TEMA 9

Creando un videojuego

Curso de Python

Alfredo Sánchez Sánchez
@alfsan04
www.educoteca.com

ÍNDICE

Nuestro primer programa gráfico interactivo 2
Operaciones booleanas 4
Dibujando una raqueta 7
Controlando la raqueta 11
Movimiento manteniendo la tecla pulsada 13
Movimiento con cada pulsación de la tecla 16
La pelota rebota en la raqueta 20
Puntuación 24
Game over 30
Mejorando el juego 31
Instrucciones de código usadas 32
Reto 33

Nuestro primer programa gráfico interactivo

Este tema va a ser el que más se parezca a una guía paso por paso para conseguir un programa, es decir, unas instrucciones que, si son seguidas, dan un resultado. Se trabajan muchos conceptos y es necesario hacerlo de esta forma pero, como siempre, tendrás una serie de preguntas a lo largo del documento y, además, un reto final que hará que tengas que poner de tu parte.

Al finalizar el capítulo anterior terminamos consiguiendo que una esfera rebotase por las paredes de una pantalla. A continuación tienes un código que consigue dicho efecto. Deberías haberlo conseguido con los retos del tema anterior, pero mira el código a continuación y pruébalo en Processing para asegurarte que funcione:

```
"""Declaramos listas globales para la posicion y el movimiento posicion
sera una lista de dos elementos, uno para la posicion en x y otro para
la posicion en y.movimiento sera otra lista de dos elementos, uno para
el movimiento en x y otro para el movimiento en y"""
posicion = [20, 200]
movimiento = [3, 2]
#definimos funcion setup con tamaño de pantalla
def setup():
    size(600,400)
#definimos funcion draw con fondo amarillo
def draw():
    background(227,220,7)
    #declaramos las listas globales que vamos a usar
    global posicion
    global movimiento
    #variable con tamaño del diametro de la esfera
    diametro = 30
    #rellenamos la elipse de azul
    fill(14,82,175)
    #declaramos la posicion inicial de la elipse y su tamaño
    ellipse(posicion[0],posicion[1],diametro,diametro)
    #iniciamos el movimiento en cada iteración para x e y
    posicion[0] = posicion[0] + movimiento[0]
    posicion[1] = posicion[1] + movimiento[1]
    #condicional para producir el rebote horizontal
    if posicion[0] > width - diametro/2 or posicion[0] < diametro/2:
        movimiento[0] = -movimiento[0]
    #condicional para producir el rebote vertical
    if posicion[1] > height - diametro/2 or posicion[1] < diametro/2:
        movimiento[1] = -movimiento[1]
```

Voy a hacerte un par de aclaraciones sobre el texto, para que puedas entender ciertas cosas que igual se te escapan:

Tanto para la posición como para el movimiento (*posicion* y *movimiento*) he usado dos listas. Ambas variables deben tener un valor para la coordenada X y otro para la coordenada Y. En el caso de *posicion* esos dos valores indican dónde debe el programa situar la esfera en cada iteración. En el caso de *movimiento* el valor de cada coordenada es cuánto se debe mover la esfera en cada dirección, en X (horizontal) y en Y (vertical).

Podría haber declarado varias variables para el mismo efecto, algo del estilo:

```
posicion_en_X = 20
posicion_en_Y = 200
movimiento_en_X = 3
movimiento_en_Y = 2
```

Pero la realidad es que es bastante más cómodo declarar estas variables como listas:

```
posicion = [20, 200]
movimiento = [3, 2]
```

Una vez tengo las listas creadas para acceder a sus valores y modificarlos sólo tendré que tener cuidado al llamarlas indicando a qué índice quiero acceder. Tenemos el índice 0 para el primer valor y el índice 1 para el segundo, de ahí que para sumar a la posición el movimiento utilicemos la siguiente expresión:

```
posicion[0] = posicion[0] + movimiento[0]
posicion[1] = posicion[1] + movimiento[1]
```

La nueva posición para cada índice será igual a la que tenía previamente sumándole el movimiento.

Operaciones booleanas

Otra cosa que he utilizado y es tremendamente útil es una operación booleana denominada *or*.

La operación *or* se engloba dentro de una serie de operaciones simples que se pueden introducir en la comprobación de cualquier condicional o bucle. Se denominan operaciones booleanas por provenir del Álgebra de Boole, nombre recibido en honor al matemático George Boole.

El álgebra de Boole trabaja con conjuntos, dichos conjuntos para nuestro caso, al tratarse de un sistema binario, se componen de dos estados → 1 y 0 o True y False.

Para comprender las operaciones booleanas primero vamos a profundizar un poquito más en las comparaciones que estamos haciendo en los condicionales y bucles. Vamos a llamar a estas comparaciones **eventos** (aunque el nombre evento en programación no se usa para esto, pero nos viene bien usarlo ahora).

El resultado de los eventos siempre va a ser *True* o *False*. Si el evento es correcto, el resultado es *True* y si no es *False*:

```
>>> diametro = 30
>>> diametro == 30
True
>>> diametro > 10
True
>>> diametro < 10
False
>>>
```

De esta forma, tenemos que entender que cualquier comparación que hagamos resultará en un valor binario.

Las operaciones booleanas que vamos a usar son muy sencillas. Las usaremos para combinar varias comprobaciones. Por ejemplo, en nuestro caso, hemos usado una comprobación doble para saber si la pelota se está saliendo por la parte derecha del área de visualización o se está saliendo por la izquierda.

Veamos dicho código:

```
if posicion[0] > width - diametro/2 or posicion[0] < diametro/2:
   movimiento[0] = -movimiento[0]
```

En este código tenemos dos eventos:

- El primero que la posición del centro de nuestra elipse sea mayor al ancho de nuestro área de visualización menos el radio de la elipse.
- El segundo que la posición del centro de nuestra elipse sea menor a cero más el radio de nuestra elipse.

Para los radios usamos la variable diámetro divida entre 2.

En la comprobación doble hemos usado un *or*, lo cual indica que efectuaremos el código que contiene el *if* si se cumple cualquiera de las dos comprobaciones (de ahí *or* → sí se cumple esto *or* esto otro…).

Las comprobaciones boolenas que vamos a usar con más frecuencia son dos:

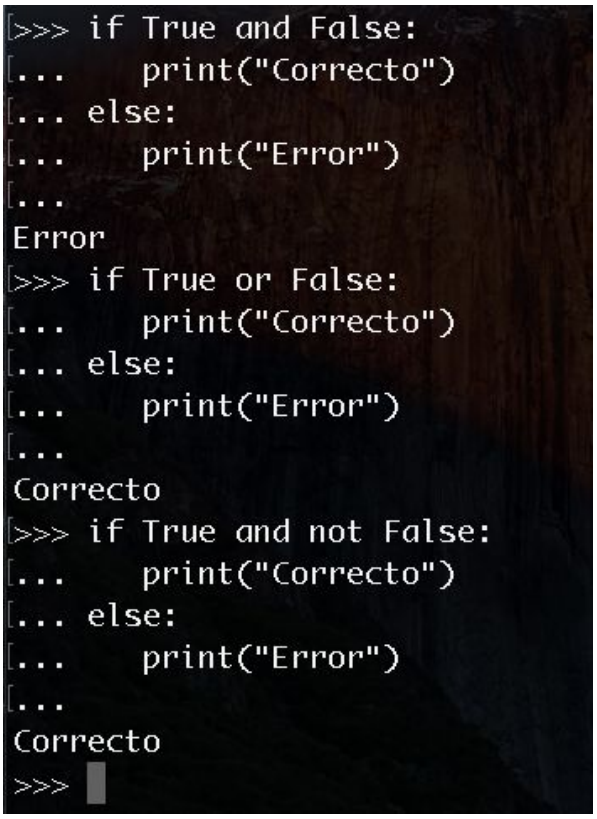
```
#para actuar si se cumplen dos eventos:
if evento1 and evento2:
   actuo
#para actuar si se cumple uno de los dos eventos:
if evento1 or evento2:
   actuo
```

Resumiendo:

- Usaré *and* cuando quiera que un código se ejecute al cumplirse dos eventos.
- Usaré *or* cuando quiera que un código se ejecute cuando se cumpla al menos un evento de una serie de ellos (también se ejecuta si se cumplen dos o más eventos).

Hay un tercer operador booleano que se puede usar y sirve para cambiar el valor de una variable booleana o una comprobación. Dicho operador es *not* y nos permite convertir en *False* una variable con valor *True*.

Veamos un pequeño ejemplo de todo lo anterior:

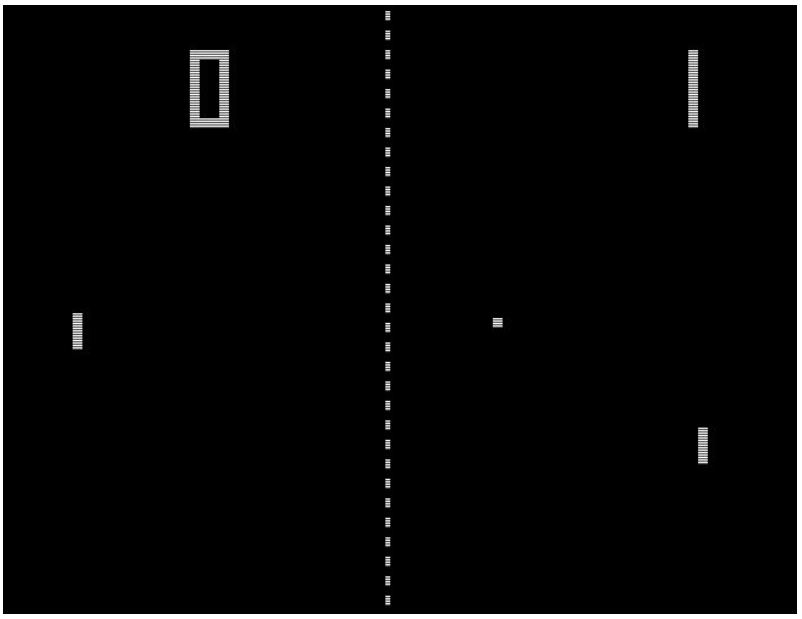


El primer *if* no se cumple porque no son *True* los dos eventos y se ha usado un *and.* En el caso del segundo, al usar un *or*, basta con que uno de los eventos tenga el valor *True* y se cumple. Por último, en el tercero usamos un *and* pero invertimos con un *not* el valor del segundo evento, que era *False*, por lo que se cumple también.

Llegados a este punto podemos seguir con nuestro juego.

Dibujando una raqueta

Vamos a crear un juego sencillo en el cual manejemos una raqueta (un rectángulo que hará las veces de raqueta) con algunas teclas del teclado y tendremos que evitar que la pelota choque con la pared en la que se encuentre la raqueta. Es decir, un pong de toda la vida.



Lo primero será diseñar un rectángulo de las proporciones adecuadas. Para crear un rectángulo, como vimos en el tema anterior, sólo debemos saber su posición tanto en X como en Y y su tamaño tanto en X como en Y:

```
rect (posicionX, posicionY, tamañoX, tamañoY)
```

Lo ideal sería referenciar tanto el tamaño como la posición a unas variables para poder modificar el aspecto del rectángulo con un simple cambio. En un pong, la raqueta sólo se mueve verticalmente, por lo que el valor de su posición en X no hace falta referenciarlo a nada, si bien el de Y va a tener que ir cambiando y para ello debe estar referido al contenido de una variable. Intenta comprender el siguiente código antes de seguir:

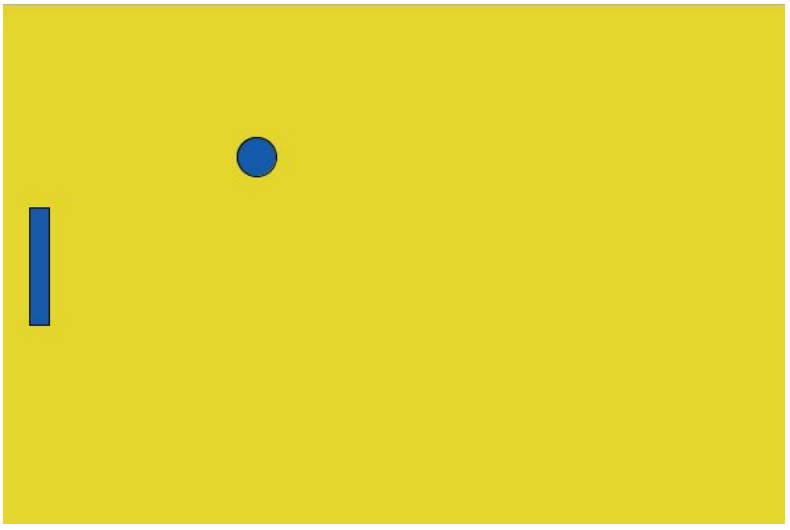
```
dimensionX = 15
dimensionY = dimensionX * 6
posicionY = height/2 - dimensionY/2
rect (20, posicionY, dimensionX, dimensionY)
```

Si introduces el código anterior en el código de la pelotita rebotando por las paredes, dentro del *draw*, verás el resultado.

La *posicionY* está rectificada para que sea justo la mitad de la altura del área de visualización (*height*). Para ello hay que darle el valor de *height/2* y a ese valor restarle la mitad de la propia altura del rectángulo, es decir, *dimensionY/2*.

Prueba el código anterior metiéndolo justo debajo de la *ellipse* en el programa que hacía rebotar a la pelotita por las paredes.

El resultado debería ser el siguiente:



Obviamente, si la pelota toca la raqueta no se ve inmutada por ello y sigue su camino, pero ya tenemos una raqueta. Los colores pueden no ser tus preferidos, pero están elegidos para ir acorde con los colores del logo de Python (con el acierto que quieras otorgarme).

Ahora que ya tenemos la raqueta, vamos a pensar en cómo podríamos moverla con las flechas del teclado.

Para empezar tendremos que pensar en si las variables de posición de la raqueta están bien puestas en la función *draw*. Esa función se ejecuta muchas veces por segundo y cada vez que se ejecute tomará todo el código que contenga y lo ejecutará, por lo que si la variable *posicionY* de la raqueta está declarada con un valor en el *draw* no podremos mover la raqueta.

El problema es que yo no puedo poner *posicionY* como global porque está referida a variables que se declaran en el *draw*. Hagamos una cosa, crearemos una variable global con valor 0 y meteremos la declaración de la variable *posicionY* en un condicional, dicho

condicional se cumplirá sólo la primera vez porque dentro del mismo a la variable global le cambiaremos el valor a 1. Observa el siguiente esquema de lo comentado:

```
variable = 0
def setup():
   size(600,400)
def draw():
   global variable
   global posicionY
   if variable == 0:
      posicionY = height/2 - dimensionY/2
      variable = 1
```

El código sería algo así, de esta forma sólo entraremos en la declaración de la variable *posicionY* la primera vez que se ejecute el *draw*. Este truco es muy usado en código para conseguir que un bucle tenga un evento o instrucción que sólo se ejecute una única vez. La variable *posicionY* debe ser mencionada como global en el draw ya que se le asignará un valor en el ámbito de un *if* para ser usada fuera del mismo.

Cuando usamos el valor de una variable que sólo toma una serie concreta de estados (la nuestra toma 2 estados) y la usamos para un condicional estamos realizando una **máquina de estados.**

Hay que indicar, también, que la variable *posicionY* es llamada en el *draw* como global, ya que ha sido declarada en un ámbito y Python no puede usarla si no lo indicamos.

Cambiemos el código del juego introduciendo lo anterior y veamos si funciona, prueba a cambiarlo sin mirar el código siguiente y luego comprueba si es correcto. Si te atascas puedes mirarlo a continuación, estás comentados los cambios en esta ocasión para su mejor comprensión, en las sucesivas veces los comentarios no aparecerán por ahorro de espacio, si te atascas vuelve a este programa para recordar para qué servía cada cosa:

```
posicion = [20, 200]
movimiento = [3, 2]
#Declaramos variable para la máquina de estados
primera_vez = 0
def setup():
    size(600,400)
def draw():
    background(227,220,7)
    global posicion
    global movimiento
    #Declaramos las variables globales que vamos a usar
```

```
    global primera_vez
    global posicionY
    diametro = 30
    fill(14,82,175)
    ellipse(posicion[0],posicion[1],diametro,diametro)
    dimensionX = 15
    dimensionY = dimensionX * 6
    """Condicional que declara la variable primera_vez en la primera
iteración"""
    if primera_vez == 0:
        posicionY = height/2 - dimensionY/2
        primera_vez = 1
    rect (20, posicionY, dimensionX, dimensionY)
    posicion[0] = posicion[0] + movimiento[0]
    posicion[1] = posicion[1] + movimiento[1]
    if posicion[0] > width - diametro/2 or posicion[0] < diametro/2:
        movimiento[0] = -movimiento[0]
    if posicion[1] > height - diametro/2 or posicion[1] < diametro/2:
        movimiento[1] = -movimiento[1]
```

¿Ya lo probaste y entendiste? Podemos seguir entonces.

Controlando la raqueta

¡Bueno! Ya tenemos lo necesario para ver cómo podemos controlar la raqueta al presionar una tecla. Para ello vamos a crear una función que se se ejecute cuando presionemos una tecla y, en función de la tecla, la raqueta subirá o bajará.

Dicha función viene incluida por defecto en Processing, aunque hay que declararla, pero no hace falta llamarla, digamos que son llamadas automáticamente al producirse un evento (de la misma forma que definimos el *draw* y el *setup* pero no los llamamos).

```
def keyPressed():
   global posicionY
   if keyCode == UP:
      posicionY = posicionY - 15
```

La función creada se activa cuando es presionada una tecla, sea cual sea. Una vez llamada habrá que indicar con qué tecla queremos que se produzca un efecto. Algunas teclas especiales tienen un código para ser llamadas. El código de la flecha hacia arriba del teclado es *UP* y se indica como *keyCode == UP*. Si queremos usar teclas de texto normal no hace falta decir *keyCode* y basta con decir *key*:

```
#Tecla especial del teclado:
keyCode == UP
#Tecla de texto:
key == "a"
```

Como verás, al presionar la flecha *UP* estamos restando 15 unidades a la *posicionY*, lo cual tiene lógica si recordamos que en Processing las magnitudes en Y crecen hacia abajo. Son 15 unidades porque tras probar he considerado que esa cantidad hacía que el movimiento fuese el adecuado en velocidad.

Prueba el código anterior situándolo al final de todo el programa para ver si funciona la tecla arriba (UP) antes de continuar.

¿Ya lo has probado? Si es así te puedes haber dado cuenta de varias cosas:

- Si dejo la tecla presionada es posible que notes que el tiempo de espera entre la primera vez que se mueve y el resto de veces que se mueve es diferente.
- Al llegar arriba nuestra raqueta no se detiene y desaparece del área de visualización.

El asunto de repetirse más lenta la segunda vez que las veces sucesivas es una cuestión de cómo está configurado el teclado. Los teclados suelen tener un retardo en la repetición, aquí tenéis el ejemplo para Mac, en Preferencias del Sistema:

En el caso de un Mac, por muy rápido que se ponga la repetición sigue habiendo cierto retardo, cosa que no es funcional. Por ello hay que tomar una decisión:

1. Si queremos que el objeto se mueva de forma continua con la tecla presionada tendremos que cambiar la forma de hacerlo…
2. …. o bien simplemente hacemos que el objeto se mueva una vez por cada vez que se presione una tecla, aunque esté presionada mucho tiempo.

Movimiento manteniendo la tecla pulsada

Para el caso 1 tendríamos simplemente que conseguir que al presionar la tecla deseada una variable global tome un valor concreto y en el *draw* hacemos que el objeto se mueva de forma continua mientras esa variable tiene el valor concreto.

Por ponerlo en modo esquema:

```
arriba = 0
def draw:
   global arriba
   if arriba == 1:
      posicionY = posicionY - 3
def keyPressed():
   if keyCode == UP:
      arriba = 1
```

Intenta introducir el código anterior en el programa, sustituyendo a las partes creadas para la función *keyPressed( )* y pruébalo. Recuerda poner la variable *arriba* en el comienzo del programa.

Si te fijas, estoy restando 3 unidades a la variable *posicionY* porque de esta forma la raqueta se mueve mucho más rápido.

Ahora sólo quedaría ver la forma en la que podemos detener la raqueta, tanto al llegar arriba como al soltar la tecla *UP*.

Para conseguir que se detenga al llegar arriba basta con indicar que sólo reste a *posicionY* si el rectángulo no ha llegado arriba, lo cual es el valor 0, por lo que bastará con añadir un *and* en el *if* que hace subir la raqueta:

```
arriba = 0
def draw:
   global arriba
   if arriba == 1 and posicionY > 0:
      posicionY = posicionY - 3
def keyPressed():
   if keyCode == UP:
      arriba = 1
```

Pruébalo para ver si realmente no consigues subir la raqueta más allá del área de visualización.

Para el evento que se produce al soltar una tecla usaremos la función *keyReleased( )* que funciona igual que la anterior, solo que entra en acción cuando una tecla deja de estar presionada, de forma que:

```
def keyReleased():
   if keyCode == UP:
      arriba = 0
```

Con esa función pararemos la raqueta al soltar la tecla *UP*. Prueba a incluirla en el código. A continuación pondré cómo quedaría todo el código con la tecla *DOWN* incluida, pero intenta hacerlo antes de ver cómo sería. Si no lo consigues míralo en el código completo que hay a continuación:

```
#Declaro todas las variables simultáneamente
posicion,movimiento,primera_vez,arriba,abajo = [20,200],[3,2],0,0,0
def setup():
    size(600,400)
def draw():
    background(227,220,7)
    global posicion, movimiento, posicionY, primera_vez, arriba, abajo
    diametro = 30
    fill(14,82,175)
    ellipse(posicion[0],posicion[1],diametro,diametro)
    dimensionX = 15
    dimensionY = dimensionX*6
    if primera_vez == 0:
        posicionY = height/2 - dimensionY/2
        primera_vez = 1
    rect (20, posicionY, dimensionX, dimensionY)
    posicion[0] = posicion[0] + movimiento[0]
    posicion[1] = posicion[1] + movimiento[1]
    if posicion[0] > width - diametro/2 or posicion[0] < diametro/2:
        movimiento[0] = -movimiento[0]
    if posicion[1] > height - diametro/2 or posicion[1] < diametro/2:
        movimiento[1] = -movimiento[1]
    #movemos la raqueta si hay que hacerlo
    if arriba == 1 and posicionY > 0:
        posicionY = posicionY - 3
    elif abajo == 1 and posicionY < height - dimensionY:
        posicionY = posicionY + 3
```

```
#funcion que activa el movimiento al presionar tecla UP y DOWN
def keyPressed():
    global arriba, abajo
    if keyCode == UP:
        arriba = 1
    elif keyCode == DOWN:
        abajo = 1
#funcion que detiene el movimiento al soltar tecla UP y DOWN
def keyReleased():
    global arriba, abajo
    if keyCode == UP:
        arriba = 0
    elif keyCode == DOWN:
        abajo = 0
```

Intenta observar cómo he optimizado la declaración de variables y las indicaciones sobre variables globales, aunando en una sola línea tanto declaraciones como variables globales que van a ser usadas:

```
#declaración conjunta de variables
posicion,movimiento,primera_vez,arriba,abajo = [20,200],[3,2],0,0,0
#indicación de varias variables globales
global posicion, movimiento, posicionY, primera_vez, arriba, abajo
```

Prueba el código e intenta familiarizarte con él. Hemos ido construyendo todo el código de forma conjunta a lo largo del tema. Si hay algo que no tengas claro vuelve a mirarlo hasta que lo entiendas antes de seguir. A continuación vamos a ver cómo hacemos el programa para que sólo avance una cantidad por cada vez que se presiona la tecla arriba o abajo.

Movimiento con cada pulsación de la tecla

Ahora el asunto es jugar con una variable que nos permita hacer el efecto de subir o bajar una única vez por cada vez que lo presionamos. Esa variable puede llamarse *teclaSuelta*. El movimiento, en esta ocasión, se va a producir en la función *keyPressed( )*, condicionándolo a que esté una tecla pulsada y que además la variable *teclaSuelta* tenga el valor *True*. Si, vamos a usar una booleana y así te acostumbras a ellas.

De esta forma iniciaremos la variable *teclaSuelta* con valor *True* y al presionar la tecla arriba moveremos la raqueta una cantidad de píxels:

```
teclaSuelta = True

def keyPressed():
   global teclaSuelta
   if keyCode == UP and teclaSuelta == True:
      posicionY = posicionY - 10
      teclaSuelta = False
```

Para probar el código anterior tendrás que modificar el programa que cambiamos para que se moviese la raqueta de forma fluida, guárdalo y copia el siguiente y pruébalo:

```
posicion,movimiento,primera_vez,teclaSuelta=[20,200],[3,2],0,True
def setup():
    size(600,400)
def draw():
    background(227,220,7)
    global posicion, movimiento, variable, posicionY, arriba, abajo
    diametro = 30
    fill(14,82,175)
    ellipse(posicion[0],posicion[1],diametro,diametro)
    dimensionX = 15
    dimensionY = dimensionX*6
    if primera_vez == 0:
        posicionY = height/2 - dimensionY/2
        primera_vez = 1
    rect (20, posicionY, dimensionX, dimensionY)
    posicion[0] = posicion[0] + movimiento[0]
    posicion[1] = posicion[1] + movimiento[1]
    if posicion[0] > width - diametro/2 or posicion[0] < diametro/2:
        movimiento[0] = -movimiento[0]
    if posicion[1] > height - diametro/2 or posicion[1] < diametro/2:
```

```
        movimiento[1] = -movimiento[1]
#funcion que activa el movimiento al presionar tecla UP
def keyPressed():
    global teclaSuelta, posicionY
    if keyCode == UP and teclaSuelta == True:
        posicionY = posicionY - 10
        teclaSuelta = False
```

Al probarlo, verás que la raqueta sube sólo la primera vez que presionamos *UP* y luego deja de subir. Es obvio que queda programar la función *keyReleased( )* que vuelva a cambiar la variable *teclaSuelta* a *True* cuando soltemos la tecla UP:

```
def keyPressed():
   global teclaSuelta
   if keyCode == UP and teclaSuelta == True:
      posicionY = posicionY - 10
      teclaSuelta = False
def keyReleased():
   global teclaSuelta
   if keyCode == UP:
      teclaSuelta = True
```

El código anterior, cada vez que soltamos la tecla *UP*, cambia el valor de la variable *teclaSuelta* a *True*, por lo que podemos hacer subir la raqueta un número de píxels con cada pulsación de la tecla *UP*.

Hasta ahora no hemos visto cómo podemos imprimir cosas en Processing. Pues bien, dentro de la interfaz de Processing hay una Consola, en la parte inferior, que muestra valores y ahí se mostrará todo lo que queramos imprimir:

En esa consola, por ejemplo, podemos imprimir el valor que tiene una variable. Prueba a incluir el siguiente código en las funciones y ejecútalo para que veas que ahí se puede mostrar el valor de una variable:

```
def keyPressed():
   global teclaSuelta
   if keyCode == UP and teclaSuelta == True:
      posicionY = posicionY - 10
      teclaSuelta = False
      print(teclaSuelta)
def keyReleased():
   global teclaSuelta
   if keyCode == UP:
      teclaSuelta = True
      print(teclaSuelta)
```

Ahora simplemente queda, igual que en el caso anterior, hacer que se detenga al llegar arriba y que baje al presionar la tecla *DOWN*:

```
def keyPressed():
   global teclaSuelta, posicionY, dimensionY
   if keyCode == UP and teclaSuelta == True:
      if posicionY > 0:
         posicionY = posicionY - 10
         teclaSuelta = False
   elif keyCode == DOWN and teclaSuelta == True:
      if posicionY < height + dimensionY:
         posicionY = posicionY + 10
         teclaSuelta = False
def keyReleased():
   global teclaSuelta
   if keyCode == UP or keyCode == DOWN:
      teclaSuelta = True
```

Volvamos a incidir en el ámbito de las variables:

- Al declarar una variable fuera de cualquier ámbito ya es declarada como global.
- Al declarar una variable en un ámbito se destruye al acabar el ámbito a no ser que la declaremos como global, en cuyo caso podrá ser usada fuera de ese ámbito.

```
variableGlobal = True
def setup():
```

```
    variableGlobal = False
def draw():
    global nuevaVariableGlobal
    nuevaVariableGlobal = True
```

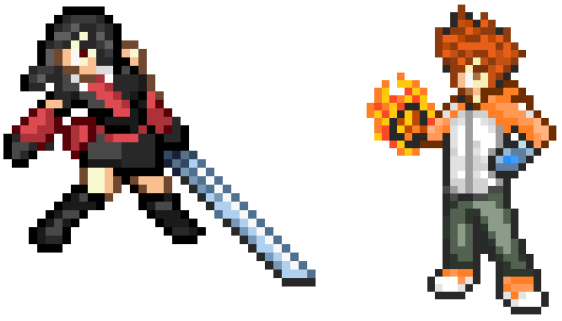
La variable declarada en el ámbito *setup* no es global porque no se ha indicado y está declarada dentro de un ámbito. Las otras dos son globales, la primera por dónde está declarada y la segunda por estar declarada en un ámbito indicando que es global.

Una vez hemos entendido esto último podemos seguir con nuestro programa. Deberías ser capaz de añadir todo lo planteado al programa general y que funcionase correctamente. Estamos sumando y restando 10 unidades a la posición de la raqueta con cada pulsación en las teclas *UP* y *DOWN*, quizá te interese que se muevan más rápido o más lento, lo dejo a tu elección.

Una vez lo has incorporado y has probado las dos opciones, movimiento fluido y movimiento por pulsaciones, vamos a seguir con el videojuego.

La pelota rebota en la raqueta

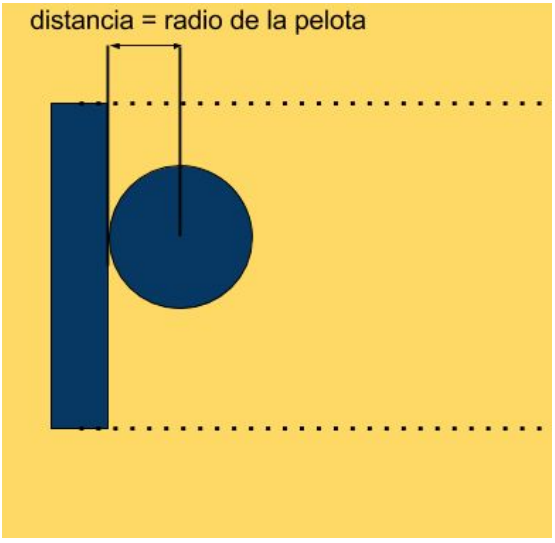
Aquí llegamos a un punto complejo en la creación de todo videojuego que vamos a analizar brevemente: el choque entre objetos. Los objetos que creamos no siempre tienen una forma muy definida y es complicado producir un evento que se inicie cuando se tocan, pongamos un ejemplo:



Si queremos que ocurra algo cuando los dos objetos se toquen tendríamos que definir cada posible punto del perímetro de cada objeto y hacer comparaciones lógicas entre todos los puntos, lo cual es muy complejo. Por ello, se suelen simplificar los contornos de los objetos como distancias desde un punto del mismo.

De esta forma, simplificamos el objeto de cara a interactuar con el mismo. Por eso, en algunos videojuegos, aunque parezca que estás tocando un objeto no se produce el evento que debería producirse o al revés, no lo estás tocando pero se inicia el evento.

Para nuestro juego va a ser sencillo, porque simplemente tendremos que seguir el siguiente esquema para que rebote la pelota al tocar el rectángulo:

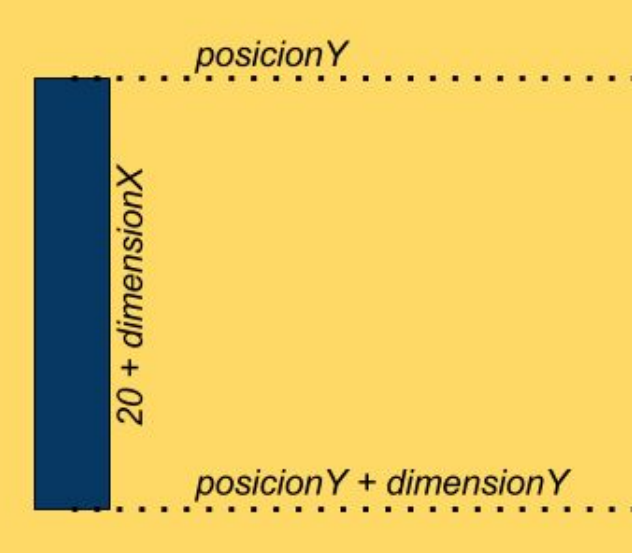


En el dibujo se puede apreciar que para considerar que la pelota ha tocado la raqueta tendremos que considerar una serie de aspectos:

1. El centro de la pelota debe estar entre la posición superior e inferior de la raqueta (zona marcada con línea de puntos).
2. La distancia entre el borde derecho de la raqueta y el centro de la pelota debe ser igual (o menor) al radio de la esfera (o elipse).

En el caso 2 he puesto una distancia igual o menor porque, al ir la esfera moviéndose en cantidades mayores a 1 puede pasar que en una iteración del bucle *draw* aun no haya tocado la raqueta y en la siguiente ya esté dentro de ella, aunque al ser cantidades pequeñas visualmente no se aprecia, si que hay que tener en cuenta esa posibilidad a la hora de programar.

Teniendo en cuenta esos dos efectos veamos cómo podríamos programar el rebote, para ello debemos tener en cuenta todos los puntos de la raqueta tal como la hemos programado:



Con lo cual, podemos ya definir la situación para la cual la pelota debería cambiar de dirección, pero tenemos que tener en cuenta una cosa. Comprobaremos primero que la pelota está a una distancia de la raqueta menor al radio, y si se cumple dicho evento comprobaremos en ese momento si estamos entre las dos líneas discontinuas del esquema anterior.

Esto es así porque puede darse la situación que movamos la raqueta tras este chequeo y la pelota, pese a estar inicialmente fuera de la raqueta, pase a estar dentro (y dentro quiere decir metida dentro) y el programa la haga rebotar. Por ello sólo podremos hacer que rebote al tocar la raqueta si se producen ambas situaciones la primera vez que la pelota esté lo suficientemente cerca de tocar la raqueta.

Prueba a incluir el siguiente código en el programa, al final del ámbito *draw*, y pruébalo. Intenta conseguir llegar tarde a la pelota y ver qué ocurre cuando llegas tarde por muy poco.

```
"""Condicional para verificar si la pelota está tocando la raqueta
cumpliéndose lo anterior"""
if posicion[0] <= 20 + dimensionX + diametro/2:
"""Condicional para verificar si el centro de la pelota está entre los dos
extremos de la raqueta"""
   if posicion[1] >= posicionY and posicion[1] <= posicionY + dimensionY:
      movimiento[0] = -movimiento[0]
```

Habrás podido ver que hay que hacer algo para que no ocurra ese efecto por el cual la pelota está todo el rato cambiando de dirección dentro de la raqueta hasta conseguir salir de ella… Básicamente hace eso porque le estamos diciendo que cambie de dirección en esa situación en la cual está por detrás de la raqueta y entre sus extremos verticales.

Si lo piensas un poco le hemos dicho que si está por detrás de una posición en X y entre dos posiciones en Y cambie de dirección, no hemos indicado que eso sólo ocurra cuando esté la pelota avanzando hacia la pared y no lo haga cuando ya haya rebotado… ¿Se te ocurre alguna forma de solucionarlo?

Lo más fácil será indicar al programa que sólo cambie de dirección si el *movimiento[0]* (que es el movimiento en el eje X) es negativo, es decir, menor que cero. Si es positivo, significará que la pelota se está trasladando de izquierda a derecha y no necesitamos que rebote en la raqueta. Para decir que sea negativo usaremos la comparación *movimiento[0] < 0*

```
def draw():
"""Condicional para verificar si la pelota está tocando la raqueta
cumpliéndose lo anterior"""
if posicion[0] <= 20 + dimensionX + diametro/2 and movimiento[0] < 0:
"""Condicional para verificar si el centro de la pelota está entre los dos
extremos de la raqueta"""
   if posicion[1] >= posicionY and posicion[1] <= posicionY + dimensionY:
      movimiento[0] = -movimiento[0]
```

Prueba la modificación explicada y mira a ver si todo es correcto. Parece que sí, ¿verdad?

Puntuación

Ahora sólo quedaría la puntuación. Cada vez que la pelota toque la pared izquierda debemos restar una vida a las vidas iniciales que tengamos. Vamos a partir de 5 vidas y crearemos una variable global llamada *vidas* con valor 5.

A continuación tendremos que restarle una vida al programa cada vez que la pelotita toque la pared izquierda. Fíjate que hemos hecho un evento que haga cambiar de dirección a la pelotita cada vez que toca un borde lateral, por lo que no podemos simplemente poner que cuando toque el borde lateral reste una vida, necesitamos indicar que, además de tocar el borde lateral, tiene que estar la pelotita a la izquierda. ¿Qué tal algo así?

```
if posicion[0] > width - diametro/2 or posicion[0] < diametro/2:
   movimiento[0] = -movimiento[0]
   if posicion[0] < 100:
      vidas = vidas - 1
```

Indicando que la *posicion[0]* sea menor que 100 estamos diciendo que la pelotita tiene que estar más bien a la izquierda. Debería funcionar, obviamente habrá que crear la variable *vidas* como global e indicar en el *draw* que vamos a usarla. Podemos imprimir el valor de la variable vidas en la *Consola* de momento, intenta conseguirlo antes de seguir.

¿Conseguido? La parte a añadir sería:

```
vidas = 5

def draw():
   global vidas
   if posicion[0] > width - diametro/2 or posicion[0] < diametro/2:
      movimiento[0] = -movimiento[0]
      if posicion[0] < 100:
         vidas = vidas - 1
         print(vidas)
```

¿Lo has probado? Deja que vayan bajando las vidas para ver qué ocurre y cómo se muestra el texto.

Antes de ver qué hacemos con las vidas negativas vamos a mostrar la cantidad de vidas de una forma un poco más graciosa.

Lo primero que intentaremos será que no ponga un simple número, queremos un texto en el cual ponga algo del estilo *Vidas: X*. El problema ya ha aparecido antes, *vidas* es una variable que contiene un entero (*int*) y no se puede combinar con un string. Intenta pensar cómo la cambiarías para producir el efecto antes de leer la solución. También ten en cuenta que vidas debería seguir siendo entero después de imprimirla porque hay que restarle unidades… ¿Lo tienes claro?

```
vidas = 5
def draw():
   global vidas
   if posicion[0] > width - diametro/2 or posicion[0] < diametro/2:
      movimiento[0] = -movimiento[0]
      if posicion[0] < 100:
         vidas = vidas - 1
         vidas = str(vidas)
         print("Vidas: "+vidas)
         vidas = int(vidas)
```

Pruébalo para que veas que funciona y que Python en Processing sigue teniendo las mismas bases vistas en el resto de temas.

Ahora queda que las vidas se muestren en el área de visualización y no en la *Consola*, así que vamos a ver cómo podemos escribir texto en el área de visualización. Lo primero de todo es comprender cómo funciona el código que nos permite mostrar texto:

```
#Cargamos el tipo de fuente que queremos usar y lo guardamos en una
variable
fuente = loadFont("miletra.vlw")
"""Llamamos a la función de texto indicando como parámetros el tipo de
texto y el tamaño del mismo"""
textFont(fuente,30)
#Escribimos el texto que queramos e indicamos en qué posición de la
pantalla aparecerá
text("Texto a mostrar", posicion_en_X, posicion_en_Y)
```

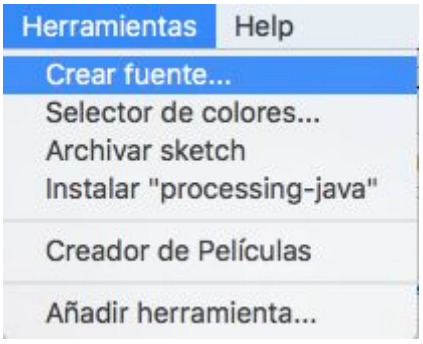
Es decir, el proceso tiene tres pasos:

1. Cargamos un tipo de fuente (es decir, un tipo de letra, como los que tiene cualquier editor de texto).

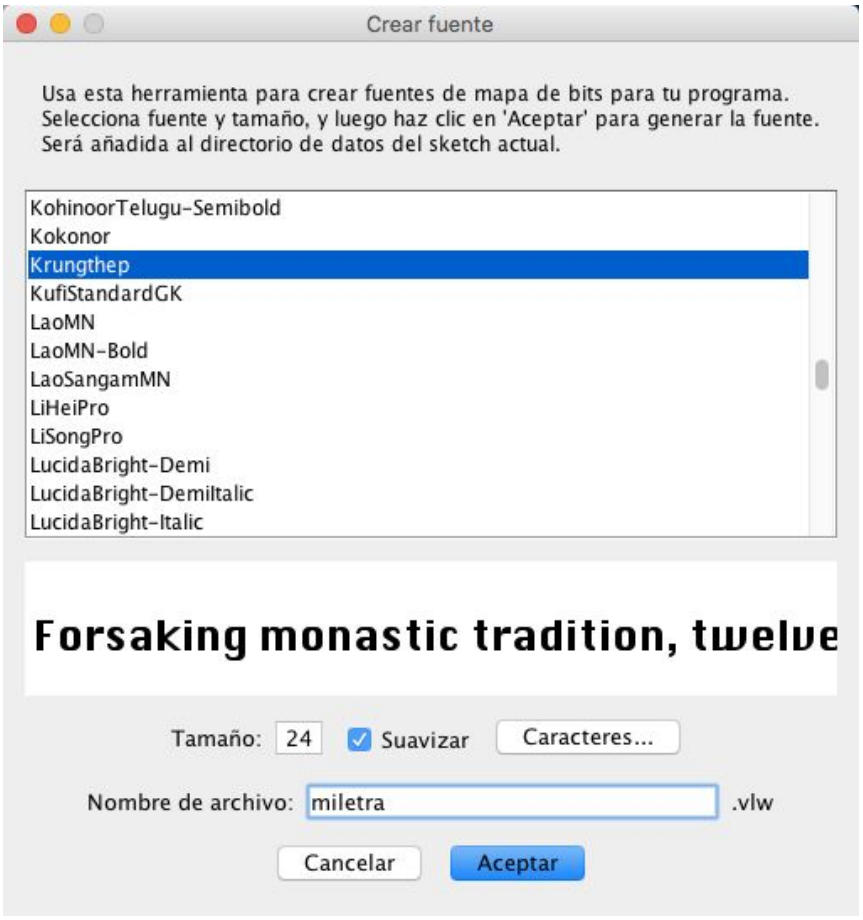
2. Indicamos que queremos usar ese tipo de fuente y un tamaño como texto.

3. Indicamos el texto que queremos escribir y su posición.

El único problema es que el tipo de fuente tiene que estar en la carpeta donde esté guardado nuestro script de Python para Processing. Por suerte Processing tiene una herramienta para crear archivos de código fuente (con extensión .vlw). Sólo tienes que ir al menú superior *Herramientas* y seleccionar *Crear fuente…*



Se abrirá una ventana en la que tendremos que elegir un tipo de fuente, un nombre con el que guardarla y un tamaño:



Yo he seleccionado un tipo de fuente similar a las que usaban los videojuegos antiguos y he seleccionado el tamaño 24. He guardado el archivo como *miletra.vlw* y en la carpeta donde tenía guardado el script aparece lo siguiente:

Se ha creado la carpeta *data* y dentro contiene:

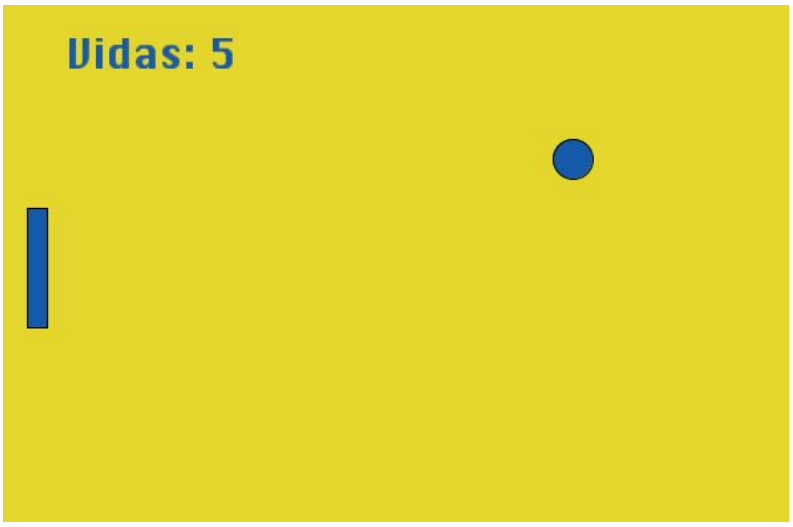
Intenta comprobar que todo es igual en tu carpeta.

Ahora que ya tenemos el tipo de fuente tenemos que escribir el código en el programa. Anteriormente estábamos escribiendo el valor de la variable *vidas* al restarle una unidad, pero si queremos escribirla en el área de visualización debemos tener en cuenta que en cada iteración del bucle *draw* **se va a refrescar el fondo** y por lo tanto lo que escribamos no va a ser permanente a no ser que lo escribamos cada vez que refresquemos el fondo.

Por ello, el cambio de valor de la variable *vidas* seguirá haciéndose en el condicional que cambia de dirección la pelotita, pero el texto que debe aparecer vamos a escribirlo siempre al comienzo del *draw*:

```
def draw():
    #Dibujo el color de fondo
    background(227,220,7)
    #Indico que voy a usar la variable vidas como global
    global vidas
    #Cargo la fuente que voy a usar en la variable fuente
    fuente = loadFont("miletra.vlw")
    """Llamo a la función textFont indicando como parámentros el tipo de
fuente y el tamaño"""
    textFont(fuente,30)
    #Convierto en string la variable vidas
    vidas = str(vidas)
    #Imprimo el texto que quiero mostrar indicando su posición
    text("Vidas: "+vidas, 50, 50)
    #Vuelvo a convertir a entero la variable vidas
    vidas = int(vidas)
```

Prueba el código anterior situándolo con el resto de instrucciones ya creadas las veces que necesites y prueba a crear una nueva fuente y cambiar la posición del texto antes de seguir.



Si te fijas, el color del texto es azul pese a no haber indicado nada… Para el color de texto sigue siendo válido el *fill* de la elipse. La primera vez que se ejecuta sale con el color por defecto y posteriormente se rellena de azul como la pelotita y la raqueta. La primera ejecución es tan rápida que ese color no se aprecia, aunque lo correcto sería poner el color de relleno al principio.

Es importante entender que la función *textFont* y la función *text* deben estar en el mismo ámbito, si no no funcionará correctamente el código. Al querer imprimir la variable con su valor actualizado deben estar en el *draw*. Si el texto fuese fijo y el fondo no cambiase en cada iteración podrían estar en el *setup*.

Game over

Bueno… ¿qué hacemos cuando la cantidad de vidas llegue a cero? Vamos a hacer algo muy sencillo pero laborioso:

- Si la cantidad de vidas es mayor que cero ejecuta el código normal.
- Si la cantidad de vidas es igual a cero pon el fondo negro y escribe Game Over.

¿Por qué digo que es laborioso? Pues bien, si queremos introducir en un condicional todo el código creado tendremos que indentar todo el código que hemos creado dentro del *draw* y hay que hacerlo manualmente porque la opción de indentación de Processing no funciona con selección múltiple muy correctamente.

Teniendo esto claro el código a incluir será:

```
def draw():
   global vidas
   if vidas > 0:
       #Aquí incluyo todo mi programa
   elif vidas == 0:
      background(0)
      fill(255)
      fuente = loadFont("miletra.vlw")
      textFont(fuente,50)
      text("Game over", 150, 200)
```

Intenta añadir los cambios al programa y ver qué ocurre. Deja que las vidas lleguen a cero. Si quieres hacer cambios en la parte de *vidas == 0* y ver cómo quedan sin tener que esperar a que se produzca el evento, cambia la variable vidas a 0 inicialmente para que aparezca directamente esa parte y poder ir modificando valores y tamaños manualmente.

Bueno, parece que ya tenemos la primera parte de nuestro juego. Como dije al principio iba a ser un proceso muy guiado porque quería un resultado muy concreto, pero ahora te toca a tí practicar todo esto haciendo la actividad.

Processing es un simulador, no has realizado un juego que pueda ser ejecutado fuera de Processing pero si te muestra todas las posibilidades de la programación con Python para realizar aplicaciones como juegos y cómo es la salida gráfica.

Mejorando el juego

No vamos a realizar más mejoras pero sí que vamos a ver algunas opciones que harían de nuestro juego más interesante. No nos engañemos, con un poco de práctica puede resultar imposible perder dado que la pelota va rebotando siempre de la misma forma y a la misma velocidad:

- La pelota podría ir más rápido según va transcurriendo el tiempo.
- Al rebotar con la raqueta podría cambiar ligeramente el ángulo de rebote de alguna forma para hacer más difícil el juego.
- Podríamos poner objetos intermedios que hiciesen rebotar la pelota.
- Podríamos hacer que esos objetos se moviesen.
- Podríamos hacer que todo ello ocurriese de forma gradual en niveles.
- Podríamos hacer una puntuación que fuese sumando también por cada vez que devuelves la pelota con la raqueta.
- Podríamos poner música y efectos.

Como verás, podríamos añadir muchas cosas, es cuestión del interés de cada uno seguir evolucionando este juego, aunque en la actividad final tendrás que realizar alguna mejora.

Instrucciones de código usadas

```
#Operación booleana que devuelve True si se cumplen dos eventos
   evento1 and evento2
"""Operación booleana que devuelve True si se cumple un evento, otro o
ambos"""
   evento1 or evento2
"""Operación que convierte en False un evento que devuelve True y
viceversa"""
   not evento
#Función que se activa al presionar una tecla
   def keyPressed():
#Instrucción para comprobar si una tecla especial está presionada
   keyCode == codigo_de_la_tecla
#Instrucción para comprobar si una tecla está presionada
   key == "tecla"
#Función que se activa al soltar una tecla
   def keyReleased():
#Cargar el tipo de fuente para usarla y aplicarla con un tamaño
   fuente = loadFont("fuente_elegida.vlw")
   textFont(fuente,tamaño)
#Escribir un texto en pantalla
   text("Texto a mostrar", posicion_en_X, posicion_en_Y)
```

Reto

Debes mejorar el juego creado haciendo que puedan jugar dos jugadores. Hemos usado las flechas para mover la raqueta izquierda y las flechas están a la derecha del teclado. Tendrás que cumplir con los siguientes criterios:

- Se mostrarán las vidas de ambos jugadores.
- Cuando un jugador pierda se acabará el juego y se indicará qué jugador ha ganado (y no Game over).
- Cada jugador moverá su raqueta con diferentes teclas. Recuerda que para teclas especiales usamos *keyCode* y para teclas de texto *key* a secas con la letra entre comillas.