El código del *plug-in* VisDa paso a paso

Javier Melenchón

PID_00201066

UOC
Universitat Oberta de Catalunya

Índice

1. Introducción

En este capítulo, vamos a explicar el código de dibujado o *rendering* de la aplicación. Ofreceremos dos versiones del mismo: una versión que se puede ejecutar de forma independiente y otra que se encuentra integrada con el *plug-in* propuesto. Los objetivos son triples: mostrar la construcción del código de dibujado, conocer las posibilidades básicas de procesos de forma práctica y conocer las partes principales de un código con interacción y animaciones.

La explicación se hace sobre la versión independiente y, al final, comentaremos cómo integrarlo con el *plug-in*. De esta manera, conseguimos que el lector pueda ir probando el código mientras se le anima a construirlo. Construiremos el código de forma incremental empezando por un ejemplo muy sencillo hasta llegar a tener todo el código entero con todas las funcionalidades y características. Durante esta construcción, también mostraremos diferentes características básicas de Processing que vale la pena conocer.

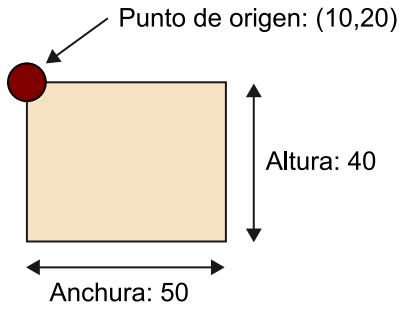
Os recomendamos ir probando cada trozo de código propuesto para poder comprobar su funcionamiento. El desarrollo se ha realizado mediante la versión 1.5.1 de Processing, la más estable del momento. Al final de cada trozo de código, se adjunta un identificador entre paréntesis y antecedido por dos contrabarras: este identificador indica el *sketch* de Processing asociado al código para que lo podáis abrir y ejecutar. Podréis acceder a las distintas carpetas con los códigos descargando el fichero “recursos_visda.zip”.

2. Dibujar y rellenar formas

Dibujar en Processing es muy sencillo. Basta con escribir una línea para poder dibujar, por ejemplo, un cuadrado o un círculo:

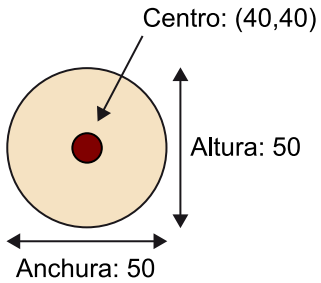
```
rect(10,20,50,40);
//(_1_010)
```

Figura 1. Dibujo de un cuadrado




```
ellipse(40,40,50,50);
//(_1_020)
```

Figura 2. Dibujo de un círculo



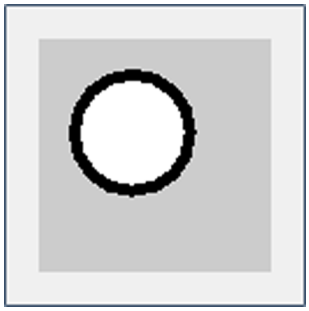
Recordad que una elipse con una anchura y altura iguales es un círculo.

2.1. El contorno de la forma

Podemos cambiar las diferentes propiedades de las formas anteriores con la misma simplicidad. Por ejemplo, para cambiar el grueso del contorno del círculo anterior, solo tenemos que especificar la siguiente línea antes de dibujarlo.

```
strokeWeight(5);
ellipse(40,40,50,50);
//(_1_030)
```

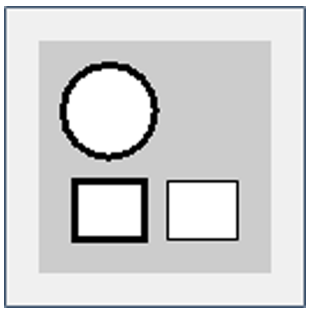
Figura 3. Círculo con contorno grueso



Debemos tener en cuenta que, cuando se cambia el grosor, todo lo que se dibuje a continuación se pintará con el grosor especificado. Si queremos dibujar otra figura con un grueso anterior, lo tenemos que volver a especificar.

```
strokeWeight(3);
ellipse(30,30,40,40);
rect(15,60,30,25);

strokeWeight(1);
rect(55,60,30,25);
//(_1_040)
```

Figura 4. Diferentes formas con diferente grosor de contorno



Así, para no escribir tantas líneas, intentaremos dibujar todas las figuras con características (en este ejemplo, de grosor de contorno) similares de forma seguida.

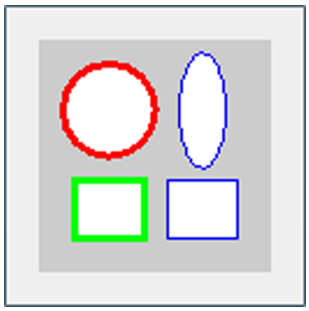
2.2. Colores

Otra característica que se puede cambiar es el color, tanto del relleno como de la línea:

```
strokeWeight(3);
stroke(#FF0000);
ellipse(30,30,40,40);
stroke(#00FF00);
rect(15,60,30,25);
```

```
strokeWeight(1);
stroke(#0000FF);
rect(55,60,30,25);
ellipse(70,30,20,50);
//(_1_050)
```

Figura 5. Figuras con contornos de colores y gruesos diferentes

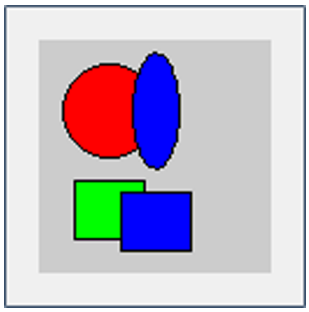


El color lo hemos especificado con formato hexadecimal, el típico formato que solemos encontrar en las páginas web. En el último ejemplo, dado que tenemos dos figuras que queremos pintar con contorno delgado y de color azul, las hemos pintado seguidas, para tener que especificar el grosor y el color una sola vez.

También podemos especificar el color del relleno. Para no hacer el código demasiado largo, pondremos todos los contornos iguales. Si no se especifica ningún contorno, se toma el negro y el grosor 1 por defecto:

```
fill(#FF0000);
ellipse(30,30,40,40);
fill(#00FF00);
rect(15,60,30,25);
fill(#0000FF);
rect(35,65,30,25);
ellipse(50,30,20,50);
//(_1_060)
```

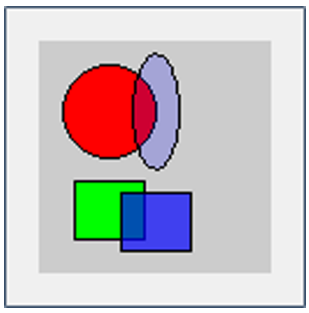
Figura 6. Figuras con rellenos de diferentes colores



Los objetos se dibujan después de que otros se muestren encima. Si queremos aplicar transparencias a los colores, solo tenemos que añadir un segundo valor a la función `fill` entre 0 (totalmente transparente) y 255 (totalmente opaco):

```
fill(#FF0000);
ellipse(30,30,40,40);
fill(#00FF00);
rect(15,60,30,25);
fill(#0000FF,175);
rect(35,65,30,25);
fill(#0000FF,50);
ellipse(50,30,20,50);
//(_1_070)
```

Figura 7. Figuras con rellenos de diferentes colores y diferentes transparencias

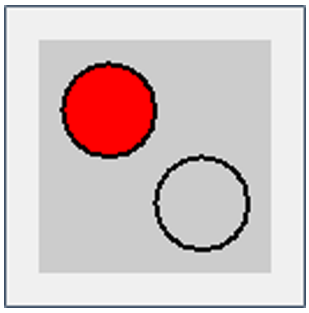


En este caso, hemos añadido otra instrucción de `fill`, ya que queríamos que fuera mayor el nivel de transparencia para la elipse que para el rectángulo. Se puede ver que la zona solapada del círculo rojo de fondo se ve más que la del rectángulo verde.

También es posible no rellenar los objetos para obtener, por ejemplo, una circunferencia:

```
strokeWeight(2);
fill(#FF0000);
ellipse(30, 30,40,40);


noFill();
ellipse(70,70,40,40);
//(_1_080)
```

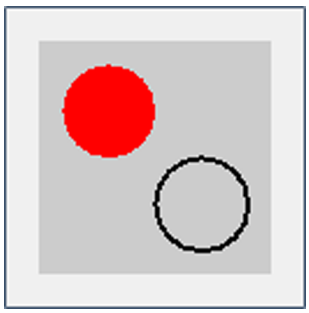
Figura 8. Un círculo rojo con contorno negro y una circunferencia negra



De forma similar, se puede dibujar un círculo sin contorno:

```
strokeWeight(2);
noFill();
ellipse(70,70,40,40);


noStroke();
fill(#FF0000);
ellipse(30,30,40,40);
//(_1_090)
```

Figura 9. Un círculo rojo y una circunferencia negra



En este caso, hemos cambiado el orden de dibujado porque, al poner el comando `noStroke()` al principio, nos hubiera obligado a poner otro comando `stroke(#000000)` para forzar el dibujo de un contorno negro en la circunferencia. Así escribimos menos y aprovechamos los valores que Processing nos pone por defecto.

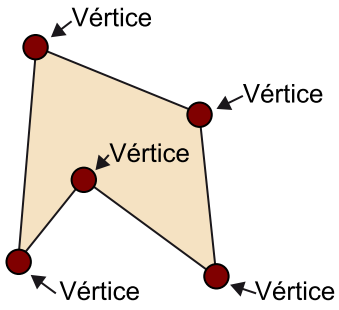
2.3. Objetos complejos

A continuación, vamos a dibujar un polígono. Al contrario que con los círculos, elipses y rectángulos, vamos a necesitar más de una línea para especificar un polígono. Básicamente, tenemos que definir sus vértices, especificando cuándo empieza y cuándo acaba la lista de los mismos:

```
beginShape();
vertex(20,10);
```

```
vertex(70,30);
vertex(75,80);
vertex(35,50);
vertex(15,75);
endShape(CLOSE);
//(_1_100)
```

Figura 10. Dibujo de un polígono

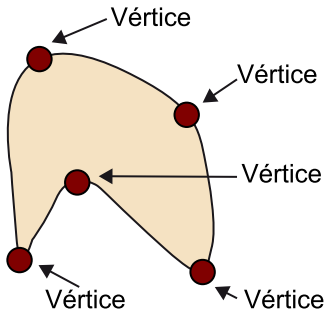


Como en las figuras anteriores, podemos modificar el contorno y el relleno del mismo modo, especificándolo antes de empezar el dibujo del polígono. Intentad borrar la palabra `CLOSE` del código anterior y pensad cómo obtener el mismo dibujo sin ella (solución: repitiendo el primer vértice al final de la lista).

A continuación, vamos a dibujar una forma curva. Estos tipos de formas se pueden especificar de varias maneras. Aquí proponemos utilizar el `curveVertex`. Para definir una de esas formas y que quede cerrada, solo tenemos que poner la lista de puntos (o vértices) por los que queremos que pase, repitiendo los tres primeros al final de la lista. Igual que con el polígono, hay que delimitar la lista de vértices con las mismas dos instrucciones:

```
beginShape();
curveVertex(20,10);
curveVertex(70,30);
curveVertex(75,80);
curveVertex(35,50);
curveVertex(15,75);
curveVertex(20,10);
curveVertex(70,30);
curveVertex(75,80);
endShape();
//(_1_110)
```

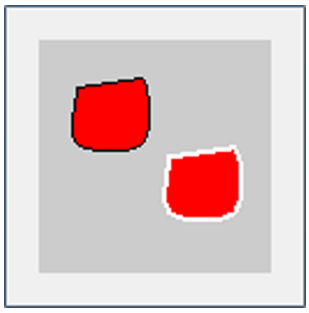
Figura 11. Dibujo de una forma curva que pasa por un conjunto de puntos



Finalmente, para acabar esta primera sección, dibujaremos el objeto que va a identificar a cada ítem de información del *plug-in* de visualización, nuestra persona:

```
fill(#FF0000,220);
strokeWeight(0.5);
beginShape();
curveVertex(16, 16);
curveVertex(16, 20);
curveVertex(16, 44);
curveVertex(44, 44);
curveVertex(44, 16);
curveVertex(20, 16);
endShape(CLOSE);

strokeWeight(2);
stroke(#FFFFFF);
beginShape();
curveVertex(56, 46);
curveVertex(56, 50);
curveVertex(56, 74);
curveVertex(84, 74);
curveVertex(84, 46);
curveVertex(60, 46);
endShape(CLOSE);
//(_1_120)
```

Figura 12. Dibujo de la persona, sin seleccionar (arriba) y seleccionada (abajo)



Hemos dibujado la persona de una categoría concreta y con sus dos posibles apariencias, sin seleccionar (con contorno negro delgado) y seleccionado (con contorno blanco grueso). Notad que el relleno tiene una pequeña cantidad de transparencia.

3. Escribir texto

En esta sección, vamos a ver cómo escribir (o mejor dicho dibujar) texto y cambiar algunas de sus características.

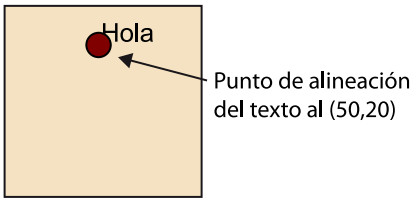
Antes de nada, tenemos que empezar seleccionando la fuente. Esto se puede hacer de varias formas, aquí proponemos una que aprovecha las fuentes instaladas en el sistema. Seleccionamos una fuente bastante común, la helvética, con un tamaño inicial de 16 puntos (píxeles) de altura.

```
textFont(createFont("Helvetica",16));
```

Una vez seleccionada la fuente, podemos escribir textos y situarlos en la zona de dibujo, del mismo modo que las formas que hemos visto en el apartado 1.

```
textFont(createFont("Helvetica",16));
text("Hola",50,20);
//(_2_010)
```

Figura 13. Texto *Hola* dibujado desde el punto (50, 20)



3.1. Tamaño y color

Podemos jugar también con el tamaño y el color del texto; el funcionamiento es el mismo que tienen las características de grosor y color en las formas, solo que los textos no tienen contorno. Para especificar el tamaño del texto, podemos usar `textSize`.

```
textFont(createFont("Helvetica",16));
fill(#FF0000);
text("Hola",50,20);
textSize(20);
text("Test",15,55);
fill(#222299);
textSize(10);
text("Casa",65,80);
//(_2_020)
```

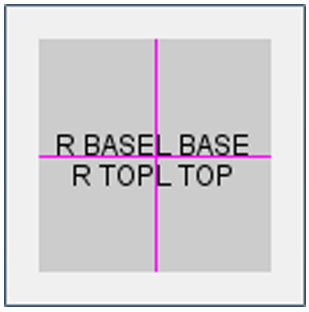
Figura 14. Varios textos con tamaños y colores diferentes

3.2. Alineación

Una propiedad del texto es la alineación. Dado un punto de localización del texto, hay que definir dónde queda este. ¿Sobre el punto? ¿Bajo el punto? ¿A la izquierda del punto? ¿A la derecha? Por defecto, se alinea arriba y a la izquierda, lo que quiere decir que el punto de referencia se encuentra en la esquina inferior izquierda de la primera letra del texto.

```
textFont(createFont("Helvetica",12));
fill(#000000);
text("L BASE",50,50);
textAlign(LEFT,TOP);
text("L TOP",50,50);
textAlign(RIGHT);
text("R BASE",50,50);
textAlign(RIGHT,TOP);
text("R TOP",50,50);
stroke(#FF00FF);
line(50,0,50,100);
line(0,50,100,50);
//(_2_030)
```

Figura 15. Textos con diferentes alineaciones: L (izquierda o LEFT), R (derecha o RIGHT), TOP (superior o TOP), BASE (línea base, valor por defecto)



En la figura 15, se pueden ver cuatro ejemplos de alineaciones. Para aumentar la claridad, se han dibujado dos líneas de color magenta: el punto (50, 50) corresponde a la intersección de ambas líneas y es también el punto donde

se sitúan todos los textos. Mirando el código, se puede ver qué alineación corresponde a cada texto. Por ejemplo, la alineación (`LEFT`) (la más común) corresponde al texto "L BASE" y la alineación (`RIGHT,TOP`), al "R TOP".

3.3. Colocación del texto

A continuación, escribiremos el texto correspondiente a la primera persona de nuestro *plug-in*. Como queremos que el texto se sitúe en la parte inferior, necesitamos conocer la altura total de la zona de dibujo. El valor de la altura se encuentra con la palabra clave `height`. La línea inferior la dibujaremos a esa altura y la línea superior, a esa altura menos lo que ocupa una línea para que no se solapen. Lo que ocupa una línea es precisamente el tamaño del texto.

```
textFont(createFont("Helvetica",12));
textAlign(LEFT);
fill(#000000);
textSize(12);
text("Xavier Queralt",0,height);
textSize(10);
text("Entrevistas",0,height-12);
//(_2_040)
```

Figura 16. Dos líneas de texto situadas en la parte de abajo de la zona de dibujo

El texto anterior queda demasiado ajustado, tanto en los bordes de la zona de dibujo como entre ellos. Podemos dar un poco de espacio añadiendo márgenes a la hora de especificar la altura de cada línea de texto. Dejaremos un margen con los bordes igual al tamaño de la fuente grande (valor de 12) y un espacio entre líneas de medio tamaño de la fuente (valor de 6).

```
textFont(createFont("Helvetica",12));
textAlign(LEFT);
fill(#000000);
textSize(12);
text("Xavier Queralt",12,height-12);
textSize(10);
text("Entrevistas",12,height-30);
```

```
//(_2_050)
```

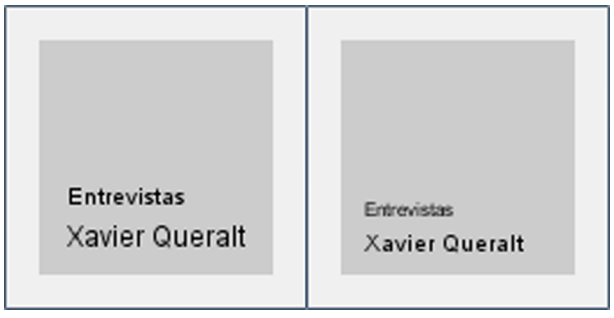
Figura 17. Dos líneas de texto con márgenes y espaciado entre ellos

Vemos que el código anterior tiene muchos números, pero en realidad todos hacen referencia a la misma cantidad. En vez de ponerlos tantas veces, asociaremos un nombre al número. Esto lo haremos creando una variable, asignándole el valor y después poniendo el nombre de la variable, así el código se vuelve más claro.

```
int medidaFuente = 12;
textFont(createFont("Helvetica",medidaFuente));
textAlign(LEFT);
fill(#000000);
textSize(medidaFuente);
text("Xavier Queralt",medidaFuente,height-medidaFuente);
textSize(medidaFuente*0.8);
text("Entrevistas",medidaFuente,height-medidaFuente*2.5);
//(_2_051)
```

Notad que multiplicamos por 0,8 para obtener 10 y por 2,5 para obtener el número 30. Con esto conseguimos poder cambiar el tamaño de todo solo cambiando un valor, el de medidaFuente. Probad a cambiar el valor por 10, por ejemplo.

Figura 18. El mismo dibujo, pero con tamaños de fuentes base diferentes; notad que los márgenes y espaciados son proporcionales al tamaño de los textos



3.4. Dibujos y texto simultáneos

A continuación, vamos a añadir líneas verticales acompañadas del número de año, de forma similar a como lo queremos construir en el *plug-in* de visualización. No obstante, la zona de dibujo se nos empieza a quedar pequeña; aprovecharemos para introducir el comando `size`, que nos permitirá establecer las medidas de la zona de dibujo que queramos.

```
int medidaFuente = 16;
size(200,200);
textFont(createFont("Helvetica",medidaFuente));
textAlign(LEFT);
fill(#000000);
textSize(medidaFuente);
text("Xavier Queralt",medidaFuente,height-medidaFuente);
textSize(medidaFuente*0.8);
text("Entrevistas",medidaFuente,height-medidaFuente*2.5);

fill(#555555);
strokeWeight(1);
stroke(#000000,105);
textAlign(LEFT,TOP);
textSize(medidaFuente/2);

line(10,10,10,height-4*medidaFuente);
text("2001",10+medidaFuente/4,10);
line(120,10,120,height-4*medidaFuente);
text("2002",120+medidaFuente/4,10);
//(_2_060)
```

Figura 19. Dibujo de dos líneas anuales y una etiqueta

Tal como podemos observar, hemos añadido dos bloques de código, un primero con la configuración de la línea y el texto y un segundo que dibuja las líneas y escribe los dos números de años. Pero ¿y si quisiéramos poner unos cuantos años más? ¿Tendríamos que replicar todo el código? No. Por eso, podemos

usar funciones. Hasta ahora, hemos estado trabajando con el modo *static* de Processing, que admite instrucciones tal y como las hemos ido poniendo en los ejemplos. Pasaremos a introducir el modo activo, que nos da más flexibilidad, puesto que nos permite definir funciones.

```
int medidaFuente = 16;

void setup(){
  size(200,200);
  textFont(createFont("Helvetica",medidaFuente));
  textAlign(LEFT);
  fill(#000000);
  textSize(medidaFuente);
  text("Xavier Queralt",medidaFuente,height-medidaFuente);
  textSize(medidaFuente*0.8);
  text("Entrevistas",medidaFuente,height-medidaFuente*2.5);
  dibujarLineaAnual(2001,10);
  dibujarLineaAnual(2002,40);
  dibujarLineaAnual(2003,70);
  dibujarLineaAnual(2004,100);
  dibujarLineaAnual(2005,130);
  dibujarLineaAnual(2006,160);
  dibujarLineaAnual(2007,190);
}

void dibujarLineaAnual(int valor, int posicion){
  fill(#555555);
  strokeWeight(1);
  stroke(#000000,105);
  textAlign(LEFT,TOP);
  textSize(medidaFuente/2);
  line(posicion,10,posicion,height-4*medidaFuente);
  text(valor,posicion+medidaFuente/4,10);
}
//(_2_070)
```

Figura 20. Dibujo de siete líneas anuales y una etiqueta

Este código obtiene un resultado similar al de la figura 19, pero dibujando muchas más líneas. Cada línea que añadimos se dibuja con una línea de código, puesto que hemos definido una función que se llama `dibujarLineaAnual`, que recibe dos valores y los utiliza en su interior para dibujar. De este modo, reutilizamos código y no escribimos tanto. Si todavía queremos escribir menos, podemos utilizar un bucle (`for`) en lugar de escribir las siete líneas con `dibujarLineaAnual`, cambiando:

```
...
  dibujarLineaAnual(2001,10);
  dibujarLineaAnual(2002,40);
  dibujarLineaAnual(2003,70);
  dibujarLineaAnual(2004,100);
  dibujarLineaAnual(2005,130);
  dibujarLineaAnual(2006,160);
  dibujarLineaAnual(2007,190);
...
```

por:

```
...
  for (int i=0;i<7;i++){
    dibujarLineaAnual(2001+i,10+30*i);
  }
...
//(_2_071)
```

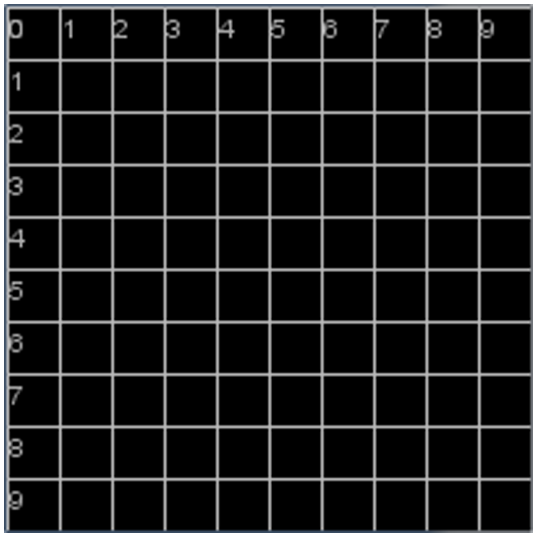
Los "..." los utilizamos en estos materiales textuales para no tener que presentar de nuevo todo el código superior o inferior en la explicación. No es ninguna instrucción de Processing y, si lo introducimos como código, obtendremos un error.

4. Transformaciones

Antes de empezar a ver las transformaciones, vamos a crear una cuadrícula que nos ayudará a entender mejor su comportamiento dentro de Processing. Utilizaremos el conocimiento que ya tenemos sobre dibujo de líneas y texto, pero, además, añadiremos una nueva instrucción para poder poner un color de fondo y dejar atrás el gris que aparece por defecto. Una posible cuadrícula se puede crear como sigue:

```
int medidaFuente = 10;

void setup(){
  size(200,200);
  background(#000000);
  textFont(createFont("Helvetica",medidaFuente));
  stroke(#BBBBBB);
  fill(#AAAAAA);

  textAlign(LEFT,TOP);
  for (int i=0;i<11;i++){
      dibujarLineaVertical(i*20);
  }
  textAlign(LEFT,TOP);
  for (int i=0;i<11;i++){
      dibujarLineaHorizontal(i*20);
  }
}

void dibujarLineaVertical(int x){
  line(x,0,x,height);
  text(x/20,x+medidaFuente/8,medidaFuente/8);
}

void dibujarLineaHorizontal(int y){
  line(0,y,width,y);
  text(y/20,medidaFuente/8,y+medidaFuente/8);
}
//(_3_010)
```

Figura 21. Cuadrícula de 10x10

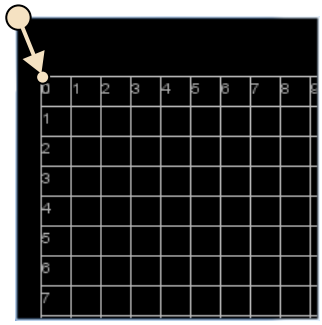


4.1. Traslación

En el ejemplo, hemos construido una cuadrícula dibujando 10 líneas verticales y 10 líneas horizontales equiespaciadas. Si queremos desplazar la cuadrícula, solo tenemos que utilizar la función translate antes de ejecutar el dibujo:

```
...
void setup(){
  size(200,200);
  background(#000000);
  textFont(createFont("Helvetica",medidaFuente));
  stroke(#BBBBBB);
  fill(#AAAAAA);

  translate(15,38);

  textAlign(LEFT,TOP);
  for (int i=0;i<11;i++){
      dibujarLineaVertical(i*20);
  }
  textAlign(LEFT,TOP);
  for (int i=0;i<11;i++){
      dibujarLineaHorizontal(i*20);
  }
}
...
//(_3_020)
```

Figura 22. Cuadrícula desplazada o trasladada 15 píxeles a la derecha y 38 abajo



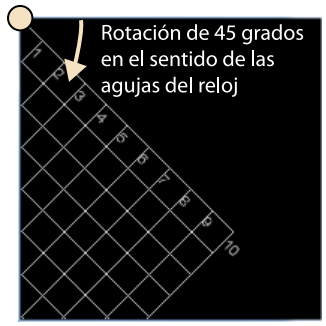
Podemos probar diferentes valores de desplazamientos o traslación, incluso valores negativos. Una traslación es un tipo de transformación que podemos realizar en Processing. Pero podemos realizar dos tipos más: rotación y escala.

4.2. Rotación

Veamos el siguiente ejemplo de cómo hacer una rotación:

```
...
void setup(){
  size(200,200);
  background(#000000);
  textFont(createFont("Helvetica",medidaFuente));
  stroke(#BBBBBB);
  fill(#AAAAAA);


  rotate(radians(45));


  textAlign(LEFT,TOP);
  for (int i=0;i<11;i++){
      dibujarLineaVertical(i*20);
  }
  textAlign(LEFT,TOP);
  for (int i=0;i<11;i++){
      dibujarLineaHorizontal(i*20);
  }
}
...
//(_3_030)
```

Figura 23. Cuadrícula rotada 45 grados en el sentido de las agujas del reloj



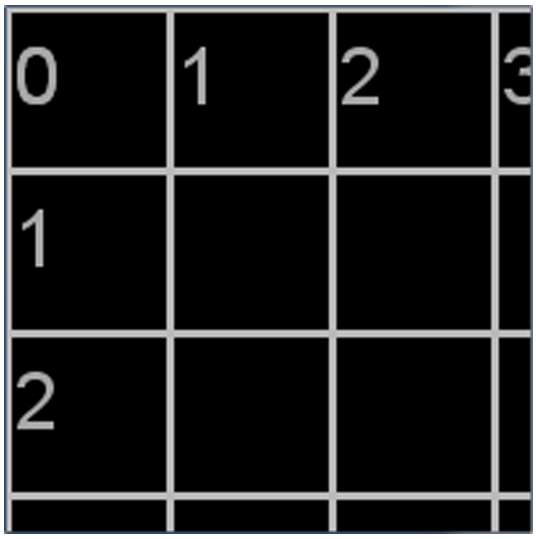
El sentido de las rotaciones es siempre el de las agujas del reloj. Podemos ejecutar rotaciones en el sentido contrario especificando los grados con valor negativo.

4.3. Escala

La escala es el tercer y último tipo de transformación que vamos a ver:

```
...
void setup(){
  size(200,200);
  background(#000000);
  textFont(createFont("Helvetica",medidaFuente));
  stroke(#BBBBBB);
  fill(#AAAAAA);

  scale(3.1);

  textAlign(LEFT,TOP);
  for (int i=0;i<11;i++){
      dibujarLineaVertical(i*20);
  }
  textAlign(LEFT,TOP);
  for (int i=0;i<11;i++){
      dibujarLineaHorizontal(i*20);
  }
}
...
//(_3_040)
```

Figura 24. Cuadrícula escalada en un factor de 3,1



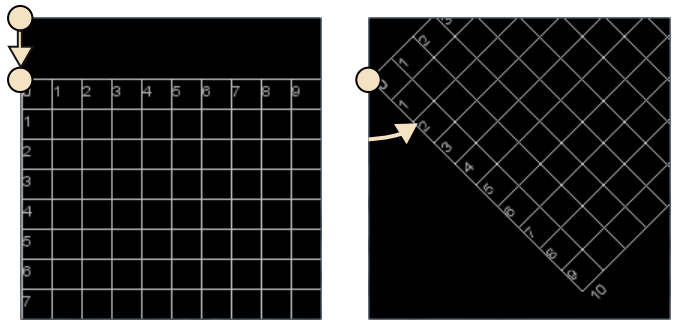
Tal como vemos, la escala es una posible manera de emular zooms cuando trabajamos en dos dimensiones. Es importante que notemos que las transformaciones de rotación y escala desempeñan su función sobre un punto de referencia. Cuando giramos un objeto, necesitamos pivotar sobre un punto y, cuando hacemos algo grande o pequeño, lo hacemos centrándonos en un punto concreto. En el caso de Processing, el punto de referencia es el (0, 0), en los ejemplos la esquina superior izquierda. Si nos fijamos, tanto la rotación como el escalado se han llevado a cabo pivotando sobre ese punto. Pero ¿qué pasa si queremos hacerlo sobre otro punto? A continuación, vamos a ver el comportamiento de Processing al ejecutar diferentes transformaciones seguidas, y esto nos permitirá pivotar sobre el punto que queramos.

4.4. Acumulación de transformaciones: el orden sí importa

Vamos a llevar a cabo una comparación. Haremos una traslación y una rotación y visualizaremos su resultado al realizarlas en dos órdenes diferentes:

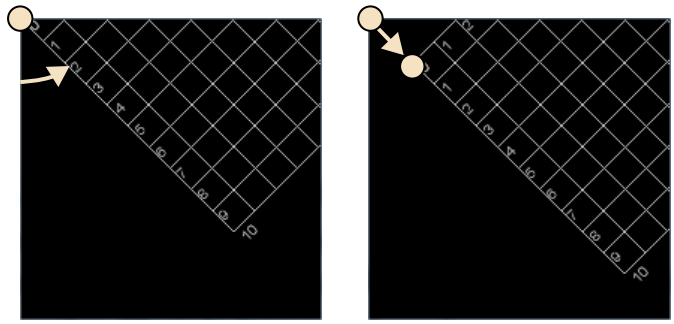
```
...
  translate(0,40);
  rotate(radians(-45));
...
//(_3_051), (_3_053)
```

Figura 25. Traslación de 40 píxeles (dos casillas de la cuadrícula) seguida de rotación de 45 grados en el sentido contrario a las agujas del reloj



```
...
  rotate(radians(-45));
  translate(0,40);
...
//(_3_052), (_3_054)
```

Figura 26. Rotación de 45 grados en el sentido contrario a las agujas del reloj seguida de una traslación de 40 píxeles (dos casillas de la cuadrícula)



Una rotación siempre cambia la orientación de los ejes (también conocido como el sistema de referencia) y la traslación se hace en relación con estos ejes. Con este ejemplo, podemos ver que, cuando realizamos una traslación después de una rotación, como la rotación hace cambiar los ejes, el desplazamiento producido se ejecuta sobre los ejes girados y esto provoca que el desplazamiento de la figura 26 no sea en vertical, sino en diagonal. En el caso de hacer primero la traslación, esta lo que hace es mover el punto de pivotaje, por eso la rotación subsiguiente se realiza alrededor del (`0,40`).

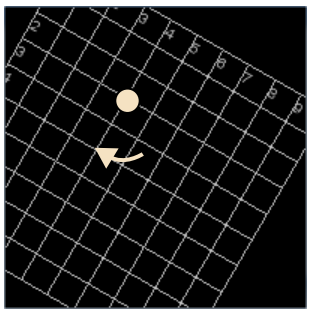
4.5. Cambio del punto de pivotaje en rotaciones y escalados

Podemos combinar varias transformaciones para hacer rotaciones alrededor del punto que deseemos. Por ejemplo, para dar un giro de 30 grados alrededor del punto (3, 4) de la cuadrícula necesitamos tres transformaciones seguidas como las siguientes:

```
...
  translate(80,60);
  rotate(radians(30));
```

```
    translate(-80,-60);
  ...
  //(_3_060)
```

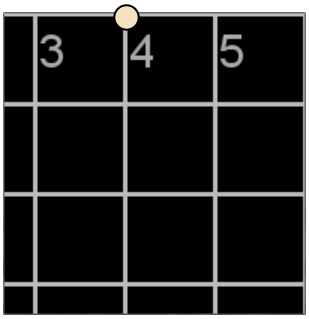
Figura 27. Rotación de 30 grados en el sentido de las agujas del reloj pivotando sobre el punto (80, 60), que es el punto (4, 3) de la cuadrícula



La rotación se hace entre medio y las dos traslaciones tienen los mismos desplazamientos, solo que en la segunda traslación se cambia el signo de las magnitudes. El hecho de que sea (80, 60) se debe a que cada unidad de cuadrícula que hemos dibujado ocupa 20 píxeles. De modo similar se puede proceder con la escala:

```
  ...
    translate(80,0);
    scale(3);
    translate(-80,-0);
  ...
  //(_3_070)
```

Figura 28. Escalado triple pivotando sobre el punto (80, 0), que es el punto (4, 0) de la cuadrícula



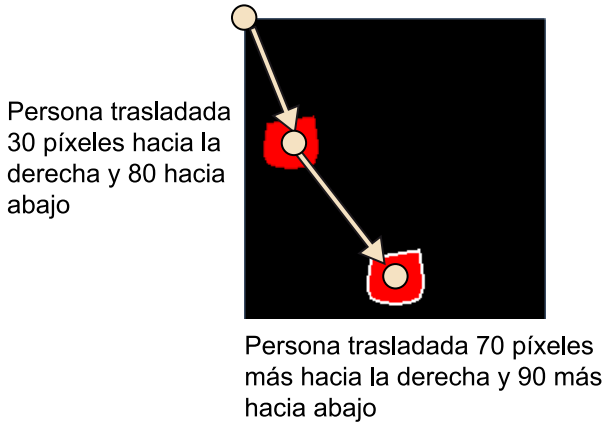
En este último caso, hemos escogido un punto de altura 0 para poder ver las etiquetas de la cuadrícula.

4.6. Uso de transformaciones

Ahora que conocemos las transformaciones, podríamos pensar en dibujar a una persona alrededor del origen de coordenadas (el punto (`0,0`)) y después situarla donde queramos mediante una transformación de traslación. Definiremos la función de dibujar a la persona y así escribiremos menos código si dibujamos varias. También definiremos la variable `medidaPersona` para tener identificado en una zona del código el tamaño de las personas. La persona la podremos dibujar seleccionada (con un contorno blanco) o no seleccionada (con un contorno negro).

```
int medidaPersona = 32;

void setup(){
  size(200,200);
  background(#000000);
  dibujarPersona(30,80,0);
  dibujarPersona(70,90,1);
}

void dibujarPersona(int tx, int ty, int seleccionada){
  translate(tx,ty);

  strokeWeight(2);
  fill(#FF0000);
  if (seleccionada==1){
     stroke(#FFFFFF);
  } else {
     stroke(#000000,50);
  }

  beginShape();
  curveVertex(-medidaPersona/2, -medidaPersona/2);
  curveVertex(-medidaPersona/2, -5*medidaPersona/14);
  curveVertex(-medidaPersona/2, medidaPersona/2);
  curveVertex(medidaPersona/2, medidaPersona/2);
  curveVertex(medidaPersona/2, -medidaPersona/2);
  curveVertex(5*medidaPersona/14, -medidaPersona/2);
  endShape(CLOSE);
}
//(_3_080)
```

Figura 29. Dibujo de dos personas cada una trasladada a partir del sistema de referencia anterior



Tal como podemos ver en el código del ejemplo de la figura 29, como cada figura tiene su traslación, al trasladarla con un `translate`, también estamos modificando el sistema de referencia. Cuando situamos a la segunda persona, la traslación la tenemos que hacer teniendo en cuenta la traslación hecha en la primera persona. Esto puede llegar a ser más complicado de lo necesario según vamos añadiendo personas. Lo que queremos es que cada persona tenga su traslación, pero que no se modifique el sistema de referencia. La manera de obtener este comportamiento la veremos en el apartado siguiente.

5. Sistemas de referencia

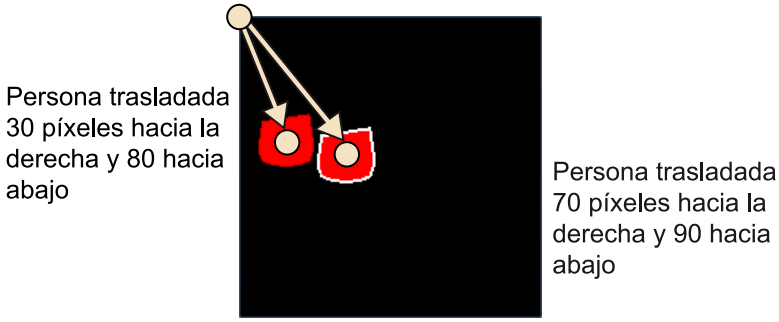
La sección anterior finalizó con la situación en la que, cada vez que realizábamos una transformación, el sistema de referencia quedaba modificado. Esto nos puede ser de utilidad a veces y en otros casos nos puede dar más complicaciones. Cuando utilizamos funciones que nos dibujen objetos, las pensamos y codificamos una sola vez. Queremos que tengan el funcionamiento diseñado allí donde las queramos poner. En el caso del apartado anterior, esto no se cumplía. Cosas como estas añaden complejidad a la hora de codificar un programa.

Si una transformación cambia el sistema de referencia, ¿cómo puedo volver al sistema de referencia anterior sin deshacer la transformación? En Processing, se consigue con el uso de dos funciones: `pushMatrix()` y `popMatrix` (). No vamos a entrar en lo que significan, sino simplemente en una forma eficaz de utilizarlas en nuestra aplicación. La idea es que `pushMatrix` guarda el sistema de referencia vigente y `popMatrix` sustituye al sistema de referencia vigente por el último que hemos guardado con `pushMatrix`. Si las utilizamos en la función de dibujar persona que producía la figura 29, quedaría así:

```
...
void dibujarPersona(int tx, int ty, int seleccionada){
  pushMatrix();
  translate(tx,ty);

  strokeWeight(2);
  fill(#FF0000);
  if (seleccionada==1){
     stroke(#FFFFFF);
  } else {
     stroke(#000000,50);
  }

  beginShape();
  curveVertex(-medidaPersona/2, -medidaPersona/2);
  curveVertex(-medidaPersona/2, -5*medidaPersona/14);
  curveVertex(-medidaPersona/2, medidaPersona/2);
  curveVertex(medidaPersona/2, medidaPersona/2);
  curveVertex(medidaPersona/2, -medidaPersona/2);
  curveVertex(5*medidaPersona/14, -medidaPersona/2);
  endShape(CLOSE);

  popMatrix();
}
```

```
...
//(_4_010)
```

Figura 30. Dos personas en dos posiciones diferentes respecto al mismo sistema de referencia



¿Notamos la diferencia entre las figuras 29 y 30? El hecho de restaurar el sistema de referencia apenas acabar de dibujar a la persona (habíamos guardado el original antes de empezar a dibujar) permite que la función dibuje a personas en la posición que digamos siempre respecto al mismo sistema de referencia, que en este caso es el (0, 0).

A continuación, redibujaremos la figura 19 utilizando `pushMatrix` y `popMatrix`. Además, utilizaremos funciones, como venimos haciendo desde hace algunos ejemplos.

```
int medidaPersona = 16;
int medidaFuente = 16;
int espacioAnual = medidaPersona*4;

void setup(){
  size(200,200);
  background(#000000);
  textFont(createFont("Helvetica",medidaFuente));
  pushMatrix();
  translate(medidaFuente/2,medidaFuente/2);
  dibujarLineasAnuales(2001,2011);
  popMatrix();
  escribeTexto("Entrevistas","Xavier Queralt");
}

void dibujarLineasAnuales(int inicio, int fin){
  for (int i=inicio;i<=fin;i++){
     dibujarLineaAnual(i, (i-inicio)*espacioAnual);
  }
}

void dibujarLineaAnual(int valor, int posicion){
  pushMatrix();
```

```
  translate(posicion,0);
  fill(#AAAAAA);
  strokeWeight(1);
  stroke(#FFFFFF,105);
  textAlign(LEFT,TOP);
  textSize(medidaFuente/2);
  line(0,0,0,height-4*medidaFuente);
  text(valor,medidaFuente/4,0);

  popMatrix();
}

void escribeTexto(String categoría, String nombre){
  textAlign(LEFT);
  fill(#FFFFFF);
  textSize(medidaFuente);
  text(nombre,medidaFuente,height-medidaFuente);
  textSize(medidaFuente*0.8);
  text(categoría,medidaFuente,height-2.5*medidaFuente);
}
//(_4_020)
```

Figura 31. Líneas de los años y etiqueta inferior de ejemplo

En este código, no hay que hacer algo muy diferente a lo anterior (excepto la separación entre las líneas, que ahora es mayor). En vez de dibujar a dos personas, lo que hacemos es dibujar una serie de líneas verticales con etiqueta del año y dos líneas de texto debajo. Fijaos en el uso de `pushMatrix` y `popMatrix` en la función `setup`. El dibujo de las líneas se encuentra entre ellas y después viene el dibujo del texto. Todas las líneas se dibujan bajo el sistema de referencia fijado por `translate(medidaFuente/2,medidaFuente/2);` pero el texto se dibuja bajo el sistema de referencia (0, 0) original, ya que está

después de `popMatrix`, que reemplaza el sistema de coordenadas puesto por el *translate* por el que se había guardado con el `pushMatrix`, que era el sistema de referencia (0, 0) original.

La función que dibuja las líneas contiene el mismo bucle que pintaba la figura 20 (adaptado un poco para permitir la especificación de un año inicial y otro final). La función que pinta cada línea vertical con su etiqueta es la misma que la asociada a la figura 20, pero si nos fijamos, veremos que hemos delegado la responsabilidad de situar horizontalmente la línea a una transformación (`translate(posicion,0));` como queremos que la transformación no nos cambie el sistema de referencia cuando acabe la función, utilizamos de nuevo `pushMatrix` y `popMatrix`.

Como regla general, aconsejamos el uso de `pushMatrix` y `popMatrix` siempre que dibujéis un objeto independiente y lo pongáis justo antes de la transformación y justo después de los dibujos. Por otro lado, empezamos a tener un código de más de 40 líneas y en él podemos distinguir una parte de dibujado y otra de escritura de texto. Vamos a crear dos pestañas nuevas en el IDE de Processing para cada una de estas dos partes, así ordenamos el código (lo tenéis en el `sketch _4_021`).

6. Incorporación de datos

Hasta ahora, hemos estado dibujando a personas a partir de datos concretos. En este punto, vamos a tratar la parte más abstracta del código, aquella que en sí misma no produce ningún resultado visual directamente, pero que es imprescindible para el código de dibujado: el tratamiento de los datos que se van a visualizar.

En este ejemplo, vamos a trabajar con un conjunto de datos predefinidos. En apartados posteriores, se verá cómo adaptar el código para poder admitir datos entregados a través de código php y el acceso a una base de datos.

Así, crearemos una nueva pestaña en el IDE de Processing que contendrá la variable `String[][] datos` con un contenido predefinido (ved el `sketch_5_010`). Cuando accedamos a esta variable lo haremos así:

```
datos[i][j]
```

donde `i` simboliza el número de registro o número de ficha y la `j` indica el campo de la ficha. En el contenido predefinido, hay 314 fichas de seis campos cada una. Debemos tener en cuenta que, en Processing, las indexaciones empiezan por 0, por lo tanto existirán las fichas desde la 0 hasta la 313 y los campos desde el 0 hasta el 5. Nuestra persona será cada una de las fichas contenida en `datos`.

6.1. El tiempo Unix

El cuarto campo de las personas de los datos contiene la fecha en el formato de tiempo Unix (o tiempo POSIX). Este formato indica el número de segundos transcurridos desde las 00:00 del 1 de enero de 1970. Es una manera como otra de tener la fecha hasta una precisión de segundos en un solo número. La fecha también aparece en los terceros campos, en formato alfanumérico. No obstante, debido a que será más fácil y eficaz trabajar con un único valor que con una cadena de caracteres, utilizaremos el tiempo Unix para manejar internamente los datos y poder visualizarlos en la línea de tiempo.

6.2. Ordenación de los datos por tiempo

Dado que la visualización de datos que queremos hacer dibuja a las personas ordenadas en el tiempo, parece lógico pensar que si los datos están ordenados por tiempo podremos dibujarlas más fácilmente. De este modo, dado que no

podemos asegurar que los datos que manejamos tengan ninguna ordenación, los vamos a ordenar nosotros mismos. Para hacerlo, utilizaremos el algoritmo QuickSort, que se acompaña en estos materiales:

```
datos = ordenar(datos,0,datos.length-1);
//(_5_020)
```

Esta línea la podemos poner en la función `setup` antes de escribir ni dibujar nada. Lo que hace es recibir la variable `datos` y devolverla ordenada. Podéis encontrar el código en los materiales, donde veréis que hay una pestaña donde aparece el código del algoritmo de ordenación. El resultado lo guardaremos en la misma variable, que modificaremos para los nuevos datos ordenados.

Vale la pena también marcar la aparición de `length`. En una lista, nos dice el número de posiciones del mismo. Dado que `datos` es una lista (técnicamente, *array*) de fichas (o personas), denotado por el primer `[]`, podemos saber el número de personas con la propiedad `length`. Lo mismo podemos hacer para una persona concreta, por ejemplo, `datos[0].length` nos dice el número de campos que tiene la primera persona.

6.3. Información a partir de los datos

Ahora que tenemos los datos ordenados, buscaremos el margen de años. Precisamente, como los tenemos ordenados desde el más antiguo hasta el más reciente, solo tendremos que tomar el primero y el último y ver sus años.

```
void margenAnual(){
  int epochPrimero, epochUltimo;
  epochPrimero = int(datos[0][3]);
  epochUltimo = int(datos[datos.length-1][3]);
  primero = epochPrimero/31558464+1970;
  último = epochUltimo/31558464+1970;
}
//(_5_030)
```

Este código lo pondremos también en una pestaña nueva. El número `31558464` es el número promedio de segundos que contiene un año. `primero` y `ultimo` son dos variables globales que habremos definido en la pestaña del código de la función `setup`, que quedará como sigue, después de añadir la ordenación de los datos (recordemos que `datos` se encuentra en una pestaña propia, dada su extensión):

```
int medidaPersona = 16;
int medidaFuente = 16;
int espacioAnual = medidaPersona*4;
int primero;
int último;
```

```
void setup(){
  size(200,200);
  background(#000000);
  textFont(createFont("Helvetica",medidaFuente));
  datos = ordenar(datos,0,datos.length-1);
  margenAnual();
  pushMatrix();
  translate(medidaFuente/2,medidaFuente/2);
  dibujarLineasAnuales(primero,último+1);
  popMatrix();
  escribeTexto(datos[0][4],datos[0][0]);
}
//(_5_030)
```

Figura 32. La línea de tiempo y el texto de la primera persona

6.4. Colocación de las personas

El siguiente paso es dibujar a las personas en su orden de tiempo, pero antes necesitamos saber la posición horizontal y vertical de las personas. La posición horizontal será proporcional al tiempo y la vertical irá en función de la categoría (campo sexto de la persona); sabemos que solo hay cuatro categorías diferentes. Para conocer sus posiciones horizontal y vertical:

```
void calcularPosicionesPersonas(){
  int incrementoPorApilamiento;
  posx = new int[datos.length];
  posy = new int[datos.length];
  incrementoPorApilamiento = (height-6*medidaFuente-medidaPersona)/3;
  for (int k=0;k<datos.length;k++){
     posx[k] = int((espacioAnual/31558464.0)*(float(datos[k][3])-(primero-1970)*31558464.0));
     posy[k] = int(datos[k][5])*incrementoPorApilamiento;
```

```
    }
  }
  //(_5_040)
```

posx y posy son dos *arrays* (o listas) de valores. posx contiene las posiciones horizontales de las personas y posy, las verticales. Estas dos variables las añadimos a la lista de variables globales de nuestra aplicación. incrementoPorApilamiento es la distancia vertical en píxeles que hay entre categorías; a la hora de calcularla tenemos que saber el espacio vertical disponible, que es igual a la longitud de la línea vertical menos los márgenes que dejamos (dos veces lo que ocupa la fuente de margen vertical y horizontal y lo que ocupa una persona, puesto que las personas se colocan según su centro y tienen una altura que no es 0 –ocupan un espacio–). La posición horizontal se calcula como el excedente en segundos desde el año primero y es proporcional al espacioAnual. La posición vertical se obtiene multiplicando el índice de la categoría (aprovechando que va entre 0 y 3) por el incrementoPorApilamiento.

6.5. Dibujo de la población

Ya tenemos toda la información necesaria para poder dibujar a todas las personas. Para hacerlo, utilizaremos el código de dibujarPersona de la figura 32, pero sustituyendo fill(#FF0000) por:

```
...
  switch(categoría){
   case 0:
      fill(#FF0000);
      break;
   case 1:
      fill(#00FF00);
      break;
   case 2:
      fill(#0000FF);
      break;
   case 3:
      fill(#FF00FF);
      break;
  }
...
//(_5_050)
```

y añadiendo una variable a la cabecera, que sea categoria:

```
...
void dibujarPersona(int tx, int ty, int categoría, int seleccionada){
...
```

```
//(_5_050)
```

El código para dibujar la población será el siguiente y se insertará en la función `setup` justo después de `dibujarLineasAnuales`:

```
void dibujarPoblacion(){
  pushMatrix();
  translate(0,medidaFuente+medidaPersona/2);
  for (int i=0;i<datos.length;i++){
     dibujarPersona(posx[i],posy[i],int(datos[i][5]),0);
  }
  popMatrix();
}
//(_5_050)
```

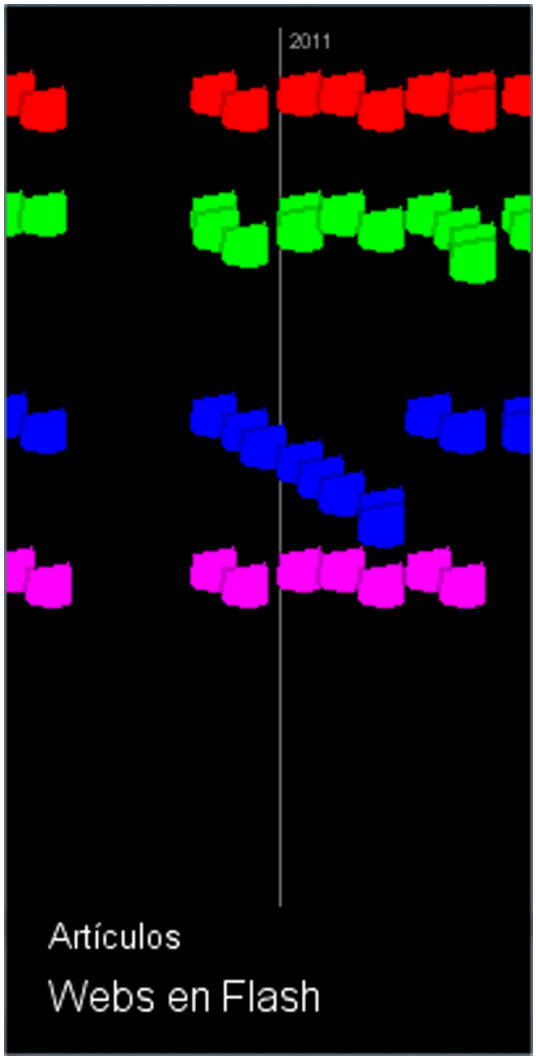
Figura 33. Dibujo de todas las personas, las líneas anuales y el texto de la primera persona

En la figura 33, hemos establecido un `espacioAnual` más pequeño para poder ver todas las personas, pero queda claro que está demasiado condensado y lo tendremos que espaciar también verticalmente. Para hacerlo, se propone modificar el código de `calcularPosicionesPersonas`, que detalla algo más la posición vertical de las personas en función de si una persona está solapada con su inmediatamente anterior de la misma categoría:

```
void calcularPosicionesPersonas(){
  float incrementoPorApilamiento;
  int[] maxposy = new int[4];
  int i,j;
  posx = new int[datos.length];
  posy = new int[datos.length];
  for (int k=0;k<datos.length;k++){
     posx[k] = int((espacioAnual/31558464.0)*(float(datos[k][3])-(primero-1970)*31558464.0));
  }
```

```
  for (int categoría=0;categoría<4;categoría++){
     i = 0;
     maxposy[categoría] = 0;
     while(int(datos[i][5])!=categoría && i<datos.length){
      i++;
  }
  for (j = i+1; j<datos.length;j++){
     if (int(datos[j][5])==categoría){
       if ((posx[j]-posx[i])<medidaPersona){
         posy[j] = posy[i]+1;
         if(posy[j]>maxposy[categoría]){
           maxposy[categoría] = posy[j];
         }
       }
       i = j;
     }
    }
  }
  maxposy[1] = maxposy[0]+maxposy[1];
  maxposy[2] = maxposy[1]+maxposy[2];
  maxposy[3] = maxposy[2]+maxposy[3];
  incrementoPorApilamiento = float(height-5*medidaFuente-4*medidaPersona)/(maxposy[3]-1.0);

  int offset;
  for (i=0;i<datos.length;i++){
    if (int(datos[i][5])>0){
      offset = int(maxposy[int(datos[i][5])-1]*incrementoPorApilamiento+(int(datos[i][5]))
               *medidaPersona);
    } else {
      offset = 0;
    }
    posy[i] = int(posy[i]*incrementoPorApilamiento+offset);
  }
}
//(_5_060)
```

Figura 34. Las personas que se encuentran en torno al año 2011



Aparte de crear el espacio de visualización el doble de alto, se han espaciado los años en doce veces el tamaño de la persona. En la figura 34, se puede ver el resultado aplicando:

```
translate(medidaFuente/2-9.5*espacioAnual,medidaFuente/2);
```

en vez de:

```
translate(medidaFuente/2,medidaFuente/2);
```

para poder ver una zona donde están las cuatro categorías. Notad que, cuando las personas se solapan horizontalmente entre ellas, van desplazándose hacia abajo para aumentar la zona visible.

7. Animaciones

Queremos darle un toque más vivo a la visualización de los datos que proponemos. Para hacerlo, recurriremos a las animaciones, haciendo que las personas se muevan un poco, eliminando el efecto estático que presenta la visualización que llevamos hasta este punto.

Antes de nada, vamos a dibujar a una persona que se mueve de izquierda a derecha de la zona de la pantalla y vuelve a entrar cuando sale. Para hacerlo, necesitamos colocar el código de dibujado dentro de la función `draw` y establecer la velocidad de refresco con `frameRate`:

```
int medidaPersona = 16;
int x;
float velocidadX;

void setup(){
  size(200,200);
  background(#000000);
  x = -medidaPersona/2;
  velocidadX = 2;
 frameRate(25);
}

void draw(){
  background(#000000);
  if (x>(width+medidaPersona/2)){
     x = -medidaPersona/2;
  } else {
     x=int(x+velocidadX);
  }
  dibujarPersona(x,100,0,0);
}
//(_6_010)
```

La función `draw` se llama un número de veces por segundo, especificado por `frameRate`. En este ejemplo, cada vez que se llama la función `draw` se hace lo siguiente:

- Borrar la pantalla con `background(#000000)`.

- Encontrar la nueva posición; si no ha salido por la derecha, la posición se actualiza sumándole la velocidad; en caso contrario, se restablece al valor inicial (especificado inicialmente en `setup`).

- Se dibuja la persona con la posición actual.

Así, como vemos, podemos lograr una animación redibujando constantemente la escena con objetos que cambian de posición. Podemos fijarnos en que, en este ejemplo, la velocidad de la persona depende de las variables `velocidadX` y del valor que hemos especificado a `frameRate`. Cuanto más `frameRate` y cuanta más `velocidadX`, más rápido. Recordemos que, para lograr una escena fluida, necesitamos un mínimo de 25 imágenes por segundo.

Ahora, vamos a cambiar el modo en el que la persona se mueve, haciéndola oscilar verticalmente:

```
void draw(){
  background(#000000);
  x = int(x + velocidadX);
  dibujarPersona(100,int(100+10*cos(radians(x))),0,0);
}
//(_6_020)
```

En este caso, nos ahorramos la distinción de si sumar o restablecer la `x`, ya que una oscilación es para emular con una función sinusoidal y un ángulo, que puede tener un valor ilimitado y que, en este caso, emula la `x`. Como curiosidad, si ponemos una velocidad de 14,4 y 25 imágenes por segundo, obtendremos una oscilación por segundo en la animación.

Intentad variar la velocidad y las imágenes por segundo para experimentar diferentes velocidades. Probad a disminuir las imágenes por segundo a menos de 25 para dejar de experimentar una animación fluida.

7.1. Animación del conjunto de personas

Vamos a tomar el código que produce la figura 34 y le añadiremos un efecto de animación de oscilación a las personas. Para hacerlo, colocaremos el código de dibujado dentro de la función draw, estableceremos las imágenes por segundo a 25 y modificaremos un poco la función de dibujo de la población:

```
void setup(){
  size(200,400);
  background(#000000);
  textFont(createFont("Helvetica",medidaFuente));
  datos = ordenar(datos,0,datos.length-1);
  margenAnual();
  calcularPosicionesPersonas();
```

```
  angulo = 0.0;
  velocidadOscilacion = 7.2;
  amplitudOscilacion = 3;
  frameRate(25);
}

void draw(){
  angulo = angulo + velocidadOscilacion;
  background(#000000);
  pushMatrix();
  translate(medidaFuente/2-9.5*espacioAnual,medidaFuente/2);
  dibujarLineasAnuales(primero,último+1);
  dibujarPoblacion();
  popMatrix();
  escribeTexto(datos[0][4],datos[0][0]);
}

// A la pestanya dibujado
void dibujarPoblacion(){
  pushMatrix();
  translate(0,medidaFuente+medidaPersona/2);
  for (int i=0;i<datos.length;i++){
     pushMatrix();
     translate(0,amplitudOscilacion*cos(radians(angulo+i+ int(datos[i][5])*80)));
     dibujarPersona(posx[i],posy[i],int(datos[i][5]),0);
     popMatrix();
  }
  popMatrix();
}
//(_6_030)
```

Hemos añadido tres variables: `angulo`, `velocidadOscilacion` y `amplitudOscilacion`. La tercera nos marcará si el recorrido de la oscilación de la persona es grande o corto. Notemos que todo el código de dibujado ha pasado a la función de `draw`. Para añadir el efecto de oscilación, en la función de `dibujarPoblacion` hemos añadido una transformación de traslación antes de dibujar la persona que depende del valor del `angulo` actual, de la propia persona `(i)` y de su categoría (`int(datos[i][5])*80`); así conseguimos que las diferentes personas no oscilen a la vez y las diferentes categorías, tampoco.

8. Interacción con el ratón

Ya solo nos queda añadir la interacción de la aplicación. El dispositivo para interaccionar será el ratón. Processing está específicamente preparado y contiene una serie de variables que almacenan la posición y el estado de los botones del ratón, así como funciones que se llaman cada vez que hay una acción del mismo (moverse y pulsar un botón, entre otros).

8.1. Las coordenadas del ratón

A continuación, vamos a dibujar a una persona en la posición donde se encuentra el ratón dentro de la zona de dibujo. Introduciremos las variables donde Processing guarda la posición del ratón, llamadas `mouseX` y `mouseY` y utilizaremos el código de `dibujarPersona` que ya tenemos:

```
int medidaPersona = 16;

void setup(){
  size(200,200);
  background(#000000);
  frameRate(25);
}

void draw(){
  background(#000000);
  dibujarPersona(mouseX,mouseY,0,0);
}
//(_7_010)
```

`mouseX` y `mouseY` contienen el valor de la posición horizontal y vertical respectivamente de la posición del ratón en cada momento. Si el ratón sale fuera de la zona de dibujo, contienen el último valor que tenía mientras estaba dentro. Probad a eliminar la línea de de `background(#000000)` y experimentad qué comportamiento obtiene el código.

8.2. Los botones del ratón

Processing no solo tiene variables que disponen de información sobre el ratón. También tiene funciones que se llaman cuando pasan según qué acciones (o eventos) hagamos con el ratón. Por ejemplo, las funciones `mousePressed()` y `mouseReleased()` se activan cuando se aprieta o se suelta un botón del

ratón (el que sea). En el ejemplo anterior, haremos que cambie de categoría (y, por lo tanto, de color) cuando tengamos algún botón del ratón apretado, y que vuelva a la categoría inicial cuando el botón se suelte:

```
int medidaPersona = 16;
int activado = 0;

void setup(){
  size(200,200);
  background(#000000);
  frameRate(25);
}

void draw(){
  background(#000000);
  dibujarPersona(mouseX,mouseY,activado,0);
}

void mousePressed(){
  activado = 1;
}

void mouseReleased(){
  activado = 0;
}
//(_7_020)
```

El código anterior también se puede diseñar utilizando la variable `mousePressed` (en vez de la función). Esta variable, de forma similar a `mouseX` y `mouseY`, vale `true` si se encuentra algún botón apretado y false en caso contrario. El código queda más corto que en el caso anterior, pero en función del caso nos irá mejor controlar los clics del ratón con las funciones y, en otros casos, con la variable:

```
int medidaPersona = 16;

void setup(){
  size(200,200);
  background(#000000);
  frameRate(25);
}

void draw(){
  background(#000000);
  if (mousePressed==true){
     dibujarPersona(mouseX,mouseY,1,0);
  } else {
```

```
      dibujarPersona(mouseX,mouseY,0,0);
    }
  }
  //(_7_030)
```

8.3. *Rollover*

Detectar cuándo el ratón se encuentra sobre determinados objetos o zonas de la zona de dibujo es un comportamiento necesario en muchas aplicaciones. Para hacerlo, es necesario comprobar dónde se encuentra el ratón cada vez que redibujamos la zona, es decir, cada vez que se llama la función `draw`. A continuación, vamos a dibujar a una persona y comprobaremos si el ratón está encima de ella: cuando así sea, la dibujaremos seleccionada (con un contorno de color diferente) y deseleccionada en caso contrario:

```
int medidaPersona = 16;
int posicionX = 100;
int posicionY = 100;

void setup(){
  size(200,200);
  background(#000000);
  frameRate(25);
}

void draw(){
  background(#000000);
  if (detectarPersona()==true){
     dibujarPersona(posicionX,posicionY,0,1);
  } else {
     dibujarPersona(posicionX,posicionY,0,0);
  }
}

boolean detectarPersona(){
  int limiteSuperior = posicionY-medidaPersona/2;
  int limiteInferior = posicionY+medidaPersona/2;
  int limiteIzquierda = posicionX-medidaPersona/2;
  int limiteDerecha = posicionX+medidaPersona/2;
  if (mouseX>=limiteIzquierda && mouseX<=limiteDerecha && mouseY>=limiteSuperior &&
      mouseY<=limiteInferior) {
     return true;
  } else {
     return false;
  }
}
```

```
//(_7_040)
```

El procedimiento clave es ver si el ratón está dentro de los cuatro límites (superior, inferior, izquierdo y derecho) que definen la caja imaginaria que contiene a la persona (caja conocida como *bounding box*). Esto se hace con el condicional (`if`) que se encuentra en `detectarPersona`.

8.4. Uso de la posición del ratón

Al final del apartado 4.3.6, teníamos el dibujo de todo nuestro mundo. Pero para visualizarlo todo debíamos hacer los espacios entre años (`espacioAnual`) muy pequeños o hacer la ventana de dibujado muy ancha. Hay otra alternativa para poder verlo todo y se basa en la navegación: lo que tenemos dibujado es una escena concreta del mundo; si navegamos por el mundo, podremos cambiar la escena y seremos capaces de verlo entero sin necesidad de hacer la zona de dibujo más grande ni disminuir el espacio entre los años. El comportamiento que haremos será desplazar la línea de tiempo a la izquierda si ponemos el ratón en el extremo izquierdo de la zona de dibujo y desplazarlo a la derecha si lo ponemos en la zona extrema derecha.

Partimos del código final que teníamos al final de la sección 4.3.6 y le añadimos una función que nos detecte si el ratón está en los extremos de la zona de dibujo y nos devuelva cuán extremo se encuentra. Esta función la situaremos dentro de una nueva pestaña que llamaremos interacción. La idea será variar la siguiente traslación fija, que se encuentra dentro de `draw`:

```
translate(medidaFuente/2-9.5*espacioAnual,medidaFuente/2);
```

utilizando una variable `panHorizontal`, que cambiará según lo que nos vaya devolviendo la función anterior (de detección del ratón):

```
translate(-panHorizontal,medidaFuente/2);
//(_7_050)
```

El motivo del signo negativo es que si, por ejemplo, queremos crear la sensación de desplazarnos hacia la derecha, tendremos que dibujar el mundo hacia la izquierda. La variable `panHorizontal` estará inicializada en `setup` al valor que tenga la primera persona y le restará un margen igual al tamaño de dos personas:

```
panHorizontal = posx[0]-medidaPersona*2;
//(_7_050)
```

Esta variable se actualizará en la función `draw` usando la función de detección del ratón, antes de dibujar ni ejecutar ninguna transformación:

```
panHorizontal = panHorizontal + detectarMouse();
```

```
//(_7_050)
```

La función de `detectarMouse` devolverá la modificación de la traslación horizontal, de forma que, cuanto más cerca del margen estemos, más modificación provocaremos:

```
float detectarMouse(){
  if (mouseX<width*0.2 && panHorizontal>posx[0]-medidaPersona*2){
     return mouseX-width*0.2;
  }
  if (mouseX>width*0.8 && panHorizontal<posx[posx.length-1]-width+2*medidaFuente){
     return mouseX-width*0.8;
  }
  return 0;
}
//(_7_050)
```

Hemos tomado unos márgenes del 20% de la anchura de la zona de dibujo. Si no se encuentran dentro de los márgenes se devolverá 0, lo que provoca una traslación horizontal nula. Si nos encontramos dentro de los márgenes, cuanto más extrema se encuentre la posición del ratón, más grande será el valor devuelto. Si se encuentra a la izquierda, el valor será negativo (hará decrecer el valor de la traslación horizontal provocada por `panHorizontal`) y, si se encuentra a la derecha, positivo (haciendo crecer el valor de la traslación horizontal).

8.5. Dibujo selectivo

Con el código que llevamos hasta este punto estamos dibujando todo el mundo, pero solo visualizamos una ventana concreta. La pregunta es ¿hay que dibujarlo todo si solo veremos una parte? La respuesta es que no. Lo que no se ve no hay que dibujarlo. Así hacemos nuestro código más eficaz y no consumimos recursos de la máquina inútilmente (dibujar algo que no se ve no tiene mucho sentido). Para hacerlo, tenemos que modificar la función `dibujarPoblación` y hacer que solo dibuje a las personas que salgan en la zona del dibujo. Tenemos información de la anchura de la zona de dibujo (`width`) y también tenemos la información de la traslación horizontal (`panHorizontal`). Dado que tenemos los datos ordenados, podemos empezar dibujando a la persona que se encuentre con un valor de traslación igual a `panHorizontal` (menos un pequeño margen) e ir dibujando a las siguientes hasta llegar a la que se encuentre en `panHorizontal+width`, más un margen (o hasta que llegue a la última persona):

```
void dibujarPoblacion(){
  int i = 0;
  pushMatrix();
  translate(0,medidaFuente+medidaPersona/2);
```

```
    while(posx[i]<panHorizontal-medidaPersona){
       i++;
    }
    while(posx[i]<panHorizontal+width+medidaPersona){
       pushMatrix();
       translate(0,amplitudOscilacion*cos(radians(angulo+i+ int(datos[i][5])*80)));
       dibujarPersona(posx[i],posy[i],int(datos[i][5]),0);
       popMatrix();
       i++;
       if (i>=datos.length){
         break;
       }
    }
    popMatrix();
  }
  //(_7_060)
```

Este es un tipo de cambio en el código que no tiene ningún impacto visual directo, sino indirecto. El hecho de que liberemos a la máquina de hacer cálculos quiere decir que podemos aprovechar para dibujar más objetos o efectos visuales que enriquezcan la aplicación.

8.6. Detección de la persona con *rollover*

A continuación, vamos a codificar el comportamiento del *rollover* para saber si el ratón está sobre alguna persona en concreto. En este caso, la marcaremos como seleccionada y la pintaremos con un contorno blanco para saber que estamos encima de ella. Para llevarlo a cabo, haremos los siguientes añadidos: primero, comprobar para todas las personas que dibujamos (siguiendo con la filosofía de antes, no hay que mirar si estamos encima de personas que no salen en la zona de dibujo) que estamos encima de alguna de ellas; segundo, si estamos encima de alguna, dibujarla con el parámetro de `seleccionada` a 1. Igual que en el código que ejecutamos para detectar si estábamos encima de una persona, la comprobación la haremos cada vez que redibujamos la zona de dibujo, puesto que la posición del ratón puede cambiar en cualquier momento.

Antes de nada, aumentaremos la zona visible a 400 x 400 para tener más espacio y poder observar mejor el efecto de selección de la persona. Seguidamente, definimos la función `detectarPersona(i)`, que nos devolverá 1 si el ratón se encuentra sobre la persona número `i` o 0 en caso contrario:

```
int personaActiva(int i){
  float limiteSuperior = posy[i]-medidaPersona/2+3*medidaFuente/2+
                  medidaPersona/2+amplitudOscilacion*cos(radians(angulo+i+ int(datos[i][5])*80));
  float limiteInferior = posy[i]+medidaPersona/2+3*medidaFuente/2+
                  medidaPersona/2+amplitudOscilacion*cos(radians(angulo+i+ int(datos[i][5])*80));
```

```
    float limiteIzquierda = posx[i]-panHorizontal-medidaPersona/2;
    float limiteDerecha = posx[i]-panHorizontal+medidaPersona/2;
    if (mouseX>=limiteIzquierda && mouseX<=limiteDerecha &&
        mouseY>=limiteSuperior && mouseY<=limiteInferior) {
       return 1;
    } else {
       return 0;
    }
  }
  //(_7_070)
```

Podemos ver que el código es muy similar al que hemos utilizado para el ejemplo anterior de *rollover*, solo que la `posicionX` y `posicionY` las hemos fijado acumulando todas las transformaciones de traslación que hemos realizado hasta el punto de la llamada a esta función, es decir, hemos acumulado, por horizontal y vertical, las siguientes traslaciones:

```
translate(-panHorizontal,medidaFuente/2);
translate(0,medidaFuente+medidaPersona/2);
translate(0,amplitudOscilacion*cos(radians(angulo+i+ int(datos[i][5])*80)));
```

Así, podemos utilizar el mismo código, solo que `posicionX` es igual a la acumulación siguiente:

```
-panHorizontal+0+0 = -panHorizontal
```

y `posicionY` se convierte en:

```
medidaFuente/2+medidaFuente+medidaPersona/2+ amplitudOscilacion*
   cos(radians(angulo+i+int(datos[i][5])*80)) = 3*medidaFuente/2+medidaPersona/2+
   amplitudOscilacion*cos(radians(angulo+i+int(datos[i][5])*80))
```

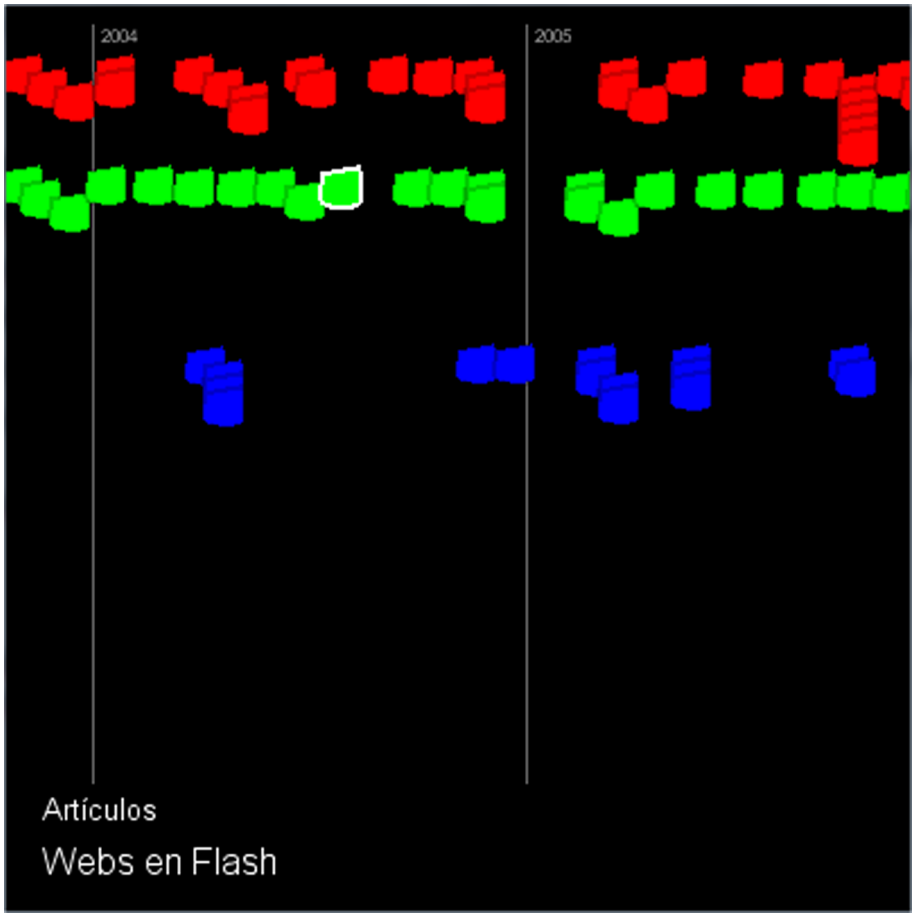
Solo hay una pequeña diferencia adicional: en este caso, devolvemos un entero (1 o 0), no un booleano (`true` o `false`), ya que lo usaremos en otra función que necesita recibir un entero y no un booleano.

Una vez tenemos la función para detectar si el ratón está encima de una persona, tenemos que utilizarla. Cuando notamos que estamos encima, tenemos que dibujar a la persona seleccionada. Dado que la función `dibujarPoblacion` pinta por pantalla a todas las personas, podemos utilizarla para que, cuando pinte a la persona, la pinte seleccionada. Solo tenemos que saber, antes de pintarla, si está seleccionada, utilizando el resultado de `detectarPersona` como cuarto parámetro de `dibujarPersona`, a través de la variable local activa:

```
void dibujarPoblacion(){
  int i = 0;
```

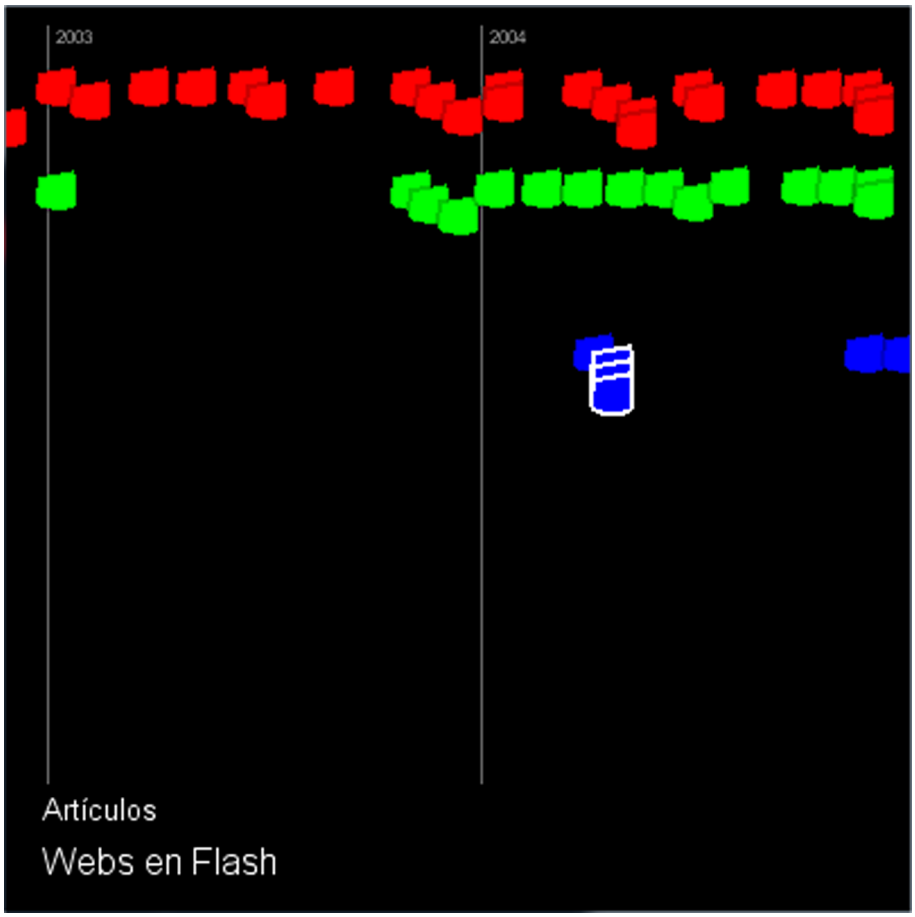
```
    int activa = 0;
...
        activa = personaActiva(i);
        dibujarPersona(posx[i],posy[i],int(datos[i][5]),activa);
...
//(_7_070)
```

Figura 35. Selección de una persona al pasar el ratón por encima



No obstante, el código que hemos propuesto tiene un problema, y es que, cuando las personas se solapan, puede llegar a seleccionar más de una, como se puede ver en la figura 36.

Figura 36. Selección de varias personas simultáneamente



Para evitar este efecto, podemos hacer que se seleccione solo la primera persona que se encuentre en la posición del ratón e ignorar el resto. Esto se puede conseguir añadiendo una variable adicional en `dibujarPoblacion`:

```
...
  int activa = 0;
  int encontrada = 0;
...
    if (encontrada == 0){
      activa = personaActiva(i);
      encontrada = activa;
    } else {
      activa = 0;
    }
    dibujarPersona(posx[i],posy[i],int(datos[i][5]),activa);
...
//(_7_080)
```

Hay un efecto adicional que tiene este código, y es que deja de comprobar si la persona está activa una vez encuentra a una, así se ahorra también tiempo de proceso de la máquina.

8.7. Escritura del texto en función de la persona detectada

Una vez hemos detectado a la persona, no solo podemos mostrarla con un contorno diferenciado. Por ejemplo, podemos mostrar el texto asociado en la zona de texto inferior utilizando la función de escritura de texto `escribeTexto` asociado a la figura 33. De hecho, como tenemos más espacio, ahora que hemos ampliado la zona de dibujo, también escribiremos en ella la fecha almacenada en datos. Modificaremos la función convenientemente y la redenominaremos para seguir el mismo formato de verbo en infinitivo del resto de funciones:

```
void escribirTexto(String categoría, String nombre, String fecha){
  textAlign(LEFT);
  fill(#FFFFFF);
  textSize(medidaFuente);
  text(nombre,medidaFuente,height-medidaFuente);
  textSize(medidaFuente*0.8);
  text(categoría + "::" + fecha,medidaFuente,height-2.5*medidaFuente);
}
```

Esta función la situaremos dentro de la función `draw`, al final. Pero para llamarla necesitamos saber qué persona es la que actualmente está seleccionada. Dado que, una vez encontramos a la persona seleccionada dejamos de buscar más, podemos incluir la siguiente línea dentro de `personaActiva`:

```
...
  if (mouseX>=limiteIzquierda && mouseX<=limiteDerecha && mouseY>=limiteSuperior &&
      mouseY<=limiteInferior) {
    persona = i;
    return 1;
  } else {
...
```

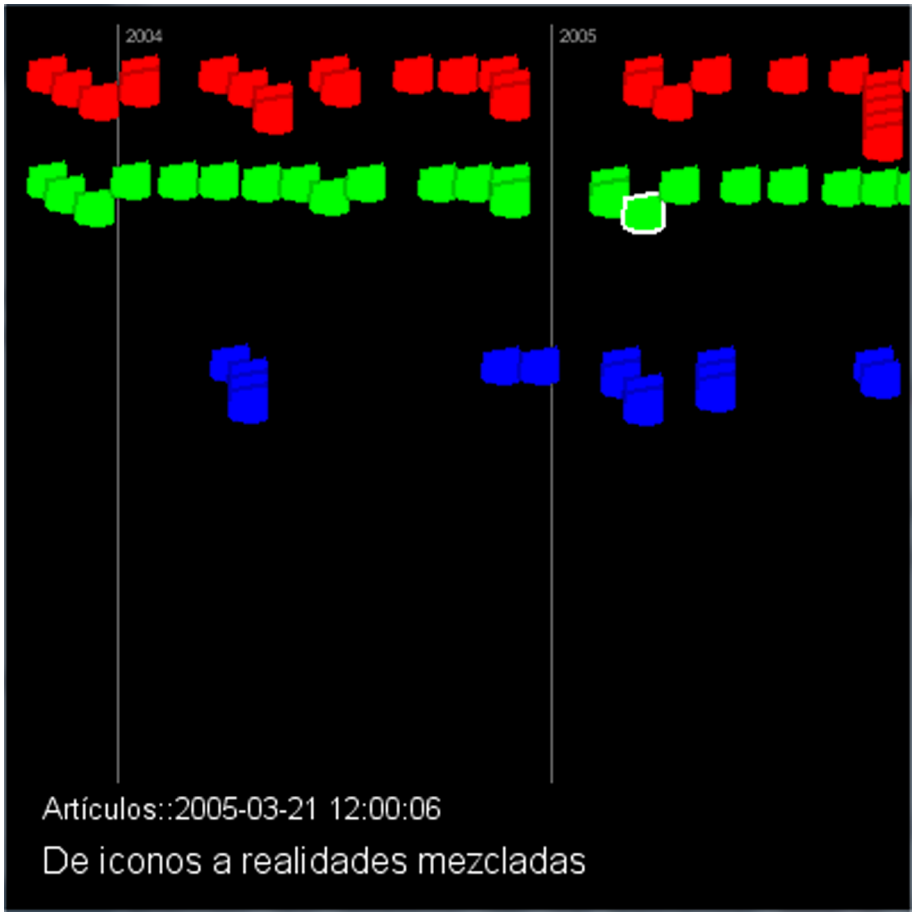
donde `persona` es una variable global que contiene la última persona por la que el ratón ha pasado por encima. Dado que al principio no hemos pasado todavía por encima de ninguna persona, indicaremos este hecho inicializándola con un valor de -1 dentro de la función `setup`.

Ahora ya podemos insertar la función de `escribirTexto` al final de la función `draw`, de la siguiente forma:

```
...
  if (persona>-1){
    escribirTexto(datos[persona][4],datos[persona][0], datos[persona][2]);
  }
}
```

```
//(_7_090)
```

Figura 37. Selección de varias personas simultáneamente



Notemos que, cuando no hemos pasado todavía por ninguna persona (cuando persona tiene un valor de -1), no escribimos ningún texto, puesto que no entramos dentro del `if`.

8.8. Enlace a páginas web

También podemos hacer que la aplicación nos abra la página web que representa a la persona si pulsamos sobre una persona. Para hacerlo, utilizaremos la función de `mouseReleased` () , que se activa cuando soltamos cualquier botón del ratón (para soltarlo, antes lo tenemos que haber pulsado). Lo hacemos así porque, si lo activamos cuando pulsamos el botón, se activa la llamada y abrimos la página web con el botón pulsado, lo que puede generar acciones no deseadas en la página web adonde entremos (no solemos navegar con el botón del ratón apretado).

```
void mouseReleased(){
  if (persona>-1){
    if (personaActiva(persona)==1){
      link(datos[persona][1]);
    }
  }
```

```
    }
    //(_7_100)
```

Cuando entramos en la función `mouseReleased()` (porque hemos levantado el dedo del botón) lo primero que hacemos es comprobar que hemos pasado por encima de alguna persona. Si es así, comprobamos que todavía nos encontramos encima de esta. Y, si es así, abrimos la dirección URL que ha almacenado en el segundo campo de la persona activa. Fijémonos en que no puede pasar que estemos encima de una persona que no sea la identificada por persona: si pulsamos sobre una persona, antes de pulsar, estamos encima de ella y esto hace que se dibuje seleccionada y quede como la última persona activa, tal como hemos visto antes.

9. Utilización del código con processing.js

Este apartado pretende dejar el código listo para poder utilizarlo con processing.js y con WordPress, según hemos visto antes en este módulo.

En la pestaña datos, dejaremos todas las variables que nos vendrán impuestas desde WordPress, de forma que solo tengamos que suprimirlas y poner el código php correspondiente. Esto nos implica que la función `dibujarPersona` cambia las constantes de color de categorías por las variables `colorCategoria0`, `colorCategoria1`, `colorCategoria2` y `colorCategoria3`; las medidas de la zona de dibujo también quedan como variables `ancho` y `alto`, y el color de fondo utilizando en `background` (en `setup` y `draw`) también queda como variable `colorFondo`. Estas siete nuevas variables, junto con los datos, quedan en la pestaña datos. Tenemos en el `sketch _8_010` estas modificaciones.

Por otro lado, el código que tenemos hasta ahora es un código organizado en pestañas en el IDE de Processing. De hecho, esta organización en pestañas es ficticia: para Processing todo el código va seguido y pone el contenido de las pestañas uno debajo del otro. Si queremos copiar todo el código en el editor que hay en WordPress, solo tenemos que hacer copiar y pegar del contenido de cada pestaña e ir poniéndolo uno debajo del anterior. Recordemos que la pestaña datos no se tiene que copiar y se tienen que igualar las ocho variables que contiene en las que contienen los datos de la base de datos de WordPress.