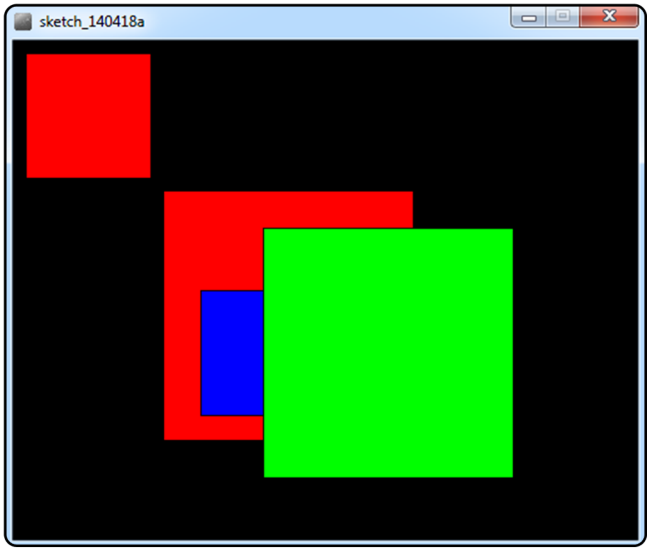
Desde el momento en que hemos definido que queríamos pintar las formas de color rojo, todas las nuevas formas que definimos serán rojas hasta que volvamos a cambiar el color. En nuestro caso, antes de dibujar el último recuadro, hemos dicho que empezaríamos a utilizar el color azul.

Es importante que tengamos esta característica presente cuando utilizamos el Processing, puesto que nos encontraremos muchos ejemplos en los que será necesario que lo tengamos claro para conseguir los resultados que queremos.

Ahora añadiremos otro recuadro de color verde:

```
fill (c_green);
rect (200, 150, 200, 200);
```

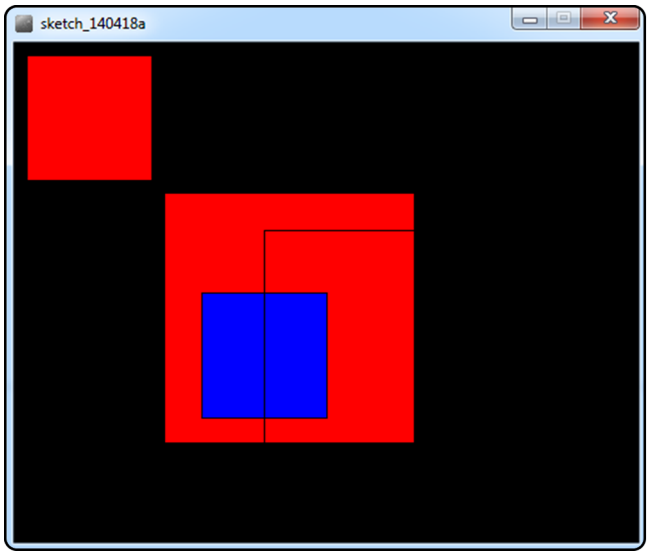
Figura 24. Función *noFill*: figura opaca



Este cuadrado que hemos añadido está tapando a los otros dos, pero lo que realmente queremos es poderlos ver, y por lo tanto, necesitamos decir que al dibujar este último recuadro no lo pinte de ningún color, que sea transparente. Para hacerlo, en lugar de utilizar la función *fill* utilizaremos la función *noFill*. Por lo tanto, eliminamos el código que hemos añadido y utilizamos el siguiente:

```
noFill ();
rect (200, 150, 200, 200);
```

Figura 25. Función *noFill*: figura transparente



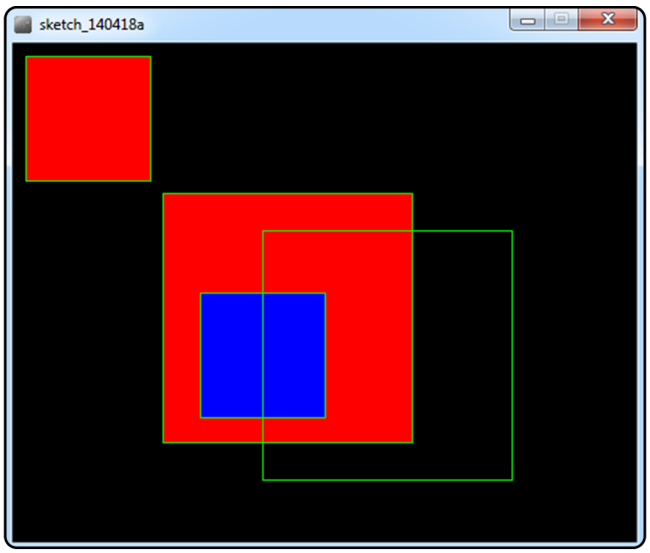
Ahora ya tenemos el cuadrado transparente, y fijaos que podemos ver que tiene el contorno de color negro. Al ser de color negro se confunde con el color de fondo, pero en el apartado siguiente lo solucionaremos.

3.2.2. Contorno de las formas

Lo primero que haremos será añadir este código al principio de la función *draw*:

```
stroke (c_green);
```

Figura 26. Función *stroke*

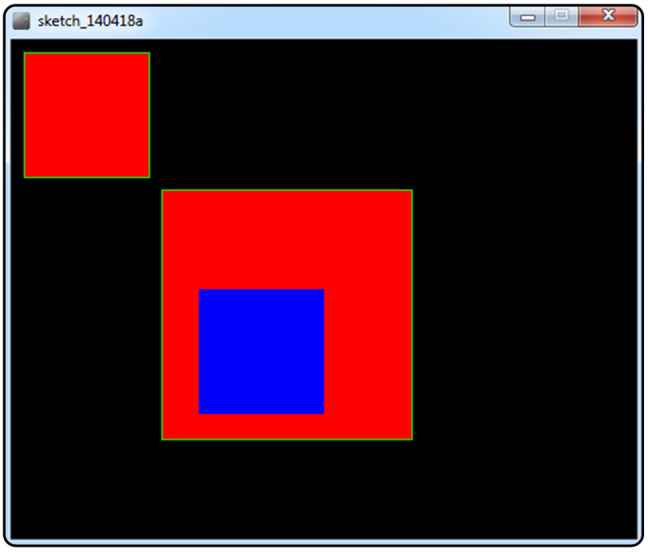


Esta función define el color de los contornos de las figuras que dibujamos, y al seleccionar el color verde, en lugar del negro que se utiliza por defecto, podemos ver claramente que todos los recuadros están contorneados.

Si, por ejemplo, quisiéramos que el cuadro de color azul no tuviera ningún contorno, justo antes de dibujarlo utilizaríamos la función *noStroke*:

```
fill (c_blue);
noStroke();
rect (150, 200, 100, 100);
```

Figura 27. Función *noStroke*



Al añadir esta instrucción, el cuadro azul ya no tiene ningún contorno, pero ¿dónde está el cuadrado transparente que estaba por encima? Pues está en el mismo lugar, pero como era transparente y ahora ya no pintamos los contornos, no lo vemos. Para solucionarlo, habrá que indicar que queremos pintar los contornos utilizando la función *stroke*, y en este caso lo haremos de color blanco. Al hacerlo, el código de la función *draw* nos queda de la siguiente manera:

```
void draw() {
   background (c_black);
   stroke (c_green);
   fill (c_red);
   rect (10, 10, 100, 100);
   rect (120, 120, 200, 200);
   fill (c_blue);
   noStroke();
   rect (150, 200, 100, 100);
   noFill ();
   stroke (c_white);
   rect (200, 150, 200, 200);
}
```

- Indicamos que el color de fondo es el negro.
- Definimos que queremos contornear de color verde y pintar de color rojo.
- Dibujamos dos cuadrados.
- Definimos que queremos pintar de color azul y que no queremos contornear.

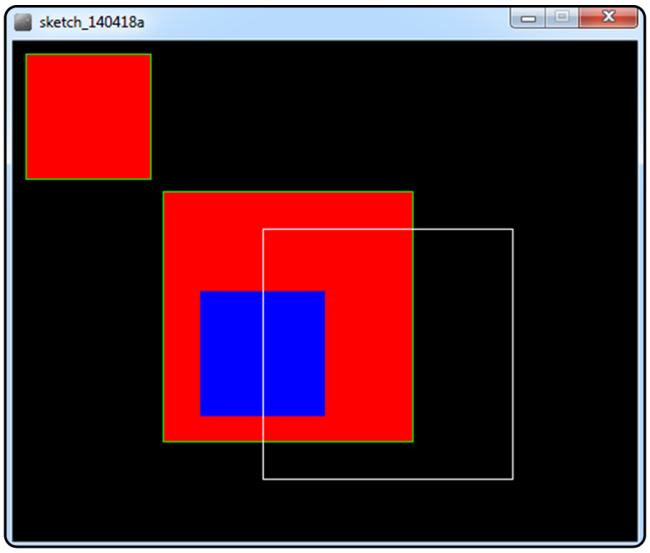
- Dibujamos el cuadrado pequeño.

- Definimos que no queremos pintar las figuras y las queremos contornear de color blanco.

- Dibujamos el cuadrado grande.

Y el resultado es este:

Figura 28. Funciones *stroke/noStroke/fill/noFill*



Con este ejemplo finalizamos la parte de gestión del color de las formas. Aparte de las funciones que hemos visto, hay que tener en cuenta la importancia del orden en que definimos los elementos y sus características, puesto que, como hemos podido ver, el resultado puede cambiar radicalmente. Esto hace que sea muy interesante planificar los diferentes elementos que aparecerán por pantalla antes de empezar a escribir el código para poderlo estructurar correctamente.

4. Animación básica

Como comentamos al empezar este curso de Processing, uno de los objetivos de este lenguaje es el de permitirnos crear animaciones de manera fácil. Para hacerlo, lo primero que nos hace falta es poder generar contenidos, como por ejemplo, las formas básicas que ya hemos visto, y el paso siguiente es aprender cómo las podemos animar.

Al hablar de animaciones, es fácil pensar en grandes producciones de películas o videojuegos, pero hay que recordar que cualquier elemento animado, como por ejemplo un simple icono que cambia si clicamos encima, es una animación. Por lo tanto, en este capítulo nos iniciaremos en técnicas de animación básica, que nos permitirán seguir evolucionando para obtener resultados más complejos.

El ejemplo que construiremos será el de una bola en movimiento; inicialmente se tratará de una animación muy básica e iremos añadiendo algunos controles extras para añadir algo más de gracia a la animación final.

El código base con el cual empezaremos a trabajar es este:

```
void setup() {
   size(800, 400);
   frameRate (20);
}

void draw() {
   background(255, 255, 255);
   fill (0, 0, 0);
}
```

Cómo podéis ver, hemos definido una ventana que pintamos de color blanco y que se actualizará 20 veces por segundo.

Ahora lo que haremos será añadir la bola que queremos animar. Al definirla, fijaos que utilizaremos variables para indicar su posición, puesto que será necesario que las podamos ir variando para indicar su posición actual.

```
int ball_x;
int ball_y;
int ball_radius;

void setup() {
   size(800, 400);
```

```
    frameRate (20);

    ball_x = 10;
    ball_y = 200;
    ball_radius = 10;
}

void draw() {
    background(255, 255, 255);
    fill (0, 0, 0);

    ellipse (ball_x, ball_y, ball_radius, ball_radius);
}
```

Figura 29. Animación: punto de partida

Ahora ya podemos ver nuestra bola situada en el punto desde donde se iniciará el movimiento. Lo que haremos a continuación será modificar su posición a lo largo del tiempo para que se vaya desplazando. Para hacerlo, añadimos esta línea de código en la función *draw()*:

```
ball_x = ball_x + 1;
```

¡Ya lo tenemos! Al ejecutar el programa, la bola se desplazará hasta la otra punta de la ventana hasta desaparecer. Podemos decir que este es el programa equivalente al "hola mundo" en el caso de las animaciones, y a partir de aquí, añadiremos algunas modificaciones para hacerlo algo más interesante.

4.1. Estructura condicional *if...else*

A continuación haremos que la bola se pare al final de la ventana. Para hacerlo, tendremos que comprobar la posición actual y actuar en consecuencia, y es por eso por lo que utilizaremos la instrucción *if...else*.

Modificamos la función *draw* para que quede de esta manera:

```
void draw() {
   background(255, 255, 255);
   fill (0, 0, 0);

   ellipse (ball_x, ball_y, ball_radius, ball_radius);

   if (ball_x < 800){
      ball_x = ball_x + 1;
   }else{
      text ("STOP", 400, 200);
   }
}
```

Analizamos lo que hemos añadido. La estructura *if...else* nos permite comprobar una condición y hacer una serie de tareas si esta se cumple u otras si no se cumple. En nuestro caso:

- *if (ball_x < 800)*: comprobamos si la posición de la bola en le eje *x* es inferior a 800, que es la anchura de nuestra ventana.

- Si la condición se cumple, la bola todavía no ha llegado al final de la ventana, incrementamos la variable *ball_x* para que la bola continúe avanzando.

- Si la condición no se cumple, la bola ha llegado al final, saltamos al bloque de código que hay en el *else*, y en este caso escribimos un texto en la pantalla y ya no incrementamos la variable de la posición de la bola.

4.2. Variables *height* y *width*

Ahora que hemos conseguido que la bola se mueva hasta el final de la ventana, cambiaremos la medida de nuestra ventana y pasaremos de 800 × 400 píxeles a 400 × 400 píxeles:

```
size(400, 400);
```

Si ejecutamos el código la bola se empezará a desplazar, pero al llegar al final de la ventana no se detendrá porque nosotros le habíamos dicho que no lo hiciera hasta llegar a la posición 800.

Para solucionar este tipo de problemas y podernos despreocupar de las dimensiones que tiene la ventana en cada momento, disponemos de las variables *height* y *width*, que contienen la información de la altura y anchura de la ventana del programa en cada momento. Modificamos la función *draw* de esta manera:

```
void draw() {
   background(255, 255, 255);
   fill (0, 0, 0);

   text ("Height: "+height+", Width: "+width, 10, 15);
   ellipse (ball_x, ball_y, ball_radius, ball_radius);

   if (ball_x < width){
      ball_x = ball_x + 1;
   }else{
      text ("STOP", 400, 200);
   }
}
```

Aparte de imprimir un texto con la información de la altura y la anchura, solo hemos cambiado la condición de *if*:

```
if (ball_x < width)
```

Ahora, en lugar de comprobar si la posición es más pequeña que 800 lo comparamos con la anchura que tiene la ventana en cada momento. Comprobad que si volvéis a cambiar las dimensiones, como por ejemplo 200 × 400 píxeles, la bola se continúa deteniendo al final de la ventana correctamente.

En general, intentaremos aprovechar estas variables para evitar que un cambio de las dimensiones de la ventana nos obliguen a repasar todo el código para adaptarlo a la medida nueva.

4.3. Simulaciones físicas

Con el ejemplo anterior hemos conseguido desplazar una bola por la pantalla a una velocidad marcada por el número de planos por segundo que estábamos dibujando (20 fps). Si en lugar de 20 fps hubiéramos indicado 10 o 30 fps, la bola se habría movido más lentamente o rápidamente, puesto que el incremento de la posición solo depende de las veces que se llame a la función *draw*.

```
ball_x = ball_x + 1;
```

Esto puede ser un problema si lo que queremos hacer es una simulación física más realista, en la que el movimiento de los objetos no se base en la velocidad de repintar de nuestro programa, sino en el tiempo.

Reescribiremos el código para que la bola caiga en lugar de moverse horizontalmente:

```
int ball_x;
int ball_y;
int ball_radius;

void setup() {
   size(400, 400);
   frameRate (20);

   ball_x = 200;
   ball_y = 10;
   ball_radius = 10;
}

void draw() {
   background(255, 255, 255);
   fill (0, 0, 0);

   ellipse (ball_x, ball_y, ball_radius, ball_radius);

   if (ball_y < height){
      ball_y = ball_y + 10;
   }else{
      text ("STOP", width/2, height/2);
   }
}
```

Al ejecutar el programa veréis que la bola cae rápidamente hasta detenerse, y lo que haremos ahora será hacer que el movimiento sea real, una simulación física de cómo se caería una bola de verdad si la soltáramos. Para hacerlo, necesitaremos dos cosas:

1) Una referencia de tiempo real, puesto que la velocidad de repintado que definimos no nos garantiza que se cumpla este tiempo. Para calcular el tiempo, utilizaremos el mismo sistema que ya hemos visto anteriormente.

Ved también

El sistema para calcular el tiempo se trata en el subapartado 2.9, sobre el tipo de datos básicos.

2) Una ecuación que nos defina el movimiento que queremos hacer. Esta puede ser la parte más complicada en función de lo que queramos hacer: en este caso se trata de la función de desplazamiento de un cuerpo acelerado. ¿La recordáis de las asignaturas de Física?

$$x = x_0 + v_0 t + \frac{1}{2} a t^2$$

En nuestro caso, esta será la más complicada que veremos, y nos servirá para calcular la posición de la bola cada vez que la tengamos que repintar. Para rellenar esta ecuación también utilizaremos la del cálculo de la velocidad, que veremos directamente en el código.

Ahora veremos las variables que utilizaremos:

```
// Time variables
int time_now;
int time_old;
int time_delta;

// Ball variables
int ball_x;
int ball_radius;
float ball_y;
float ball_y_old;
float ball_y_speed;
float ball_y_speed_old;
float ball_y_acceleration;
```

Para calcular el tiempo, definimos las tres variables que ya habíamos visto: el tiempo actual, el tiempo de la ejecución anterior de la función *draw*, y el tiempo que ha pasado entre las dos ejecuciones.

Para la bola mantenemos las variables del radio y la posición *x*, pero modificamos la variable de la posición *y* y decimos que sea de tipo *float*. Esto lo hacemos para asegurarnos de que al hacer los cálculos tengamos más precisión y la bola se mueva de una manera más suave. También hemos definido cuatro variables más que necesitaremos para hacer el cálculo: la posición *y* de la última vez que se ha movido la bola, la velocidad actual y la velocidad anterior y la aceleración.

Las inicializaciones quedan de esta manera:

```
void setup() {
   size(400, 400);
   frameRate (20);

   //Time init
   time_now = 0;
   time_old = 0;
   time_delta = 0;
```

```
    //Ball init
    ball_x = 200;
    ball_y = 10.0;
    ball_y_old = 10.0;
    ball_radius = 10;
    ball_y_speed = 0.0;
    ball_y_speed_old = 0.0;
    ball_y_acceleration = 9.8;
}
```

Aquí no hay ninguna sorpresa, inicializamos los tiempos a cero, y la posición de la bola en el punto de donde queremos que salga, con la velocidad inicial a cero y la aceleración de la Tierra.

Ahora nos toca hacer el cálculo del tiempo: utilizaremos el mismo código que ya habíamos visto y comentado, insertándolo al principio de la función *draw*:

```
time_now = millis();
time_delta = time_now - time_old;
time_old = time_now;
```

La variable *time_delta* es la que utilizaremos para hacer el cálculo de la nueva posición, y lo hacemos de esta manera:

```
    if (ball_y < height){
        ball_y_speed = ball_y_speed_old + (ball_y_acceleration * (time_delta/1000.0));
        ball_y = ball_y_old + (ball_y_speed_old * (time_delta/1000.0)) +
        ( (ball_y_acceleration * (time_delta/1000.0) * (time_delta/1000.0)) / 2.0 );

        ellipse (ball_x, ball_y, ball_radius, ball_radius);

        ball_y_speed_old = ball_y_speed;
        ball_y_old = ball_y;
    }else{
        text ("STOP", width/2, height/2);
    }
```

Mientras la bola no llegue al suelo:

- Calculamos la velocidad que tiene que tener en función de la velocidad anterior y el tiempo que ha pasado. Fijaos que estamos dividiendo la variable *time_delta* entre 1.000,0, puesto que este valor es en milisegundos y nosotros lo necesitamos expresado en segundos.

- Calculamos la nueva posición utilizando la ecuación que hemos visto.

- Dibujamos la bola en la posición nueva que acabamos de calcular.

- Actualizamos el valor de las variables *antiguas* con las que hemos calculado ahora, de esta manera ya estaremos preparados para hacer el próximo cálculo.

Y en el supuesto de que llegue abajo, hacemos que se detenga y enseñamos el mensaje en la pantalla.

A partir de aquí sería bastante sencillo modificar este código para simular el típico ejemplo de una bola disparada por un cañón. ¿Os animáis a probarlo?

4.4. Mesa de billar

Para acabar, haremos el ejemplo de la bola que va rebotando por la pantalla.

Ved también

La estructura condicional *if...else* se trata en el subapartado 4.1. de este módulo.

El código está basado en lo que hemos visto al hablar de la estructura *if...else*:

```
int ball_x;
int ball_y;
int ball_radius;

void setup() {
   size(800, 400);
   frameRate (20);

   ball_x = 10;
   ball_y = 200;
   ball_radius = 10;
}

void draw() {
   background(255, 255, 255);
   fill (0, 0, 0);

   ellipse (ball_x, ball_y, ball_radius, ball_radius);

   if (ball_x < 800){
      ball_x = ball_x + 1;
   }else{
      text ("STOP", 400, 200);
   }
}
```

Antes de ver el código de la solución intentad modificarlo vosotros mismos para conseguir que la bola vaya rebotando por las paredes continuamente.

Una posible solución podría ser esta:

```
int ball_x;
int ball_y;
int ball_radius;
int inc_x;
int inc_y;

void setup() {
   size(800, 400);
   frameRate (20);

   ball_x = 10;
   ball_y = 200;
   ball_radius = 10;

   inc_x = 5;
   inc_y = 2;
}

void draw() {
   background(255, 255, 255);
   fill (0, 0, 0);

   ball_x = ball_x + inc_x;
   ball_y = ball_y + inc_y;

   ellipse (ball_x, ball_y, ball_radius, ball_radius);

   if ( (ball_x <= 0) || (ball_x >= width)) {
      inc_x = inc_x * -1;
   }
   if ( (ball_y <= 0) || (ball_y >= height)) {
      inc_y = inc_y * -1;
   }
}
```

Para hacer el control hemos definido dos variables que nos dicen cuánto nos tenemos que desplazar hacia cada dirección. En el supuesto de que la bola sobresalga por algún lado, por ejemplo por la derecha de la ventana, multiplicamos por –1 este desplazamiento y por lo tanto, hacemos que la bola vuelva hacia atrás.

5. Interacción básica: capturar el ratón

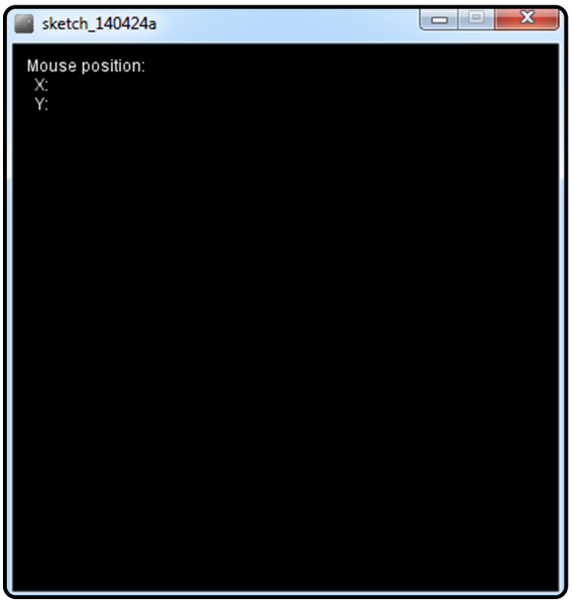
Con los ejemplos que hemos visto hasta ahora siempre hemos estado creando programas que al ejecutarlos hacían directamente la tarea definida y finalizaban, pero habrá muchos casos en los que necesitaremos que el usuario interactúe para dar indicaciones sobre lo que quiere hacer.

Para dar solución a esta necesidad veremos qué opciones nos da el Processing para gestionar el ratón, y haremos una serie de ejercicios para comprobar su funcionamiento.

El código base que utilizaremos es este:

```
void setup() {
   size(400, 400);
   frameRate (20);
}

void draw() {
   background(0, 0, 0);
   text ("Mouse position:\n  X: "+"\n  Y:",10, 20);
}
```

Figura 30. Interacción con el ratón: código inicial



De momento solo definimos la ventana y preparamos el texto en el que informaremos de la situación del ratón.

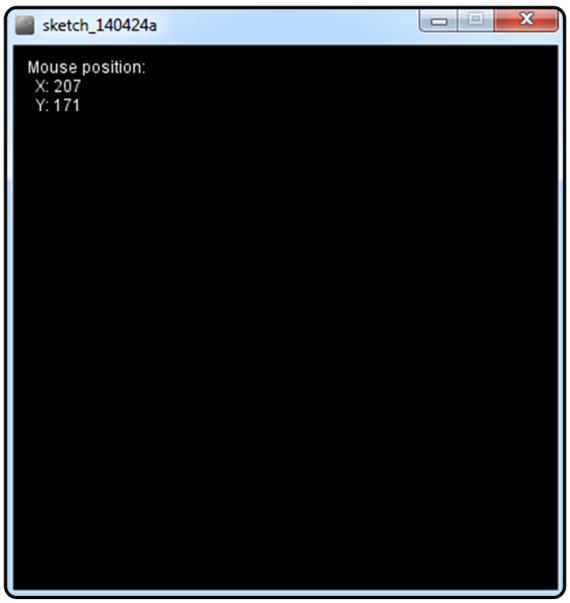
5.1. Conocemos la posición actual

Este es otro caso en que el Processing nos demuestra que está pensado para facilitarnos el trabajo, puesto que pone a nuestra disposición un par de variables que se actualizan automáticamente para indicarnos la posición del ratón en todo momento.

Estas variables son *mouseX* y mouseY, y las podemos consultar directamente desde cualquier lugar de nuestro código. Por ejemplo, si cambiamos la línea de texto que estábamos pintando y la dejamos de esta manera:

```
text ("Mouse position:\n  X: "+mouseX+"\n  Y: "+mouseY,10, 20);
```

Figura 31. Interacción con el ratón: *mouseX*/*mouseY*



De una manera muy fácil estamos enseñando en todo momento las coordenadas donde nos encontramos, y por lo tanto, podremos utilizar esta información para interactuar con el programa.

5.2. Perseguimos el ratón

Ahora que sabemos cómo podemos consultar la posición, haremos que una bola lo siga indefinidamente. Por lo tanto, lo primero que habrá que hacer es definir las variables que necesitaremos para dibujarla:

```
float ball_x;
float ball_y;
float dir_x;
float dir_y;
```

Y a continuación inicializarlas:

```
ball_x = width/2.0;
ball_y = height/2.0;
dir_x = 1.0;
dir_y = 1.0;
```

Podemos ver que tenemos dos parejas de variables:

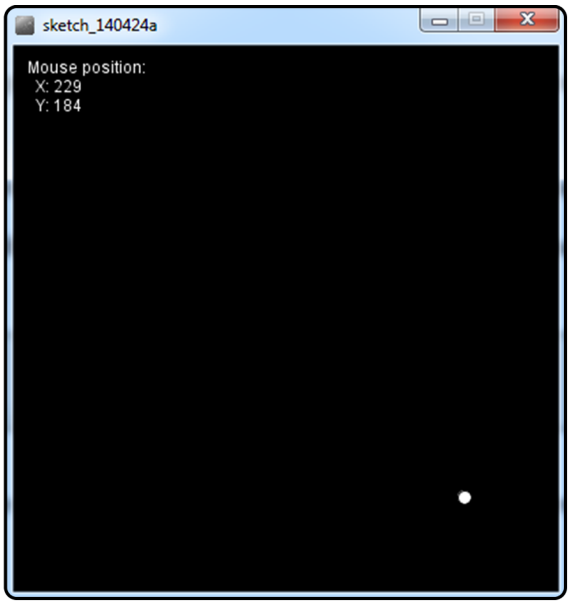
- *ball_**: nos sirve para indicar la posición actual de la bola y está inicializada en el centro de la ventana.

- *dir_**: indica hacia dónde se moverá la bola; de momento decimos que se mueva hacia abajo a la derecha.

Lo único que nos falta es dibujar la bola y calcular su desplazamiento. De momento lo hacemos sin tener en cuenta el ratón:

```
ball_x = ball_x + dir_x;
ball_y = ball_y + dir_y;
ellipse (ball_x, ball_y, 10, 10);
```

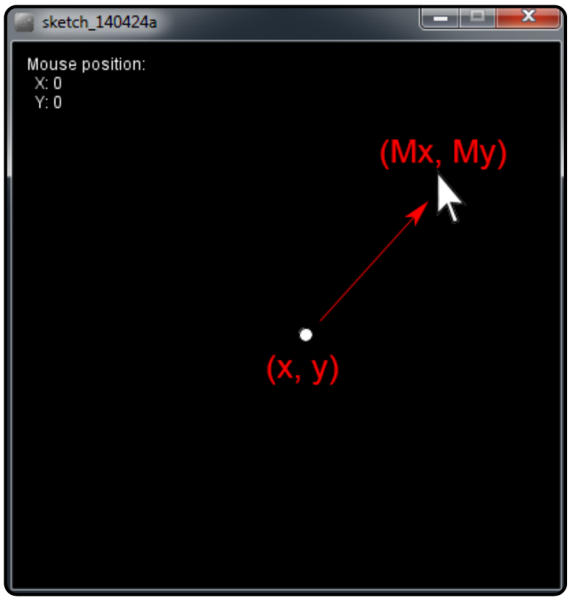
Figura 32. Perseguimos el ratón



Y ya lo tenemos, una bola que se mueve desde el centro hasta la esquina inferior derecha.

Ahora lo que tenemos que calcular es el valor de las variables *dir_x* i *dir_y* para que en cada momento la bola se mueva hacia donde está el ratón.

Figura 33. Cálculo del desplazamiento



Como podemos ver en la imagen, conocemos la posición actual de la bola y la del ratón, de manera que tenemos que calcular la diferencia que hay entre las dos posiciones (*Mx – x*, *My – y*). Traducido al código sería así:

```
void draw() {
   background(0, 0, 0);
   text ("Mouse position:\n  X: "+mouseX+"\n  Y: "+mouseY,10, 20);

   dir_x = mouseX - ball_x;
   dir_y = mouseY - ball_y;

   ball_x = ball_x + dir_x;
   ball_y = ball_y + dir_y;
   ellipse (ball_x, ball_y, 10, 10);
}
```

Hemos añadido las dos líneas que recalculan el valor de *dir_x* i *dir_y*, de forma que, en lugar de incrementar siempre la posición de la bola con un valor igual, utiliza el ratón como referencia.

5.3. Capturar los clics del ratón

Aparte de poder conocer la posición del ratón, el Processing tambe nos da las herramientas para poder conocer el estado del resto de elementos del ratón, como por ejemplo, si hemos pulsado un botón, si lo hemos desplazado, etc.

De todas las funciones y variables que podéis encontrar en la referencia utilizaremos un par para detectar cuándo hacemos clic, y hacer que la bola se mueva o quede parada. Para hacerlo, nos basaremos en este código:

```
float ball_x;
float ball_y;
float dir_x;
float dir_y;
boolean run;

void setup() {
   size(400, 400);
   frameRate (20);

   ball_x = width/2.0;
   ball_y = height/2.0;
   dir_x = 1.0;
   dir_y = 1.0;
   run = false;
}

void draw() {
   background(0, 0, 0);
   text ("Mouse position:\n  X: "+mouseX+"\n  Y: "+mouseY,10, 20);

   if (run == true){
      dir_x = mouseX - ball_x;
      dir_y = mouseY - ball_y;

      ball_x = ball_x + dir_x;
      ball_y = ball_y + dir_y;
   }
   ellipse (ball_x, ball_y, 10, 10);
}
```

Como podéis ver, es el mismo código de la bola que seguía al ratón, pero hemos hecho un par de cambios:

- Hemos añadido la variable *run*, que utilizaremos para indicar si la bola se tiene que mover o no. De momento la hemos dejada inicializada con el valor *false*, de manera que, aunque movamos el ratón, la bola no se moverá.

- Hemos añadido un bloque *if...else* que comprueba el valor de la variable *run*. En el supuesto de que el valor sea *true*, calculamos el movimiento que tiene que hacer la bola del mismo modo que habíamos visto en el ejemplo anterior.

Si ejecutáis el código tal como está, la bola se quedará quieta en el centro de la ventana. Si intentáis inicializar la variable *run* a *cert*, veréis como sí que se moverá.

Pero lo que nosotros queremos es que el valor de esta variable cambie al hacer clic con el ratón. Para hacerlo, deberemos añadir otra función a nuestro código:

```
void mousePressed() {
   run = true;
}
```

Por lo tanto, ahora tendremos la función *setup*, la función *draw*, y después d'esta hemos añadido la función *mousePressed*. Esta función es la que lo Processing ejecutará siempre que pulsamos algún botón del ratón, y en nuestro caso solo la utilizamos para hacer que la variable *run* tenga el valor *cert*.

Con este cambio, cuando ejecutamos el programa veremos que la bola está quieta en el centro de la ventana aunque movamos el ratón, pero en el momento que hacemos clic nos empezará a perseguir.

Hagámoslo algo más interesante: para conseguirlo, utilizaremos la variable *mouseButton*. A esta variable también la actualiza automáticamente el Processing y nos informa del último botón del ratón que hemos clicado. Por lo tanto, lo que podemos hacer es modificar la función *mousePressed* y hacer que al clicar con el botón izquierdo la bola se mueva y que con el derecho se pare. El cambio que tenemos que hacer es muy sencillo:

```
void mousePressed() {
   if (mouseButton == LEFT){
      run = true;
   }else{
      if (mouseButton == RIGHT){
         run = false;
      }
   }
}
```

Simplemente hay que identificar el botón que hemos clicado y actualizar la variable *run* en función de lo que queramos que pase.

5.4. Utilización de la rueda del ratón

Finalizaremos el apartado de gestión del ratón viendo cómo podemos utilizar la información del desplazamiento de la rueda. Para hacerlo, ampliaremos algo más el ejemplo y haremos que podamos ajustar la velocidad del movimiento de la bola con la rueda del ratón.

En este caso deberemos añadir otra función a nuestro código. Después de la función *mousePressed* añadid este código:

```
void mouseWheel(MouseEvent event) {
   float incr = event.getAmount();
   println(incr);
}
```

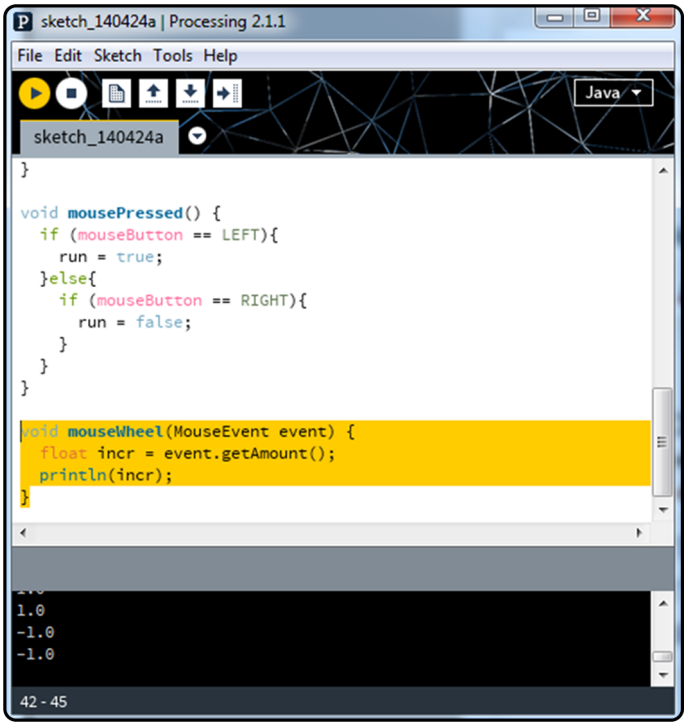
La función *mouseWheel* se ejecuta siempre que hagamos girar la rueda del ratón, y en este caso para recuperar la información de hacia dónde hemos girado la rueda, tenemos la variable *event*; más adelante veremos con más detalle cómo funcionan las funciones, de momento nos interesa saber que la variable *incr* valdrá 1 o – 1 en función del sentido en que giramos la rueda del ratón.

Ved también

Las funciones se tratan en el apartado 6 de este módulo.

Si ejecutáis el código y hacéis girar la rueda, podréis ver cómo aparece el valor en la ventana de mensajes del editor, gracias al *println* que hacemos:

Figura 34. Rueda del ratón



Lo que haremos a continuación es aprovechar esta información para modificar una variable que nos indicará la velocidad de la bola y que podrá tener un valor entre 1 y 100:

- El valor 1 indicará la velocidad más lenta.
- El valor 100 indicará la más rápida, que será la velocidad con que se está moviendo ahora la bola.

Antes de modificar realmente la velocidad de la bola, añadiremos esta variable y el código suficiente para poder comprobar que estamos modificando su valor.

Añadimos la variable:

```
float vel;
```

Y la inicializamos a la máxima velocidad:

```
vel = 100.0;
```

Además, modificamos el mensaje de texto para que también indique la velocidad actual:

```
text ("Mouse position:\n  X: "+mouseX+"\n  Y: "+mouseY+"\n  Vel: "+vel,10, 20);
```

Y ahora tendremos que modificar la función *mouseWheel* para actualizar el valor de la velocidad:

```
void mouseWheel(MouseEvent event) {
   float incr = event.getAmount();

   vel = vel + incr;
   if (vel < 1.0){
      vel = 1.0;
   }
   if (vel > 100.0){
      vel = 100.0;
   }
}
```

Ahora al ejecutar el código podréis ver cómo el valor de la velocidad va cambiando, y nunca es inferior a 1 ni superior a 100, que son los límites que queríamos tener.

Para finalizar, habrá que modificar el cálculo del movimiento de la bola para adaptarlo a la velocidad:

```
if (run == true){
      dir_x = mouseX - ball_x;
      dir_y = mouseY - ball_y;

      dir_x = (dir_x * vel) / 100.0;
      dir_y = (dir_y * vel) / 100.0;

      ball_x = ball_x + dir_x;
      ball_y = ball_y + dir_y;
}
```

Hemos añadido las dos líneas centrales. Lo que estamos haciendo es coger el desplazamiento que haría la bola y calcular el porcentaje en función de la velocidad que tenemos seleccionada. De esta manera, si la velocidad es 10, solo nos desplazaremos el 10% de la distancia que tendríamos que recorrer, y por lo tanto, la bola se moverá más lentamente que si vamos al 50% o al 100%.

Al final hemos obtenido un programa en el que, añadiendo dos funciones extras, podemos controlar con el ratón si la bola se mueve o no, hacia dónde se mueve y a qué velocidad de una manera bastante sencilla, gracias a toda la información que podemos obtener con las ayudas que nos ofrece Processing.

6. Organización del código

Para ayudar a organizar el código de un programa y hacerlo más fácil de entender y gestionar, utilizaremos dos herramientas diferentes: las funciones y los *frames*.

6.1. Las funciones

Una función es un conjunto de instrucciones que utilizamos para hacer unas acciones concretas. Hasta ahora ya hemos visto y utilizado algún ejemplo de funciones: *setup*, *draw*, *mouseWheel*, etc.

6.1.1. Estructura básica de las funciones

La estructura básica de una función es la siguiente:

```
<tipus_retorn> <nom_funció> (<parametres_que_rep>){
   <codi_de_la_funció>
   return <variable_que_retornem>
}
```

Vamos por partes:

1) Tipo de regreso: Cuando ejecutamos una función y se realizan las acciones que tenga definidas puede ser que al final nos tenga que devolver algún resultado. Por ejemplo, si tuviéramos una función que calculara el complementario de un color, querríamos que al finalizar la función nos devolviera ese color para poder usarlo. En estos casos hay que indicarlo para que todo el mundo lo sepa y el programa funcione correctamente.

De momento todas las funciones que hemos visto tenían como tipos de retorno *void*, y esto indica que esta función no devuelve ningún valor:

```
void setup(){}
```

Pero si, por ejemplo, tuviera que devolver un número entero, habría que utilizar *int* para indicarlo:

```
int funcioTornamUnNombre(){}
```

2) Nombre de la función: Es el nombre que utilizaremos cuando queramos ejecutar el código que contiene. Por convención general el nombre empieza por una letra minúscula y sin espacios en blanco. Si el nombre de la función está compuesto por varias palabras, haremos que la inicial de cada una sea en mayúsculas excepto la primera; por ejemplo: *estoEsElNombreDe unaFuncion*.

Es importante que los nombres sean autodescriptivos para facilitar el entendimiento del código.

3) Parámetros que recibe la función: Al ejecutar una función le podemos pasar tantos parámetros o variables como sean necesarios para que la función pueda hacer las tareas que tiene definidas. Siguiendo con el ejemplo de la función que calcula el complementario de un color, tendremos que pasarle el color original para que pueda hacer el cálculo, ¿no?

Para indicar los parámetros que recibe, usaremos la misma notación que utilizamos para definir las variables de nuestro programa, con la diferencia de que, si hay más de una, utilizaremos comas para separarlas:

```
void nomFuncio (int nombre, String text){}
```

En este caso la función recibe dos parámetros, uno de tipo entero y una cadena de texto, que podremos utilizar dentro de la función con los nombres que hemos definido: *número y texto*, respectivamente.

4) Código de la función: Es el conjunto de instrucciones que realizamos, y trabajaremos del mismo modo que lo hemos estado haciendo hasta ahora con las funciones *setup*, *draw*, etc.

5) Variable que devolvemos: En el caso de que la función devuelva algún parámetro lo habremos de indicar al finalizar la función utilizando la estructura:

```
return variable
```

Variable es el nombre de la variable que tiene el valor que queremos devolver. En este caso es obligatorio que el tipo de variable sea el mismo que el tipo que hemos dicho que devolvía la función. Es decir, si hemos definido la función de esta manera:

```
int tornaEnter(){}
```

La variable que devolvemos tendrá que ser de tipo *int*, puesto que si no, el programa nos dará un error.

6.1.2. Definimos la primera función

A partir del código de la bola que se mueve, que habíamos creado en el capítulo anterior, crearemos una función nueva para hacer el cálculo de la posición de la bola fuera de la función *draw*.

El código inicial es este:

```
float ball_x;
float ball_y;
float dir_x;
float dir_y;
boolean run;
float vel;

void setup() {
   size(400, 400);
   frameRate (20);

   ball_x = width/2.0;
   ball_y = height/2.0;
   dir_x = 1.0;
   dir_y = 1.0;
   run = false;
   vel = 100.0;
}

void draw() {
   background(0, 0, 0);
   text ("Mouse position:\n  X: "+mouseX+"\n  Y: "+mouseY+"\n  Vel: "+vel,10, 20);

   if (run == true){
      dir_x = mouseX - ball_x;
      dir_y = mouseY - ball_y;

      dir_x = (dir_x * vel) / 100.0;
      dir_y = (dir_y * vel) / 100.0;

      ball_x = ball_x + dir_x;
      ball_y = ball_y + dir_y;
   }
   ellipse (ball_x, ball_y, 10, 10);
}

void mousePressed() {
   if (mouseButton == LEFT){
      run = true;
```

```
    }else{
       if (mouseButton == RIGHT){
          run = false;
       }
    }
}

void mouseWheel(MouseEvent event) {
    float incr = event.getAmount();

    vel = vel + incr;
    if (vel < 1.0){
       vel = 1.0;
    }
    if (vel > 100.0){
       vel = 100.0;
    }
}
```

Podemos ver que en la función *draw* tenemos un bloque de código que calcula la posición de la bola:

```
    if (run == true){
       dir_x = mouseX - ball_x;
       dir_y = mouseY - ball_y;

       dir_x = (dir_x * vel) / 100.0;
       dir_y = (dir_y * vel) / 100.0;

       ball_x = ball_x + dir_x;
       ball_y = ball_y + dir_y;
    }
```

Lo que haremos será definir una nueva función que hará este trabajo:

```
void calculatePosition() {
    dir_x = mouseX - ball_x;
    dir_y = mouseY - ball_y;

    dir_x = (dir_x * vel) / 100.0;
    dir_y = (dir_y * vel) / 100.0;

    ball_x = ball_x + dir_x;
    ball_y = ball_y + dir_y;
}
```

Fijaos que esta función no devuelve ningún valor: simplemente se dedica a consultar y modificar las variables que habíamos definido desde el principio, y es el mismo código que teníamos dentro de la función *draw*. Ahora nos falta ejecutar esta nueva función desde donde antes teníamos este código:

```
void draw() {
   background(0, 0, 0);
   text ("Mouse position:\n  X: "+mouseX+"\n  Y: "+mouseY+"\n  Vel: "+vel,10, 20);

   if (run == true){
      calculatePosition();
   }
   ellipse (ball_x, ball_y, 10, 10);
}
```

Cómo podemos ver, dentro de la función *draw* donde antes estaba el código para calcular la posición ahora solo estamos llamando a la nueva función que hemos definido. Esto hace que la función *draw* sea más fácil de entender, puesto que en lugar de unos cálculos más o menos complejos, tenemos una función que con su nombre ya nos dice qué hace.

Otra ventaja es que podemos utilizar esta misma función desde lugares diferentes sin tener que copiar todo el código, y por lo tanto, si en algún momento hacemos un cambio se aplicará automáticamente a todos los lugares desde donde utilizamos la función.

6.1.3. Otros ejemplos

Ahora veremos algún ejemplo más, y para hacerlo, nos basaremos en este código:

```
int i;

void setup() {
   size(200, 100);
   frameRate (1);

   i = 1;
}

void draw() {
   background(0, 0, 0);
   text ("i value: "+i,10, 20);

   i = i +1;
}
```

Como podéis ver, es un programa muy sencillo que se dedica a incrementar el valor de una variable. A partir de aquí, veremos diferentes maneras de modificarlo para trabajar con funciones que reciban parámetros o no, y que devuelvan parámetros o no.

Función *plusOne*

Empezamos por el caso más fácil: crearemos una función que no recibirá ningún parámetro ni devolverá nada, y que se encargará de incrementar la variable *i*:

```
void draw() {
   background(0, 0, 0);
   text ("i value: "+i,10, 20);

   plusOne();
}

void plusOne(){
   i = i + 1;
}
```

Definimos la función indicando que no devuelve nada con *el void* delante del nombre de la función, y que no recibe ningún parámetro, puesto que no hay ninguno definido entre los paréntesis. Dentro de la función actualizamos directamente el valor de la variable *i*.

Lo último que tenemos que hacer es llamar a la función desde la función *draw* para que se ejecute e incremente el valor.

Función *plusX*

En este caso queremos crear una función que nos permita incrementar el valor de la variable con el número que queramos en cada momento; por lo tanto, nos interesará pasar este número como parámetro. Lo hacemos de la manera siguiente:

```
void draw() {
   background(0, 0, 0);
   text ("i value: "+i,10, 20);

   plusX(10);
}

void plusX (int x){
   i = i + x;
```

```
}
```

Fijaos cómo hemos definido el parámetro que recibe la función. Al ser un entero indicamos el tipo *int* y a continuación el nombre de la variable *x*. Y será esta variable la que sumaremos para incrementar el valor de *i*.

Para llamar a la función, lo único que tenemos que hacer es decir cuánto queremos incrementar el valor, y ya lo tenemos solucionado; en nuestro caso se irá incrementando de 10 en 10.

Función *addXY*

El último ejemplo será el de una función que recibe dos números y devuelve la suma de los dos; el código resultante queda así:

```
void draw() {
   background(0, 0, 0);
   text ("i value: "+i,10, 20);

   i = addXY (i, 10);
}

int addXY (int x, int y){
   int sum;
   sum = x + y;
   return sum;
}
```

En este caso, aparte de recibir dos variables, hemos definido que la función devuelva un entero (en lugar de *void* indicamos *int*), y al final de la función utilizamos el parámetro *return* para indicar que queremos volver el valor de la variable *sum*.

La llamada de la función también varía un poco; en este caso, como nos devuelve un valor deberemos asignarla a alguna variable (la *i*), y le pasamos los dos parámetros que queremos sumar (el valor actual de la *i y* el incremento que queremos aplicar).

6.2. Los *frames* y la gestión del flujo del programa

Al empezar a definir un programa lo primero a lo que hay que responder es qué queremos que haga, independientemente de cómo lo acabemos programando. Esta separación entre idea e implementación a veces es muy difícil de definir, sobre todo porque es muy fácil caer en la tentación de empezar a escribir el programa a pesar de no haber previsto todos los detalles.

Estas ansias por empezar a programar son un problema en cualquier tipo de programa, pero si intervienen elementos gráficos, estos problemas se pueden agravar, ya que el orden de cómo realizamos las tareas puede ser más crítico que cuando hagamos ciertas operaciones con datos.

Para intentar minimizar los posibles problemas, es muy interesante tener una descripción suficientemente detallada de lo que queremos conseguir, y aquí es donde los *frames* nos pueden ayudar.

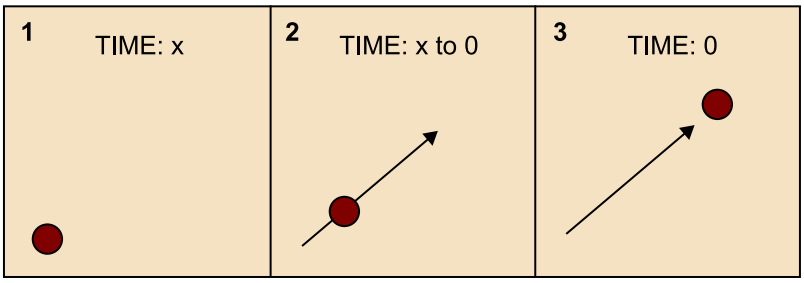
Podemos entender un *frame* como un momento concreto de nuestro programa, y se podría hacer una analogía con los *frames* que se utilizan con los programas de autoría, como por ejemplo, el Adobe Flash, en el que para cada *frame* definimos unos elementos gráficos y una serie de acciones que se pueden hacer en este momento determinado.

6.2.1. El guion animado y el diagrama de estados

Al planificar un programa será extraño si se puede resolver con un único *frame*, y es por eso por lo que, para agruparlos, utilizaremos un guion animado, del mismo modo que si estuviéramos preparando una película, y lo más importante es que contenga la descripción de todos los *frames* que componen nuestro programa.

Por ejemplo, supongamos que queremos hacer un programa en el que podremos definir el tiempo que una bola se estará moviendo, y que una vez finalice el movimiento, se quede quieta hasta que le indiquemos que vuelva a su punto inicial. El guion animado resultante podría ser similar a este:

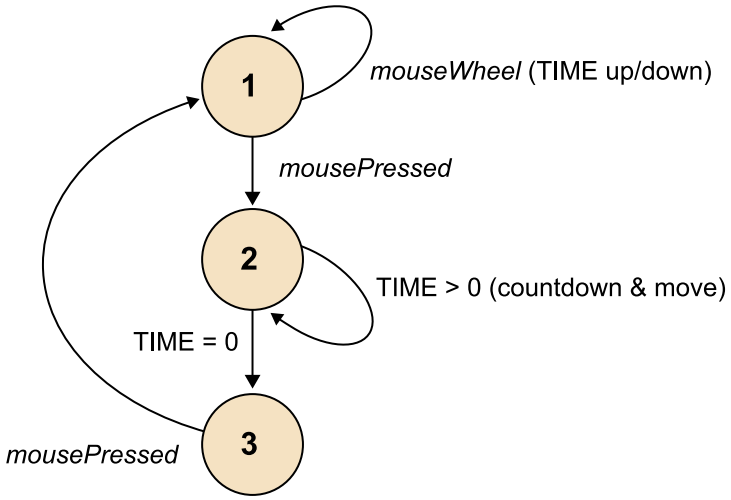
Figura 35. Guion animado



Fijaos que no hace falta que tenga gráficos increíbles, lo más importante es que nos describa los momentos clave de nuestro programa para que lo podamos tener en cuenta al crearlo.

Ahora, lo que nos hace falta es añadir qué podemos hacer en cada *frame* y cómo pasaremos de uno a otro, y para hacerlo, utilizaremos lo que se denominan *máquinas de estados*:

Figura 36. Máquina de estados



Lo que hemos hecho es coger cada *frame* y dibujarlo dentro de de un círculo. A continuación, con flechas, indicamos qué acciones o acontecimientos se pueden producir y qué consecuencias tienen. Ahora los veremos:

1) Estado o *frame* 1: la bola está quieta en la parte inferior izquierda de la ventana y podemos interactuar de dos maneras diferentes:

- Con la rueda del ratón modificamos el tiempo que querremos que se mueva la bola.
- Al hacer clic en el ratón saltaremos al *frame* 2.

2) Estado o *frame* 2: en este estado la bola se empezará a mover y se iniciarà la cuenta atrás.

- Mientras el tiempo pendiente sea superior a 0, la bola se continúa moviendo.
- Cuando el tiempo sea 0, saltaremos al estado siguiente.

3) Estado o *frame* 3: aquí la bola quedará quieta hasta que clicamos el ratón, momento en que volveremos al *frame* 1.

Con estos dos elementos tenemos claro qué queremos conseguir sin haber escrito ni una línea de código; por lo tanto, este sistema nos sirve tanto para casos como el Processing, en que tendremos que programar un código, como para casos en que tengamos una herramienta gráfica como el Adobe Flash, que nos lo permite hacer más visualmente. Lo importante de este trabajo previo es poder visualizar rápidamente nuestro programa para tener una referencia clara de lo que tenemos que hacer en cada momento, y así simplificar el paso siguiente, que es la codificación.

6.2.2. Del guion animado al código: la estructura *switch*

Ahora que tenemos el guion animado y la máquina de estados creados, nos hace falta un sistema para codificarlo de la manera más sencilla posible, y para hacerlo, utilizaremos una estructura de control que tiene ciertas similitudes con una que ya habíamos visto: *if...else*.

Con *if...else* podemos comprobar una condición y realizar una acción si se cumple u otra si no se cumple. Pero, ¿qué pasa si puede haber más opciones? Una solución es utilizar estructuras *if...else* imbricadas:

```
if (c == 1){
   //Do 1
}else{
   if (c == 2){
      //Do 2
   }else{
      if (c == 3){
         //Do 3
      }else{
         //Do 4
      }
   }
}
```

Pero este sistema nos puede generar estructuras muy cargadas y que a la larga compliquen la comprensión del código si no somos muy estructurados. La solución es utilizar la estructura de control *switch*:

```
switch(num) {
   case 0:
      println("Zero");
      break;
   case 1:
      println("One");
      break;
   default:
      println("None");
      break;
}
```

Con esta estructura antes que nada indicamos la variable que queremos comprobar, que en nuestro caso se denomina *num*. A partir de aquí siempre haremos lo mismo:

1) *caso X*: comprueba si la variable indicada es igual al valor X, y en caso afirmativo se ejecuta el código que hay a partir de este punto.

2) *break*: fijaos que la comprobación que hace *case* acaba con dos puntos en lugar de utilizar los corchetes *{}*, como es habitual. Por eso habrá que indicar al final el código que hemos de ejecutar dentro de cada *case* con la instrucción *break*. Si no lo hiciéramos, se continuarían ejecutando todas las líneas de código aunque formaran parte del *case* siguiente.

3) *default*: nos permite definir la acción por defecto que queremos hacer si no hemos encontrado ninguno *case* que cumpla con la condición.

A diferencia de la estructura *if...else*, podemos ver cómo tenemos un código mucho más claro, todo al mismo nivel, independientemente del número de condiciones que queramos comprobar, cosa que nos simplificará el seguimiento de lo que estamos haciendo en cada momento.

Ahora veremos cómo quedaría la estructura de control de nuestro guion animado:

```
int frame;

void setup() {
   size(400, 400);
   frameRate (20);

   frame = 1;
}

void draw() {
   background(0, 0, 0);

   switch (frame){
      case 1:
         //Stopped ball, time control, waiting click
         break;
      case 2:
         //Moving ball, countdown
         break;
      case 3:
         //Stopped ball, waiting click
         break;
   }
}
```

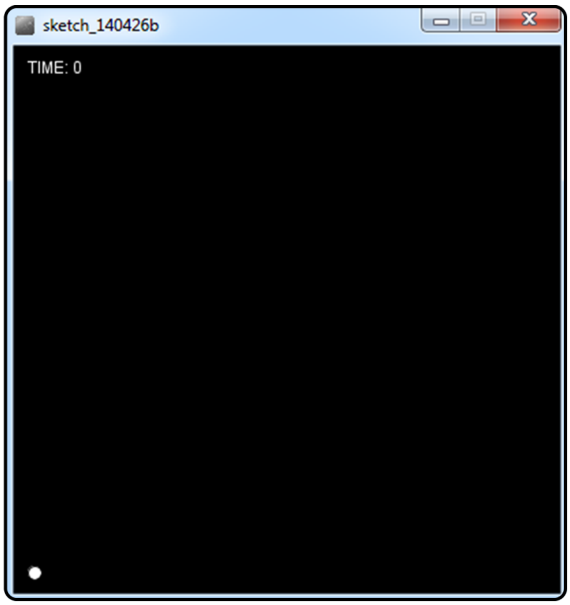
Definimos e inicializamos una variable llamada *frame* que utilizaremos para saber en cada momento en qué estado del programa nos encontramos, y a continuación traducimos nuestra máquina de estados en una estructura *switch*. De momento simplemente hemos definido la estructura, no hemos realizado ninguna acción, pero lo que es interesante es que ya tenemos un código estructurado que solo hay que completar con las acciones concretas.

Empezamos a completar el código, y lo primero que haremos será añadir los elementos gráficos que necesitamos:

```
int frame;
int ball_x;
int ball_y;
int time;
String message;
float crono;

void setup() {
   size(400, 400);
   frameRate (20);

   frame = 1;
   ball_x = 15;
   ball_y = height - 15;
   time = 0;
   message = new String ("TIME: ");
   crono = 0.0;
}

void draw() {
   background(0, 0, 0);

   switch (frame){
      case 1:
         //Stopped ball, time control, waiting click
         break;
      case 2:
      //Moving ball, countdown
         break;
      case 3:
         //Stopped ball, waiting click
         break;
   }

   ellipse (ball_x, ball_y, 10, 10);
   text (message+time, 10, 20);
```

```
}
```

Figura 37. Estructura básica de máquina de estados



Añadimos las variables para controlar la posición de la bola, el tiempo y el texto del mensaje que enseñaremos, y al final de la función *draw* dibujamos la bola y el texto.

Ahora iremos codificando lo que tenemos que hacer en cada uno de los estados o *frames*.

Estado 1

Lo primero que hemos definido es que con la rueda del ratón podamos modificar el tiempo que la bola se estará moviendo; por lo tanto, habremos de utilizar la función *mouseWheel*:

```
void mouseWheel(MouseEvent event) {
   float incr = event.getAmount();

   time = time + int(incr);
   if (time < 0){
      time = 0;
   }
}
```

A partir de la información que nos devuelve el ratón actualizamos la variable *time*, pero en este caso tenemos que hacer un trabajo extra. Nuestra variable es un número entero, mientras que la información del ratón es un número decimal, y por defecto no podemos sumar números de tipos diferentes.

Para solucionarlo utilizamos la función *int*, que se encarga de transformar el número decimal en un entero, y de esta manera sí que podemos actualizar correctamente nuestra variable.

Si ejecutáis el programa podréis comprobar cómo ya se actualiza el tiempo en la ventana y que además nunca podrá ser negativo, puesto que también hemos añadido esta comprobación a nuestro código.

Ahora tenemos que detectar cuándo clicamos el ratón para ir al estado siguiente. Para hacerlo, habíamos visto que existía la función *mousePressed*, pero en este caso utilizaremos una variable que también se llama igual (*mousePressed*) y que contiene la información de si se ha pulsado algún botón del ratón; lo haremos de esta manera:

```
void draw() {
   background(0, 0, 0);

   switch (frame){
      case 1:
         //Stopped ball, time control, waiting click
         if (mousePressed){
            crono = time;
            frame = 2;
         }
         break;
      case 2:
         //Moving ball, countdown
         break;
      case 3:
         //Stopped ball, waiting click
         break;
   }

   ellipse (ball_x, ball_y, 10, 10);
   text (message+time, 10, 20);
   text (frame, width-20, 20);
}
```

Dentro *del case 1*, comprobamos si esta variable es cierta, y en caso afirmativo decimos que el estado actual ya no es el número 1, sino que será el 2, y además inicializaremos el cronómetro con el tiempo que tenemos seleccionado.

Aparte, hemos añadido otro texto en la parte superior derecha de la ventana, que nos indica el plano actual, y de esta manera tendremos una indicación mientras estemos haciendo el programa.

Estado 2

Cuando llegamos a este estado quiere decir que hemos seleccionado un tiempo y que hemos de iniciar el movimiento de la bola, y por lo tanto, añadiremos el código para controlar el tiempo:

```
void draw() {
   background(0, 0, 0);

   clock();

   switch (frame){
      case 1:
         //Stopped ball, time control, waiting click
         if (mousePressed){
            crono = time;
            frame = 2;
         }
         break;
      case 2:
         //Moving ball, countdown
         countdown();
         time = int (crono);
         break;
```

Añadimos dos funciones nuevas, *clock*, que se encarga de contar el tiempo que ha pasado entre cada ejecución de la función *draw*, y la función *countdown*, que actualizará el tiempo que nos falta para parar la bola.

Utilizamos funciones extras para no sobrecargar la función *draw* y que continúe siendo lo máximo de clara posible. Una vez actualizado el tiempo, hacemos lo mismo con la variable *time*, que es la que utilizamos para pintar por pantalla.

La función *clock* es esta:

```
void clock() {
   time_now = millis();
   time_delta = time_now - time_old;
   time_old = time_now;
}
```

Cómo podéis ver, es el mismo código que ya habíamos utilizado anteriormente. L'única diferencia es que ahora el código está en una función aparte.

La función *countdown* es muy simple:

```
void countdown() {
   crono = crono - (time_delta / 1000.0);
}
```

Si ejecutáis el código veréis que el tiempo va disminuyendo, pero al llegar a cero no se para y continúa con tiempos negativos. Esta es una condición que ya habíamos definido en nuestra máquina de estado, y ahora la implementaremos:

```
case 2:
         //Moving ball, countdown
         countdown();
         time = int (crono);
         if (time <= 0){
            frame = 3;
         }
         break;
```

Lo único que ha hecho falta ha sido indicar que si el contador llega a cero saltamos al estado 3, y como en este estado no descontamos el tiempo, este ya no aparece con números negativos.

Ahora ya solo nos falta que la bola se mueva por la pantalla, y para hacerlo, crearemos otra función que se encargará de moverla, a la que llamaremos desde el *case*, como hemos hecho con el cronómetro.

```
case 2:
         //Moving ball, countdown
         countdown();
         time = int (crono);
         move_ball();
         if (time <= 0){
            frame = 3;
         }
         break;
```

```
void move_ball() {
   ball_x = ball_x + 1;
   ball_y = ball_y - 1;
}
```

Parte de la gracia de crear funciones extras es que las podemos reutilizar desde varios lugares del programa, y si en algún momento quisiéramos cambiar el tipo de movimiento de la bola, solo habría que modificar esta función para aplicarlo automáticamente a todos los lugares que utilicen esta función.

Si volvemos a ejecutar el código veréis cómo mientras estamos en el estado 1 podemos modificar el tiempo, y al clicar el ratón saltamos al estado 2. En este estado la bola se empieza a desplazar y el tiempo decrece hasta llegar a cero, momento en que pasamos al estado 3 automáticamente.

Estado 3

Este estado es muy simple: estaremos esperando que el usuario pulse algún botón del ratón. Cuando lo haga diremos que el estado vuelve a ser el inicial y podremos volver a controlar el tiempo que queremos que la bola se desplace.

```
case 3:
         //Stopped ball, waiting click
         if (mousePressed){
            frame = 1;
            ball_x = 15;
            ball_y = height - 15;
         }
         break;
```

Y después de codificar este último estado ya tenemos el programa hecho. Hemos comenzado definiendo qué queríamos hacer y hemos ido avanzando paso por paso para poder construir el programa de una manera ordenada. A pesar de que no es la única manera de programar, sí que es muy interesante tener alguna metodología que nos obligue de alguna manera a ser ordenados desde el punto de vista del diseño del programa (guion animado y máquina de estados), y de la codificación (funciones).

7. Trabajar con imágenes

Es el momento de añadir un poco de alegría a nuestros programas, y lo haremos utilizando imágenes. Pensad qué cantidad de elementos gráficos hay en cualquier programa actual: iconos, fondos, cursores, elementos animados, etc.

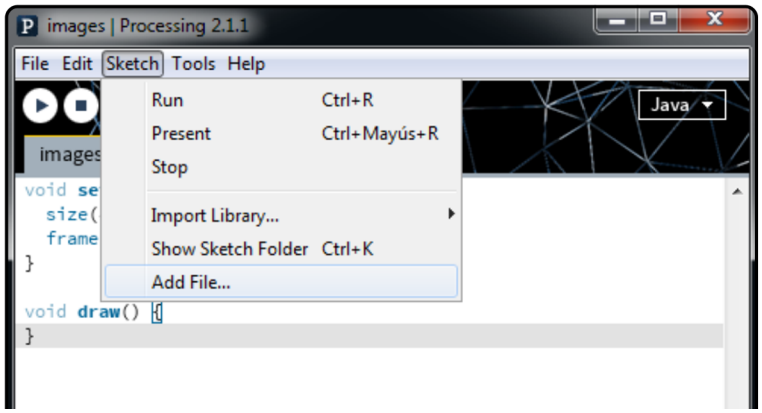
7.1. Elementos básicos

Como siempre, empezamos con un código de base para poder hacerlo evolucionar poco a poco. Si lo ejecutáis veréis que se abre una ventana de color gris:

```
void setup() {
   size(800, 331);
   frameRate (20);
}

void draw() {
}
```

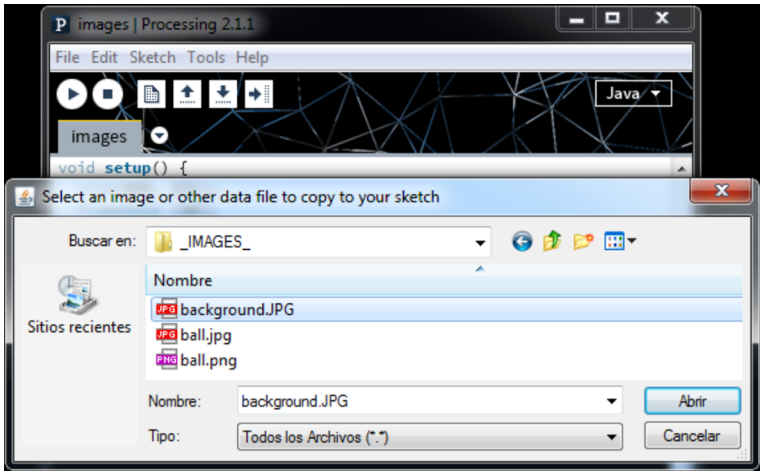
Lo siguiente que tenemos que hacer para poder utilizar imágenes es cargarlas en el editor del Processing. Esto lo hacemos yendo al menú *Sketch*, opción *Add File*.

Figura 38. Cargar imagen (1)



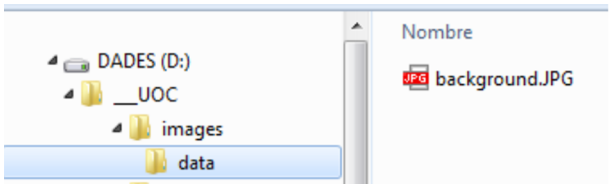
Y a continuación seleccionamos la imagen que queremos cargar:

Figura 39. Cargar imagen (2)



Esto lo que hace es copiar la imagen dentro del directorio donde está guardado el código, concretamente dentro de una carpeta llamada *data*. Si no queremos hacer este paso utilizando los menús también podemos hacer la copia de las imágenes manualmente en esta carpeta.

Figura 40. Cargar imagen (3)



Aquí podemos ver la carpeta donde está el código, que se llama *images*, y contiene la carpeta *data*, que es donde estarán las imágenes y el resto de elementos que podamos necesitar (sonidos, vídeos...).

Ahora que tenemos el fichero de la imagen en el lugar correcto para poder acceder a él, la cargaremos y la utilizaremos en nuestro programa. Modificamos el código de esta manera:

```
PImage img;

void setup() {
   size(800, 331);
   frameRate (20);

   img = loadImage ("background.JPG");
}

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

Figura 41. Cargar una imagen de fondo

Analizamos los cambios que hemos añadido:

1) Definimos una variable del tipo *Pimage*. Este tipo de dato nos permite almacenar imágenes y hacer operaciones básicas.

2) Inicializamos la variable que hemos creado utilizando la función *loadImage*. Esta función carga la imagen suponiendo que el fichero se encuentra en el directorio *data*.

Es muy importante indicar el nombre del fichero teniendo en cuenta las mayúsculas y minúsculas, puesto que si no coincide exactamente, nos dará un error y no lo cargará.

También es importante cargar las imágenes dentro de la función *setup*, puesto que el proceso puede tardar un poco, y si lo hiciéramos después mientras estamos interactuando con el programa, podría haber momentos de lentitud.

3) Para pintar la imagen en la ventana utilizamos la función *image*, y le pasamos la variable que contiene la imagen y la posición de la esquina superior izquierda de esta. En nuestro caso, la imagen es de la misma medida que la de la ventana que hemos definido.

7.2. Tipo de imágenes que podemos utilizar

Los tipos o formatos que nos pueden interesar más son estos tres:

- **JPG**: Es el formato que utilizaremos más, nos permite cargar imágenes y mantener una medida de fichero reducida.

- **GIF**: Puede ser útil para imágenes pequeñas con detalles muy finos como los de los iconos, por ejemplo, pero el uso más interesante es que podremos cargar imágenes animadas y ver esta animación al ejecutar el programa.

- **PNG**: este formato nos permite trabajar con imágenes con zonas transparentes, cosa que nos puede dar bastante juego. Veremos un ejemplo a continuación.

7.3. Añadimos una bola

Cargaremos otra imagen y la añadiremos al código de esta manera:

```
PImage img;
PImage ball;

void setup() {
   size(800, 331);
   frameRate (20);

   img = loadImage ("background.JPG");
   ball = loadImage ("ball.jpg");
}

void draw() {
   image (img, 0, 0);
   image (ball, 20, 20);
}
```

Figura 42. Una bola "maquillada"

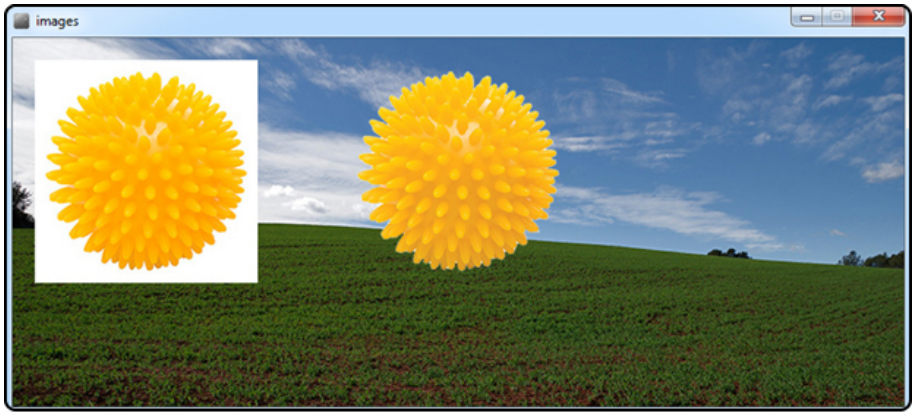
Por fin tenemos una bola más interesante que las que habíamos utilizado hasta ahora. Podríamos modificar todos los ejemplos que habíamos visto hasta ahora sustituyendo el círculo que dibujábamos para la función *image* y continuarían funcionando igual pero con unos gráficos mejorados.

Está claro que tenemos un pequeño problema, puesto que si la imagen es rectangular, los márgenes se adaptan perfectamente, pero en este caso nos aparece un margen que nos molesta. Ahora lo solucionaremos:

```
PImage img;
```

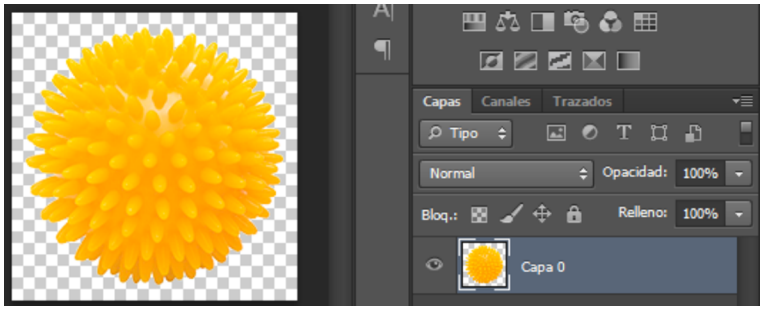
```
PImage ball;
PImage ball_alfa;

void setup() {
   size(800, 331);
   frameRate (20);

   img = loadImage ("background.JPG");
   ball = loadImage ("ball.jpg");
   ball_alfa = loadImage ("ball.png");
}

void draw() {
   image (img, 0, 0);
   image (ball, 20, 20);
   image (ball_alfa, 300, 20);
}
```

Figura 43. Imágenes transparentes



Qué diferencia, ¿no? Hemos cargado otra imagen con formato PNG que hemos preparado con el Photoshop. En este caso, lo que hemos hecho es editar la imagen para que quede en una capa con el fondo transparente. Este es el aspecto de la imagen vista desde el Photoshop:

Figura 44. Preparamos la imagen con el Photoshop

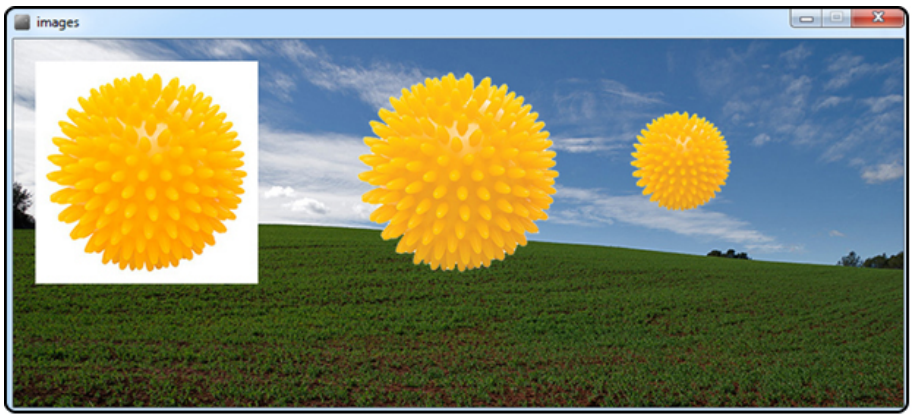


Ahora si utilizamos esta imagen la integración con el fondo es perfecta.

Para finalizar, modificaremos las dimensiones en las que pintamos esta imagen:

```
void draw() {
   image (img, 0, 0);
   image (ball, 20, 20);
   image (ball_alfa, 300, 20);
   image (ball_alfa, 550, 60, 100, 100);
}
```

Figura 45. Redimensionar la imagen



Hemos pintado la misma imagen transparente, pero esta vez en lugar de indicar solo el nombre de la variable y la posición, hemos añadido dos parámetros que nos permiten definir la medida horizontal y vertical de la imagen.

En nuestro ejemplo hemos indicado una medida de 100 × 100, y la imagen se ve correctamente porque es cuadrada, pero no hay nada que nos prohíba indicar una medida de 200 × 100, por ejemplo, y obtener una imagen deformada. Os animamos a que lo probéis.

8. Transformaciones básicas

Ahora que ya sabemos cómo podemos generar formas básicas y utilizar imágenes, es posible que tengamos que adaptarlas a ciertas situaciones. Por ejemplo, cuando desplazamos la bola quizás nos gustaría que oscilara un poco como si fuera una pelota de plástico.

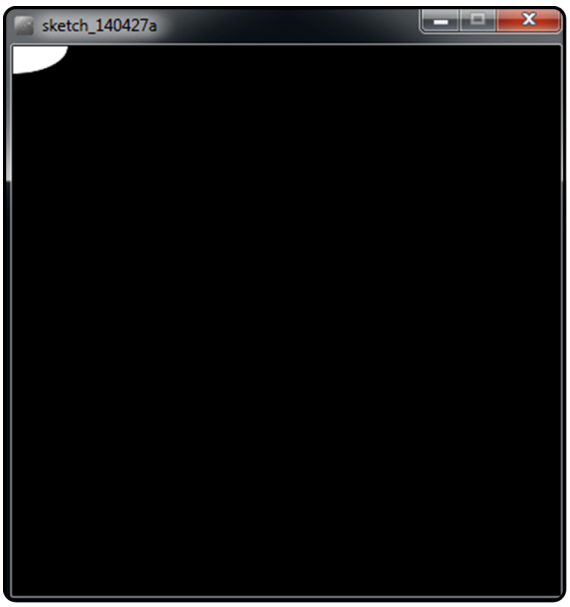
Para dar solución a esta necesidad y a otras, podemos utilizar las transformaciones.

8.1. Transformaciones

Veremos tres transformaciones básicas: rotación, traslación y escala, y para hacerlo utilizaremos este código como punto de partida, en el que dibujamos una elipse centrada en el punto (0, 0):

```
void setup() {
   size(400, 400);
   frameRate (20);
}

void draw() {
   background(0, 0, 0);
   ellipse (0, 0, 80, 40);
}
```

Figura 46. Código base de transformaciones

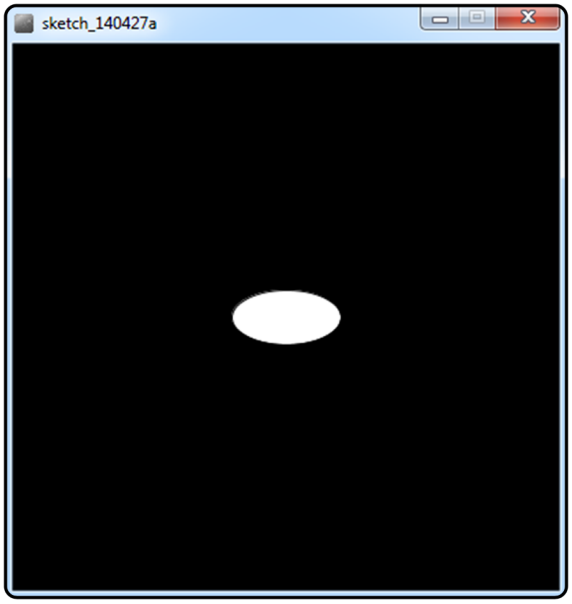


8.1.1. Traslación

Lo primero que haremos será desplazar la elipse en el centro de la ventana, y para hacerlo, utilizaremos la función *translate* de esta manera:

```
void draw() {
   background(0, 0, 0);
   translate ((width/2), (height/2));
   ellipse (0, 0, 80, 40);
}
```

Figura 47. Traslación



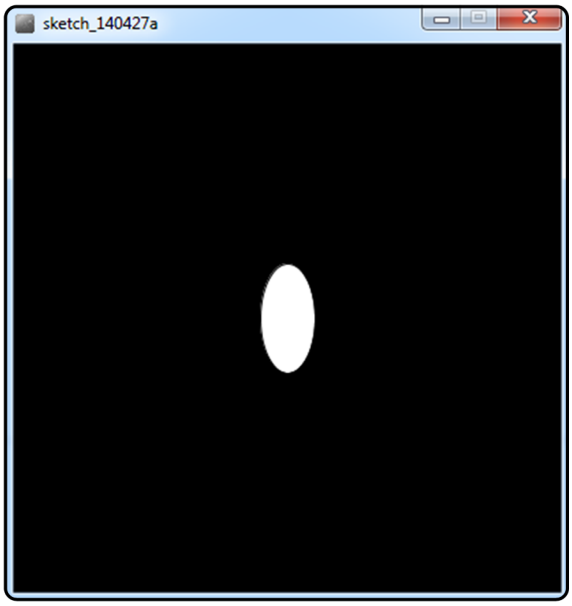
A la función solo tenemos que indicarle el desplazamiento que queremos aplicar a los elementos que dibujaremos; en nuestro caso hemos dicho que se desplazara el equivalente a la mitad de la anchura de la ventana y la mitad de la altura; por lo tanto, la figura ha quedado centrada. Una vez ejecutada esta función en todo lo que dibujamos en la ventana, se le aplicará la misma traslación.

8.1.2. Rotación

Veamos el código directamente:

```
void draw() {
   background(0, 0, 0);
   translate ((width/2), (height/2));
   rotate (HALF_PI);
   ellipse (0, 0, 80, 40);
}
```

Figura 48. Rotación



En este caso utilizamos la función *rotate* e indicaremos el ángulo que queremos rodar en grados radianes. Nosotros hemos seleccionado π 1/2, es decir, 90º. Si queréis utilizar grados, podéis hacer la conversión con la función radianes o la inversa con la función *degree*:

```
Xradians = radians (Ydegree);
Ydegree = degrees (Xradians);
```

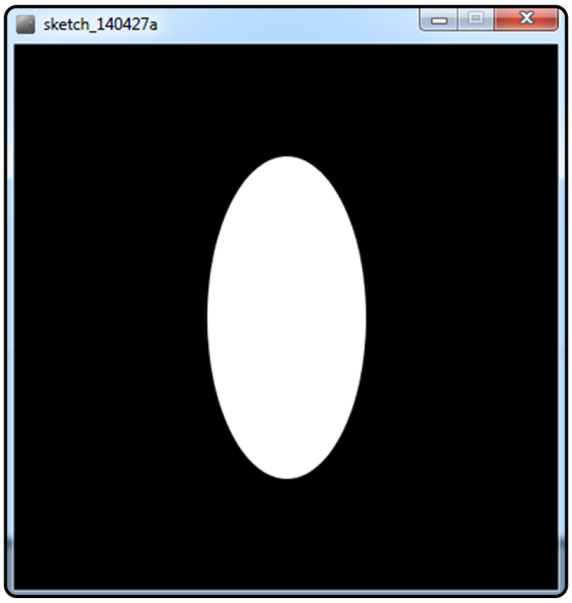
Una cosa que tenemos que tener en cuenta es que las rotaciones siempre cogen como punto de referencia las coordenadas (0, 0) y esto puede hacer que a veces no tengamos el resultado que nos esperaríamos *a priori*, pero veremos todo ello con más detalle algo más adelante.

8.1.3. Escala

Para finalizar, veremos la función *scale*, que nos permite hacer un escalado proporcional (igual para el eje *x* e *y*) o aplicar diferentes factores a cada eje para deformar los objetos:

```
void draw() {
   background(0, 0, 0);
   translate ((width/2), (height/2));
   rotate (HALF_PI);
   scale (3.0);
   ellipse (0, 0, 80, 40);
}
```

Figura 49. Escala



En este caso hemos hecho un escalado con un factor 3 (3 veces más grande) y que afecta a los dos ejes por igual. El mismo resultado lo podríamos haber obtenido con este código:

```
scale (3.0, 3.0);
```

La diferencia es que ahora hemos indicado la escala de los ejes por separado, y por lo tanto podríamos haber aplicado factores diferentes a cada uno.

Igual que en el caso de la rotación, la escala se aplica tomando como referencia el origen de coordenadas (0, 0), factor que tendremos que tener en cuenta al utilizar la transformación.

8.2. El problema de la referencia al origen de coordenadas

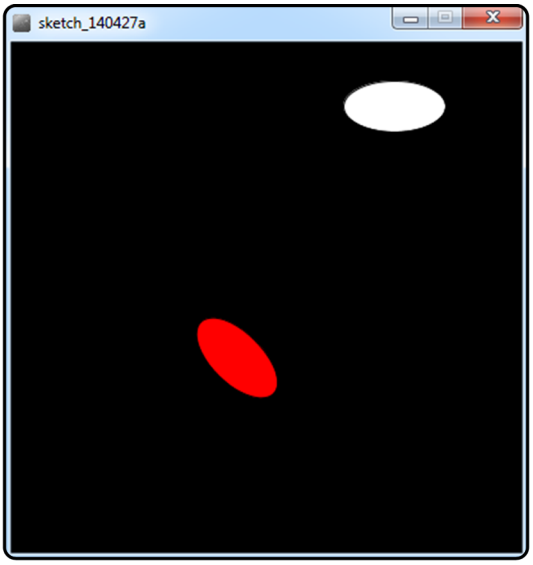
Como hemos dicho, las transformaciones de rotación y escala utilizan como referencia el origen de coordenadas, y en realidad la traslación también lo hace. Pero, ¿qué quiere decir esto?

Para verlo miramos un ejemplo:

```
void setup() {
   size(400, 400);
   frameRate (20);
}

void draw() {
   background(0, 0, 0);
   fill (255, 255, 255);
   ellipse (300, 50, 80, 40);
```

```
    fill (255, 0, 0);
    rotate (HALF_PI/2);
    ellipse (300, 50, 80, 40);
}
```

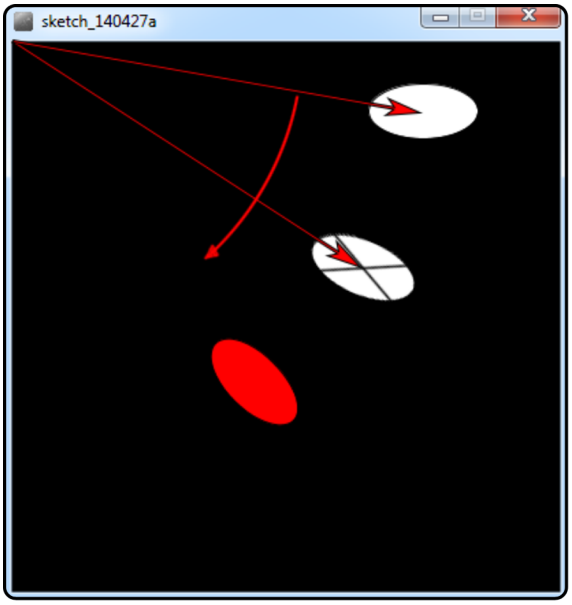
Figura 50. Transformaciones y origen de coordenadas (1)



En el código podemos ver cómo hemos dibujado dos elipses de la misma medida y en la misma posición; la única diferencia es que a la roja se le ha aplicado una rotación. El problema es que esta rotación seguramente no es la que esperábamos, y quizás nos habría gustado que la figura quedara girada en el mismo lugar, ¿no?

En realidad lo que ha pasado es que se ha rodado teniendo como origen de la transformación las coordenadas (0, 0):

Figura 51. Transformaciones y origen de coordenadas (2)



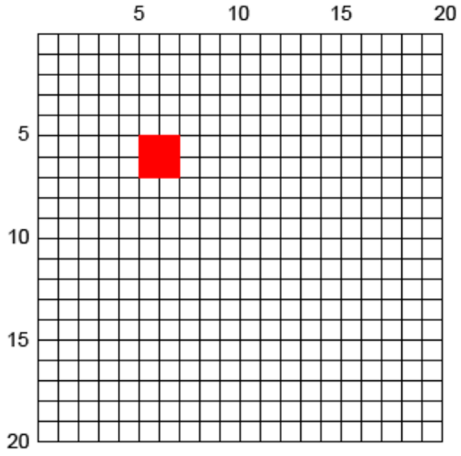
Si tenemos claro que el punto sobre el cual se rueda es el origen de coordenadas, sí que se puede entender el resultado que hemos obtenido, pero esto nos plantea una pregunta: ¿cómo podemos conseguir el resultado que esperábamos?

La solución es mover el origen de coordenadas y situarlo en el punto que queremos usar de referencia para la rotación, y para hacer esto, utilizamos la función *translate*. Qué extraño, ¿no habíamos dicho que esta función se utilizaba para desplazar lo que dibujábamos?

Pues sí, el resultado es que al utilizar *translate*, los objetos aparecen desplazados, pero no es porque los cambien las coordenadas, sino porque ha modificado el origen de coordenadas que utiliza para dibujarlos. Intentamos aclararlo con un ejemplo paso por paso; primero creamos un cuadrado:

```
rect (5, 5, 2, 2);
```

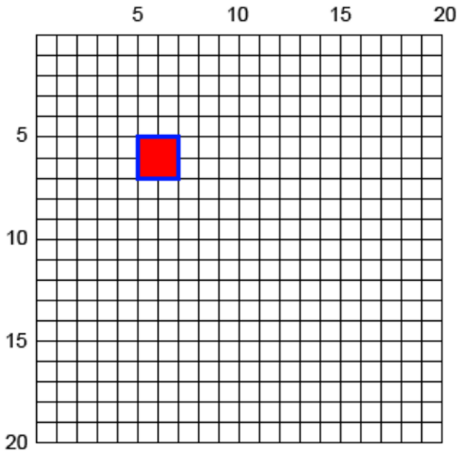
Figura 52. Transformaciones y origen de coordenadas (3)



Hasta aquí ningún misterio, ahora crearemos otro en el mismo sitio, pero en lugar de rellenarlo con un color sólido solo lo contornearemos de color azul:

```
void draw() {
   background(0, 0, 0);
   fill (255, 0, 0);
   stroke (255, 0, 0);
   rect (5, 5, 2, 2);

   noFill ();
   stroke (0, 0, 255);
   rect (5, 5, 2, 2);
}
```

Figura 53. Transformaciones y origen de coordenadas (4)

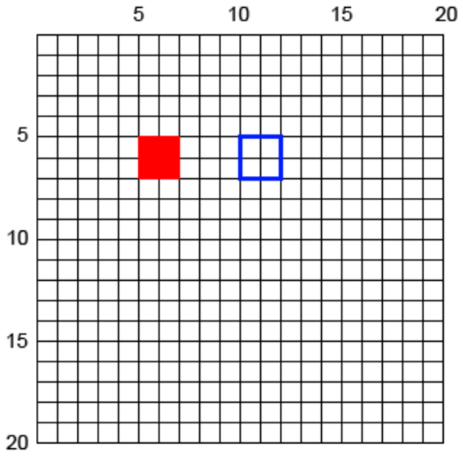


De acuerdo, ahora desplazaremos el segundo recuadro con la función *translate*:

```
void draw() {
   background(0, 0, 0);
   fill (255, 0, 0);
```

```
    stroke (255, 0, 0);
    rect (5, 5, 2, 2);

    translate (5, 0);
    noFill ();
    stroke (0, 0, 255);
    rect (5, 5, 2, 2);
}
```

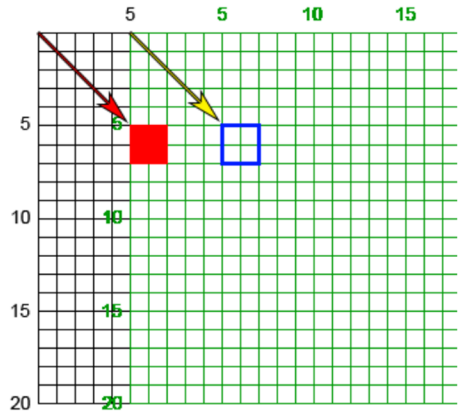
Figura 54. Transformaciones y origen de coordenadas (5)



El cuadrado se ha dibujado en la posición (10, 5), pero fijaos que nosotros lo continuamos dibujando en la posición (5, 5). En realidad, en el momento en que dibujamos el cuadrado hacemos dos cosas:

- Desplazamos el origen de coordenadas a la posición (5, 0).
- Dibujamos el cuadrado teniendo en cuenta las nuevas coordenadas de referencia.

Figura 55. Transformaciones y origen de coordenadas (6)

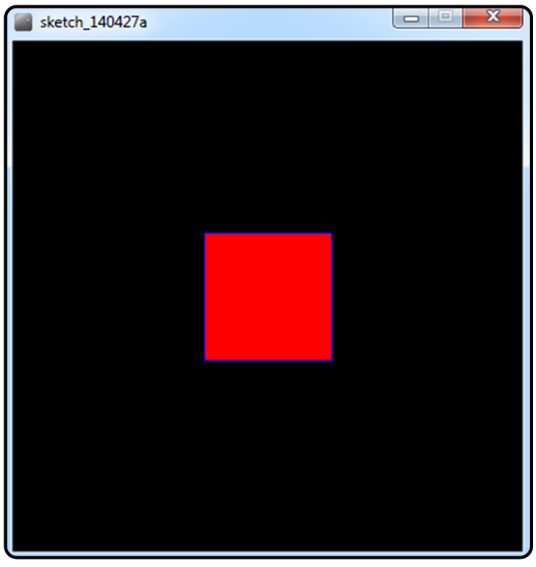


Con esta imagen lo podemos ver. El cuadro rojo utiliza las coordenadas (0, 0) originales (flecha roja), pero para dibujar el azul, primero hemos desplazado el origen de coordenadas (representadas por la matriz de color verde) y después hemos dibujado el cuadrado azul con las nuevas coordenadas de referencia (flecha amarilla).

Repetimos el proceso utilizando la rotación, y empezamos con este código:

```
void setup() {
   size(400, 400);
   frameRate (20);
}

void draw() {
   background(0, 0, 0);
   fill (255, 0, 0);
   stroke (255, 0, 0);
   rect (150, 150, 100, 100);

   noFill ();
   stroke (0, 0, 255);
   rect (150, 150, 100, 100);
}
```

Figura 56. Transformaciones y origen de coordenadas (7)

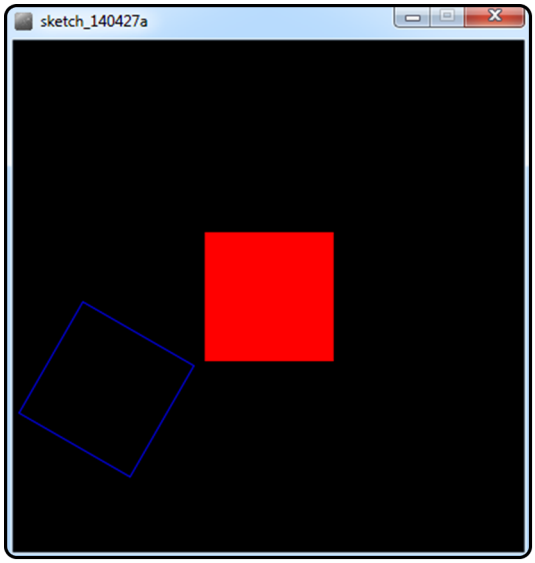


Si rodamos el cuadro azul 30 grados sin tener las coordenadas en cuenta, obtenemos esto:

```
noFill ();
stroke (0, 0, 255);
rotate (radians (30));
```

```
rect (150, 150, 100, 100);
```

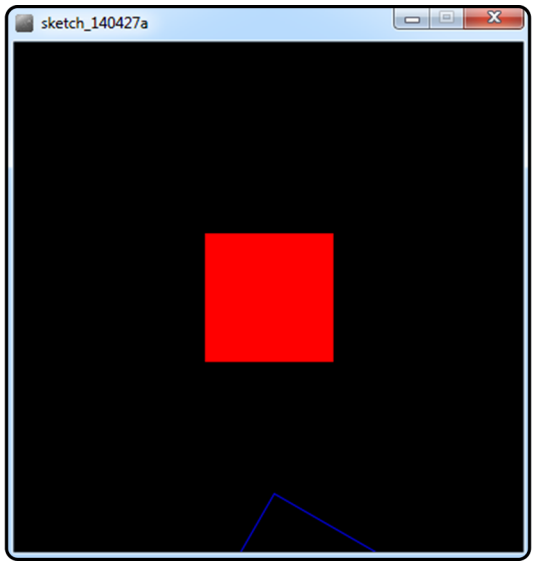
Figura 57. Transformaciones y origen de coordenadas (8)



Para solucionarlo, primero habrá que desplazar el origen de coordenadas al punto que queremos usar como centro de la rotación:

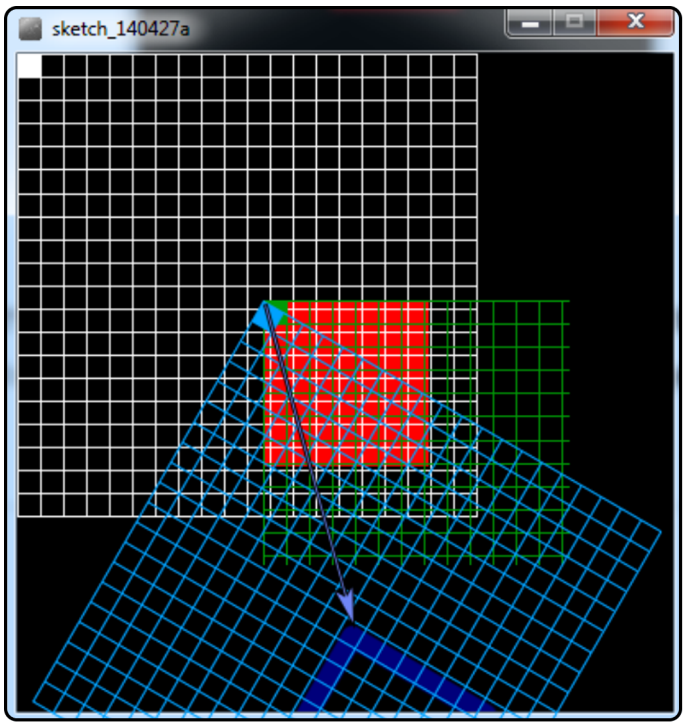
```
noFill ();
stroke (0, 0, 255);
translate (150, 150);
rotate (radians (30));
rect (150, 150, 100, 100);
```

Figura 58. Transformaciones y origen de coordenadas (9)



Pero, ¿qué ha pasado? ¿Por qué nos aparece el cuadrado tan abajo si solo lo hemos rodado? La respuesta es cómo ha quedado situado el origen de coordenadas y dónde hemos dibujado el cuadrado después:

Figura 59. Transformaciones y origen de coordenadas (10)

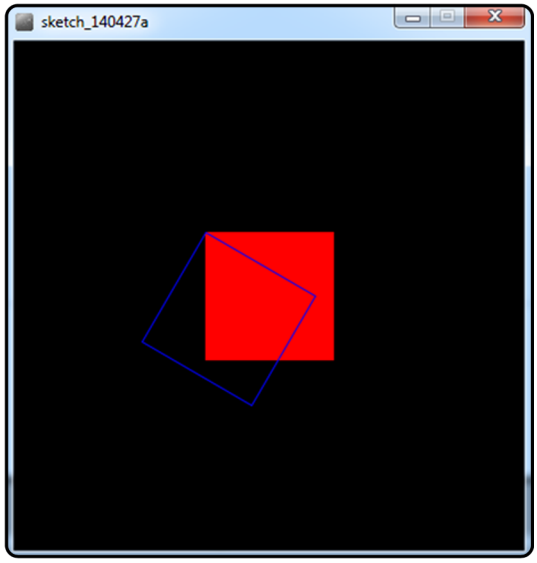


Antes que nada, hemos modificado las coordenadas originales (color blanco) y las hemos desplazado a la nueva posición (color verde), y después las hemos rodado 30 grados (color azul).

Ahora hemos dibujado el cuadrado en la posición (150, 150) de las nuevas coordenadas, pero estas coordenadas ya estaban en el lugar donde queríamos que estuviera dibujado el cuadrado, y por lo tanto, lo que habrá que hacer es dibujarlo en las coordenadas (0, 0):

```
noFill ();
stroke (0, 0, 255);
translate (150, 150);
rotate (radians (30));
rect (0, 0, 100, 100);
```

Figura 60. Transformaciones y origen de coordenadas (11)



Ahora sí, por fin tenemos el cuadrado rodado de la manera que queríamos.

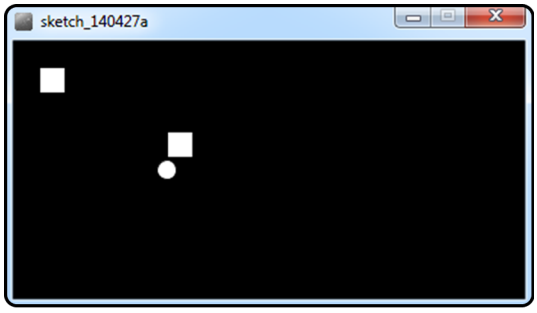
Este sistema, que para dibujar un par de cuadrados parece muy complejo, hace que cuando se trabaja con muchos elementos que dependen unos de otros, se simplifique bastante el trabajo. Es por eso por lo que tanto con el Processing como con cualquier otro sistema de programación gráfica os encontráis con soluciones de este estilo.

8.3. Ámbito de las transformaciones

Cuando trabajamos con transformaciones tenemos que tener en cuenta que en el momento en que aplicamos alguna, esta afectará a todos los elementos que se dibujen a partir de este momento. Miramos este código:

```
void setup() {
   size(400, 200);
   frameRate (20);
}

void draw() {
   background(0, 0, 0);
   rect (20, 20, 20, 20);
   translate (100,50);
   rect (20, 20, 20, 20);
   ellipse (20, 50, 15, 15);
}
```

Figura 61. Ámbito de las transformaciones



Hemos dibujado un cuadrado en la posición (20, 20) y después hemos dibujado otro que queríamos desplazar 100 puntos a la derecha y 50 puntos abajo, pero el círculo que hemos añadido después queríamos que estuviera en la posición (20, 50) y también ha aparecido desplazado.

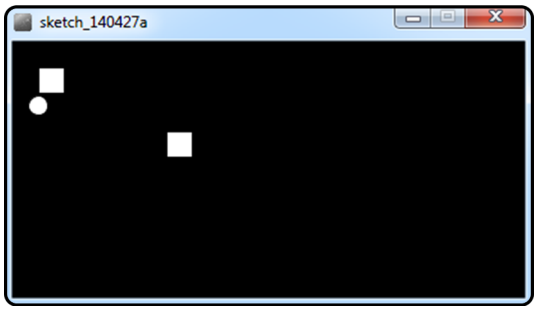
Esto es porque, desde el momento en que hemos aplicado la traslación, todo queda afectado, recordad que en realidad hemos modificado el origen de coordenadas. Para solucionar esto tenemos diferentes opciones.

8.3.1. Ordenación de los elementos en función de las transformaciones

Este sistema se basa en aplicar y dibujar los elementos en función de las transformaciones que hay que aplicar; en nuestro ejemplo el código quedaría de esta manera:

```
void draw() {
   background(0, 0, 0);
   rect (20, 20, 20, 20);
   ellipse (20, 50, 15, 15);
   translate (100,50);
   rect (20, 20, 20, 20);
}
```

Figura 62. Ordenación de las transformaciones



Con un ejemplo tan simple como este, el resultado es correcto; el problema es que no siempre podremos intercambiar el orden en el que pintamos los elementos (recordad el algoritmo del pintor). También hace que no tengamos

los elementos que dibujamos por grupos lógicos que nos simplifiquen la interpretación del código, sino una agrupación en función de unas transformaciones que hay que aplicar.

8.3.2. Invertir los efectos aplicados

Esta es una mejor solución, puesto que en lugar de reorganizar el código para adaptarlo a las transformaciones que aplicamos, desactivaremos las transformaciones que ya no queremos utilizar. Con el ejemplo se verá más claro:

```
void draw() {
   background(0, 0, 0);
   rect (20, 20, 20, 20);
   translate (100,50);
   rect (20, 20, 20, 20);

   translate (-100, -50);
   ellipse (20, 50, 15, 15);
}
```

Volvemos a dibujar el círculo al final del código, pero en este caso justo antes hemos aplicado la traslación inversa. Con esta acción hacemos que el origen de coordenadas se vuelva a desplazar a su situación original en la esquina superior izquierda de la ventana, y por lo tanto, el círculo aparece donde queríamos.

A pesar de que este sistema es mucho mejor que el anterior, en casos en los que hay múltiples transformaciones y muchos elementos para dibujar, puede ser un poco difícil de utilizar, sobre todo porque hay que hacer un seguimiento muy acotado de todas las transformaciones que hacemos y después invertir sus efectos en el orden correcto.

8.3.3. La pila de transformaciones

Para solucionar la complejidad de tener que recordar todos los efectos aplicados y el orden en el que hay que deshacerlos, tenemos la pila de transformaciones. La idea es que en esta pila podremos almacenar diferentes orígenes de coordenadas con sus transformaciones asociadas y después recuperarlos.

Para poder trabajar con la pila utilizaremos dos funciones:

- *pushMatrix*: Guardará la situación actual en la pila, con todas las transformaciones que haya aplicadas.

- *PopMatrix*: Recuperará el último estado que habíamos guardado, de manera que dejaremos de aplicar las transformaciones que teníamos definidas ahora, y volveremos a utilizar las anteriores.

Veamos un ejemplo:

```
void draw() {
   background(0, 0, 0);
   rect (20, 20, 20, 20);

   pushMatrix();
   translate (100,50);
   rect (20, 20, 20, 20);

   popMatrix();
   ellipse (20, 50, 15, 15);
}
```

Si ejecutáis el código veréis que se están dibujando los tres elementos tal como queríamos. Primero dibujamos el cuadrado en la posición (20, 20) y a continuación guardamos el estado actual, que no tiene ninguna transformación aplicada, en la pila.

A continuación aplicamos una traslación y dibujamos el segundo cuadrado, que, como es natural, aparece desplazado.

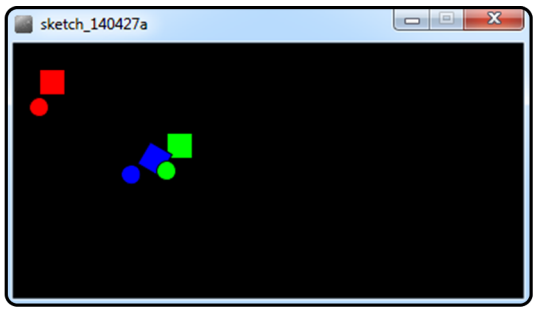
Ahora es el momento en que queremos dibujar el círculo sin ninguna transformación, antes hemos tenido que deshacer la traslación pero ahora podemos recuperar el estado anterior con *el popMatrix*. Utilizando esta función, lo que hacemos es recuperar el estado que habíamos guardado (sin transformaciones) y utilizarlo para continuar dibujando más elementos.

En la pila podemos ir poniendo y quitando tantos estados como queramos, lo único que tenemos que tener en cuenta es que siempre se tiene que hacer en orden, es decir, al hacer *el popMatrix* siempre recuperaremos el último que habíamos puesto. Miramos un ejemplo con unos cuantos elementos más:

```
void draw() {
   background(0, 0, 0);
   fill (255, 0, 0);
   rect (20, 20, 20, 20);

   pushMatrix();
   translate (100,50);
   fill (0, 255, 0);
   rect (20, 20, 20, 20);

   pushMatrix ();
   fill (0, 0, 255);
   rotate (radians(30));
   rect (20, 20, 20, 20);
```

```
    ellipse (20, 50, 15, 15);

    popMatrix();
    fill (0, 255, 0);
    ellipse (20, 50, 15, 15);

    popMatrix();
    fill (255, 0, 0);
    ellipse (20, 50, 15, 15);
}
```

Figura 63. Pila de transformaciones



Aparte de dibujar más elementos, se han codificado con colores en función de las transformaciones que tienen aplicadas para poder ver mejor cómo funciona la pila:

- Color rojo para las figuras sin ninguna transformación aplicada.
- Color verde para las figuras que solo tienen la traslación.
- Color azul para las que tienen la traslación y la rotación.

Si seguimos el código, podemos ver la evolución siguiente:

1) Dibujamos el primer cuadrado rojo sin ninguna transformación.

2) Guardamos el estado en la pila (ninguna transformación).

a) Nos desplazamos 100 puntos a la derecha y 50 abajo.

b) Dibujamos el cuadrado verde.

c) Guardamos el estado en la pila (traslación).

- Aplicamos una rotación de 30º, que se suma al desplazamiento que ya teníamos aplicado.
- Dibujamos el cuadrado azul.
- Dibujamos el círculo azul.
- Recuperamos el último estado que tenemos en la pila.

d) Volvemos a tener únicamente la traslación aplicada.

e) Dibujamos el círculo verde.

f) Recuperamos el último estado que tenemos en la pila.

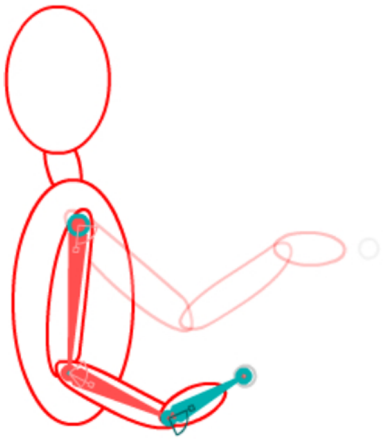
3) Volvemos a la situación de no tener ninguna transformación aplicada.

4) Dibujamos el círculo rojo.

Con este sistema podemos tener un control más sencillo de las transformaciones aplicadas, incluso si el programa que se está realizando no es complejo.

Un ejemplo en el que es prácticamente obligatorio utilizar la pila de transformaciones es con las cadenas cinemáticas (*kinematic chain*), que, por ejemplo, nos permitirían representar un brazo.

Figura 64. Cadena cinemática



Este es el ejemplo típico. Para representar el brazo tenemos tres elementos: brazo, antebrazo y mano. Cuando el brazo gira, el antebrazo y la mano también lo tienen que hacer, y por lo tanto, se ha de aplicar una transformación que les afecte a todos, pero si solo se mueve la mano, los otros no se ven afectados.

Con la pila de transformaciones, del mismo modo que hemos dibujado los cuadrados y círculos de diferentes colores en función de las transformaciones aplicadas, podríamos controlar la posición en cada momento de todos los elementos del brazo e iríamos a un nivel o al otro de la pila, en función del trozo de brazo que estuviéramos representando.

9. Bibliotecas

El Processing es un lenguaje que nos ofrece la oportunidad de crear una infinidad de programas para solucionar cualquier necesidad que podamos tener. Naturalmente, en función de la complejidad de lo que queramos obtener, el código necesario también lo será más o menos.

En algunos casos concretos, cuando aparecen necesidades para las cuales el lenguaje no tiene una solución bastante práctica o directamente no tiene ninguna solución, tenemos disponibles las bibliotecas.

Una biblioteca es una colección de código que añade funcionalidades que el lenguaje básico no tiene. Por ejemplo, el Processing nos permite dibujar rectángulos, pero no tiene ninguna función que dibuje una manzana. Una manera de solucionar esto sería crear una biblioteca que dispusiera de la función para dibujar una manzana, y de esta manera, en lugar de tenerla que dibujar con las primitivas que tiene el Processing por defecto (líneas, elipses, rectángulos...), la podríamos crear directamente, quizás con una llamada similar a esta:

```
apple (20, red);
```

Las bibliotecas que utilizamos también están programadas en Processing, o en Java, que es el lenguaje en el que se basa, y por lo tanto, también lo habríamos podido codificar nosotros. La ventaja de disponer de las bibliotecas es que las podemos hacer accesibles a cualquier programa que estemos creando y que disponen de una API (*application programming interface*) muy definida, que nos informa de lo que podemos hacer y de cómo lo tenemos que hacer, del mismo modo que tenemos la referencia de Processing.

En este apartado haremos la introducción básica a dos bibliotecas que nos pueden ser de utilidad para dar un valor añadido a nuestros programes de una manera más sencilla:

- *Minim*: Es una biblioteca que por defecto ya está cargada y que nos permite trabajar con ficheros de audio.

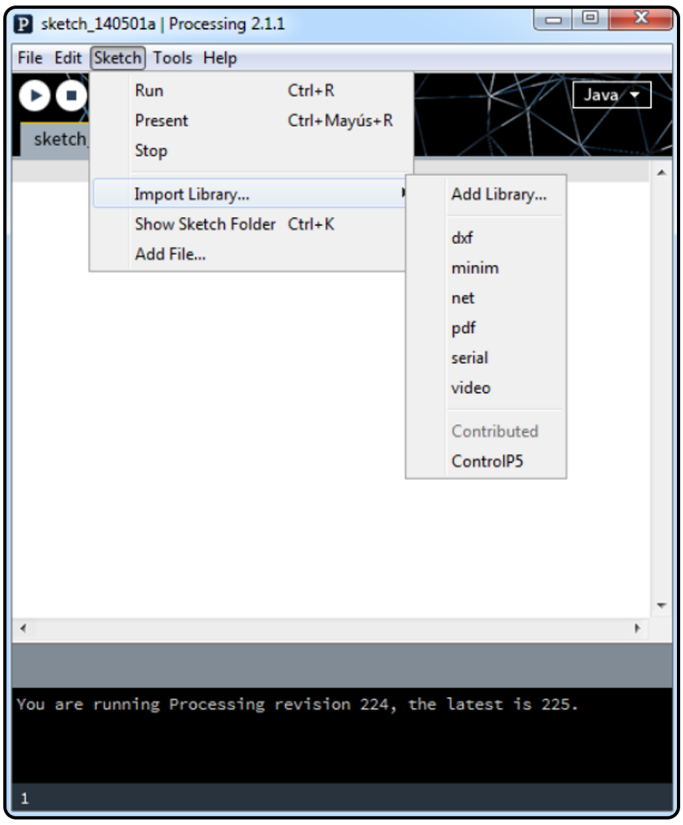
- *ControlP5*: Esta biblioteca no está cargada por defecto y nos permite crear rápidamente interfaces gráficas con botones, desplegables, etc.

Para saber qué bibliotecas están disponibles, podéis consultar la lista en la web de Processing, donde además encontraréis la descripción y cómo hay que utilizarlas.

9.1. Gestión de bibliotecas

Para saber qué bibliotecas tenemos disponibles, podemos ir al menú *Sketch*, *Import Library*, y encontraremos una lista de las que hay instaladas y podremos utilizar.

Figura 65. Lista de bibliotecas



En la imagen podéis ver que están las bibliotecas *Minim* y *ControlP5*, pero si lo comprobáis en vuestro PC, no encontraréis la *ControlP5*, y por lo tanto, será necesario ver cómo se instala.

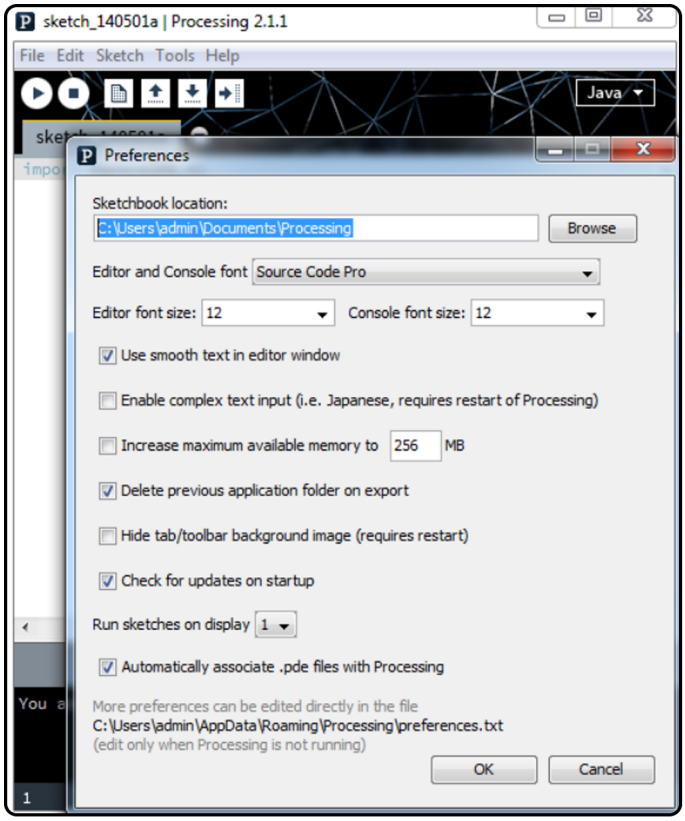
9.1.1. Instalación de la biblioteca *ControlP5*

Desde la página web donde está la lista de todas las bibliotecas, seleccionaremos la *ControlP5* y saltaremos directamente a la web oficial de esta biblioteca. En esta web, aparte de encontrar la biblioteca que hay que instalar, también encontraremos la referencia para poder utilizarla, ejemplos y mucha documentación para poder ver cómo funciona y lo que podemos llegar a hacer.

De momento nos bajaremos la última versión disponible (el fichero en formato ZIP que hay en la zona de descargas), y descomprimiremos el fichero en un directorio temporal. Al hacerlo, veréis que aparece un directorio llamado *controlP5*, que es la biblioteca, y un fichero de texto que explica cómo hay que instalarla.

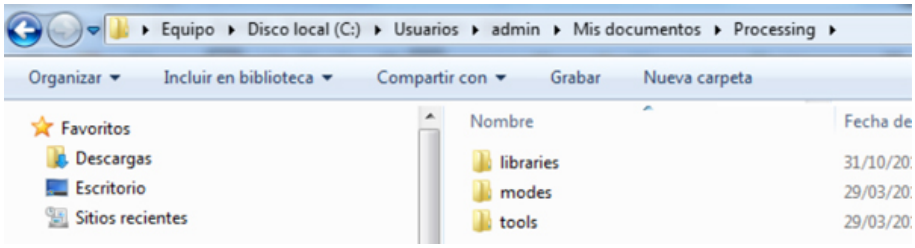
El proceso es tan sencillo como copiar el directorio dentro de la carpeta donde el Processing busca las bibliotecas, información que podemos encontrar si vamos a las preferencias del Processing:

Figura 66. Instalación de bibliotecas (1)



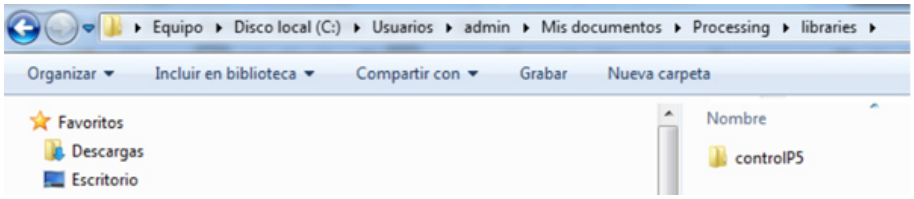
Copiamos la ruta del *sketchbook location* y vamos al explorador de ficheros:

Figura 67. Instalación de bibliotecas (2)



Aquí encontraremos el directorio *libraries*, que contiene las bibliotecas que instalaremos nosotros; entramos en el directorio y copiamos el de la biblioteca que hemos descomprimido:

Figura 68. Instalación de bibliotecas (3)



Una vez hecho esto, solo hay que reiniciar el programa para que cargue las nuevas bibliotecas y ya las podremos empezar a utilizar.

Para utilizar una biblioteca solo hay que ir a la lista de las que tenemos disponibles y seleccionarla, y veréis cómo automáticamente se añadirá un código en vuestro programa:

```
import controlP5.*;
```

Este es el aspecto que tiene con la biblioteca *controlP5*, y sirve para indicar en el programa que, aparte de las funcionalidades que tiene por defecto, tendrá acceso a otras que están contenidas dentro de la biblioteca que hemos indicado.

9.2. Biblioteca *Minim*

Como hemos dicho, esta biblioteca nos permite trabajar con sonidos. A continuación veremos los pasos básicos para poder cargar y reproducir uno.

Empezamos con este código:

```
void setup()
{
   size(400, 200);
}

void draw()
{
```

Lo primero que haremos será indicar que queremos utilizar la biblioteca y, por lo tanto, iremos al menú *Sketch*, *Import Library* y seleccionaremos la biblioteca *Minim*. Al hacerlo veréis que el código queda de esta manera:

```
import ddf.minim.spi.*;
import ddf.minim.signals.*;
import ddf.minim.*;
import ddf.minim.analysis.*;
import ddf.minim.ugens.*;
import ddf.minim.effects.*;

void setup()
{
   size(400, 200);
}

void draw()
{
```

Ha añadido una serie de llamadas *import* para cargar todas las funcionalidades de la biblioteca, y a partir de este punto, ya podemos empezar a utilizarla.

La mejor manera de saber qué podemos hacer es consultar directamente la web del creador de la biblioteca, donde, como hemos dicho, encontraremos toda la referencia y ejemplos. En nuestro caso simplemente cargaremos un fichero y lo reproduciremos; para hacerlo, necesitaremos dos variables:

```
Minim minim;
AudioPlayer player;
```

La primera es la que utilizaremos para acceder a las funcionalidades de la biblioteca, y la segunda es la que nos permitirá cargar y reproducir los ficheros de audio (de manera similar a como lo hacemos con las imágenes). Ahora las inicializaremos:

```
void setup()
{
   size(400, 200);

   minim = new Minim(this);
   player = minim.loadFile("buzzer.mp3");
}
```

Lo más interesante de este código es la manera como cargamos el fichero de audio, y el hecho de que antes de utilizarlo deberemos importar el fichero *buzzer.mp3* al *sketch* tal como hacemos con las imágenes (recordad: menú *sketch*, *add file*).

Ahora que ya tenemos el fichero cargado, solo hay que reproducirlo:

```
void setup()
{
   size(400, 200);

   minim = new Minim(this);
   player = minim.loadFile("buzzer.mp3");

   player.play();
}
```

Al ejecutar el programa sonará una sirena. Fijaos que el orden de reproducción del sonido está dentro de la función *setup*. Lo hemos hecho así porque queremos que el sonido se reproduzca una sola vez; si estuviera dentro de la función *draw*, empezaría una reproducción cada vez que se ejecutara la función *draw*. Probad a modificar el código y dejarlo así:

```
void draw()
{
   player.play();
}
```

Qué dolor de cabeza, ¿no? Quizás hemos visto algo antes que nos puede servir para mejorarlo:

```
void mousePressed() {
   player.play();
}
```

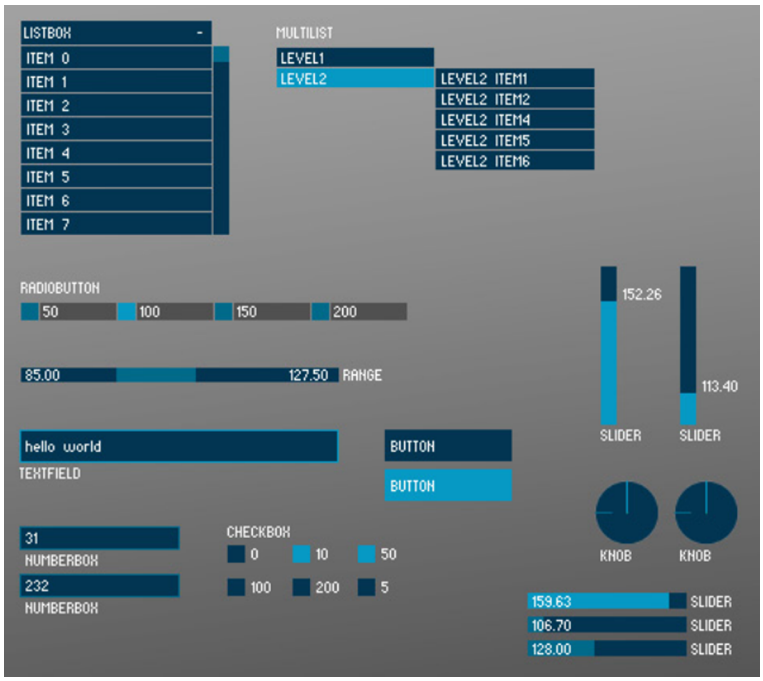
Mucho mejor, al dejar el código así solo sonará la alarma al pulsar algún botón del ratón.

Esta biblioteca tiene muchas más opciones, acordaos de echar un vistazo a la referencia oficial para verlas.

9.3. Biblioteca *ControlP5*

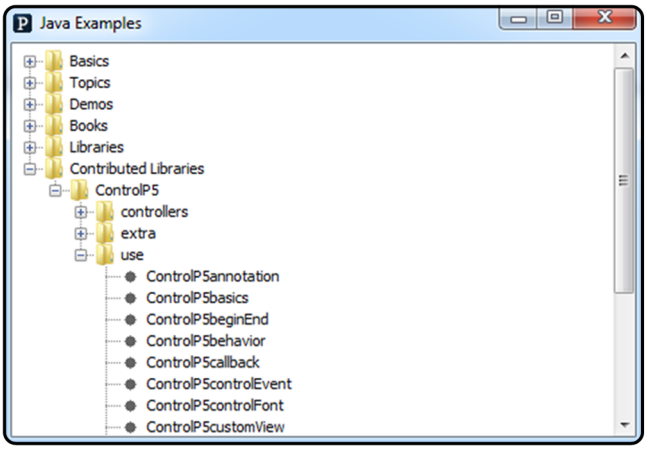
Esta biblioteca nos permite diseñar interfaces gráficas para nuestros programas de una manera bastante sencilla. A continuación tenemos una imagen en la que podemos ver los diferentes controles que nos ofrece:

Figura 69. Controles de la biblioteca *ControlP5*



Una cosa muy interesante de esta biblioteca es que tiene disponibles ejemplos básicos de todos sus elementos, y ya los tenéis instalados en vuestro PC por el hecho de haber instalado la biblioteca. Id al menú *File*, *Examples*:

Figura 70. Ejemplos disponibles



En esta lista encontraremos estos códigos de ejemplo, y como podéis ver, no solamente son de esta biblioteca, sino que hay muchos más.

A continuación veremos un programa básico en el que añadiremos un botón y que nos servirá para ver la estructura básica que podemos utilizar para construir nuestras interfaces. Como siempre, empezaremos con un código básico:

```
void setup()
{
   size(400, 200);
}

void draw()
{
   background (0, 0, 0);
}
```

El paso siguiente es indicar que utilizaremos la biblioteca *ControlP5*:

```
import controlP5.*;
```

Y ahora ya la podemos empezar a utilizar. Antes que nada definiremos e inicializaremos la variable que contendrá todos los elementos de la interfaz que queramos añadir:

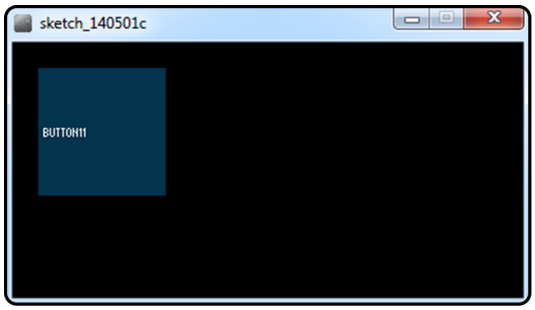
```
import controlP5.*;

ControlP5 userinterface;

void setup()
{
   size(400, 200);

   userinterface = new ControlP5 (this);
```

```
}
```

A partir de ahora utilizaremos la variable *userinterface* como referencia para añadir y modificar la interfaz que estamos creando. Añadiremos el botón:

```
void setup()
{
   size(400, 200);

   userinterface = new ControlP5 (this);

   userinterface.addButton ("button11", 1, 20, 20, 100, 100);
}
```

Figura 71. *ControlP5*: el primer botón



Al ejecutar el código nos aparece el botón que acabamos de definir, donde hemos indicado el nombre, el valor asociado que tendrá el botón (se puede considerar como un identificador), la posición (x, y) y la medida.

Lo siguiente que tenemos que hacer es detectar cuándo han pulsado el botón, y para hacerlo, añadiremos una función nueva:

```
void controlEvent(ControlEvent theEvent) {

   if(theEvent.isController()) {

      println("control event from : "+theEvent.controller().name()+", value :
                                    "+theEvent.controller().value());

      if(theEvent.controller().name().equals("button1")) {
         println ("BUTTON PRESSED");
      }
   }
}
```

Vamos por partes:

1) Función *controlEvent*: Esta función se encarga de escuchar todos los acontecimientos o acciones que se hacen sobre los elementos que pertenecen al usuario. Por ejemplo, si se mueve una ventana, se desplaza el ratón o clicamos el botón.

En el caso del ratón, teníamos funciones específicas (por ejemplo, *mousePressed*) que hacían que no fuera necesario utilizar esta función, pero en el caso de *controlP5*, esta será la mejor manera de gestionar lo que está pasando.

2) Comprobación *if(theEvent.isController())*: Con esta comprobación filtramos todos los acontecimientos que llegan y solo gestionamos los que son de tipo *controller*, que son los asociados a los elementos de nuestra interfaz.

3) Información que recibimos: Una vez que sabemos que el acontecimiento es de la interfaz, enseñamos la información del nombre y el valor que tiene en la ventana de mensajes del Processing. Esto lo hacemos simplemente para poder hacer un seguimiento de lo que está pasando mientras hacemos las pruebas.

4) Comprobar quién ha generado el event: Esta es la parte importante. Con la comprobación *if(theEvent.controller().name().equals("button1"))* miramos cuál de los elementos que hemos creado ha generado el acontecimiento. En nuestro caso solo tenemos el botón, de manera que comprobamos si quien ha generado el acontecimiento se llama *button1*.

Si tuviéramos más elementos, añadiríamos tantos bloques de comprobación *if(theEvent.controller().name().equals("nombre_elemento"))* como fuera necesario.

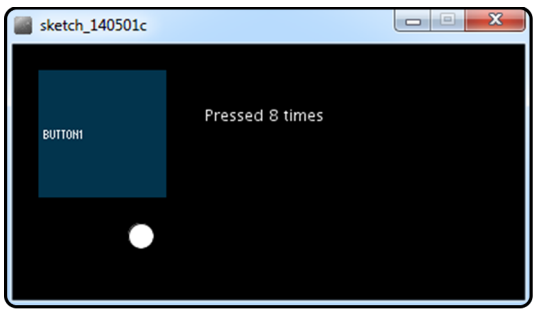
5) Acción para hacer: Dentro de cada bloque asociado a cada elemento estará el código que hay que ejecutar al utilizar este elemento; en nuestro caso escribimos un mensaje de texto diciendo que hemos pulsado el botón. Si el código que hay que ejecutar es un poco complejo, será muy recomendable crear una función aparte y llamarla desde aquí para simplificar la comprensión del código y tenerlo mejor organizado.

Ahora que tenemos el botón creado y sabemos cuándo lo pulsamos, añadimos un poco de código para que pase algo más interesante:

```
import controlP5.*;

ControlP5 userinterface;

int ball_x;
int count;

void setup()
{
```

```
    size(400, 200);

    userinterface = new ControlP5 (this);
    userinterface.addButton ("button1", 1, 20, 20, 100, 100);

    ball_x = 20;
    count = 0;
}

void draw()
{
    background (0, 0, 0);

    text ("Pressed "+count+" times", 150, 60);
    ellipse (ball_x, 150, 20, 20);
}

void controlEvent(ControlEvent theEvent) {
    if(theEvent.isController()) {
        println("control event from : "+theEvent.controller().name()+", value :
                                        "+theEvent.controller().value());

        if(theEvent.controller().name().equals("button1")) {
            println ("BUTTON PRESSED");
            pressedButton1();
        }
    }
}

void pressedButton1(){
    ball_x = ball_x + 10;
    count = count + 1;
}
```

Figura 72. *ControlP5*: botón funcional



Este es un ejemplo muy básico en el que podemos ver la estructura que utilizaremos para trabajar con esta biblioteca. Teniendo en cuenta que el objetivo del *controlP5* es simplificar la creación de interfaces, nos puede parecer que el

código que hemos tenido que utilizar es bastante complejo, y es cierto, pero es mucho más sencillo que tener que crear y gestionar estos elementos gráficos nosotros mismos.

El código que hemos creado utiliza las opciones más básicas de un botón para ver la interacción más básica que podemos obtener, y a partir de aquí, es muy interesante ver los ejemplos disponibles para poder sacar el máximo provecho a esta biblioteca.

Por ejemplo, aquí tenéis el aspecto de una práctica hecha con el Processing por un compañero que ha cursado la asignatura de *Integración digital de contenidos*, en la que cada uno de los elementos de control se ha elaborado con esta biblioteca:

Figura 73. *ControlP5*: Ejemplo práctico