Práctica 1 - Pong

David Alberto Medina Medina
Modesto Fernando Castrillon Santana

5 de febrero de 2019

Índice general

Introducción

Processing es un es un IDE *opensource* que utiliza *Java* como lenguaje de programación. Este proyecto está desarrollado y mantenido por la *Processing Foundation* que sirve como soporte de aprendizaje para instruir a estudiantes de todo el mundo en el mundo de la codificación dentro del contexto de las artes visuales.

El objetivo de este proyecto de laboratorio consiste en utilizar las herramientas de dibujo que nos proporciona el entorno de programación de *Processing* para programar nuestra versión moderna del famosísimo *Pong*, juego desarrollado por el ingeniero Allan Alcorn de *Atari, Inc.* el 29 de Noviembre de 1971 (*Wikipedia: Pong*, s.f.).

Durante el desarrollo de esta experiencia, aprenderemos a utilizar las diferentes herramientas gráficas de *Processing* así como la utilización de librerías de sonido y aplicación de sencillos algoritmos de colisiones que utilizaremos en la implementación de las físicas de nuestro juego (Souto, s.f.).

Método y materiales

2.1. Materiales

Los materiales software que han sido utilizados durante el desarrollo de esta práctica son los siguientes:

- *Processing* como framework de desarrollo.
- La librería *Sound* de *Processing*.
- Archivos de sonido *wav* originales (Caprani, 2014) de *Pong*: `ping_pong_8bit_plop.wav`, `ping_pong_8bit_beeep.wav`, `ping_pong_8bit_peeeeeep.wav`

2.2. Método

2.2.1. Clase `Pong`

La clase `Pong` contiene todos los parámetros y métodos que son llamados durante la instanciación y toda la vida útil del juego. Las primeras líneas de código están dedicadas a la declaración y/o instanciación de todos los parámetros de inicialización del juego además de importar la librería *Sound*.

```
1  import processing.sound.*;
2
3
4  private SoundFile hitWallSound;
5  private SoundFile hitPaddleSound;
6  private SoundFile pointSound;
7
8  World world;
9  Player playerL;
10 Player playerR;
11 Ball ball;
12
13 static final int WIDTH = 800;
14 static final int HEIGHT = 800;
15
16 static final int background = 0;
17 static final int elementColor[] = {255, 255, 255};
18 static final int margin = 6;
19 static final int netLines = 30;
20 static final int fps = 40;
21
22 static final int xScore1 = WIDTH/4 - 50;
23 static final int yScore1 = 150;
```

```
24 static final int xScore2 = 3*WIDTH/4 - 50;
25 static final int yScore2 = 150;
26
27 static final int playerWidth = 10;
28 static final int playerHeight = 50;
29 static final int playerMargin = 50;
30 static final int playerSpeed = 50;
31 static final int[] playerColor = {255, 255, 255};
32
33 static final int ballPos = 30;
34 static final int diameter = 20;
35 static final int ballSpeed = 5;
36 static final int[] ballColor = {255, 255, 255};
37
38 static final int scoreSize = 128;
```

El método `setup()` establece la configuración inicial de nuestro juego. El tamaño de la ventana de nuestro juego se establece a 800x800 píxeles realizando una llamada al método `size()` de *Processing*.

A continuación, se inicializamos los parámetros del mundo de nuestro juego al inicializar la clase `World`.

Los jugadores o raquetas izquierda y derecha –`playerL` y `playerR`– son instaciados llamando a la clase `Player`. Posteriormente, se ajusta su posición relativa en la pantalla de juego.

La pelota se inicializa llamando a la clase `Ball` la cual será generada en el mundo.

Finalmente, se instancian las variables que contendrán los archivos de audio que reproduciremos cuando la bola choque contra una raqueta, una pared o cuando algún jugador marque un punto –`ping_pong_8bit_beeep.wav`, `ping_pong_8bit_plop.wav` y `ping_pong_8bit_peeeeeep.wav`, respectivamente.

```
1  void setup() {
2    size(800, 800);
3
4    world = new World(background, elementColor, margin, netLines, fps
       , xScore1, yScore1, xScore2, yScore2, scoreSize);
5
6    playerR = new Player(0, 0, playerWidth, playerHeight,
       playerMargin, playerSpeed, playerColor);
7    playerR.x1 = width - playerR.x2 - playerR.margin;
8    playerR.y1 = height/2 - playerR.x2/2;
9
10   playerL = new Player(0, 0, playerWidth, playerHeight,
       playerMargin, playerSpeed, playerColor);
11   playerL.x1 = playerL.x2 + playerL.margin;
12   playerL.y1 = height/2 - playerL.x2/2;
13
14   ball = new Ball(ballPos, diameter, ballSpeed, ballSpeed, 1, 1,
       ballColor);
15   ball.spawn();
16
17   hitWallSound = new SoundFile(this, "ping_pong_8bit_plop.wav");
18   hitPaddleSound = new SoundFile(this, "ping_pong_8bit_beeep.wav");
19   pointSound = new SoundFile(this, "ping_pong_8bit_peeeeeep.wav");
20 }
```

El objetivo del método `draw()` consiste en actualizar el estado del juego en cada frame por segundo –40 fps–, actualizando el fondo de nuestro mundo, la

posición de las raquetas de los jugadores y la bola, y la puntuación en caso de que algún haya marcado un punto.

Se controla qué jugador a marcado un punto con un vector booleano de dos elementos que devuelve el método `refresh()` de la clase `Ball` siendo la primera posición del vector `true` si el jugador de la izquierda a marcado un punto o `true` el elemento de la segunda posición del vector en caso de que sea el jugador de la derecha quién marcó el punto.

```
void draw() {
  world.refresh();
  playerL.refresh();
  playerR.refresh();
  boolean[] isPoint = xBallMotion();
  yBallMotion();
  ball.refresh();
  delimitBallMotion();
  for(int i = 0; i < 2; i++) {
    if(isPoint[i]) world.refreshScore(i);
  }
}
```

El último método de la clase `Pong`, `keyPressed()`, gestiona los eventos de teclado responsables del control de las raquetas izquierda y derecha. El jugador izquierdo puede mover la raqueta hacia arriba y abajo con los botones `W` y `S` mientras que el jugador de la derecha hará lo mismo con los botones `ARRIBA` y `ABAJO`.

```
void keyPressed() {
  if(key == CODED) {
    if(keyCode == UP && playerR.y1 > (0 + world.margin)) playerR.y1
     -= playerR.speed;
    else if(keyCode == DOWN && playerR.y1 < (height - (world.margin
     + playerR.y2 + world.margin))) playerR.y1 += playerR.speed;
  } else {
    if(key == 'w' && playerL.y1 > (0 + world.margin)) playerL.y1 -=
     playerL.speed;
    else if(key == 's' && playerL.y1 < (height - (world.margin +
    playerL.y2 + world.margin))) playerL.y1 += playerL.speed;
  }
}
```

El método `delimitBallMotion()` mantiene la posición delimitida dentro del margen vertical de la pantalla de juego.

```
public void delimitBallMotion() {
  if(ball.y < (ball.DIAMETER/2 + world.margin)) ball.y = ball.
    DIAMETER/2 + world.margin;
  else if(ball.y > height - (ball.DIAMETER/2 + world.margin)) ball.
    y = height - (ball.DIAMETER/2 + world.margin);
}
```

El movimiento horizontal de la bola es gestionado por el método `xBallMotion()`. Si la bola colisiona con alguna raqueta, esta debe rebotar en dirección contraria cambiando el signo del atributo de clase `xDir`. Una colisión tiene lugar cuando la distancia horizontal de cualquier jugador a la bola es negativa y la posición vertical de la bola se encuentra dentro de la región que ocupa el alto de cada raqueta (Souto, s.f.).

Si la distancia horizontal entre la bola y la raqueta es mucho más negativa que 10 veces el ancho de la raqueta, entonces se anota un punto al jugador del

lado contrario, asignando a `true` la posición 0 un vector booleano si el punto es para el jugador de la izquierda o un `true` en la posición 1 si el punto le pertenece al jugador de la derecha, y se hace reaparecer la bola en el centro del mundo del juego con su dirección apuntando hacia el lado del jugador perdedor, de modo que la bola se dirigirá hacia el lado del perdedor.

```
private boolean[] xBallMotion() {
  boolean[] result = {false, false};
  ball.x += ball.xSpeed*ball.xDir;
  int collisionRight = playerR.x1 - (ball.x + ball.DIAMETER);
  int collisionLeft = ball.x - (playerL.x1 + ball.DIAMETER +
    playerL.x2);

  boolean isCollisionRight = collisionRight < -playerR.x2 && ball.y
     >= playerR.y1 && ball.y <= (playerR.y1 + playerR.y2);
  boolean isCollisionLeft = collisionLeft < -playerR.x2 && ball.y
    >= playerL.y1 && ball.y <= (playerL.y1 + playerL.y2);
  if(isCollisionRight || isCollisionLeft) {
    hitPaddleSound.play();
    ball.xDir = -ball.xDir;
  }
  else if(collisionRight < -playerR.x2*10) {
    pointSound.play();
    ball.xDir = 1;
    ball.spawn();
    result[1] = true;
  } else if(collisionLeft < -playerL.x2*10) {
    pointSound.play();
    ball.xDir = -1;
    ball.spawn();
    result[0] = true;
  }
  return result;
}
```

El movimiento de la bola en vertical es gestionado en el método `yBallMotion()`. Las colisiones son calculadas siguiendo el mismo algoritmo descrito en el método `xBallMotion()`. Si la bola toca la pared superior o inferior rebotará en dirección contraria.

```
private void yBallMotion() {
  ball.y += ball.ySpeed*ball.yDir;
  int collisionTop = ball.y - (ball.DIAMETER/2 + world.margin);
  int collisionBottom = height - (ball.DIAMETER/2 + ball.y + world.
    margin);
  boolean isCollisionTop = collisionTop <= 0;
  boolean isCollisionBottom = collisionBottom <= 0;
  if(isCollisionTop || isCollisionBottom) {
    hitWallSound.play();
    ball.yDir = -ball.yDir;
  }
```

2.2.2. Clase `World`

Esta clase contiene todos las rutinas responsables de la inicialización del mundo del juego: paredes, línea divisoria central o red de aparición de la bola, paredes superior e inferior, y puntación de los jugadores.

El constructor toma como parámetros de inicialización:

background Color del fondo de la pantalla de juego.

elementColor Vector que contiene el color de los elementos del mundo: superior e inferior.

margin Grosor de las paredes superior e inferior.

netLines Número de líneas que conforman la red o línea central por donde reaparecerá la bola del juego.

fps Frames por segundos del mundo.

xScore1 y xScore2 Posición relativa en pantalla sobre el eje horizontal de la puntación de cada jugador.

yScore1 e yScore2 Posición relativa en pantalla sobre el eje vertical de la puntación de cada jugador.

scoreSize Tamaño de fuente de los contadores de cada jugador.

score1 y score2 Se inicializan los marcadores de cada jugador a 0.

```
1  public int margin;
2  public int background;
3  public int[] elementColor;
4  public int netLines;
5  public int score1;
6  public int xScore1;
7  public int yScore1;
8  public int score2;
9  public int xScore2;
10 public int yScore2;
11 public int scoreSize;
12 public int fps;
13
14 public World(int background, int[] elementColor, int margin, int
     netLines, int fps, int xScore1, int yScore1, int xScore2, int
     yScore2, int scoreSize) {
15   this.background = background;
16   this.elementColor = elementColor;
17   this.margin = margin;
18   this.netLines = netLines;
19   this.fps = fps;
20
21   score1 = 0;
22   this.xScore1 = xScore1;
23   this.yScore1 = yScore1;
24
25   score2 = 0;
26   this.xScore2 = xScore2;
27   this.yScore2 = yScore2;
28
29   this.scoreSize = scoreSize;
30 }
```

El método `refresh()` permite recargar el estado del fondo o mundo de juego configurando el color de la línea y relleno –métodos `stroke()` y `fill()`– de los elementos que conforman el mundo: contadores de puntuación, paredes y red. El ratio de refresco del estado de juego se ajusta con el método `frameRate()`.

Finalmente, se contruye el mundo, la red y la puntuación de los jugadores llamando a los siguientes métodos: `buildWorld()`, `buildNet()` y `buildScore()`.

```
public void refresh() {
  stroke(elementColor[0], elementColor[1], elementColor[2]);
  fill(elementColor[0], elementColor[1], elementColor[2]);
  frameRate(fps);
  buildWorld();
  buildNet();
  buildScore();
}
```

El método `buildWorld()` ajusta el color del fondo del juego con el método de *Processing*, `background()`, así como el grosor de los elementos del mundo llamando al método `strokeWeight()`. Se generan además las paredes superior e inferior del juego.

```
private void buildWorld() {
  background(background);
  strokeWeight(margin);
  line(0,0,width,0);
  line(0,height,width,height);
}
```

La función principal de `buildNet()` consiste en pintar en la pantalla las líneas de la red central del juego. Dibujará tantas líneas equiespaciadas como indique la propiedad de clase `netLines`.

```
private void buildNet() {
  for(int i = 0; i < netLines; i++) {
    int startY = i*(height/(netLines*2) + netLines);
    int endY = startY + height/(netLines*2);
    line(width/2, startY, width/2, endY);
  }
}
```

La clase `World` contiene dos miembros de clase responsable de la gestión del contador de puntos de cada jugador –`score1` y `score2`. Dependiendo del número que se pase por parámetro se sumará 1 a la puntuación del jugador de la izquierda o derecha –0 ó 1, respectivamente.

```
public void refreshScore(int i) {
  switch(i) {
    case 1:
      score1++;
      break;
    case 0:
      score2++;
      break;
  }
}
```

2.2.3. Clase `Player`

`Player` es la clase responsable de la creación de las paletas de cada jugador en la pantalla de juego. Su constructor inicializa las siguientes propiedades de clase:

x1 e y2 Son las propiedades que gestionan la posición en el eje horizontal y vertical de las raquetas.

x1 e y2 Ajustan ancho y el alto de la raqueta del jugador.

margin Establece el margen de las raquetas con los extremos laterales de la pantalla de juego.

speed Esta propiedad contiene la velocidad con la que se mueven las raquetas de los jugadores.

playerColor Contiene un vector que contiene el valor RGB del color de las raquetas de los jugadores.

```
1     public int x1;
2     public int y1;
3     public int x2;
4     public int y2;
5     public int speed;
6     public int[] playerColor;
7     public int margin;
8
9     public Player(int x1, int y1, int x2, int y2, int margin, int
      speed, int[] playerColor) {
10        this.x1 = x1;
11        this.y1 = y1;
12        this.x2 = x2;
13        this.y2 = y2;
14        this.margin = margin;
15        this.speed = speed;
16        this.playerColor = playerColor;
17    }
```

El método `refresh()` pinta la raqueta del jugador llamando a la función `rect()` en la posición indicada por las variables `x1`, `x2`, `y1` e `y2`. Antes de llamar al método `rect()`, se comprueba si la nueva posición de la raqueta escapa de los límites superior o inferior de la pantalla para evitar que la raqueta salga por "fuera"de la pantalla.

```
1     public void refresh() {
2       stroke(playerColor[0], playerColor[1], playerColor[2]);
3       fill(playerColor[0], playerColor[1], playerColor[2]);
4
5       if(y1 > height) y1 = height-y2;
6       else if(y1 < 0) y1 = 0;
7
8       rect(x1, y1, x2, y2);
9     }
```

2.2.4. Clase `Ball`

En primer lugar se importan las librerías `Math`, `Random` y `Sound` que son utilizadas por la clase `Ball`.

```
1 import java.lang.Math;
2 import java.util.Random;
3 import processing.sound.*;
```

Esta clase contiene las siguientes propiedades:

ballPos La posición de la bola en el mundo.

DIAMETER El diámetro de la bola.

xSpeed e ySpeed La velocidad de la bola en la horizontal y vertical.

xDir e yDir La posición horizontal y vertical de la bola.

ballColor Vector que contiene los valores RGB del color de la bola.

```
    public int x;
    public int y;
    public final int DIAMETER;
    public int xSpeed;
    public int ySpeed;
    public int xDir;
    public int yDir;
    public int[] ballColor;


    public Ball(int ballPos, int DIAMETER, int xSpeed, int ySpeed,
    int xDir, int yDir, int[] ballColor) {
        this.x = ballPos;
        this.y = ballPos;
        this.DIAMETER = DIAMETER;
        this.xSpeed = xSpeed;
        this.ySpeed = ySpeed;
        this.xDir = xDir;
        this.yDir = yDir;
        this.ballColor = ballColor;
    }
```

El método `spawn()` hace reaparecer la bola en la línea central del juego o red dirigiéndose hacia la dirección del jugador ganador. La bola reaparecerá en un punto aleatoria a lo largo de la línea de la red del tablero.

```
    public void spawn() {
      x = width/2;
      y = (int)random(height);

      Random rand = new Random();
      yDir = (int)Math.pow(-1, rand.nextInt(3) + 1);
    }
```

Con `refresh()` la bola actualiza su posición en pantalla al llamar al método `ellipse()`.

```
    public void refresh() {
      stroke(ballColor[0], ballColor[1], ballColor[2]);
      fill(ballColor[0], ballColor[1], ballColor[2]);
      ellipse(x, y, DIAMETER, DIAMETER);
    }
```

Resultados y discusión

Una vez implementado nuestro código, el juego ya está listo para ejecutarse. Se ha intentado reproducir el *Pong* de 1972 en la medida de lo posible (andys arcade, 2014). Puede observarse que el juego está diseñado para jugarse para dos jugadores donde el número de máximo de puntos por jugador es ilimitado. Los sonidos de la bola son los originales del juego, generados por ondas cuadradas de diferente frecuencia (Caprani, 2014). Se ha intentado conservar en la medida de lo posible el estilo retro del juego original manteniendo ese fondo de pantalla negro con los elementos del juego resaltados en un color blanco. Como cambios a destacar, se ha optado por usar una bola redonda, que contrasta con la bola cuadrada del juego original, y una fuente distinta de los números del marcador, mucho más modernos que en el juego original.

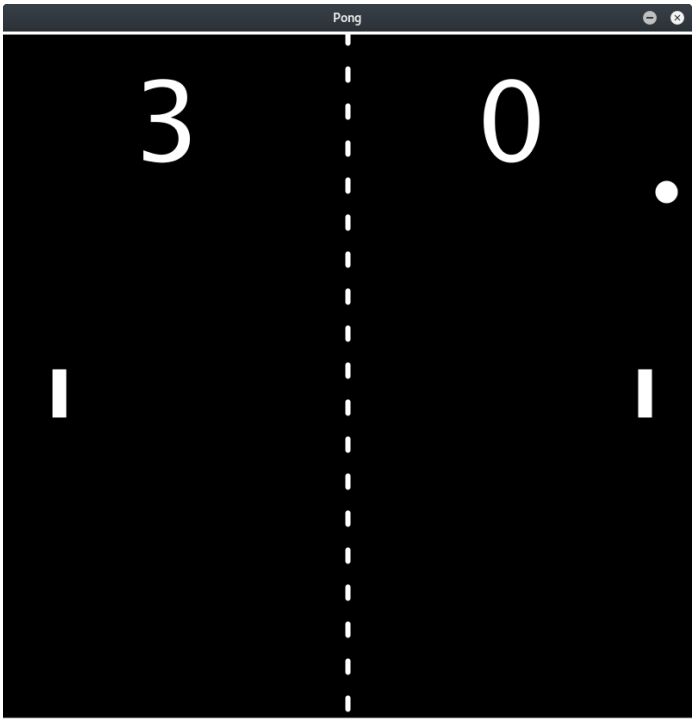

Figura 3.1: *Pong* diseñado con *Processing*

Conclusiones

En este proyecto de laboratorio hemos aprendido a utilizar el entorno de desarrollo *Processing* y a programar usando las funciones gráficas para dibujar figuras geométricas plana sobre el lienzo de juego. Queda de manifiesto la facilidad de uso de las librerías y funciones que dispone *Processing* para implementar cualquier proyecto gráfico.

Con el objeto de hacer el juego funcional y atractivo, hemos aprendido a usar la librería *Sound* para reproducir los efectos de sonido del juego y hemos aprendido a utilizar algoritmos sencillos de detección de colisiones para permitir que la bola rebote contra los elementos del juego.

Referencias

andys arcade. (2014). *Original Atari PONG (1972) arcade machine gameplay video.* https://www.youtube.com/watch?v=fiShX2pTz9A. Youtube. (Accessed: 2019-01-30)

Caprani, O. (2014). *The PONG Game.* http://cs.au.dk/~dsound/DigitalAudio.dir/Greenfoot/Pong.dir/Pong.html. Aarhus University: Computer Science Department. (Accessed: 2019-01-30)

Souto, N. (s.f.). *Toptal: Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects.* https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects. (Accessed: 2019-01-30)

Wikipedia: Pong. (s.f.). https://en.wikipedia.org/wiki/Pong. (Accessed: 2019-01-30)