

CREANDO INTERFACES DE USUARIO

GUIÓN DE PRÁCTICAS
2018/19

3 de octubre de 2019

Modesto Fernando Castrillón Santana, José Daniel Hernández Sosa
Universidad de Las Palmas de Gran Canaria
Escuela de Ingeniería en Informática

Esta obra está bajo licencia de Creative Commons Reconocimiento - No Comercial 4.0 Internacional 

Índice general

1. Introducción a Processing	7
1.1. Introducción	7
1.2. Instalación y entorno de programación	7
1.3. Programación en modo básico	9
1.3.1. Dibujo de primitivas básicas	9
1.3.2. Variables	13
1.3.3. Tipos de datos	14
1.3.4. Repeticiones	16
1.3.5. Dibujar un tablero de ajedrez	17
1.4. Programación en modo continuo	19
1.4.1. El bucle infinito de ejecución	19
1.4.2. Control	24
1.4.3. Interacción	30
1.4.3.1. Teclado	31
1.4.3.2. Ratón	33
1.4.4. Sonido	37
1.4.5. Exportando la pantalla	38
1.5. Otras referencias y fuentes	40
1.6. Tarea	40
2. Superficie de revolución	43
2.1. PShape	43
2.2. P3D	47
2.3. Sólido de revolución	50
2.3.1. Rotación de un punto 3D	52
2.4. Tarea	53

3. Transformaciones	55
3.1. Transformación básicas 2D	55
3.1.1. Traslación	57
3.1.2. Rotaciones	60
3.1.3. Escalado	62
3.2. Concatenación de transformaciones	62
3.3. Transformaciones básicas 3D	63
3.4. Objetos de archivo	66
3.5. Texto e imágenes	67
3.6. Tarea	69
4. Modelos cámara	71
4.1. Proyecciones	71
4.1.1. Ortográfica	72
4.1.2. Perspectiva	74
4.2. La cámara	76
4.3. Oclusión	80
4.4. Tarea	81
5. Iluminación y texturas	83
5.1. Iluminación	83
5.1.1. Luces	84
5.1.2. Materiales	91
5.2. Texturas	93
5.2.1. Imágenes	96
5.3. Shaders	98
5.4. Tarea	98
6. Procesamiento de imagen y vídeo	101
6.1. Captura de vídeo	101
6.2. Operaciones básicas	102
6.2.1. Píxeles	102
6.2.2. OpenCV	103
6.2.3. Grises	103
6.2.4. Umbralizado	105
6.2.5. Bordes	107
6.2.6. Diferencias	110

6.3. Detección	112
6.3.1. Caras	112
6.3.2. Personas	117
6.3.2.1. Kinect	117
6.3.2.2. Openpose	121
6.4. Galería	123
6.5. Tarea	124
7. Introducción a la síntesis y procesamiento de audio	127
7.1. Síntesis	127
7.2. Análisis	136
7.3. Minim	138
7.3.1. Efectos	141
7.4. Tarea	146
8. Introducción a p5.js	147
8.1. p5.js	147
8.1.1. Eventos	149
8.1.2. 3D	150
8.1.3. Imágenes	153
8.1.4. Sonido	154
8.1.5. Cámara	155
8.1.6. No todo es lienzo	155
8.2. Otras herramientas	156
8.3. Tarea	156
9. Introducción a la programación con Arduino	159
9.1. Arduino	159
9.1.1. Hardware	159
9.1.2. Software	161
9.1.3. Instalación	162
9.2. Programación	162
9.3. Algunos recursos útiles	163
9.3.1. Control del tiempo	163
9.3.2. Interrupciones	163
9.3.3. Funciones matemáticas	164
9.3.4. Generación de números aleatorios	164

9.3.5. Procesamiento de texto	164
9.4. Algunas fuentes/ejemplos adicionales	165
9.5. Tarea	165
10.Arduino y Processing	167
10.1.Comunicaciones	167
10.2.Lectura de sensores en Arduino	168
10.2.1. Conversión analógica/digital	168
10.2.2. Sensor de luz	169
10.2.3. Sensor de distancia	169
10.2.4. Giróscopos, acelerómetros, magnetómetros	169
10.3.Comunicación entre Arduino y Processing	169
10.4.Algunas fuentes/ejemplos	171
10.5.Tarea	171

Práctica 1

Introducción a Processing

1.1. INTRODUCCIÓN

Processing [Processing Foundation](#) [Accedido Enero 2019a] es un proyecto de código abierto con fines creativos basado en el lenguaje Java, y concebido como un cuaderno de dibujo para estudiantes, artistas informáticos, programadores y diseñadores. La facilidad sintáctica de Java, y la enorme comunidad existente, sirven de gran apoyo, ofreciendo un conjunto de herramientas para la creación de aplicaciones creativas. Su diseño pretende facilitar la programación que integre imágenes, animación, sonido e interacción, ofreciendo un entorno de desarrollo para prototipado rápido que además de las posibilidades de visualización, permite la integración de sensores y actuadores. Siendo además factible el desarrollo para Android, p5.js, Python, etc.

Es ampliamente utilizado en la mencionadas comunidades tanto para el aprendizaje, como la creación de prototipos y la producción audiovisual. Cubre por tanto necesidades no sólo para enseñar los fundamentos de programación, sino también como cuaderno de prototipos software, o herramienta de producción profesional. Processing está disponible en el siguiente [enlace](#)¹, si bien en la sección 1.5 se relacionan otros recursos con ejemplos, demos y bibliografía.

1.2. INSTALACIÓN Y ENTORNO DE PROGRAMACIÓN

La instalación requiere previamente realizar la descarga a través del mencionado enlace, y descomprimir. Una vez instalado, al lanzar se presenta la interfaz, que en caso de no estar

¹<http://processing.org/>

español, puede escogerse a través del menú con *File* → *Preferences*. Es posible acceder a diversos ejemplos a través de la barra de menú *Archivo* → *Ejemplos*.

El entorno de desarrollo de Processing (PDE), ver figura 1.1, consiste en un editor de texto para escribir código, un área de mensajes, una consola de texto, fichas de gestión de archivos, una barra de herramientas con botones para las acciones comunes, y una serie de menús. Cuando se ejecuta un programa, se abre en una nueva ventana llamada la ventana de visualización (*display window*).

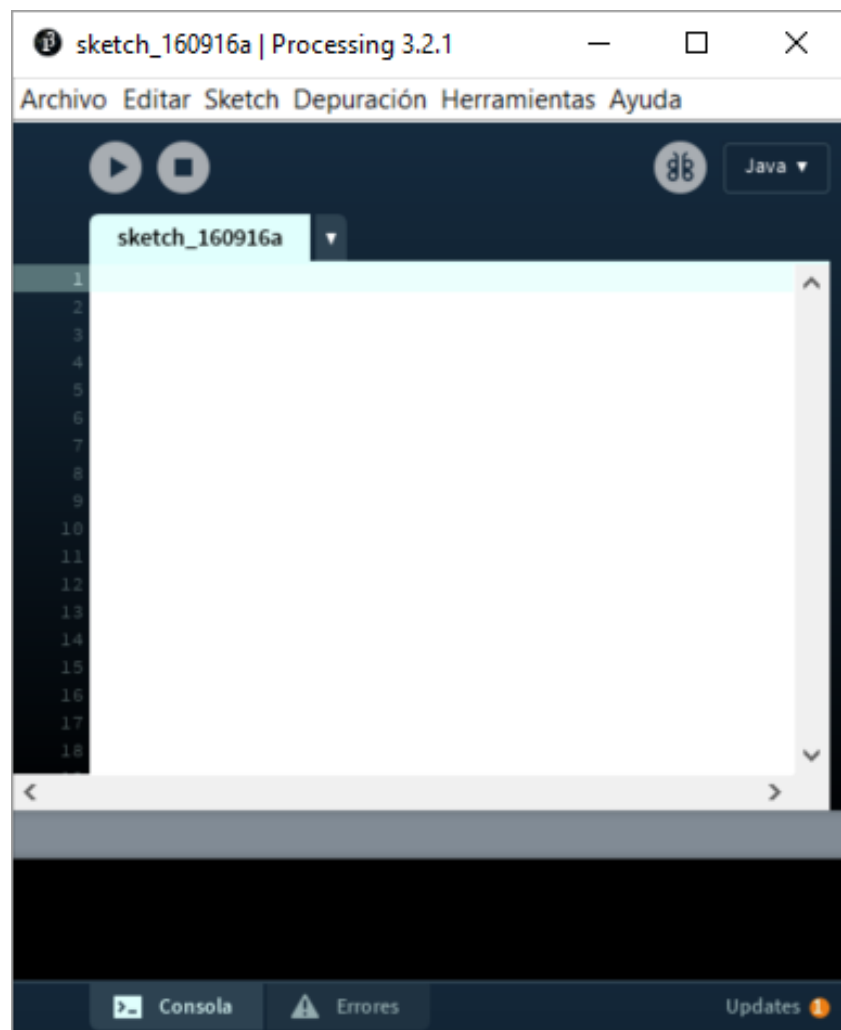


Figura 1.1: Imagen del entorno de programación, PDE, de Processing.



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

El área de mensajes, en la parte inferior, ofrece información de la salida de texto del

programa en ejecución, al hacer uso de las funciones `print()` y `println()`, además de mensajes de error, tanto en ejecución como durante la edición. Las utilidades para la depuración integradas en el entorno están disponibles desde la versión 2.0b7.

Los botones de la barra de herramientas permiten ejecutar y detener programas, ver Tabla 1.1. Comandos adicionales se encuentran dentro de la barra de menú: *Archivo*, *Editar*, *Sketch*, *Depuración*, *Herramientas*, *Ayuda*. Los submenús son sensibles al contexto, lo que significa que sólo los elementos pertinentes a la labor que se está llevando a cabo están disponibles.

Cuadro 1.1: Resumen de comandos habituales

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

1.3. PROGRAMACIÓN EN MODO BÁSICO

Processing distingue dos modos de programación: el básico, y el continuo, se describen brevemente ambos. El modo básico permite la elaboración de imágenes estáticas, es decir que no se modifican. Líneas simples de código tienen una representación directa en la pantalla.

1.3.1. Dibujo de primitivas básicas

Un ejemplo mínimo de dibujo de una línea entre dos puntos de la pantalla se presenta en el listado 1.1.

Listado 1.1: Ejemplo de dibujo de una línea

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

Se debe tener en cuenta que se emplea el sistema de coordenadas cartesiano, como es habitual, teniendo su origen en la esquina superior izquierda. Esto quiere decir que para cualquier dimensión de ventana, la coordenada [0,0] se corresponde con la esquina superior izquierda como se ilustra en la figura 1.2.

Processing también puede dibujar en tres dimensiones. En el plano imagen, la coordenada z es cero, con valores z negativos moviéndose hacia atrás en el espacio, ver figura 1.3. Cuando se realiza el dibujo en 3D simulado, la cámara se coloca en el centro de la pantalla.

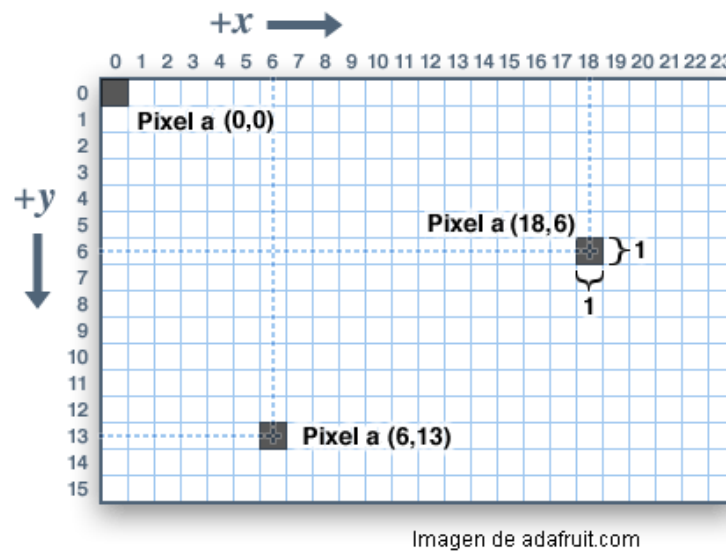


Figura 1.2: Sistema de coordenadas de la pantalla.

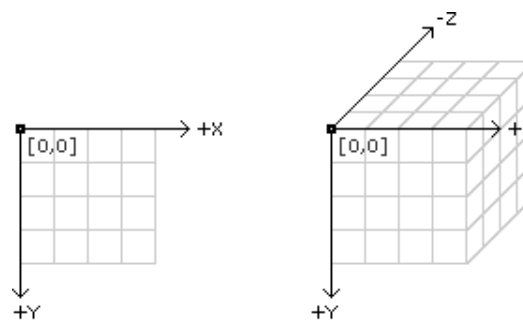


Figura 1.3: Sistema de coordenadas en 2 y 3 dimensiones (imagen de processing.org).

Un siguiente paso es dibujar dos líneas, modificando el color del pincel para cada una de ellas con la función *stroke*. En el listado 1.2, se especifica el color con una tripleta RGB. A través del [enlace²](http://www.w3schools.com/colors/colors_rgb.asp) puede practicarse con el espacio de color RGB (rojo, verde y azul) modificando los valores de cada canal. RGB es el modelo de color por defecto, si bien puede adoptarse otro con *colormode*.

Listado 1.2: Dibujo de dos líneas modificando el color

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

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

Además de la tripla RGB, señalar que el comando *stroke*, puede especificar un único

²http://www.w3schools.com/colors/colors_rgb.asp

valor, entre 0 y 255, que se interpreta como tono de gris, p.e. *stroke(0)*; especifica el color negro, y *stroke(255)*; el blanco. También la combinación RGB puede indicarse como un valor hexadecimal *stroke(#9ACD32)*;

Cabe destacar que es posible establecer de forma arbitraria el tamaño de la ventana de visualización con el comando *size*, como se muestra en el listado 1.3.

Listado 1.3: Una línea en una ventana de 640 × 360 píxeles

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

Los ejemplos previos fijan el color de las líneas, una posibilidad es asignarlo a partir de valores aleatorios haciendo uso de *random*, tal y como se muestra el listado 1.4. El valor entre paréntesis fija el valor máximo admisible.

Listado 1.4: Dibujo de una línea con color aleatorio

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

Una vez comprendido el dibujo de líneas, puede plantearse pintar un cuadrado haciendo uso de cuatro líneas. ¿Cómo se deducen los puntos extremos de cada segmento? Puede ser útil usar papel y lápiz.

El listado 1.5 dibuja el cuadrado considerando como esquinas superior izquierda e inferior derecha respectivamente los puntos (30,30) y (60,60). El ejemplo además define un color de fondo con *background*, y un grosor de línea con *strokeWeight*.

Listado 1.5: Dibujo de un cuadrado con cuatro líneas

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

Como es habitual, existen comandos que facilitan el dibujo de primitivas sencillas, el listado 1.6 muestra el comando *rect* para dibujar en este caso un cuadrado de 10 × 10. Para conocer todas las primitivas 2D, ver *2D primitives* en *Ayuda* → *Referencia*.

Listado 1.6: Dibujo de un cuadrado

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

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

El color de relleno se define con *fill()*, afectando a las primitivas a partir de ese momento, ver listado 1.7. Al igual que el resto de comandos que definen un color, la especificación de un único valor se interpreta como nivel de gris (0 negro, 255 blanco). Si se indicaran 4 valores, el último de ellos define la transparencia, el canal alfa. Las funciones *noFill* y *noStroke* cancelan respectivamente el relleno y el borde de las primitivas.

Listado 1.7: Dibujo de un cuadrado sólido

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

A modo de resumen, el listado 1.8 muestra el uso de varias primitivas 2D.

Listado 1.8: Usando varias primitivas 2D

```
size(450,450);  
  
stroke(128);  
fill(128);  
  
ellipse(200,300,120,120); //Por defecto modo con coordenadas del centro y diámetro  
  
stroke(255,0,255);  
noFill();  
strokeWeight(2);  
ellipse(400,300,60,60);  
  
stroke(123, 0, 255);  
strokeWeight(10);  
ellipse(40,123,60,60);  
  
stroke(0);  
strokeWeight(1);  
line(40,123,400,300);  
  
triangle(10, 240, 50, 245, 24, 280);  
  
fill(0);  
rect(190,290,30,50);  
  
stroke(255,0,0);  
fill(255,0,0);  
bezier(5,5,10,10,310,320,320,20);
```

Con los comandos ya conocidos, sería posible componer un dibujo estático combinando varias primitivas y colores (*rect*, *ellipse*, *line*, ...). Incluso podría reproducir el Mondrian de la figura 1.4.

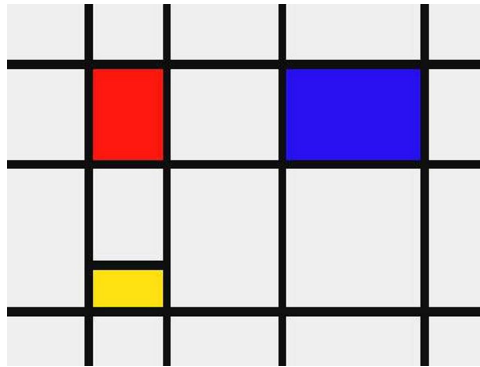


Figura 1.4: Piet Mondrian, *Composición con rojo, amarillo y azul* (1930).

1.3.2. Variables

El uso de variables aporta muchas posibilidades en la escritura de código. Para comenzar, utilizamos algunas de las variables presentes durante la ejecución, como son las dimensiones de la ventana que se almacenan respectivamente en las variables *width* y *height*. Dichas variables se utilizan en el listado 1.9, para pintar una estrella simple en el centro de la ventana, con independencia de las dimensiones fijadas con *size()*.

Listado 1.9: Dibujo de una estrella o asterisco

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

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

Una gran ventaja del uso de variables es que un programador puede definir y utilizar sus propias variables a conveniencia. En Processing es necesario declarar cualquier variable antes de utilizarla. En el listado 1.10 utiliza **I** para establecer el tamaño de la estrella.

Listado 1.10: Dibujo de una estrella variable

```
int I=10;

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

```
line (width/2,height/2-1,width/2,height/2+1);  
line (width/2+1,height/2,width/2-10,height/2);
```

El ejemplo del listado 1.11 dibuja una línea y un círculo de un determinado radio, haciendo uso de la función *ellipse* definiendo previamente el color de relleno.

Listado 1.11: Dibujo de un círculo

```
int Radio = 50 ;  
  
size (500,500);  
background (0);  
stroke (80);  
line (230,220,285,275);  
fill (150,0,0);  
ellipse (210,100,Radio, Radio);
```

1.3.3. Tipos de datos

Processing está basado en Java, por lo que deben asumirse cualquier característica de dicho lenguaje en Processing. Varios tipos de variables se muestran en el listado 1.12. Tener presente que las variables se deben declarar explícitamente y asignarles valores antes de llamar o de realizar una operación con ellas.

Listado 1.12: Ejemplos de tipos de variables

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

El listado 1.13 incluye ejemplos de accesos a vectores.

Listado 1.13: Uso de vectores

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

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

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

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

Processing incluye la clase *ArrayList* de Java, que no requiere conocer su tamaño desde el inicio. De esta forma se facilita añadir objetos a la lista, ya que el tamaño de la lista aumenta o decrece de forma automática, ver listado [1.14](#).

Listado 1.14: Uso del tipo ArrayList

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

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

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

ArrayList myVectorList ;
myVectorList = new ArrayList();

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

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

// Bucle para acceder a objetos usando ArrayList.size() y ArrayList.get()
```

```
for(int i = 0; i < myVectorList.size(); i++){
PVector V = (PVector) myVectorList.get(i); // ojo con el cast (PVector)
println(V);
}
```

1.3.4. Repeticiones

El código del listado 1.15 crea una ventana de dimensiones 800×800 en la que pintamos líneas verticales de arriba a abajo separadas entre ellas 100 píxeles. Recordar que la coordenada y de la parte superior es 0, y la inferior es 800 o *height* si usamos la variable correspondiente.

Listado 1.15: Dibujo de varias líneas verticales

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

Claramente las llamadas a la función *line* son todas muy similares, sólo varían las coordenadas x de los dos puntos. Los lenguajes de programación facilitan la especificación de llamadas repetidas por medio del uso de bucles. Una versión más compacta del dibujo de las líneas verticales se muestra en el listado 1.16. El bucle define una variable, i , a la que asignamos un valor inicial 100, un valor final, 700, y la forma en que se va modificando i con cada ejecución, en este caso añadiendo 100.

Listado 1.16: Dibujo de varias líneas verticales con un bucle

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

Las sentencias repetitivas son particularmente útiles cuando las repeticiones son cientos o miles.

1.3.5. Dibujar un tablero de ajedrez

Esta sección aborda el dibujo de un tablero de ajedrez, que contiene 64 casillas, como muestra la figura 1.5. El listado 1.17 Fija el fondo a blanco, dibujando los cuatro recuadros negros de la primera fila del tablero.

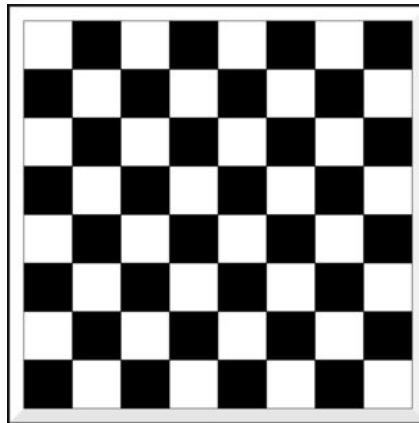


Figura 1.5: Rejilla estilo tablero de ajedrez.

Listado 1.17: Dibujo de las casillas de la primera fila

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

El listado 1.18 aplica un bucle *for* para pintar esas cuatro casillas negras.

Listado 1.18: Dibujo de las casillas de la primera fila con un bucle

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

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

El listado 1.19 da el salto para dibujar cuatro filas, cada una con su bucle particular.

Listado 1.19: Dibujo de una rejilla con varios bucles

```
size(800,800);
background(255);
```

```
fill(0);
//Primera fila

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

Como realmente cada bucle es similar a los demás, el listado 1.20 anida bucles, quedando más compacto.

Listado 1.20: Dibujo de una rejilla de ajedrez

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

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

Restan las otras cuatro filas, que resultan de una leve variación ya que se alternan los tonos blancos y negros. como muestra el listado 1.21.

Listado 1.21: Dibujo de un tablero de ajedrez

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

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

A lo largo de este apartado, se han mostrado las posibilidades del modo básico para componer una imagen estática, en la que es posible modificar los argumentos de las llamadas,

eliminar comandos, o añadir otros, pero el resultado es siempre estático, no hay movimiento, ver listado 1.22.

Listado 1.22: Modo básico

```
int Radio = 50;
void setup()
{
  size(500,500);
  background(0);
  stroke(80);
  line(230, 220, 285, 275);
  fill(150,0,0);
  ellipse(210, 100, Radio, Radio);
}
```

1.4. PROGRAMACIÓN EN MODO CONTINUO

1.4.1. El bucle infinito de ejecución

El modo continuo, cuenta con dos métodos básicos:

- *setup()* se ejecuta una única vez al lanzar el programa
- *draw()* se ejecuta por defecto de forma continua, permitiendo la escritura de funciones personalizadas, y hacer uso de la interacción.

El ejemplo el listado 1.23 usa ambos métodos, delimitando con llaves las instrucciones asociadas a cada uno de ellos. En concreto el método *setup* realiza la inicialización, en este caso fija las dimensiones de la ventana. Por otro lado, el método *draw* se encarga de dibujar. En este ejemplo, *draw* pinta una línea con color y posición aleatoria. La gran diferencia del modo continuo es que las instrucciones contenidas en el método *draw* no se ejecutan una única vez, sino que se *llama* a dicho método de forma reiterada cada segundo, por defecto 60.

Listado 1.23: Ejemplo de dibujo de líneas de color y posición aleatorios

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

void draw()
{
  stroke(random(255),random(255),random(255));
}
```

```
line(random(width),random(height),random(width),random(height));  
}
```

Haciendo uso únicamente del método *setup*, el resultado es equivalente al modo básico, dado que dicho método se ejecuta una única vez, ver listado 1.24.

Listado 1.24: Ejemplo básico

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

El código del listado 1.25 dibuja cuatro círculos en la pantalla y utiliza además una función propia llamada *circles()*. Observa el modo en que se define, y el uso de las llaves para delimitar las instrucciones contenidas en la función. En este caso concreto, el código de *draw()* sólo se ejecuta una vez, porque en *setup()* se llama a la función *noLoop()*, que cancela la repetición, resultando equivalente al modo básico.

Listado 1.25: Ejemplo de uso del método *draw*

```
void setup() {  
  size(200, 200);  
  noStroke();  
  background(255);  
  fill(0, 102, 153, 204);  
  smooth();  
  noLoop();  
}  
  
void draw() {  
  circles(40, 80);  
  circles(90, 70);  
}  
  
void circles(int x, int y) {  
  ellipse(x, y, 50, 50);  
  ellipse(x+20, y+20, 60, 60);  
}
```

En general la ejecución reiterada tiene sentido cuando exista un cambio en aquello que dibujamos ya sea por movimiento, modificación del color, etc. El ejemplo del listado 1.26 dibuja líneas desde un punto fijo, con el otro extremo aleatorio, variando también el color de forma aleatoria.

Listado 1.26: Dibujo de líneas desde un punto

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

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

En el listado 1.27 se fuerza a que sean líneas aleatorias, pero verticales y blancas.

Listado 1.27: Dibujo de líneas blancas verticales

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

void draw() {
  stroke(255);

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

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

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

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

Como se comentaba anteriormente, el color de fondo de la ventana se fija con la llamada al método *background*. Este comando nos puede ser útil si queremos forzar que se borre la pantalla antes de dibujar de nuevo, ver listado 1.28.

Listado 1.28: Ejemplo de dibujo con borrado

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

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

```
}
```

El ejemplo del listado 1.29 introduce el uso del método *frameRate*, que fija el número de llamadas al método *draw* por segundo. En este caso dibuja líneas aleatorias, pero observa la tasa de refresco.

Listado 1.29: Ejemplo de dibujo de líneas a distinta frecuencia

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

void draw() {
  background(51);

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

La tasa de refresco puede además controlarse a través de una serie de funciones:

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

El código del listado 1.30 varía la tasa de refresco en base al número de ejecuciones y el evento de pulsado de un botón del ratón.

Listado 1.30: Controlando la tasa de refresco

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

```
}  
}  
void mousePressed() {  
    loop();  
    frame = 0;  
}
```

El listado 1.31 ilustra un ejemplo de llamada a *redraw*.

Listado 1.31: Ejemplo con redraw

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

Como se ha comentado anteriormente, por defecto se establece el modo continuo con una frecuencia de 60 llamadas por segundo al método *draw()*- El código del listado 1.32 provoca el desplazamiento de un círculo por la ventana.

Listado 1.32: Desplazamiento del círculo

```
int Radio = 50;  
int cuenta = 0;  
  
void setup()  
{  
    size(500,500);  
    background(0);  
}  
  
void draw()  
{  
    background(0);  
    stroke(175,90);  
    fill(175,40);  
    println("Iteraciones: " + cuenta);  
    ellipse(20+cuenta, height/2, Radio, Radio);  
    cuenta++;  
}
```

Finalizamos el apartado señalando que además de dibujar elementos geométricos, el listado 1.33 muestra que es posible escribir texto.

Listado 1.33: Hola mundo

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

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

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

1.4.2. Control

En el ejemplo del listado 1.32 se desplaza un círculo por la ventana haciendo uso de una variable. El listado 1.34 presenta una variante algo más reducida.

Listado 1.34: Manejo de variables para dibujar circunferencias

```
int cir_x=0;

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

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

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

La modificación mostrada en el listado 1.35 evita que desaparezca la figura a la derecha de la ventana. Conociendo las dimensiones de la ventana, controlamos el valor de la posición, reiniciando el valor de la coordenada **x** del círculo cuando llega al borde derecho.

Listado 1.35: Controlando el regreso de la figura

```
int cir_x=0;

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



```
void draw() {
  background(127);

  ellipse(cir_x,50,50,50);

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

El listado 1.36 integra dos círculos a distintas velocidades.

Listado 1.36: Con dos círculos

```
float cir_rap_x = 0;
float cir_len_x = 0;

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

void draw() {
  background(27,177,245);

  fill(193,255,62);
  ellipse(cir_len_x,50, 50, 50);
  cir_len_x = cir_len_x + 1;

  fill(255,72,0);
  ellipse(cir_rap_x,50, 50, 50);
  cir_rap_x = cir_rap_x + 5;

  if(cir_len_x > 400) {
    cir_len_x = 0;
  }
  if(cir_rap_x > 400) {
    cir_rap_x = 0;
  }
}
```

El listado 1.37 modifica la forma del círculo más lento de forma aleatoria cuando se dan ciertas circunstancias.

Listado 1.37: Simulando un latido

```
float cir_rap_x = 0;
float cir_len_x = 0;

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

```
noStroke();
}

void draw() {
  background(27,177,245);

  float cir_len_tam = 50;

  if(random(10) > 9) {
    cir_len_tam = 60;
  }

  fill(193,255,62);
  ellipse(cir_len_x,50, cir_len_tam, cir_len_tam);
  cir_len_x = cir_len_x + 1;

  fill(255,72,0);
  ellipse(cir_rap_x,50, 50, 50);
  cir_rap_x = cir_rap_x + 5;

  if(cir_len_x > 400) {
    cir_len_x = 0;
  }
  if(cir_rap_x > 400) {
    cir_rap_x = 0;
  }
}
```

Se aprecia cierto efecto de pausa, se debe a que la condición no tiene en cuenta el radio para hacer reaparecer la figura. En el listado 1.38 lo modifica con una aparición más ajustada.

Listado 1.38: Aparición ajustada

```
int cir_x=0;

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

  noStroke();

  fill(100,255,32);
}

void draw() {
  background(127);

  ellipse(cir_x,50,50,50);

  cir_x=cir_x+1;

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

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

En los ejemplos previos, la modificación de la coordenada x es siempre un incremento. El listado 1.39 produce un efecto de rebote al llegar al extremo, modificando el signo del movimiento aplicado, para que pueda ser incremento o decremento. De esta forma se consigue que el círculo rebote en ambos lados.

Listado 1.39: Provocando el rebote

```
int pos=0;  
int mov=1;  
  
void setup() {  
    size(400,400);  
}  
  
void draw() {  
    background(128);  
    ellipse (pos,50,30,30);  
    pos=pos+mov;  
  
    if (pos>=400 || pos<=0)  
    {  
        mov=-mov;  
    }  
}
```

El listado 1.40 integra un *jugador*, que se desplaza en vertical acompañando al ratón, y también puede alterar el movimiento del círculo cuando haya choque.

Listado 1.40: Frontón

```
int posX=0;  
int posY=50;  
int D=30;  
int ancho=20;  
int alto=50;  
int mov=5;  
  
void setup() {  
    size(400,400);  
}  
  
void draw() {  
    background(128);  
    ellipse (posX,posY,D,D);
```

```

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

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

```

Añadimos un marcador contabilizando el número de veces que no controlamos la *pelota*. En el listado 1.41 introducimos una variable contador, inicialmente a cero, que modifica su valor cada vez que haya un choque con la pared de la derecha.

Listado 1.41: Integrando el marcador

```

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

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

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

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

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

```

```
}  
text("Goles "+goles , width/2-30, 20);  
}
```

Sería también posible mostrar un mensaje cada vez que haya un *gol*, es decir, cuando se toque la pared derecha. Mostrarlo únicamente al modificar el tanteo, hará que casi no se vea, es por ello que lo hacemos creando un contador que se decrementa cada vez que se muestra el mensaje, ver listado 1.42.

Listado 1.42: Celebrando el gol

```
int posX=0;  
int posY=50;  
int D=30;  
int ancho=20;  
int alto=50;  
int mov=5;  
int goles=0;  
int muestragol=0;  
  
void setup() {  
  size(400,400);  
}  
  
void draw() {  
  background(128);  
  ellipse(posX,posY,D,D);  
  
  //Donde se encuentra el jugador  
  int jugx=width-50;  
  int jugy=mouseY-30;  
  rect(jugx , jugy , ancho , alto );  
  
  posX=posX+mov;  
  //verificando si hay choque  
  if (posX>=400 || posX<=0 || (mov>0 && jugy<=posY+D/2 && posY-D/2<=jugy+alto && jugx<=posX+D/2 && posX-D/2<=jugx+ancho))  
  {  
    mov=-mov;  
    //Si choca con la derecha, es gol  
    if (posX>=400)  
    {  
      goles=goles+1;  
      muestragol=40;  
    }  
  }  
  text("Goles "+goles , width/2-30, 20);  
  if (muestragol>0)  
  {  
    text("GOOOOL", width/2-30, height-50);  
    muestragol=muestragol-1;  
  }  
}
```

Introducir el movimiento en los dos ejes requiere modificar la coordenada y de la pelota. El código del listado 1.43 lo integra, manteniendo el rebote en las paredes, ahora también inferior y superior.

Listado 1.43: Rebote de la bola

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

void setup() {
  size(400, 200);
  stroke(214,13,255);
  strokeWeight(7);
}
void draw() {
  background(33,234,115);
  ellipse(cir_x, cir_y, 40, 40);
  cir_x = cir_x + mov_x;
  cir_y = cir_y + mov_y;

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

1.4.3. Interacción

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

1.4.3.1. Teclado

Las acciones de teclado puede utilizarse como evento en sí o para recoger los detalles de la tecla pulsada. Por ejemplo la variable booleana *keyPressed* se activa si una tecla se presiona, en cualquier otro caso es falsa. Además, dicha variable estará activa mientras mantengamos la tecla pulsada. El código del listado 1.44 hace uso de dicha variable para desplazar una línea.

Listado 1.44: Ejemplo básico

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

Es posible recuperar la tecla pulsada. La variable *key* es de tipo char y almacena el valor de la tecla que ha sido presionada recientemente. En el listado 1.45 se muestra la tecla pulsada.

Listado 1.45: Muestra la tecla

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

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

Cada letra tiene un valor numérico en la Tabla ASCII, que también podemos mostrar, tal y como se realiza en el listado 1.46.

Listado 1.46: El código ASCII de la tecla

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

void draw() {
  background(200);
  if (keyPressed == true)
```

```
{
  int x = key;
  text(key+" ASCII "+x, 20, 20 );
}
```

Una particularidad interesante es poder identificar las teclas especiales como por ejemplo: flechas, *Shift*, *Backspace*, tabulador y otras más. Para ello, lo primero que debemos hacer es comprobar si se trata de una de estas teclas comprobando el valor de la variable *key* == *CODED*. El listado 1.47 utiliza las teclas de las flechas cambie la posición de una figura dentro del lienzo.

Listado 1.47: Ejemplo básico

```
color y = 35;
void setup() {
  size(100, 100);
}
void draw() {
  background(204);
  line(10, 50, 90, 50);
  if (key == CODED)
  {
    if (keyCode == UP)
    {
      y = 20;
    }
    else
    if (keyCode == DOWN)
    {
      y = 50;
    }
  }
  else
  {
    y = 35;
  }
  rect(25, y, 50, 30);
}
```

También es posible detectar eventos individualizados, es decir el pulsado de una tecla en concreto. El código del listado 1.48 realiza una acción al pulsar la tecla t.

Listado 1.48: Ejemplo básico evento de teclado

```
boolean drawT = false;

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



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

1.4.3.2. Ratón

En ejemplos previos hemos visto que es posible acceder a las coordenadas del ratón, simplemente haciendo uso de las variables adecuadas desde el método *draw()* como en el código de listado 1.49.

Listado 1.49: Ejemplo de uso de las coordenadas del ratón

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

Es posible usar el evento, es decir, la función que asociada con el mismo, como en el listado 1.50 que modifica el color del fondo.

Listado 1.50: Ejemplo de cambio del tono de fondo

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

```
background(gray);
}
void mousePressed() {
  gray += 20;
}
```

El ejemplo del listado 1.51 emplea el evento de pulsado para pasar del modo básico al continuo.

Listado 1.51: Paso a modo continuo

```
void setup()
{
  size(200, 200);
  stroke(255);
  noLoop();
}

float y = 100;
void draw()
{
  background(0);
  line(0, y, width, y);
  line(0, y+1, width, y+1);
  y = y - 1;
  if (y < 0) { y = height; }
}

void mousePressed()
{
  loop();
}
```

Como puede verse, *mousePressed()* es una función enlazada a la acción de pulsar un botón del ratón. Siendo posible controlar también cuando se suelta, se mueve, o si se arrastra el ratón con un botón pulsado, ver listado 1.52.

Listado 1.52: Ejemplo evento de arrastre

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

```
    moveX = mouseX;
    moveY = mouseY;
}
void mouseDragged() { // Mueve el negro
    dragX = mouseX;
    dragY = mouseY;
}
```

Como ya se ha mencionado anteriormente, las coordenadas de ratón se almacenan en las variables *mouseX* y *mouseY*, ver listado 1.53.

Listado 1.53: Coordenadas del ratón

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

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

Ambas son variables que contienen las posiciones actuales del ratón, las previas están disponibles en *pmouseX* y *pmouseY*. Podemos utilizar las coordenadas del ratón para pintar a mano alzada, ver listado 1.54.

Listado 1.54: Pintado con el «pincel»

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

    background(128);
}

void draw() {
    point(mouseX, mouseY);
}
```

Es posible restringir el pintado a sólo cuando se pulse un botón del ratón, empleando una estructura condicional, ver listado 1.55.

Listado 1.55: Pintado cuando se pulsa

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

    background(128);
}

void draw() {
```

```
if (mousePressed == true) {  
    point(mouseX,mouseY);  
}  
}
```

El listado 1.56 utiliza el teclado, en concreto las teclas del cursor arriba y abajo, para cambiar grosor del pincel (y pintamos círculo), y cualquier otra tecla para alterar el color. Se identifica primero si se ha pulsado una tecla, y luego la tecla en concreto pulsada.

Listado 1.56: Grosor y color del pincel con las teclas

```
int grosor=1;  
int R=0,G=0,B=0;  
  
void setup() {  
    size(400,400);  
    background(128);  
}  
  
void draw() {  
  
    if (mousePressed == true) {  
        point(mouseX,mouseY);  
    }  
  
    if (keyPressed == true) {  
        if (keyCode == UP) {  
            grosor = grosor+1;  
            strokeWeight(grosor);  
        }  
        else  
        {  
            if (keyCode == DOWN) {  
                if (grosor>1){  
                    grosor = grosor-1;  
                    strokeWeight(grosor);  
                }  
            }  
            else  
            {  
                R=(int)random(255);  
                G=(int)random(255);  
                B=(int)random(255);  
            }  
        }  
    }  
  
    //Muestra del pincel  
    noStroke();  
    fill(128);  
    rect(4,4,grosor+2,grosor+2);  
    fill(R,G,B);  
    ellipse(5+grosor/2,5+grosor/2,grosor,grosor);  
}
```

```
stroke(R,G,B);  
}
```

En el listado 1.57 el radio del pincel depende del valor de x o y del ratón.

Listado 1.57: Radio del pincel dependiente de ratón

```
void setup() {  
  size(640, 360);  
  noStroke();  
  rectMode(CENTER);  
}  
  
void draw() {  
  background(51);  
  fill(255, 204);  
  rect(mouseX, height/2, mouseY/2+10, mouseY/2+10);  
  fill(255, 204);  
  int inversaX = width-mouseX;  
  int inversaY = height-mouseY;  
  rect(inversaX, height/2, (inversaY/2)+10, (inversaY/2)+10);  
}
```

1.4.4. Sonido

De cara a la tarea, señalar que es posible realizar la reproducción de sonidos presentes en disco como se muestra en el listado 1.58. Una posible fuente de archivos wavs es el [enlace³](http://freewavesamples.com/). Tener en cuenta que será necesario instalar la biblioteca de sonido de la *Processing Foundation* a través del menú *Herramientas->Añadir herramientas* y buscando *Sound* en la pestaña *Libraries*.

Listado 1.58: Reproduciendo sonido

```
import processing.sound.*;  
  
int pos=0;  
int mov=5;  
  
SoundFile sonido;  
  
void setup() {  
  size(400,400);  
  
  sonido = new SoundFile(this, "E:/Intro Programacion con Processing Experto/Bass-Drum-1.wav");  
}
```

³<http://freewavesamples.com/>

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

  pos=pos+mov;

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

Al trabajar en modo continuo, pueden presentarse efectos extraños al reproducir sonido dada su duración, a menos que se lance el sonido a través de un hilo con el método *thread*, tal y como se muestra en el listado 1.59.

Listado 1.59: Latencia del sonido

```
import processing.sound.*;

int pos=0;
int mov=5;

SoundFile sonido;

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

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

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

  pos=pos+mov;

  if (pos>=400 || pos<=0){
    mov=-mov;
    thread ("Suena");
  }
}

void Suena( ) {
  sonido.play ( ) ;
}
```

1.4.5. Exportando la pantalla

Para salvar la pantalla en cada uteración, existe el comando *saveFrame*, si no se le indican argumentos, salva en formato tiff de forma correlativa. El listado 1.60, lo hace en formato png.

Listado 1.60: Salva la pantalla como fotogramas

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

void draw()
{
  background(200);
  line(0,0,mouseX,mouseY);

  saveFrame("fotograma-###.png");
}
```

De cara a exportar un gif animado, es necesario previamente instalar `GifAnimation` a través del `menú Herramientas` y buscando `gif` en la pestaña `Libraries`. El listado `code:processing-gif` presenta un pequeño ejemplo que salva lo que ocurra en pantalla hasta que pulsemos un botón del ratón.

```
\begin{lstlisting}[frame=single,caption={Exportando un gif animado},label=code:processing-gif]
import gifAnimation.*;
```

```
GifMaker fichergif;
```

```
void setup()
{
  size(400,400);
  stroke(0);

  // gif
  fichergif = new GifMaker(this, "animacion.gif");
  fichergif.setRepeat(0); // anima sin fin
}
```

```
void draw()
{
  background(200);
  line(0,0,mouseX,mouseY);

  fichergif.addFrame();
}
```

```
void mousePressed() {
  fichergif.finish(); // Finaliza captura y salva
}
```

1.5. OTRAS REFERENCIAS Y FUENTES

Señalar que a través del entorno de desarrollo de Processing están disponibles un nutrido número de ejemplos, accesibles seleccionando en la barra de menú *Archivo* → *Ejemplos*, con lo que aparece un desplegable donde por ejemplo puedes descender *Topics* → *Motion* → *Bounce*, *Basics* → *Input* → *Storinginput*. *Fun programming* propone una introducción a la programación con Processing Pazos [Accedido Enero 2019], como también lo hace el libro Nyhoff and Nyhoff [2017]. Otra interesante fuente de códigos ejemplo es *Open Processing @openprocessing* [Accedido Enero 2019].

Galerías de desarrollos realizados con Processing están disponibles en la propia página a través del enlace Processing exhibition archives Processing Foundation [Accedido Enero 2019b]. Un foro más amplio que describe propuestas desarrolladas no únicamente con Processing es *Creative Applications Network* CreativeApplications.Net [Accedido Enero 2019].

Bibliografía adicional sobre Processing, desde los primeros libros de la década pasada Greenberg [2007], Reas and Fry [2007], hasta obras más recientes Noble [2012], Runberg [2015], de Byl [2017]. El libro de Runberg cubre también openFrameworks y Arduino, siendo accesible desde la universidad a través del enlace.

1.6. TAREA

Realizar de forma individual, un juego similar al Pong para dos jugadores, ver figura 1.6. La propuesta realizada debe incluir rebote, marcador, sonido, movimiento inicial aleatorio, etc.

La entrega se debe realizar a través del campus virtual, remitiendo una memoria o el enlace a un *screencast*, que además de describir las decisiones adoptadas para la solución propuesta, identifique al estudiante, además de las referencias y herramientas utilizadas; y que cuidando el formato y estilo, debe incluir enlace al código desarrollado (p.e. github), con su correspondiente *README*, y opcionalmente un gif animado mostrando la ejecución.

BIBLIOGRAFÍA

CreativeApplications.Net. Creative applications network, Accedido Enero 2019. URL <http://www.creativeapplications.net/>.

Penny de Byl. *Creating Procedural Artworks with Processing*. Penny and Daniel de Byl, 2017.

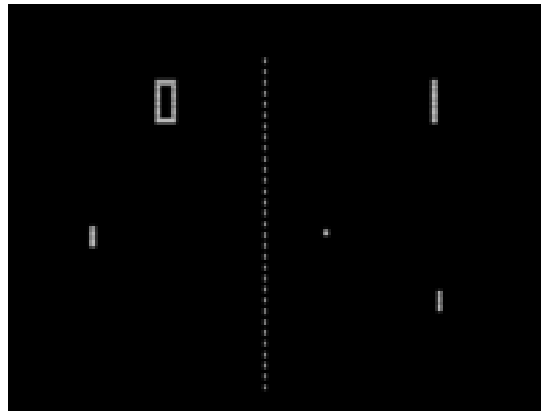


Figura 1.6: Interfaz clásica de Pong⁴.

Ira Greenberg. *Processing. Creative Coding and computational art*. friendof, 2007.

Joshua Noble. *Programming Interactivity*. O'Reilly, 2012. URL <http://shop.oreilly.com/product/0636920021735.do>.

Jeffrey L. Nyhoff and Larry R. Nyhoff. *Processing: An Introduction to Programming*. Chapman and Hall/CRC, 2017.

@openprocessing. Openprocessing, Accedido Enero 2019. URL <https://www.openprocessing.org/>.

Abe Pazos. Fun programming, Accedido Enero 2019. URL <http://funprogramming.org/>.

Processing Foundation. Processing, Accedido Enero 2019a. URL <http://processing.org/>.

Processing Foundation. Processing exhibition archives, Accedido Enero 2019b. URL <https://processing.org/exhibition/>.

Casey Reas and Ben Fry. *Processing: a programming handbook for visual designers and artists*. MIT Press, 2007. URL <https://processing.org/img/learning/Processing-Sample-070607.pdf>.

Derek Runberg. *The sparkfun guide to processing*. Sparkfun, 2015.

Práctica 2

Superficie de revolución

2.1. PSHAPE

La práctica anterior describe las primitivas básicas 2D para el dibujo de objetos tales como rectángulos y elipses. Una opción más avanzada para la creación de formas arbitrarias es hacer uso de variables *PShape* [Shiffman \[Accedido Enero 2019b\]](#), que por otro lado permiten acelerar el proceso de dibujado, en particular cuando los objetos crecen en complejidad.

Listado 2.1: Dibujando un rectángulo con *rect*

```
void setup() {
  size(400, 400);
}
void draw() {
  background(50);
  stroke(255);
  fill(127);
  rect(mouseX, mouseY, 100, 50);
}
```

El listado [2.1](#) sigue las pautas de la práctica anterior, haciendo uso de *rect* para dibujar un rectángulo que acompaña al movimiento del puntero sobre la ventana. Como ejemplo ilustrativo de una variable *PShape*, el listado [2.2](#) muestra el código para obtener el mismo resultado sin necesidad de la función *rect*, si bien introduciendo la traslación 2D con la función *translate*.

Listado 2.2: Dibujando un rectángulo como variable *PShape*

```
PShape rectangle;

void setup() {
  size(400, 400, P2D);
  //La forma
```

```
rectangle = createShape(RECT, -50, -25, 100, 50);
// Aspectos de dibujo
rectangle.setStroke(color(255));
rectangle.setStrokeWeight(4);
rectangle.setFill(color(127));
}
void draw() {
  background(50);
  // Situamos en el puntero
  translate(mouseX, mouseY);
  shape(rectangle);
}
```

Como se muestra en el listado 2.2, la modificación de características de color de una *PShape* requiere utilizar los métodos *setFill*, *setStroke*, *setStrokeWeight*, etc. En este sentido, el listado 2.3 selecciona el color de relleno de la forma según la posición del puntero.

Listado 2.3: Modificando el color de relleno de una variable *PShape*

```
PShape rectangle;

void setup() {
  size(400, 400, P2D);
  rectangle = createShape(RECT, -50, -25, 100, 50);
  rectangle.setStroke(color(255));
  rectangle.setStrokeWeight(4);
  rectangle.setFill(color(127));
}

void draw() {
  background(50);
  translate(mouseX, mouseY);
  rectangle.setFill(color(map(mouseX, 0, width, 0, 255)));
  shape(rectangle);
}
```

La flexibilidad ofrecida por las variables tipo *PShape* aparece al poder definir los vértices que componen el objeto de forma arbitraria. El tutorial disponible en la web [Shiffman \[Accedido Enero 2019b\]](#) incluye el ejemplo de una estrella (también disponible como ejemplo *Archivo->Ejemplos->Topics->Create Shapes->PolygonPShape*), adaptado en el listado 2.4, y que define los vértices que delimitan a la forma entre llamadas a *beginShape* y *endShape*, esta última con *CLOSE* como argumento para cerrar la línea poligonal.

Listado 2.4: Polígono arbitrario, una estrella, con *PShape*

```
PShape star;

void setup() {
  size(400, 400, P2D);

  // La variable
  star = createShape();
```

```
star.beginShape();
// El pincel
star.fill(102);
star.stroke(255);
star.strokeWeight(2);
// Los puntos de la forma
star.vertex(0, -50);
star.vertex(14, -20);
star.vertex(47, -15);
star.vertex(23, 7);
star.vertex(29, 40);
star.vertex(0, 25);
star.vertex(-29, 40);
star.vertex(-23, 7);
star.vertex(-47, -15);
star.vertex(-14, -20);
star.endShape(CLOSE);
}

void draw() {
  background(51);
  // Movimiento con el puntero
  translate(mouseX, mouseY);
  // Dibujamos
  shape(star);
}
```

A partir de la forma de estrella, otro ejemplo disponible es *PolygonPShapeOOP2* (*Archivo->Ejemplos->Topics->Create Shapes*). El resultado del código presente en el listado 2.5, hace uso de la clase *Polygon*, ver listado 2.6, mostrando múltiples estrellas en movimiento vertical descendente.

Listado 2.5: Múltiples estrellas

```
ArrayList<Polygon> polygons;

void setup() {
  size(640, 360, P2D);

  // La forma
  PShape star = createShape();
  star.beginShape();
  star.noStroke();
  star.fill(0, 127);
  star.vertex(0, -50);
  star.vertex(14, -20);
  star.vertex(47, -15);
  star.vertex(23, 7);
  star.vertex(29, 40);
  star.vertex(0, 25);
  star.vertex(-29, 40);
  star.vertex(-23, 7);
```

```
star.vertex(-47, -15);
star.vertex(-14, -20);
star.endShape(CLOSE);

// Lista de objetos
polygons = new ArrayList<Polygon>();

// Lista de objetos PShape
for (int i = 0; i < 25; i++) {
    polygons.add(new Polygon(star));
}
}

void draw() {
    background(255);

    // Dibujamos
    for (Polygon poly : polygons) {
        poly.display();
        poly.move();
    }
}
```

La mencionada clase *Polygon* se encarga de definir la posición y velocidad inicial de cada forma añadida a la lista de objetos.

Listado 2.6: Clase *Polygon*

```
class Polygon {
    // The PShape object
    PShape s;
    // The location where we will draw the shape
    float x, y;
    // Variable for simple motion
    float speed;

    Polygon(PShape s_) {
        x = random(width);
        y = random(-500, -100);
        s = s_;
        speed = random(2, 6);
    }

    // Simple motion
    void move() {
        y += speed;
        if (y > height+100) {
            y = -100;
        }
    }

    // Draw the object
    void display() {
        pushMatrix();
```

```
    translate(x, y);
    shape(s);
    popMatrix();
  }
}
```

2.2. P3D

Los ejemplos anteriores trabajan en 2D. Processing ofrece diversos modos de reproducción como son: SVG, PDF, P2D y P3D. P2D y P3D en concreto hacen uso de hardware compatible con OpenGL, permitiendo mayor rendimiento al dibujar. P2D, que ha aparecido en alguno de los listados previos, es el modo de reproducción optimizado para dos dimensiones, mientras que P3D nos permite trabajar en tres dimensiones [Shiffman \[Accedido Enero 2019a\]](#). Para ambos modos, la calidad del resultado puede configurarse con los métodos *smooth* y *hint*, si bien lo veremos en prácticas posteriores.

Existen formas tridimensionales básicas como la esfera y el prisma, respectivamente los métodos *sphere* y *box*, ver listado [2.7](#).

Listado 2.7: Cubo y esfera

```
size(640, 360, P3D);
background(0);

noFill();
stroke(255);

//Prisma
translate(width*0.2, height*0.15, 0);
box(100);

//Esfera
translate(width/2, height*0.35, 0);
sphere(100);
```

Sin embargo, su potencia y flexibilidad, como ya mencionamos, viene dada por poder definir objetos a través de vértices arbitrarios. Como primer ejemplo, el listado [2.8](#) muestra una pirámide formada por cuatro caras triangulares, cada una con 3 puntos tridimensionales. Al mover el puntero observamos el efecto de la proyección en perspectiva.

Listado 2.8: Pirámide de cuatro lados

```
PShape obj;

void setup() {
  size(600, 600, P3D);
```

```
// La variable
obj=createShape();
// El pincel
obj.beginShape();
obj.noFill();
// Puntos de la forma
obj.vertex(-100, -100, -100);
obj.vertex( 100, -100, -100);
obj.vertex(  0,   0,  100);

obj.vertex( 100, -100, -100);
obj.vertex( 100,  100, -100);
obj.vertex(  0,   0,  100);

obj.vertex( 100, 100, -100);
obj.vertex(-100, 100, -100);
obj.vertex(  0,   0,  100);

obj.vertex(-100, 100, -100);
obj.vertex(-100, -100, -100);
obj.vertex(  0,   0,  100);
obj.endShape();
}

void draw() {
  background(255);
  //Movemos con puntero
  translate(mouseX, mouseY);
  //Muestra la forma
  shape(obj);
}
```

La pareja *beginShape-endShape* sin argumentos asume una serie de vértices consecutivos que conforman una línea poligonal. La especificación de un parámetro a la hora de crear la forma permite indicar el tipo de elementos que definen los vértices a continuación: *POINTS*, *LINES*, *TRIANGLES*, *TRIANGLE_FAN*, *TRIANGLE_STRIP*, *QUADS*, o *QUAD_STRIP* (más detalles [aquí](#)¹):

- *beginShape-endShape(CLOSE)*: Cierra la línea poligonal, uniendo el último vértice con el primero, aplicando el color de relleno.
- *beginShape(POINTS)-endShape*: Cada vértice es un punto, no se conectan con líneas.
- *beginShape(LINES)-endShape*: Cada dos puntos definen un segmento independiente.
- *beginShape(TRIANGLES)-endShape*: Cada grupo de tres puntos define un triángulo. Aplica relleno.

¹https://processing.org/reference/beginShape_.html

- *beginShape(TRIANGLE_STRIP)-endShape*: Los triángulos no son independientes entre sí, cada nuevo vértice compone un triángulo con los últimos dos vértices del triángulo anterior. Aplica relleno.
- *beginShape(TRIANGLE_FAN)-endShape*: El primer vértice está compartido por todos los triángulos. Aplica relleno.
- *beginShape(QUADS)-endShape*: Cada cuatro puntos definen un polígono. Aplica relleno.
- *beginShape(QUAD_STRIP)-endShape*: Se reutilizan los dos últimos vértices del polígono anterior. Aplica relleno.

Como ejemplo de creación de una forma propia, el listado 2.9 muestra una serie enlazada de triángulos creada con la opción *TRIANGLE_STRIP*.

Listado 2.9: Ejemplo de uso de *TRIANGLE_STRIP* con *PShape*

```
PShape obj;

void setup() {
  size(600, 600, P3D);

  // La variable
  obj=createShape();
  // El pincel
  obj.beginShape(TRIANGLE_STRIP);
  obj.fill(102);
  obj.stroke(255);
  obj.strokeWeight(2);
  // Puntos de la forma
  obj.vertex(50, 50, 0);
  obj.vertex(200, 50, 0);
  obj.vertex(50, 150, 0);
  obj.vertex(200, 150, 0);
  obj.vertex(50, 250, 0);
  obj.vertex(200, 250, 0);
  obj.vertex(50, 350, 0);
  obj.vertex(200, 350, 0);
  obj.vertex(50, 450, 0);
  obj.vertex(200, 450, 0);
  obj.endShape();
}

void draw() {
  background(255);
  //Movemos con puntero
  translate(mouseX, mouseY);
  //Muestra la forma
  shape(obj);
}
```

```
}
```

La variante mostrada en el listado 2.10 modifica el valor de la coordenada z de varios vértices. Observa el efecto de la perspectiva sobre los vértices más alejados.

Listado 2.10: Ejemplo de uso de *TRIANGLE_STRIP* con profundidad

```
PShape obj;

void setup() {
  size(600, 600,P3D);

  // La variable
  obj=createShape();
  // El pincel
  obj.beginShape(TRIANGLE_STRIP);
  obj.fill(102);
  obj.stroke(255);
  obj.strokeWeight(2);
  // Puntos de la forma
  obj.vertex(50, 50, 0);
  obj.vertex(200, 50, 0);
  obj.vertex(50, 150, 0);
  obj.vertex(200, 150, -100);
  obj.vertex(50, 250, -100);
  obj.vertex(200, 250, -100);
  obj.vertex(50, 350, -200);
  obj.vertex(200, 350, -200);
  obj.vertex(50, 450, -200);
  obj.vertex(200, 450, -200);
  obj.endShape();
}

void draw() {
  background(255);
  //Movemos con puntero
  translate(mouseX, mouseY);
  //Muestra la forma
  shape(obj);
}
```

2.3. SÓLIDO DE REVOLUCIÓN

La creación de objetos 3D resulta engorrosa al ser necesario disponer de mecanismos para definir los vértices que delimitan el objeto en un escenario tridimensional, y esto debe hacerse sobre una pantalla bidimensional. Una posible simplificación del proceso viene dada a través de la creación de un objeto por medio de superficies de barrido o revolución. En ambos casos, se definen en primer lugar una serie de puntos, que conforman una curva

plana (generalmente), que bien por sucesivas traslaciones (barrido) o rotaciones (revolución) permiten definir la malla de un objeto tridimensional. De modo ilustrativo, la figura 2.1 crea dos objetos utilizando el esquema de revolución, al definir sendos perfiles que se rotan un número determinado de veces, en este caso sobre el eje y , para crear el objeto a partir de la unión de los sucesivos *meridianos* que conforman la malla del objeto.

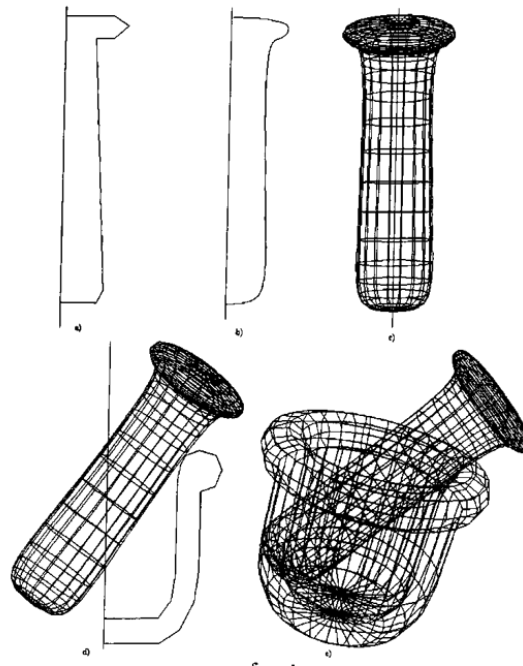


Figura 2.1: Creando una escena con superficies de revolución

Si presentamos la malla resultante *planchada* sobre un plano, tendría el aspecto de una rejilla rectangular, ver figura 2.2 izquierda, donde cada polígono o cara está delimitado por cuatro vértices. Habitualmente resulta más interesante trabajar con triángulos, que pueden crearse de forma sistemática subdividiendo cada polígono, ver figura 2.2 derecha.

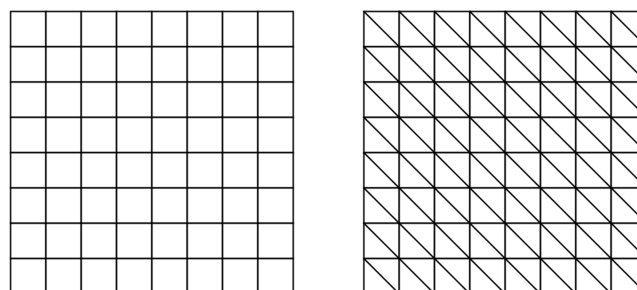


Figura 2.2: Ilustración de malla antes de tras triangularizar

2.3.1. Rotación de un punto 3D

En dos dimensiones la rotación de un punto sobre el plano cartesiano se ilustra en la figura 2.3. Siguiendo la regla de la mano derecha, al rotar un ángulo θ el punto p con coordenadas (x_1, y_1) , las coordenadas resultantes (x_2, y_2) tras la rotación

$$x_2 = x_1 \cdot \cos\theta - y_1 \cdot \operatorname{sen}\theta \quad (2.1)$$

$$y_2 = x_1 \cdot \operatorname{sen}\theta + y_1 \cdot \cos\theta$$

O su equivalente en forma matricial con premultiplicación:

$$(x_2, y_2) = (x_1, y_1) \cdot \begin{bmatrix} \cos\theta & \operatorname{sen}\theta \\ -\operatorname{sen}\theta & \cos\theta \end{bmatrix} \quad (2.2)$$

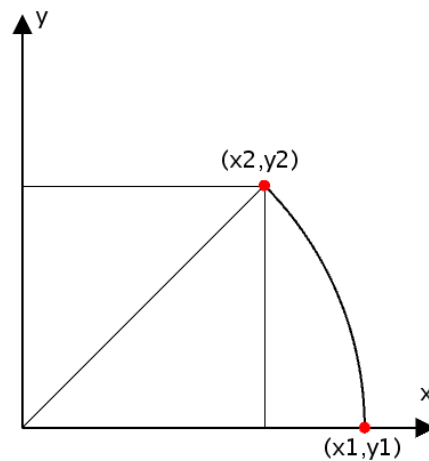


Figura 2.3: Rotación 2D de un ángulo θ

Extensible de forma análoga a tres dimensiones, donde por simplicidad asumiremos una rotación de un punto alrededor del eje vertical y , de forma similar al ejemplo mostrado en la figura 2.1. De esta forma, las coordenadas rotadas de un punto 3D rotado un ángulo θ sobre el eje y siguen las siguientes expresiones:

$$\begin{aligned}x_2 &= x_1 \cdot \cos\theta - z_1 \cdot \sin\theta \\y_2 &= y_1 \\z_2 &= x_1 \cdot \sin\theta + z_1 \cdot \cos\theta\end{aligned}\tag{2.3}$$

$$(x_2, y_2, z_2) = (x_1, y_1, z_1) \cdot \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

Un sólido de revolución requiere la rotación un número de veces de los puntos del perfil proporcionado para su obtención.

2.4. TAREA

Crear un prototipo que recoja puntos de un perfil al hacer clic con el ratón sobre la pantalla, para finalmente crear un objeto tridimensional por medio de una superficie de revolución, almacenando la geometría resultante en una variable de tipo *PShape*, ver a modo de ilustración la figura 2.4. El prototipo permitirá crear sólidos de revolución de forma sucesiva, si bien se almacena únicamente el último definido.

La entrega se debe realizar a través del campus virtual, remitiendo una memoria o el enlace a un *screencast*, que además de describir las decisiones adoptadas, identifique al remitente, además de las referencias y herramientas utilizadas, cuidando el formato y estilo, debe incluir enlace al código desarrollado (p.e. github), con su correspondiente *README*, y opcionalmente un gif animado mostrando la ejecución.

BIBLIOGRAFÍA

Daniel Shiffman. P3D tutorial, Accedido Enero 2019a. URL <https://processing.org/tutorials/p3d/>.

Daniel Shiffman. PShape tutorial, Accedido Enero 2019b. URL <https://processing.org/tutorials/pshape/>.

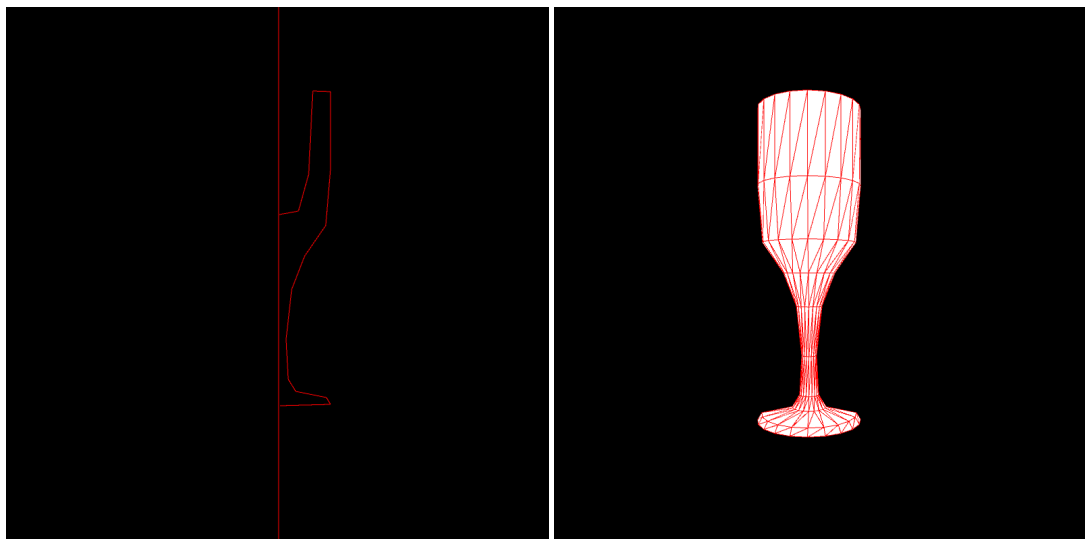


Figura 2.4: Perfil y sólido de revolución resultante

Práctica 3

Transformaciones

3.1. TRANSFORMACIÓN BÁSICAS 2D

En la práctica precedente se ha descrito la utilización de las variables tipo *PShape* para la definición de objetos arbitrarios. A la hora de dibujar, tanto si hacemos uso de primitivas disponibles (2D o 3D) o modelos creados con variables *PShape*, es posible modificar su posición, tamaño y pose haciendo uso de las funciones proporcionadas por Processing para su transformación. Son tres las transformaciones básicas disponibles para escalado, traslación y rotación; ya sean estas últimas en 2D o 3D sobre cada uno de los tres eje principales, respectivamente los métodos *scale*, *translate*, *rotate*, *rotateX*, *rotateY* y *rotateZ*.

El listado 3.1 hace uso de transformaciones en 2D para dibujar varios cuadrados. Observa sin embargo, que si bien la llamada a la función *rect* siempre usa los mismos parámetros, (0,0,100,100), el resultado es claramente diferente, y no únicamente en el color, que se ha modificado de forma expresa para poder ilustrar el efecto sobre cada recuadro.

Si bien el método *translate* se ha utilizado en ejemplos previos, en este ejemplo concreto se aplica en diversas ocasiones, admitiendo dos o tres parámetros según si trabajamos con dos (P2D), como en este caso, o tres dimensiones (P3D). El método *scale* realiza un escalado pudiendo tener de uno a tres argumentos, según si el escalado se realiza a todas las coordenadas por igual, o de forma diferenciada a *x* e *y* (P2D), o *x*, *y* y *z* (P3D). Por último, la rotación con el método *rotate* indica el ángulo en radianes, pudiendo emplearse el método *radians* para facilitar expresarlo en grados.

Listado 3.1: Transformaciones sobre un recuadro

```
size(500,500,P2D);  
  
translate(100,100);
```

```
//Recuadro sin transformar
rect(0,0,100,100);

//Recuadro trasladado rojo
fill(255,0,0);
translate(200,200);
rect(0,0,100,100);

//Recuadro trasladado y escalado verde
fill(0,255,0);
scale(0.7);
rect(0,0,100,100);

//Recuadro trasladado, escalado y rotado azul
fill(0,0,255);
rotate(radians(225));
rect(0,0,100,100);
```

Es importante observar que las transformaciones se acumulan, es decir cada nueva primitiva dibujada se presenta tras realizar sobre ella todas las transformaciones previamente expresadas en el código. Sin embargo, existe la posibilidad de evitar, según nos convenga, dicho efecto de acumulación haciendo uso de los métodos *pushMatrix* y *popMatrix*. Estos métodos aportan flexibilidad, permitiendo trabajar con transformaciones de forma independiente para cada objeto o grupo de ellos.

Por un lado, *pushMatrix* conserva la matriz de coordenadas en dicho momento, con lo que no seguiría acumulando en ella las próximas transformaciones, mientras que *popMatrix* restablece la última matriz de transformación almacenada. El listado 3.2 ilustra su utilización con un resultado diferente al ejemplo anterior al aplicar a los recuadros rojo y verde las transformaciones entre una pareja *pushMatrix-popMatrix*. Veremos en cualquier caso más detalles más adelante dentro de este mismo guion de práctica.

Listado 3.2: Transformaciones sobre un recuadro con *pushMatrix* y *popMatrix*

```
size(500,500,P2D);

//Trasladamos todo
translate(100,100);

//Recuadro sin transformar
rect(0,0,100,100);

//Recuadro trasladado rojo
pushMatrix();
fill(255,0,0);
translate(200,200);
rect(0,0,100,100);
```



```
//Recuadro trasladado y escalado verde
fill(0,255,0);
scale(0.7);
rect(0,0,100,100);
popMatrix();

//Recuadro trasladado, escalado y rotado azul
fill(0,0,255);
rotate(radians(225));
rect(0,0,100,100);
```

Con el objetivo de reforzar la comprensión de las matrices de transformación, las siguientes subsecciones dan más detalles sobre las transformaciones básicas.

3.1.1. Traslación

El efecto real de las funciones de transformación es la aplicación de una modificación de los ejes de coordenadas.

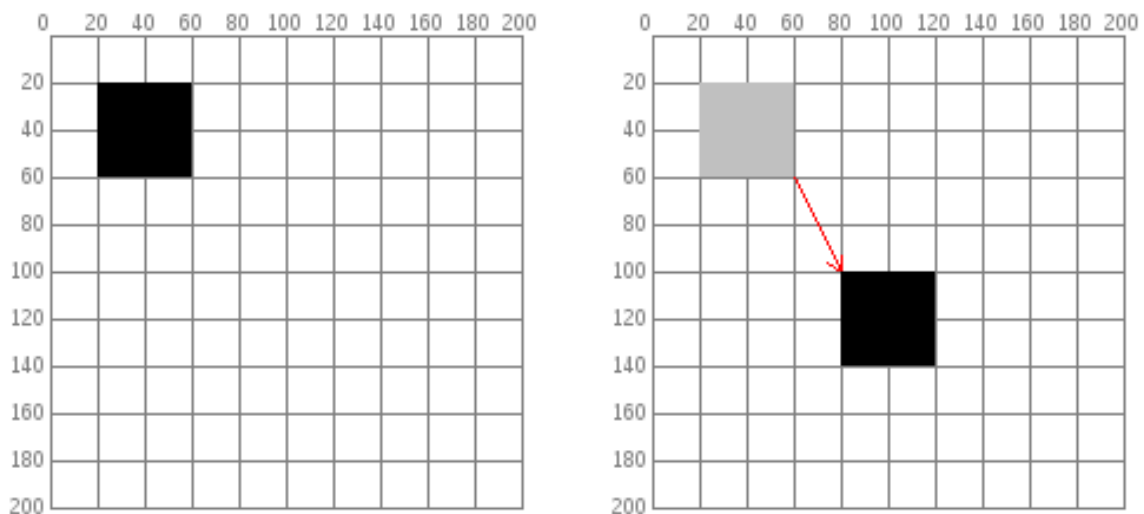


Figura 3.1: Cuadrado antes y después trasladar (fuente imagen Eisenberg [Accedido Febrero 2019]).

Conocida esta circunstancia, para dibujar un cuadrado colocado inicialmente en las coordenadas (20,20), desplazado 60 unidades a la derecha y 80 unidades hacia abajo, existen dos posibilidades:

- Cambiar las coordenadas directamente en la llamada a *rect* mediante la suma a los puntos iniciales: *rect*(20 + 60, 20 + 80, 40, 40), ilustrado en la figura 3.1.

- Desplazando el sistema de coordenadas con la función *translate*, manteniendo la misma llamada *rect(20,20,40,40)*. Los ejes antes y después de trasladarlos se ilustran en la figura 3.2.

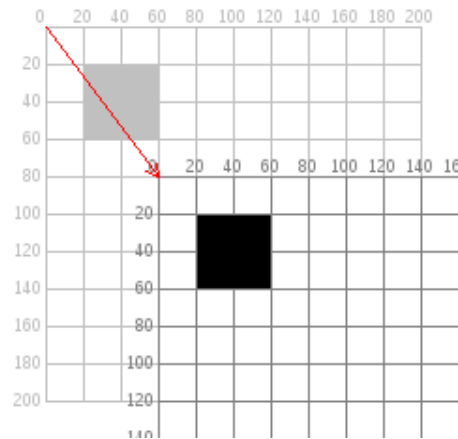


Figura 3.2: Traslación del sistema de referencia (fuente imagen Eisenberg [Accedido Febrero 2019]).

En el segundo caso, el cuadrado no se mueve de su posición, su esquina superior izquierda se encuentra todavía en (20,20), por el contrario, el sistema de coordenadas se modifica. El listado 3.3 muestra ambas formas.

Listado 3.3: Dibujando el cuadrado trasladando y con traslación del sistema de referencia

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

  // El nuevo recuadro usa las mismas coordenadas que el primero pero antes se mueve el sistema de
  referencia
  fill(0, 0, 255, 128);
  pushMatrix(); //Salva el sistema de coordenadas actual
  translate(60, 80);
  rect(20, 20, 40, 40);
  popMatrix(); //vuelve al sistema de coordenadas original
}
```

Si bien para un ejemplo simple como el anterior, modificar el sistema de coordenadas puede parecer engorroso o excesivo, las transformaciones facilitan la operación al crecer el

número de objetos y su reutilización. El listado 3.4 presenta dos variantes mostrando lo que significa dibujar una hilera de casas con ambos esquemas. Se utiliza un bucle que llama a la función *house()*, que recibe como parámetros la ubicación, *x* e *y*, de la esquina superior izquierda de cada casa. La segunda opción simplifica la especificación de las coordenadas para cada primitiva de transformación, al estar las tres desplazadas.

Listado 3.4: Dibujo de varias casas

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

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

*****

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

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

3.1.2. Rotaciones

Tal y como se menciona anteriormente, la rotación 2D se aplica con la función `rotate()`, requiriendo como argumento el número de radianes a rotar. Si un círculo completo en grados son 360° , en radianes se corresponde con 2π , ver figura 3.3.

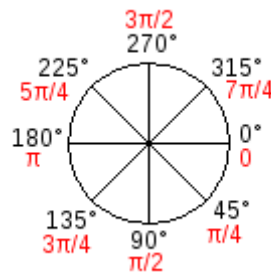


Figura 3.3: Ángulos de rotación en grados (negro) y radianes (rojo) (fuente imagen Eisenberg [Accedido Febrero 2019])

Recordar que la función `radians()` toma un número de grados como argumento y lo convierte a radianes, mientras que la función `degrees()` convierte radianes a grados. A modo de ejemplo, el listado 3.5 rota un cuadrado 45 grados en sentido horario. Al no estar una de sus esquinas en el origen, el resultado de la rotación se ilustra en la figura 3.4.

Listado 3.5: Ejemplo de aplicación de la rotación 2D

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

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

Si el propósito es rotar el objeto alrededor de un punto específico, por ejemplo una esquina del recuadro o el centro del mismo, será necesario trasladar antes el sistema de coordenadas. A modo de ejemplo, el listado 3.6 realiza las acciones necesarias para rotar sobre la esquina superior izquierda, que serían las siguientes:

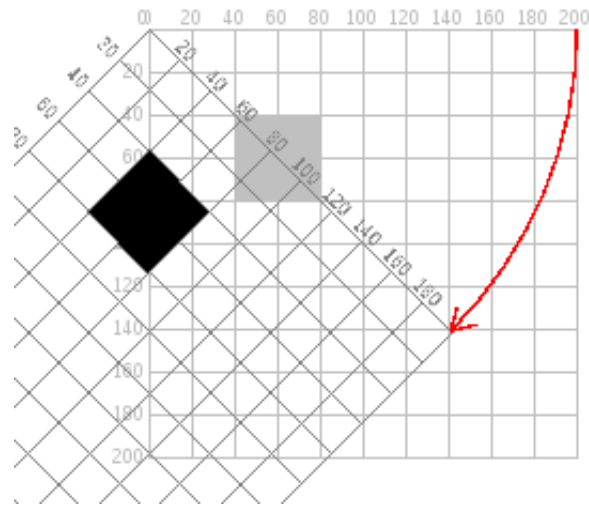


Figura 3.4: Resultados de la rotación del listado 3.5

1. Trasladar el origen del sistema de coordenadas de (0, 0) a la esquina superior izquierda del cuadrado.
2. Girar $\pi/4$ radianes (45°)
3. Dibujar el cuadrado.

Listado 3.6: Rotación sobre el pivote

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

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

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

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

3.1.3. Escalado

Como última transformación básica, ilustramos el escalado. A modo de ilustración, el listado 3.7 modifica el aspecto que vemos del cuadrado.

Listado 3.7: Escalado

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

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

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

Si bien pudiera parecer que el cuadrado se ha movido, no ha ocurrido realmente. Su esquina superior izquierda se encuentra todavía en la posición (20,20). También se puede ver que las líneas son más gruesas. Eso no es una ilusión óptica, las líneas son en realidad dos veces más gruesas, porque el sistema de coordenadas ha crecido al doble de su tamaño original.

3.2. CONCATENACIÓN DE TRANSFORMACIONES

Al realizar múltiples transformaciones, el orden es importante. No hay conmutatividad, una rotación seguida de una traslación y de un escalado no dará los mismos resultados que una traslación seguida de una rotación y un escalado, ver lo que ocurre en el listado 3.8.

Listado 3.8: Encadenando transformaciones

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

  pushMatrix();
  fill(255, 0, 0); // cuadrado rojo
  rotate(radians(30));
  translate(70, 70);
}
```

```
scale(2.0);
rect(0, 0, 20, 20);
popMatrix();

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

Cada vez que haces una rotación, traslación, o escalado, la información necesaria para realizar la transformación se acumula en una matriz de transformación. Esta matriz contiene toda la información necesaria para hacer cualquier serie de transformaciones. Y esa es la razón por la que se usan las funciones *pushMatrix()* y *popMatrix()*, para poder obviarla, o deshacer fácilmente operaciones, si fuera necesario.

Estas dos funciones nos permiten manejar una pila de sistemas de referencia, *pushMatrix()* pone el estado actual del sistema de coordenadas en la parte superior de dicha pila, mientras que *popMatrix()* extrae el último estado de dicha matriz almacenado en la pila. El ejemplo anterior utiliza *pushMatrix()* y *popMatrix()* para asegurarse de que el sistema de coordenadas estaba *limpio* antes de cada parte del dibujo.

Mencionar que en Processing, el sistema de coordenadas se restaura a su estado original (de origen en la parte superior izquierda de la ventana, sin rotación ni escalado) cada vez que la función *draw()* se ejecuta. Si fuera necesario también es posible resetearla desde programa con la llamada a *resetMatrix*, e incluso muestra la matriz actual con *printMatrix*.

3.3. TRANSFORMACIONES BÁSICAS 3D

Las diferencias entre las transformaciones 2D y 3D no son excesivas, realmente para trabajar en tres dimensiones basta con pasar tres argumentos a las funciones de transformación, con la salvedad de que para las rotaciones haremos uso de las funciones *rotateX()*, *rotateY()*, o *rotateZ()*.

A modo de ejemplo, el listado 3.9 hace uso de una primitiva 3D, la esfera, para mostrarla levemente rotada permitiendo observar su *polo norte*.

Listado 3.9: Dibujando una esfera levemente rotada

```
void setup()
{
  size(500,500,P3D);
}
```

```
stroke(0);
}

void draw()
{
  background(200);

  // Esfera
  translate(width/2, height/2, 0);
  rotateX(radians(-45));
  sphere(100);
}
```

En el listado 3.10 introducimos movimiento, incorporando en el método *draw* una rotación sobre el eje *y* variable, que pudiera ilustrar la autorrotación *diaria* de un planeta.

Listado 3.10: Dibujando una esfera levemente rotada con movimiento

```
float ang;

void setup()
{
  size(500,500,P3D);
  stroke(0);

  // Inicializa
  ang=0;
}

void draw()
{
  background(200);

  // Esfera
  translate(width/2, height/2, 0);
  rotateX(radians(-45));
  rotateY(radians(ang));
  sphere(100);

  // Resetea tras giro completo
  ang=ang+0.25;
  if (ang>360)
    ang=0;
}
```

Como añadido, el listado 3.11 incluye además un *satélite* en órbita geostacionaria.

Listado 3.11: Dibujando una esfera levemente rotada con movimiento

```
float ang;

void setup()
```



```
{
  size(500,500,P3D);
  stroke(0);

  // Inicializa
  ang=0;
}

void draw()
{
  background(200);

  // Esfera
  translate(width/2, height/2, 0);
  rotateX(radians(-45));
  rotateY(radians(ang));
  sphere(100);

  //Resetea tras giro completo
  ang=ang+0.25;
  if (ang>360)
    ang=0;

  //Objeto orbitando geoestacionario
  translate(-width*0.25,0,0);
  box(10);
}
```

Finalmente, en el listado 3.12 se muestra el código con el *satélite* sin una órbita geoestacionaria.

Listado 3.12: Dibujando una esfera levemente rotada con movimiento

```
float ang;
float angS;

void setup()
{
  size(500,500,P3D);
  stroke(0);

  // Inicializa
  ang=0;
  angS=0;
}

void draw()
{
  background(200);

  // Esfera
  translate(width/2, height/2, 0);
```

```
rotateX(radians(-45));

//Planeta
pushMatrix();
rotateY(radians(ang));
sphere(100);
popMatrix();

//Resetea tras giro completo
ang=ang+0.25;
if (ang>360)
  ang=0;

//Objeto
pushMatrix();
rotateZ(radians(angS));
translate(-width*0.25,0,0);
box(10);
popMatrix();

//Resetea tras giro completo
angS=angS+0.25;
if (angS>360)
  angS=0;
}
```

3.4. OBJETOS DE ARCHIVO

Como utilidades, Processing dispone de utilidades para la carga de objetos svg (2D) y obj (3D). El listado 3.13 ilustra la carga de un objeto de archivo svg (fuente archivo¹). El archivo a cargar debe estar en la carpeta *data* del prototipo haciendo uso de la función *loadShape*.

Listado 3.13: Carga un archivo svg mostrando su contenido

```
PShape svg;

void setup() {
  size(600, 600, P2D);
  svg = loadShape("lake_inkscape_curve_fill.svg");
}

void draw() {
  background(255);
  scale(0.7);
  shape(svg);
}
```

¹<http://people.sc.fsu.edu/~jburkardt/data/svg/svg.html>

Cualquier objeto cargado puede ser sometido a transformaciones. El listado 3.14 carga un objeto 3D desde un archivo 3D (fuente archivo²), realizando varias transformaciones sobre el mismo antes de visualizarlo.

Listado 3.14: Carga un archivo obj mostrando su contenido

```
PShape obj;

void setup() {
  size(600, 600, P3D);
  obj = loadShape("lamp.obj");
}

void draw() {
  background(255);
  translate(mouseX, mouseY, 0);
  scale(30);
  rotateX(radians(180));
  shape(obj);
}
```

3.5. TEXTO E IMÁGENES

Las transformaciones no sólo se aplican sobre vértices y aristas, texto e imágenes son otros elementos de interés. El listado 3.15 muestra un mensaje tras aplicarle transformaciones 3D.

Listado 3.15: Hola mundo con transformaciones

```
void setup() {
  size(600, 600, P3D);
  stroke(0);
  fill(0);
}

void draw() {
  background(255);
  translate(mouseX, mouseY, 0);

  rotateX(radians(45));
  rotateY(radians(22));
  scale(3);
  text("Hola mundo", 10, 10);
}
```

²<http://people.sc.fsu.edu/~jburkardt/data/obj/obj.html>

Un ejemplo de las variadas posibilidades, se muestra en el listado 3.16, cuya ejecución que puede hacer las delicias de algunos fans (fuente archivo³).

Listado 3.16: Texto al estilo Star wars ()

```
final int COLOR_MAX = 255;
final char DELIMITER = '\n'; // delimiter for words

final int WORDS_PER_LINE = 5;
final int MAX_TEXT_SIZE = 40;
final int MIN_TEXT_SIZE = 0;

// where to draw the top of the text block
float textYOffset = 500+MAX_TEXT_SIZE; // 500 must match setup canvas size
// start PAST the bottom of the screen so that the
// text comes in instead of just appearing

final float TEXT_SPEED = 0.5; // try changing this to experiment

// story to tell!
final String STORY_TEXT = "A long time ago, in a galaxy far, far "+DELIMITER+
"away.... It is a period of civil war. Rebel spaceships, "+DELIMITER+
"striking from a hidden base, have won their first victory "+DELIMITER+
"against the evil Galactic Empire. During the battle, rebel "+DELIMITER+
"spies managed to steal secret plans to the Empire\'s " +DELIMITER+
"ultimate weapon, the DEATH STAR, an armored space station " +DELIMITER+
"with enough power to destroy an entire planet. Pursued by " +DELIMITER+
"the Empire\'s sinister agents, Princess Leia races home " +DELIMITER+
"aboard her starship, custodian of the stolen plans that " +DELIMITER+
"can save her people and restore freedom to the galaxy....";

String[] storyLines;

void setup()
{
  size(500, 500, P3D);
  textYOffset = height;

  fill(250,250,0);
  textAlign(CENTER,CENTER);
  textSize(MAX_TEXT_SIZE+MIN_TEXT_SIZE);
}

void draw()
{
  background(0);
  translate(width/2,height/2);
  rotateX(PI/3);
  text(STORY_TEXT,0,textYOffset);
  // Make the text slowly crawl up the screen
  textYOffset -= TEXT_SPEED;
}
```

³<https://forum.processing.org/two/discussion/23576/star-wars-text>

Para hacer algo similar con imágenes necesitamos hacer uso de variables de tipo *PImage*. El listado 3.17 sirve de ejemplo para mostrar el proceso de carga y visualización, tras varias transformaciones.

Listado 3.17: Hola mundo con transformaciones

```
PImage img;

void setup() {
  size(600, 600, P3D);
  imageMode(CENTER);
  //Carga de la imagen
  img=loadImage("sample.png");
}

void draw() {
  background(255);
  translate(mouseX, mouseY, 0);

  rotateX(radians(45));
  rotateY(radians(22));
  scale(3);
  //Muestra la imagen
  image(img, 0, 0);
}
```

3.6. TAREA

Crear un prototipo que muestre un sistema planetario en movimiento que incluya una estrella, al menos cinco planetas y alguna luna, integrando primitivas 3D, texto e imágenes (p.e. imagen de fondo). Además debe incluir una *nave espacial* cuya localización y orientación pueda modificarse de forma interactiva por medio del teclado/ratón.

La entrega se debe realizar a través del campus virtual, remitiendo una memoria o el enlace a un *screencast*, que además de describir las decisiones adoptadas, identifique al remitente, las referencias y herramientas utilizadas, cuidando el formato y estilo, debe incluir enlace al código desarrollado (p.e. github), con su correspondiente *README*, y un gif animado, de al menos 5 segundos, que ilustre ejecución del prototipo.

BIBLIOGRAFÍA

J David Eisenberg. 2D transformations, Accedido Febrero 2019. URL <https://processing.org/tutorials/transform2d/>.

Práctica 4

Modelos cámara

La práctica anterior aborda los elementos necesarios para aplicar transformaciones 2D y 3D a objetos, texto e imágenes. El primer paso en la representación de una escena gráfica es justamente la aplicación de transformaciones a los modelos de los objetos presentes en dicha escena. Una vez que los modelos han sido posicionados en el espacio tridimensional, procede localizar la o las cámaras, para finalmente proyectar los puntos al plano de proyección en dos dimensiones.

En esta práctica se resumen las posibilidades disponibles tanto para configurar la proyección a aplicar, como la especificación de una cámara. Procedemos en primer lugar a describir las proyecciones, para posteriormente abordar la cámara.

4.1. PROYECCIONES

Asumiendo una posición de la cámara fija, aspecto que dejamos para la siguiente sección, se describen las posibilidades para realizar proyecciones. Nos centramos en los dos grandes grupos de proyecciones, ambos presentes en Processing [Shiffman \[Accedido Enero 2019\]](#): ortográfica y perspectiva. Ambas proyecciones se alternan con sus respectivas configuraciones por defecto en el listado 4.1, que dibuja un cubo en el centro de la ventana, permitiendo cambiar el modo de proyección entre ortográfica y perspectiva al hacer clic con el ratón. Observar que la proyección perspectiva modifica el tamaño del objeto en función de su distancia, no así la ortográfica. Las siguientes subsecciones analizan ambos tipos de proyección con mayor detalle.

Listado 4.1: Alternando entre proyección ortográfica y perspectiva

```
int mode;
```

```
void setup ()
{
  size (800, 800, P3D);
  ortho ();
  mode=0;
}

void draw ()
{
  background(200);

  //Muestra modo proyección actual
  fill (0);
  if (mode == 1) {
    text ("PERSPECTIVA", 20,20);
  } else {
    text ("ORTOGRAFICA", 20,20);
  }

  //Dibuja objeto en el centro de la ventana
  noFill ();
  translate (width/2, height/2, 0);
  box(200);
}

void mouseClicked () {
  if (mode == 0) {
    mode=1;
    perspective ();
  } else {
    mode = 0;
    ortho ();
  }
}
```

4.1.1. Ortografía

Como se comenta anteriormente. en una proyección ortográfica dos objetos iguales aparecen del mismo tamaño independientemente de su distancia. La función *ortho* realmente no requiere parámetros, configuración por defecto, a menos que precisemos indicar el volumen de recorte. En este último caso se especifican cuatro o seis parámetros; dependiendo de si definimos un recuadro de recorte, o un cubo completo de recorte. indicando los planos que delimitan al recuadro/cubo: izquierdo, derecho, inferior, superior, cercano y lejano el cubo. El listado 4.2 muestra un cubo con proyección ortográfica, que nos permite ver las dimensiones de ancho y alto del cubo. No hay información de su profundidad, en realidad no se aprecia si es un cuadrado, un cubo o una caja de mayor profundidad. Para visualizarlo se define una proyección ortográfica con cuatro parámetros, no se definen los planos cercano y lejano.

Los parámetros definidos tienen en cuenta las dimensiones de la ventana. En este ejemplo concreto, los valores coinciden justamente con los definidos por defecto, es decir asignados en la llamada a *ortho* sin parámetros.

Listado 4.2: Proyección ortográfica

```
size(400, 400, P3D);
ortho(-width/2, width/2, -height/2, height/2);
noFill();
translate(width/2, height/2, 0);
box(100);
```

En caso de especificar el recuadro o cubo de recorte, realmente se mapean las coordenadas del mismo a las esquinas de la ventana de visualización. De esta forma si levemente modificamos el ejemplo del listado 4.2 y modificamos los planos que delimitan en x e y , como en el listado 4.2 el resultado de la visualización, recuerda a un escalado. No hemos constatado utilidad para los parámetros referidos a los planos cercano y lejano en estos ejemplos básicos.

Listado 4.3: Proyección ortográfica con recuadro de recorte

```
size(400, 400, P3D);
ortho(-100, 100, -100, 100);
noFill();
translate(width/2, height/2, 0);
box(100);
```

En los ejemplos previos, sólo se aprecian dos dimensiones del objeto tridimensional, un cubo, dado que vemos únicamente una de las tapas. El listado 4.4 además de desplazar el cubo al centro, lo rota en función del tiempo alrededor del eje y y para verlo en movimiento, pudiendo *intuirse* dos caras, lo cual permite apreciar su fondo.

Listado 4.4: Proyección ortográfica con movimiento del cubo

```
float ang;

void setup()
{
  size(400, 400, P3D);
  noFill();
  ang=0;
}

void draw()
{
  background(200);
  ortho(-width/2, width/2, -height/2, height/2);
  translate(width/2, height/2, 0);
  rotateY(radians(ang));
  box(100);
}
```

```
ang+=0.5;
if (ang==360) ang=0;
}
```

Finalmente en el listado 4.5 son dos las rotaciones aplicadas, con lo que se aprecian tres de las caras, si bien al ser un modelo de alambres persiste la ambigüedad.

Listado 4.5: Proyección ortográfica tras dos rotaciones

```
float ang;

void setup()
{
  size(400, 400, P3D);
  noFill();
  ang=0;
}

void draw()
{
  background(200);
  ortho(-width/2, width/2, -height/2, height/2);
  translate(width/2, height/2, 0);
  rotateX(-PI/6);
  rotateY(radians(ang));
  box(100);

  ang+=0.5;
  if (ang==360) ang=0;
}
```

4.1.2. Perspectiva

La proyección de perspectiva se establece con la función *perspective* que requiere ninguno o cuatro parámetros. El primero establece el ángulo de vista, el segundo la relación de aspecto alto ancho, y los dos últimos, los planos cercano y lejano de recorte en z. En caso de no especificar parámetros, como en el listado 4.1, se adoptan los valores por defecto, siendo equivalente a la llamada *perspective(PI/3.0, width/height, cameraZ/10.0, cameraZ*10.0)*, donde *cameraZ* es $((height/2.0) / \tan(PI*60.0/360.0))$. El listado 4.6 presenta el mínimo ejemplo del cubo proyectado con la perspectiva por defecto.

Listado 4.6: Proyección en perspectiva

```
void setup()
{
  size(800, 800, P3D);
  perspective();
}
```

```

void draw ()
{
  background(200);

  //Dibuja objeto en el centro de la ventana
  noFill ();
  translate (width/2, height/2, 0);
  box(200);
}

```

Para ilustrar lo que aportan el ángulo de visión y la relación de aspecto, se presenta el listado 4.7, donde mover el puntero a lo largo del eje x afecta al primero, y hacer clic al segundo.

Listado 4.7: Perspectiva con modificación del ángulo de visión, y de la relación de aspecto

```

int aspecto;
float cameraZ;

void setup()
{
  size(800, 800, P3D);
  //Valores por defecto de la perspectiva
  aspecto=width/height;
  cameraZ= ((height/2.0) / tan(PI*60.0/360.0));
}

void draw ()
{
  background(200);

  perspective(mouseX/ float (width) * PI/2, aspecto, cameraZ/10.0, cameraZ*10.0);

  //Dibuja objeto en el centro de la ventana
  noFill ();
  translate (width/2, height/2, 0);
  box(200);
}

void mouseClicked() {
  //Modifica la relación de aspecto
  if (aspecto > width/height) {
    aspecto=width/height;

  } else {
    aspecto = 2*width/height;
  }
}

```

Para la especificación de los planos de recorte en z , se asume al observador en $z=0$ mirando hacia al lado positivo de z , por lo que ambos deben ser positivos. El listado 4.8

mueve el cubo tras definir los planos de recorte en z, pudiendo observar el efecto de recorte en el objeto.

Listado 4.8: Proyección en perspectiva con especificación de los planos de recorte en profundidad

```
float muevez;

void setup() {
  size(800,800,P3D);
  noFill();
  perspective(PI/2,width/height,0.1,900);
  muevez=0;
}

void draw() {
  background(200);

  translate(width/2,height/2,muevez);
  box(200);

  muevez-=0.5;
}
```

Una alternativa para especificar el volumen de recorte es hacer uso de la función *frustum*. Una llamada *frustum* afecta a la perspectiva utilizada, al especificar los planos que delimitan el volumen de recorte. Señalar que el plano cercano debe ser mayor que 0, y menos que el lejano.

4.2. LA CÁMARA

Con un comportamiento idéntico a la función *gluLookAt* de OpenGL, Processing proporciona la función *camera*, que permite establecer la localización de la cámara en el espacio tridimensional, la dirección hacia donde mira, y su vertical. En total nueve argumentos: tres para la localización del ojo, tres para la dirección en la que mira el ojo, y los tres últimos las coordenadas del vector vertical.

La llamada sin argumentos es equivalente a *camera(width/2.0, height/2.0, (height/2.0) / tan(PI*30.0 / 180.0), width/2.0, height/2.0, 0, 0, 1, 0)*; que básicamente coloca el ojo a la altura del centro de la pantalla, si bien desplazado en z, mirando hacia el centro de la pantalla, y con el vector vertical. El listado 4.9 coloca el ojo mirando hacia el centro de la ventana, justamente donde colocamos el cubo.

Listado 4.9: Vista desde la cámara

```
void setup()
{
```

```
size(800, 800, P3D);
camera();
}

void draw ()
{
  background(200);

  noFill();
  translate(width/2, height/2, 0);
  box(200);
}
```

Como muestra de la configuración de los parámetros, en el listado 4.10 se utiliza el evento de teclado, reconociendo las teclas de cursores para modificar la posición de la cámara, si bien mantenemos hacia donde mira y la vertical.

Listado 4.10: Vista desde la cámara

```
int px,py;

void setup()
{
  size(800, 800, P3D);
  px=0;
  py=0;
}

void draw ()
{
  background(200);

  //Configuración de la cámara
  camera(width/2.0-px, height/2.0-py, (height/2.0) / tan(PI*30.0 / 180.0), width/2.0, height/2.0, 0, 0,
    1, 0);

  noFill();
  translate(width/2, height/2, 0);
  box(200);
}

void keyPressed() {
  if (key == CODED) {
    if (keyCode == UP) {
      py+=10;
    }
    else
    {
      if (keyCode == DOWN) {
        py-=10;
      }
      else
      {

```

```
    if (keyCode == LEFT) {
        px-=10;
    }
    else
    {
        if (keyCode == RIGHT) {
            px+=10;
        }
    }
}
}
```

El punto al que se mira se modifica de nuevo de forma interactiva , a través de las teclas del cursor, en el listado 4.11.

Listado 4.11: Vista desde la cámara modificando hacia donde mira

```
int px,py;

void setup()
{
    size(800, 800, P3D);
    px=0;
    py=0;
}

void draw ()
{
    background(200);

    //Configuración de la cámara
    camera(width/2.0, height/2.0, (height/2.0) / tan(PI*30.0 / 180.0), width/2.0-px, height/2.0-py, 0, 0,
        1, 0);

    noFill();
    translate(width/2, height/2, 0);
    box(200);
}

void keyPressed() {
    if (key == CODED) {
        if (keyCode == UP) {
            py+=10;
        }
        else
        {
            if (keyCode == DOWN) {
                py-=10;
            }
            else
            {
                if (keyCode == LEFT) {
```

```
        px-=10;
    }
    else
    {
        if (keyCode == RIGHT) {
            px+=10;
        }
    }
}
}
```

Para finalizar las opciones de la cámara, el listado 4.12 modifica la vertical del ojo, con lo que cambia la vista obtenida. El resultado será equivalente a rotar el cubo, dado que al no existir otros objetos en la escena ni iluminación, no se aprecia la diferencia.

Listado 4.12: Cámara cambiando su vertical

```
float vx,vy,ang;

void setup()
{
    size(800, 800, P3D);
    ang=0;
}

void draw ()
{
    background(200);
    //Vertical de partida (0,1,0)
    vx=-sin(radians(ang));
    vy=cos(radians(ang));

    //Configuración de la cámara
    camera(width/2.0, height/2.0, (height/2.0) / tan(PI*30.0 / 180.0), width/2.0, height/2.0, 0, vx, vy, 0)
        ;

    noFill();
    translate(width/2, height/2, 0);
    box(200);

    ang=ang+0.25;
    if (ang>360) ang=0;
}
```

Indicar finalmente que toda cámara puede configurar el modo de proyección que se le aplica a través de los modos de proyección mencionados en la sección previa.

4.3. OCLUSIÓN

Como elemento de mejora del realismo en la reproducción, a la hora de representar objetos con relleno, Processing aplica por defecto el algoritmo de ocultación *z-buffer*. Sin embargo, es posible activar o desactivar su acción con la función *hint* pasando como argumento respectivamente *ENABLE_DEPTH_TEST* o *DISABLE_DEPTH_TEST* en cada caso. Por defecto está activado en el modo P3D.

Los listados 4.13 y 4.14 muestran la diferencia de comportamiento. El primer ejemplo, listado 4.13, muestra el cubo con una proyección ortográfica, que por defecto aplica ocultación.

Listado 4.13: Cubo con ocultación

```
size(400, 400, P3D);
ortho(-width/2, width/2, -height/2, height/2);
fill(255);
translate(width/2, height/2, 0);
rotateX(-PI/6);
rotateY(PI/3);
box(100);
```

Si en el listado 4.13 se dibuja el cubo con las caras traseras ocultas. Al desactivar el z-buffer, ver el listado 4.14, se obtiene una imagen en la que se ven todas las aristas del cubo, una figura ambigua.

Listado 4.14: Un cubo

```
size(400, 400, P3D);
ortho(-width/2, width/2, -height/2, height/2);

hint(DISABLE_DEPTH_TEST);

fill(255);
translate(width/2, height/2, 0);
rotateX(-PI/6);
rotateY(PI/3);
box(100);
```

La utilización del z-buffer permite considerar al dibujar múltiples objetos sus intersecciones. El listado 4.15 dibuja dos cubos que intersectan en el espacio, y el resultado de la ocultación permite tener una mejor referencia de su localización en el espacio tridimensional.

Listado 4.15: Dos cubos

```
size(400, 400, P3D);
ortho(-width/2, width/2, -height/2, height/2);

pushMatrix();
```



```
fill(255);
translate(width/2, height/2, 0);
rotateX(-PI/6);
rotateY(PI/3);
box(100);
popMatrix();

pushMatrix();
fill(128);
translate(width/2-15, height/2-20, 55);
rotateX(-PI/6);
rotateY(PI/3);
box(50);
popMatrix();
```

La desactivación del z-buffer, permite dibujar como un pintor, lo último siempre aparece *arriba*.

4.4. TAREA

La tarea anterior abordaba la definición de un sistema planetario que incluía una nave. Diseñar un mecanismo de interacción que permite colocar en la localización de la nave una cámara, facilitando por tanto la modificación de su localización, además de facilitar orientarla para indicar el punto al que se desea mirar, y la vertical, con el objetivo de dar tener la sensación de navegación desde el punto de vista de la nave. El prototipo final debe permitir que se alterne entre una vista general y la vista desde la nave.

La entrega se debe realizar a través del campus virtual, remitiendo una memoria o el enlace a un screencast que además de describir las decisiones adoptadas, identifique al remitente, las referencias y herramientas utilizadas, cuidando el formato y estilo. Como material adicional debe incluirse el enlace al código desarrollado (p.e. github), con su correspondiente *README*, y un gif animado, de al menos 5 segundos, que ilustre ejecución del prototipo.

BIBLIOGRAFÍA

Daniel Shiffman. P3D tutorial, Accedido Enero 2019. URL <https://processing.org/tutorials/p3d/>.

Práctica 5

Iluminación y texturas

5.1. ILUMINACIÓN

Un notorio paso en la mejora del realismo de una escena con objetos tridimensionales es el uso de la iluminación, aspecto integrado en el modo de reproducción P3D [Shiffman \[Accedido Enero 2019\]](#).

Como primer ejemplo básico, el listado 5.1 activa la iluminación con su configuración por defecto al pulsar un botón del ratón, desactivándola al soltarlo. La ejecución permite observar un cubo en rotación eterna. El color de relleno se aplica cuando no se activa iluminación, mientras que se ve afectado por la orientación de cada cara en relación a la fuente de luz cuando se activa la iluminación con la llamada a la función *lights*. Por cierto, *noLights* la desactiva, teniendo sentido en el caso de querer que determinados objetos tengan en cuenta la iluminación, y otros no.

Listado 5.1: Alternando entre cubo con y sin iluminación

```
float ang;

void setup()
{
  size(800, 800, P3D);
  ang=0;
}

void draw ()
{
  background(200);

  if (mousePressed) {
    lights ();
  }
}
```

```
//Dibuja objeto en el centro de la ventana
translate(width/2, height/2, 0);
rotateX(radians(-30));
rotateY(radians(ang));
box(200);

ang+=1;
if (ang>360) ang=0;
}
```

Cuando hablamos de iluminación de adopta el modelo de reflexión de Phong, que combina características de la fuente de luz y el material. En Processing será posible configurar aspectos relativos a: la intensidad de ambiente, el número de fuentes de luz, su dirección, el decaimiento de la luz, la reflexión especular, además de características de reflexión de los materiales. Cuando se activa la iluminación, incluyendo la llamada a *lights* como en el ejemplo previo, se adoptan la siguiente configuración de iluminación por defecto:

- Luz ambiente: equivalente a la llamada *ambientLight(128, 128, 128)*
- Dirección de la luz: equivalente a la llamada *directionalLight(128, 128, 128, 0, 0, -1)*
- Decaimiento de la luz: equivalente a la llamada *lightFalloff(1, 0, 0)*
- Reflexión especular: equivalente a la llamada *lightSpecular(0, 0, 0)*

Las modificaciones del modo por defecto deben integrarse en *draw*, ya que se resetea cualquier nueva configuración en cada nueva ejecución de dicha función. En los siguientes apartados se presentan opciones para evitar la configuración por defecto.

5.1.1. Luces

Para los ejemplos de configuración de la iluminación, adoptamos como objeto 3D la esfera que permite apreciar en mayor medida las posibilidades del comportamiento frente a la luz que un cubo, al ser un objeto con una superficie curvada. El listado 5.2 recuerda el modo de dibujar una esfera en el centro de la ventana, si bien en este caso se incrementa el nivel de detalle de la malla del objeto con la función *sphereDetail*, con el objetivo de obtener mejores resultados al iluminar. Observa las diferencias comentando dicha llamada.

Listado 5.2: Esfera con mayor detalle

```
void setup ()
{
  size(800, 800, P3D);
  fill(204);
}
```

```
sphereDetail(60);
}

void draw ()
{
  background(128);

  translate(width/2, height/2, 0);
  sphere(150);
}
```

La iluminación de ambiente no tiene definida dirección, e intenta simular la luz que llega a la superficie de los objetos por reflexión difusa de la luz tras rebotar en todos los elementos de la escena, aspecto no contemplado en el modelo de iluminación/reflexión que nos ocupa. La función admite tres o seis argumentos, los tres primeros definen el color, según del espacio de color activo, y los tres últimos localizan su posición. Con el objetivo de poder comparar, el listado 5.3 muestra una escena con tres esferas desplazadas, iluminadas con las condiciones de iluminación establecidas por defecto.

Listado 5.3: Tres esferas con iluminación por defecto

```
float ang;

void setup ()
{
  size(800, 800, P3D);
  ang=0;
  noStroke();
  sphereDetail(60);
}

void draw ()
{
  background(200);

  if (mousePressed) {
    lights();
  }

  //Dibuja objetos
  pushMatrix();
  translate(width/4, height/2, 0);
  rotateX(radians(-30));
  rotateY(radians(ang));
  sphere(75);
  popMatrix();

  pushMatrix();
  translate(width/2, height/2, 0);
  rotateX(radians(-30));
  rotateY(radians(ang));
```

```
sphere(75);
popMatrix();

pushMatrix();
translate(3*width/4, height/2, 0);
rotateX(radians(-30));
rotateY(radians(ang));
sphere(75);
popMatrix();

ang+=1;
if (ang>360) ang=0;
}
```

El listado 5.4 muestra el efecto de configurar la luz ambiental. En principio se muestran las esferas con la iluminación por defecto, si bien al hacer clic se establece una intensidad ambiente rojiza, de mayor o menor intensidad dependiendo de la posición del puntero. La rotación de las esferas es prácticamente imperceptible.

Listado 5.4: Esferas con modo por defecto y alternativa con únicamente intensidad ambiente con componente roja variable

```
float ang;

void setup()
{
  size(800, 800, P3D);
  ang=0;
  noStroke();
  sphereDetail(60);
}

void draw()
{
  background(200);

  if (mousePressed) {
    float val=(float)mouseX/(float)width*(float)255;
    ambientLight((int)val,0,0);
  }
  else
  {
    lights();
  }

  //Dibuja objetos
  pushMatrix();
  translate(width/4, height/2, 0);
  rotateX(radians(-30));
  rotateY(radians(ang));
  sphere(75);
  popMatrix();
}
```

```

pushMatrix();
translate(width/2, height/2, 0);
rotateX(radians(-30));
rotateY(radians(ang));
sphere(75);
popMatrix();

pushMatrix();
translate(3*width/4, height/2, 0);
rotateX(radians(-30));
rotateY(radians(ang));
sphere(75);
popMatrix();

ang+=1;
if (ang>360) ang=0;
}

```

La función *directionalLight* define una luz direccional, es decir una fuente de luz que viene desde una dirección específica. Cualquier luz afectará a la superficie dependiendo del ángulo entre la normal a la superficie del objeto y la dirección de la fuente de luz. La función dispone de seis parámetros, definiendo en primer lugar el color de la fuente de luz en los tres primeros, y la dirección de la luz en los restantes. El listado 5.5 presenta las esferas con iluminación por defecto, activando, al hacer clic, además de la iluminación ambiental rojiza, una fuente de luz direccional, con mayor componente verde, que viene desde un lateral.

Listado 5.5: Luz direccional

```

float ang;

void setup()
{
  size(800, 800, P3D);
  ang=0;
  noStroke();
  sphereDetail(60);
}

void draw()
{
  background(200);

  if (mousePressed) {
    float val=(float)mouseX/(float)width*(float)255;
    ambientLight((int)val,0,0);
    directionalLight(50, 200, 50, -1, 0, 0);
  }
  else
  {
    lights();
  }
}

```

```

}

//Dibuja objetos
pushMatrix();
translate(width/4, height/2, 0);
rotateX(radians(-30));
rotateY(radians(ang));
sphere(75);
popMatrix();

pushMatrix();
translate(width/2, height/2, 0);
rotateX(radians(-30));
rotateY(radians(ang));
sphere(75);
popMatrix();

pushMatrix();
translate(3*width/4, height/2, 0);
rotateX(radians(-30));
rotateY(radians(ang));
sphere(75);
popMatrix();

ang+=1;
if (ang>360) ang=0;
}

```

Las luces direccionales son luces localizadas en el infinito. Mayor flexibilidad la aporta la función *spotLight* que además de definir color y dirección, requiere parámetros para localizar la fuente de luz, el ángulo del cono de luz, y la concentración de la luz en dicho cono. El listado 5.6 añade a la escena con iluminación no por defecto una luz, que al mover el puntero llegará a provocar la presencia del reflejo sobre la superficie de la esfera.

Listado 5.6: Luz localizada

```

float ang;

void setup()
{
  size(800, 800, P3D);
  ang=0;
  noStroke();
  sphereDetail(60);
}

void draw()
{
  background(200);

  if (mousePressed) {
    float val=(float)mouseX/(float)width*(float)255;

```



```
    ambientLight((int)val,0,0);
    directionalLight(50, 200, 50, -1, 0, 0);
    spotLight(204, 153, 0, mouseX, mouseY, 500, 0, 0, -1, PI/2, 600);
}
else
{
    lights ();
}

//Dibuja objetos
pushMatrix();
translate(width/4, height/2, 0);
rotateX(radians(-30));
rotateY(radians(ang));
sphere(75);
popMatrix();

pushMatrix();
translate(width/2, height/2, 0);
rotateX(radians(-30));
rotateY(radians(ang));
sphere(75);
popMatrix();

pushMatrix();
translate(3*width/4, height/2, 0);
rotateX(radians(-30));
rotateY(radians(ang));
sphere(75);
popMatrix();

ang+=1;
if (ang>360) ang=0;
}
```

Para luces no localizadas en el infinito, la función *pointLight* fija una luz con un cono de 180° , simplificando la definición de luces localizadas, al requerir únicamente definir su color y posición, ver el listado 5.7.

Listado 5.7: Luz puntual

```
float ang;

void setup()
{
    size(800, 800, P3D);
    ang=0;
    noStroke();
    sphereDetail(60);
}

void draw()
{
```

```
background(200);

if (mousePressed) {
  float val=(float)mouseX/(float)width*(float)255;
  ambientLight((int)val,0,0);
  directionalLight(50, 200, 50, -1, 0, 0);
  pointLight(204, 153, 0, mouseX, mouseY, 400);
}
else
{
  lights();
}

//Dibuja objetos
pushMatrix();
translate(width/4, height/2, 0);
rotateX(radians(-30));
rotateY(radians(ang));
sphere(75);
popMatrix();

pushMatrix();
translate(width/2, height/2, 0);
rotateX(radians(-30));
rotateY(radians(ang));
sphere(75);
popMatrix();

pushMatrix();
translate(3*width/4, height/2, 0);
rotateX(radians(-30));
rotateY(radians(ang));
sphere(75);
popMatrix();

ang+=1;
if (ang>360) ang=0;
}
```

El color de la luz con reflexión especular se fija con la función *lightSpecular* requiriendo los tres valores del espacio de color como parámetros. El listado 5.8 establece el uso del reflejo especular. Apreciar las diferencias cuando se activa y cuando no.

Listado 5.8: Reflexión especular

```
float ang;

void setup()
{
  size(800, 800, P3D);
  ang=0;
  noStroke();
  sphereDetail(60);
}
```

```
}  
  
void draw ()  
{  
  background(200);  
  
  if (mousePressed) {  
    pointLight(204, 153, 0, mouseX, mouseY, 400);  
    lightSpecular(100, 100, 100);  
    directionalLight(0.8, 0.8, 0.8, 0, 0, -1);  
  }  
  else  
  {  
    lights();  
  }  
  
  //Dibuja objetos  
  pushMatrix();  
  translate(width*0.3, height/2, 0);  
  rotateX(radians(-30));  
  rotateY(radians(ang));  
  sphere(75);  
  popMatrix();  
  
  translate(width*0.6, height/2, 0);  
  rotateX(radians(-30));  
  rotateY(radians(ang));  
  float s = mouseX / float(width);  
  sphere(75);  
  
  ang+=1;  
  if (ang>360) ang=0;  
}
```

5.1.2. Materiales

El material del objeto también influye en la reflexión que se observa al incidir la luz sobre él. Para especificar características del material de un objeto están disponibles las funciones *ambient*, *emissive*, *specular* y *shininess* que configuran la respuesta a la iluminación de ambiente, reflexión difusa y especular, de las primitivas dibujadas a continuación. El listado 5.9 dibuja varias esferas, variando las características de la reflexión difusa y especular.

Listado 5.9: Variedad de materiales

```
float ang;  
  
void setup()  
{  
  size(800, 800, P3D);  
  ang=0;
```

```
noStroke();
sphereDetail(60);
}

void draw ()
{
  background(200);

  if (mousePressed) {
    lights();

    lightSpecular(100, 100, 100);
    directionalLight(0.8, 0.8, 0.8, 0, 0, -1);
  }

  //Dibuja objetos
  emissive(0,0,0);

  pushMatrix();
  translate(width*0.25, height*0.3, 0);
  rotateX(radians(-30));
  rotateY(radians(ang));
  specular(100, 100, 100);
  shininess(100);
  sphere(75);
  popMatrix();

  pushMatrix();
  translate(width*0.50, height*0.3, 0);
  rotateX(radians(-30));
  rotateY(radians(ang));
  specular(100, 100, 100);
  shininess(10);
  sphere(75);
  popMatrix();

  pushMatrix();
  translate(width*0.75, height*0.3, 0);
  rotateX(radians(-30));
  rotateY(radians(ang));
  specular(100, 100, 100);
  shininess(1);
  sphere(75);
  popMatrix();

  emissive(10,0,50);
  pushMatrix();
  translate(width*0.25, height*0.6, 0);
  rotateX(radians(-30));
  rotateY(radians(ang));
  specular(0, 0, 100);
  shininess(10);
  sphere(75);
  popMatrix();
```

```

emissive(50,0,50);
pushMatrix();
translate(width*0.5, height*0.6, 0);
rotateX(radians(-30));
rotateY(radians(ang));
specular(0, 0, 100);
shininess(10);
sphere(75);
popMatrix();

emissive(0,50,0);
pushMatrix();
translate(width*0.75, height*0.6, 0);
rotateX(radians(-30));
rotateY(radians(ang));

specular(0, 0, 50);
shininess(10);
sphere(75);
popMatrix();

ang+=1;
if (ang>360) ang=0;
}

```

Las características de un material pueden utilizarse en combinación con las funciones *pushStyle* y *popStyle* para que tengan efecto sólo en los objetos entre ambas llamadas.

5.2. TEXTURAS

Asignar texturas a una forma requiere de la función *texture* especificando en los vértices el mapeo de cada uno con respecto a las coordenadas u y v de la imagen. Con *textureMode* (*IMAGE* o *NORMAL*) se especifica si se trabaja en coordenadas de la imagen o normalizadas (0,1). El último modo evita tener claramente presentes las coordenadas de la imagen. Un ejemplo ilustrativo se muestra en el listado 5.10, que aplica una textura sobre una cara poligonal, compuesta por cuatro vértices. Si nuestra forma tuviera varias caras, tendrán que asociarse las coordenadas de la textura para cada cara poligonal. La llamada a la función *texture* debe estar entre *beginShape* y *endShape* para tener efecto.

Listado 5.10: Textura sobre recuadro

```

PImage img;

void setup() {
  size(640, 360, P3D);

```

```

    img = loadImage("logoulpgc.png");
}

void draw() {
    background(0);
    translate(width / 2, height / 2);
    textureMode(NORMAL);
    beginShape();
    texture(img);
    vertex(-100, -100, 0, 0, 0);
    vertex( 100, -100, 0, 1, 0);
    vertex( 100,  100, 0, 1, 1);
    vertex(-100,  100, 0, 0, 1);
    endShape();
}

```

La función *textureWrap* permite establecer si la textura se aplica una única vez o de forma cíclica en base al tamaño de la superficie sobre la que se mapea. El listado 5.11 hace uso del modo *REPEAT* que repite la textura en su caso, al contrario del modo por defecto *CLAMP*.

Listado 5.11: Textura sobre recuadro

```

PImage img;

void setup() {
    size(640, 360, P3D);
    img = loadImage("logoulpgc.png");
}

void draw() {
    background(0);
    translate(width / 2, height / 2);
    textureMode(NORMAL);
    textureWrap(REPEAT);
    beginShape();
    texture(img);
    vertex(-100, -100, 0, 0, 0);
    vertex( 100, -100, 0, 2, 0);
    vertex( 100,  100, 0, 2, 2);
    vertex(-100,  100, 0, 0, 2);
    endShape();
}

```

Para objetos tridimensionales más complicados, existen funcionalidades incluidas como la mostrada en el listado 5.12 para el caso de un elipsoide. ¿Qué ocurre si se aplica iluminación?

Listado 5.12: Textura sobre PShape 3D

```

PImage img;
PShape globo;
float ang;

```

```
void setup() {
  // Load an image
  size(600, 600, P3D);
  img = loadImage("logoulpgc.png");

  beginShape();
  globo = createShape(SPHERE, 150);
  globo.setStroke(255);
  globo.scale(1.85, 1.09, 1.15);
  globo.setTexture(img);
  endShape(CLOSE);

  ang=0;
}

void draw() {
  background(0);
  translate(width / 2, height / 2);
  rotateY(radians(ang));

  shape(globo);
  ang=ang+1;
  if (ang>360) ang=0;
}
```

Incluimos un ejemplo también con una tira de triángulos, que exige probablemente una mayor concentración a la hora de asociar vértices y el mapa de textura, ver listado [5.13](#).

Listado 5.13: Textura sobre PShape 3D

```
PImage img;

void setup() {
  size(600, 600, P3D);
  img = loadImage("logoulpgc.png");
}

void draw() {
  background(0);
  translate(width / 2, height / 2);
  rotateX(radians(360*mouseX/width));
  rotateY(radians(360*mouseY/height));
  textureMode(NORMAL);
  beginShape(TRIANGLE_STRIP);
  texture(img);
  vertex(-100, -300, 0, 0, 0);
  vertex( 100, -300, 0, 1, 0);
  vertex(-100, -100, 0, 0, 1);
  vertex( 100, -100, 0, 1, 1);
  vertex(-100,  100, 0, 0, 0);
  vertex( 100,  100, 0, 1, 0);
  endShape();
}
```

```
}
```

Un ejemplo final asocia como textura los fotogramas capturados por la cámara, podría hacerse también con los de un vídeo cargado de disco. El listado 5.14 proyecta la imagen sobre un recuadro.

Listado 5.14: Textura sobre PShape 3D

```
import processing.video.*;

PImage img;
Capture cam;

void setup() {
  size(800, 800, P3D);
  //Cámara
  cam = new Capture(this, 640, 480);
  cam.start();
}

void draw() {
  if (cam.available()) {
    background(0);
    cam.read();

    translate(width / 2, height / 2);
    rotateX(radians(mouseX/2));
    rotateY(radians(mouseY/2));
    textureMode(NORMAL);
    beginShape();
    texture(cam);
    vertex(-200, -200, 0, 0, 0);
    vertex(200, -200, 0, 1, 0);
    vertex(200, 200, 0, 1, 1);
    vertex(-200, 200, 0, 0, 1);
    endShape();
  }
}
```

A nivel interno OpenGL maneja la textura a distintas resoluciones, es lo que se conoce como *mipmap*, para en cada momento aplicar la textura sobre la figura con el menor *aliasing*. Processing se configura para la mayor calidad.

5.2.1. Imágenes

Si bien ya hemos tratado varios ejemplos con lectura de imágenes, describimos algún ejemplo adicional. Como ya se conoce, la función *loadImage* facilita la carga de imágenes desde disco, que puede ser visualizada con *image* como ilustra el listado 5.15.

Listado 5.15: Carga de imagen

```
PImage img;

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

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

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

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

Listado 5.16: Imagen que tiembla

```
PImage img;

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

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

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

La función *tint* permite variar la intensidad de la visualización, es la correspondiente a *stroke* o *fill* para primitivas gráficas.

Listado 5.17: Variando la intensidad de la imagen

```
PImage img;

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

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

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

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

Listado 5.18: Ciclo de animación

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

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

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

  frameRate (6);
}

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

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

5.3. SHADERS

Una mayor flexibilidad adicional se obtiene programando el *shader* propio [Colubri \[Accedido Febrero 2019\]](#). Los *shaders* definen el modo en el que se aplica la iluminación, texturas, etc., por lo que su programación abre la puerta para usuarios avanzados de buscar nuevas posibilidades.

Quedando fuera de los objetivos de esta práctica, confiamos en abordarlo más adelante.

5.4. TAREA

Componer una escena con objetos tridimensionales que incluya texturas, luces y movimiento de cámara.

La entrega se debe realizar a través del campus virtual, remitiendo una memoria o el enlace a un *screencast* que además de describir las decisiones adoptadas, identifique al remitente, las referencias y herramientas utilizadas, cuidando el formato y estilo. Como material adicional debe incluirse el enlace al código desarrollado (p.e. github), con su correspondiente

README, y un gif animado, de al menos 5 segundos, que ilustre ejecución del prototipo.

BIBLIOGRAFÍA

Andres Colubri. Shaders, Accedido Febrero 2019. URL <https://processing.org/tutorials/pshader/>.

Daniel Shiffman. P3D tutorial, Accedido Enero 2019. URL <https://processing.org/tutorials/p3d/>.

Práctica 6

Procesamiento de imagen y vídeo

6.1. CAPTURA DE VÍDEO

Por defecto la instalación de Processing no incluye las bibliotecas de vídeo, por lo que es necesario añadirlas para los ejemplos de esta práctica. Recordar que las contribuciones o bibliotecas se añaden a través de *Herramientas* → *Añadir herramientas* → *Libraries*. En esta ocasión buscaremos *Video*, escogiendo la identificada como *Video | GStreamer based video library for Processing*. Además de instalar la biblioteca, incorpora su correspondiente batería de ejemplos.

Un primer ejemplo de captura y visualización se incluye en el listado 6.1 que en caso de hacer clic, desplaza de forma leve y aleatoria la posición de la imagen. Como en este ejemplo, una vez capturado el fotograma, podremos hacer con él cualquier acción aplicable a una imagen, ver sección 5.2.1.

Listado 6.1: Captura de cámara, con tembleque al hacer clic

```
import processing.video.*;

Capture cam;

void setup() {
  size(640 , 480, P3D);
  //Cámara
  cam = new Capture(this , width , height);
  cam.start();
}

void draw() {
  if (cam.available()) {
    background(0);
    cam.read();
  }
}
```

```
//comportamiento diferenciado si hay clic
if (mousePressed)
  //Desplaza la imagen de forma aleatoria al mostrarla
  image(cam,random(-5,5),random(-5,5));
else
  image(cam,0,0);
}
}
```

Los siguientes apartados pretenden mostrar algunas pinceladas de operaciones posibles con los fotogramas de un vídeo.

6.2. OPERACIONES BÁSICAS

6.2.1. Píxeles

Una imagen se compone de píxeles, en el listado 6.2 se muestra el modo de acceder a dicha información, requiriendo una llamada al método *loadPixels* para habilitar el acceso, acceder propiamente a través de la variable *pixels*, y finalmente actualizar con el método *updatePixels*. En este ejemplo concreto se aplica un umbralizado, para cada píxel de la mitad superior de la imagen, se suman sus tres componentes, y si superan el valor $255 * 1,5$ se pinta en blanco, y negro en caso contrario.

Listado 6.2: Captura de cámara mostrando la mitad superior umbralizada

```
import processing.video.*;

Capture cam;
int dimension;

void setup() {
  size(640, 480);
  //Cámara
  cam = new Capture(this, width, height);
  cam.start();
  //Obtiene el número de píxeles de la imagen
  dimension = cam.width * cam.height;
}

void draw() {
  if (cam.available())
  {
    background(0);
    cam.read();

    //Carga píxeles para poder operar con ellos
    cam.loadPixels();
    //Recorre la parte superior de la imagen
```

```
for (int i=1;i<dimension/2;i++)
{
    //Suma las tres componentes del píxel
    float suma=red(cam.pixels[i])+green(cam.pixels[i])+blue(cam.pixels[i]);

    //Umbraliza, asigna blanco o negro, en base a comparar el valor intermedio
    if (suma<255*1.5)
    {
        cam.pixels[i]=color(0, 0, 0);
    }
    else
    {
        cam.pixels[i]=color(255, 255, 255);
    }
}
//Actualiza pixeles
cam.updatePixels();
}
//Muestra la imagen
image( cam ,0 ,0 );
}
```

6.2.2. OpenCV

De cara a poder realizar procesamiento de imágenes tanto básico como elaborado, es aconsejable hacer uso de utilidades existente, como es el caso de la biblioteca OpenCV [OpenCV team \[Accedido Marzo 2019\]](#). Para los ejemplos mostrados a continuación hemos hecho uso de *CVImage* [Chung \[Accedido Marzo 2019a\]](#). Indicar que a través del menú de herramientas se facilita la instalación de la adaptación realizada por Greg Borenstein para OpenCV 2.4.5; si bien al no estar actualizada para las versiones recientes de OpenCV, optamos por la versión compilada para Java puesta a disposición por Bryan Chung para OpenCV 4.0.0, la mencionada *CVImage* [Chung \[Accedido Marzo 2019a\]](#). Su instalación requiere descomprimir y copiar a la carpeta denominada *libraries* de nuestra instalación de Processing. No es posible mantener ambas librerías instaladas de forma simultánea, ya que entran en conflicto.

6.2.3. Grises

Un primer ejemplo muestra el uso de la variable de tipo *CVImage* para obtener su versión en tonos de grises. Como veremos posteriormente en diversos ejemplos, la imagen capturada en color, para diversos operaciones es necesario convertirla a escala de grises, el listado 6.3 recupera y muestra la imagen de grises correspondiente a la captura.

El código presenta varias novedades, al hacer uso de la variable tipo *CVImage*, requerir la utilización de la biblioteca OpenCV, y la copia de una variable *Mat* con valores *unsigned byte*

a otra *CVImage*. El resultado muestra tanto la imagen de entrada, como la convertida a grises.

Listado 6.3: Muestra imagen capturada en RGB y grises

```
import java.lang.*;
import processing.video.*;
import cvimage.*;
import org.opencv.core.*;

Capture cam;
CVImage img,auximg;

void setup() {
  size(1280, 480);
  //Cámara
  cam = new Capture(this, width/2, height);
  cam.start();

  //OpenCV
  //Carga biblioteca core de OpenCV
  System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
  println(Core.VERSION);
  //Crea imágenes
  img = new CVImage(cam.width, cam.height);
  auximg=new CVImage(cam.width, cam.height);
}

void draw() {
  if (cam.available()) {
    background(0);
    cam.read();

    //Obtiene la imagen de la cámara
    img.copy(cam, 0, 0, cam.width, cam.height,
    0, 0, img.width, img.height);
    img.copyTo();

    //Imagen de grises
    Mat gris = img.getGrey();

    //Copia de Mat a CVImage
    cpMat2CVImage(gris,auximg);

    //Visualiza ambas imágenes
    image(img,0,0);
    image(auximg,width/2,0);

    //Libera
    gris.release();
  }
}

//Copia unsigned byte Mat a color CVImage
void cpMat2CVImage(Mat in_mat,CVImage out_img)
```



```

{
    byte[] data8 = new byte[cam.width*cam.height];

    out_img.loadPixels();
    in_mat.get(0, 0, data8);

    // Cada columna
    for (int x = 0; x < cam.width; x++) {
        // Cada fila
        for (int y = 0; y < cam.height; y++) {
            // Posición en el vector ID
            int loc = x + y * cam.width;
            // Conversión del valor a unsigned basado en
            // https://stackoverflow.com/questions/4266756/can-we-make-unsigned-byte-in-java
            int val = data8[loc] & 0xFF;
            // Copia a CVImage
            out_img.pixels[loc] = color(val);
        }
    }
    out_img.updatePixels();
}

```

6.2.4. Umbralizado

El ejemplo de acceso a los píxeles, listado 6.2, aplicaba un umbralizado a la parte superior de la imagen capturada. El listado 6.4 hace uso de las utilidades para tal fin presentes en OpenCV, en concreto la función *threshold*. El código modifica el valor del umbral aplicado según la posición del puntero. Las personas más curiosas pueden analizar las opciones de dicha función en la documentación.

Listado 6.4: Umbralizado dependiente de la posición del puntero

```

import java.lang.*;
import processing.video.*;
import cvimage.*;
import org.opencv.core.*;
import org.opencv.imgproc.Imgproc;

Capture cam;
CVImage img,auximg;

void setup() {
    size(1280, 480);
    //Cámara
    cam = new Capture(this, width/2, height);
    cam.start();

    //OpenCV
    //Carga biblioteca core de OpenCV
    System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
}

```

```
println(Core.VERSION);
//Crea imágenes
img = new CVImage(cam.width, cam.height);
auximg=new CVImage(cam.width, cam.height);
}

void draw() {
    if (cam.available()) {
        background(0);
        cam.read();

        //Obtiene la imagen de la cámara
        img.copy(cam, 0, 0, cam.width, cam.height,
        0, 0, img.width, img.height);
        img.copyTo();

        //Imagen de grises
        Mat gris = img.getGrey();

        //Umbraliza con umbral definido por la posición del ratón
        Imgproc.threshold(ggris, ggris, 255*mouseX/width, 255, Imgproc.THRESH_BINARY);

        //Copia de Mat a CVImage
        cpMat2CVImage(ggris, auximg);

        //Visualiza ambas imágenes
        image(img, 0, 0);
        image(auximg, width/2, 0);

        //Libera
        gris.release();
    }
}

//Copia unsigned byte Mat a color CVImage
void cpMat2CVImage(Mat in_mat, CVImage out_img)
{
    byte[] data8 = new byte[cam.width*cam.height];

    out_img.loadPixels();
    in_mat.get(0, 0, data8);

    // Cada columna
    for (int x = 0; x < cam.width; x++) {
        // Cada fila
        for (int y = 0; y < cam.height; y++) {
            // Posición en el vector ID
            int loc = x + y * cam.width;
            //Conversión del valor a unsigned basado en
            //https://stackoverflow.com/questions/4266756/can-we-make-unsigned-byte-in-java
            int val = data8[loc] & 0xFF;
            //Copia a CVImage
            out_img.pixels[loc] = color(val);
        }
    }
}
```

```
}  
  out_img.updatePixels();  
}
```

6.2.5. Bordes

La detección de contornos o bordes es un proceso habitual sobre imágenes, el resultado de la aplicación del conocido detector de Canny se muestra en el listado 6.5. El código es bastante similar al anterior, modificando la función aplicada, dado que la imagen de la derecha muestra en blanco los contornos detectados haciendo uso de la función *Canny*.

Listado 6.5: Aplicación de Canny sobre una imagen

```
import processing.video.*;  
import cvimage.*;  
import org.opencv.core.*;  
import org.opencv.imgproc.Imgproc;  
  
Capture cam;  
CVImage img,auximg;  
  
void setup() {  
  size(1280, 480);  
  //Cámara  
  cam = new Capture(this, width/2, height);  
  cam.start();  
  
  //OpenCV  
  //Carga biblioteca core de OpenCV  
  System.loadLibrary(Core.NATIVE_LIBRARY_NAME);  
  println(Core.VERSION);  
  //Crea imágenes  
  img = new CVImage(cam.width, cam.height);  
  auximg=new CVImage(cam.width, cam.height);  
}  
  
void draw() {  
  if (cam.available()) {  
    background(0);  
    cam.read();  
  
    //Obtiene la imagen de la cámara  
    img.copy(cam, 0, 0, cam.width, cam.height,  
            0, 0, img.width, img.height);  
    img.copyTo();  
  
    //Imagen de grises  
    Mat gris = img.getGrey();  
  
    //Aplica Canny
```

```

    Imgproc.Canny( gris , gris ,20 ,60,3);

    //Copia de Mat a CVImage
    cpMat2CVImage( gris ,auximg);

    //Visualiza ambas imágenes
    image(img,0,0);
    image(auximg, width/2,0);

    gris.release();
}
}

//Copia unsigned byte Mat a color CVImage
void cpMat2CVImage(Mat in_mat,CVImage out_img)
{
    byte[] data8 = new byte[cam.width*cam.height];

    out_img.loadPixels();
    in_mat.get(0, 0, data8);

    // Cada columna
    for (int x = 0; x < cam.width; x++) {
        // Cada fila
        for (int y = 0; y < cam.height; y++) {
            // Posición en el vector ID
            int loc = x + y * cam.width;
            //Conversión del valor a unsigned basado en
            //https://stackoverflow.com/questions/4266756/can-we-make-unsigned-byte-in-java
            int val = data8[loc] & 0xFF;
            //Copia a CVImage
            out_img.pixels[loc] = color(val);
        }
    }
    out_img.updatePixels();
}
}

```

Una gradación del resultado de los contornos se obtiene como resultado del código del listado 6.6, que a partir de los gradientes en x e y , estima el valor total. El resultado, al ser degradado, da sensación de mayor estabilidad. Con respecto a los ejemplos previos, se modifica el procesamiento realizado, el bloque es similar.

Listado 6.6: Resultado de aplicación del cálculo del gradiente

```

import processing.video.*;
import cvimage.*;
import org.opencv.core.*;
import org.opencv.imgproc.Imgproc;

Capture cam;
CVImage img,auximg;

void setup() {

```

```
size(1280, 480);
//Cámara
cam = new Capture(this, width/2, height);
cam.start();

//OpenCV
//Carga biblioteca core de OpenCV
System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
println(Core.VERSION);
//Crea imágenes
img = new CVImage(cam.width, cam.height);
auximg=new CVImage(cam.width, cam.height);
}

void draw() {
  if (cam.available()) {
    background(0);
    cam.read();

    //Obtiene la imagen de la cámara
    img.copy(cam, 0, 0, cam.width, cam.height,
    0, 0, img.width, img.height);
    img.copyTo();

    //Imagen de grises
    Mat gris = img.getGrey();

    //Gradiente
    int scale = 1;
    int delta = 0;
    int ddepth = CvType.CV_16S;
    Mat grad_x = new Mat();
    Mat grad_y = new Mat();
    Mat abs_grad_x = new Mat();
    Mat abs_grad_y = new Mat();

    // Gradiente X
    Imgproc.Sobel(gris, grad_x, ddepth, 1, 0);
    Core.convertScaleAbs(grad_x, abs_grad_x);

    // Gradiente Y
    Imgproc.Sobel(gris, grad_y, ddepth, 0, 1);
    Core.convertScaleAbs(grad_y, abs_grad_y);

    // Gradiente total aproximado
    Core.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0, gris);

    //Copia de Mat a CVImage
    cpMat2CVImage(gris, auximg);

    //Visualiza ambas imágenes
    image(img, 0, 0);
    image(auximg, width/2, 0);
```

```
    gris.release();
}
}

//Copia unsigned byte Mat a color CVImage
void cpMat2CVImage(Mat in_mat,CVImage out_img)
{
    byte[] data8 = new byte[cam.width*cam.height];

    out_img.loadPixels();
    in_mat.get(0, 0, data8);

    // Cada columna
    for (int x = 0; x < cam.width; x++) {
        // Cada fila
        for (int y = 0; y < cam.height; y++) {
            // Posición en el vector ID
            int loc = x + y * cam.width;
            //Conversión del valor a unsigned basado en
            //https://stackoverflow.com/questions/4266756/can-we-make-unsigned-byte-in-java
            int val = data8[loc] & 0xFF;
            //Copia a CVImage
            out_img.pixels[loc] = color(val);
        }
    }
    out_img.updatePixels();
}
```

6.2.6. Diferencias

Para una cámara fija, el cálculo de diferencias con el modelo de fondo, o como en el listado 6.7, con el fotograma previo, indica las zonas de la imagen donde ha existido movimiento, mostrando el valor absoluto de dicha diferencia.

Listado 6.7: Diferencia entre imágenes sucesivas

```
import processing.video.*;
import cvimage.*;
import org.opencv.core.*;
import org.opencv.imgproc.Imgproc;

Capture cam;
CVImage img,pimg,auximg;

void setup() {
    size(1280, 480);
    //Cámara
    cam = new Capture(this, width/2, height);
    cam.start();

    //OpenCV
```

```
//Carga biblioteca core de OpenCV
System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
println(Core.VERSION);
img = new CVImage(cam.width, cam.height);
pimg = new CVImage(cam.width, cam.height);
auximg=new CVImage(cam.width, cam.height);
}

void draw() {
  if (cam.available()) {
    background(0);
    cam.read();

    //Obtiene la imagen de la cámara
    img.copy(cam, 0, 0, cam.width, cam.height,
    0, 0, img.width, img.height);
    img.copyTo();

    //Imagen de grises
    Mat gris = img.getGrey();
    Mat pgris = pimg.getGrey();

    //Calcula diferencias entre el fotograma actual y el previo
    Core.absdiff(gris, pgris, gris);

    //Copia de Mat a CVImage
    cpMat2CVImage(gris, auximg);

    //Visualiza ambas imágenes
    image(img,0,0);
    image(auximg,width/2,0);

    //Copia actual en previa para próximo fotograma
    pimg.copy(img, 0, 0, img.width, img.height,
    0, 0, img.width, img.height);
    pimg.copyTo();

    gris.release();
  }
}

//Copia unsigned byte Mat a color CVImage
void cpMat2CVImage(Mat in_mat,CVImage out_img)
{
  byte[] data8 = new byte[cam.width*cam.height];

  out_img.loadPixels();
  in_mat.get(0, 0, data8);

  // Cada columna
  for (int x = 0; x < cam.width; x++) {
    // Cada fila
```

```
for (int y = 0; y < cam.height; y++) {
  // Posición en el vector ID
  int loc = x + y * cam.width;
  // Conversión del valor a unsigned basado en
  // https://stackoverflow.com/questions/4266756/can-we-make-unsigned-byte-in-java
  int val = data8[loc] & 0xFF;
  // Copia a CVImage
  out_img.pixels[loc] = color(val);
}
}
out_img.updatePixels();
}
```

6.3. DETECCIÓN

Ciertamente la Visión por Computador ofrece mucho más, otras operaciones con imágenes se introducen en el tutorial de Processing [Shiffman \[Accedido Marzo 2019\]](#), si bien para posibilidades más avanzadas, una introducción a la la Visión por Computador, con el borrador accesible online, es la de Richard Szeliski [Szeliski \[2010\]](#). Desde un enfoque centrado en la interacción hombre-máquina, dado que no estamos en una asignatura de Visión por Computador, entre los aspectos más avanzados incluimos varios ejemplos basados en detección de objetos, más concretamente detección de personas. Para varios de ellos los ejemplos se basan en el blog de Bryan Chung [Chung \[Accedido Marzo 2019b\]](#), y en general requieren de detectores/clasificadores presentes en la carpeta *data*.

6.3.1. Caras

La detección de caras se incorpora en OpenCV desde la implementación de Rainer Lienhart [Lienhart and Maydt \[2002\]](#) del detector conocido de Viola y Jones [Viola and Jones \[2004\]](#). El listado 6.8 muestra la aplicación del detector de caras, incluyendo la búsqueda de ojos en la zona donde se haya localizado el rostro, haciendo uso de los detectores de ojos desarrollados en nuestro grupo e incluidos en OpenCV desde 2009 [Castrillón et al. \[2011\]](#). Como se ha mencionado, es necesario disponer de los modelos en la carpeta *data* del prototipo.

Listado 6.8: Detección de caras y ojos

```
import java.lang.*;
import processing.video.*;
import cvimage.*;
import org.opencv.core.*;
// Detectores
import org.opencv.objdetect.CascadeClassifier;
import org.opencv.objdetect.Objdetect;
```



```
Capture cam;
CvImage img;

//Cascadas para detección
CascadeClassifier face, leye, reye;
//Nombres de modelos
String faceFile, leyeFile, reyeFile;

void setup() {
    size(640, 480);
    //Cámara
    cam = new Capture(this, width, height);
    cam.start();

    //OpenCV
    //Carga biblioteca core de OpenCV
    System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
    println(Core.VERSION);
    img = new CvImage(cam.width, cam.height);

    //Detectores
    faceFile = "haarcascade_frontalface_default.xml";
    leyeFile = "haarcascade_mcs_lefteye.xml";
    reyeFile = "haarcascade_mcs_righteye.xml";
    face = new CascadeClassifier(dataPath(faceFile));
    leye = new CascadeClassifier(dataPath(leyeFile));
    reye = new CascadeClassifier(dataPath(reyeFile));
}

void draw() {
    if (cam.available()) {
        background(0);
        cam.read();

        //Obtiene la imagen de la cámara
        img.copy(cam, 0, 0, cam.width, cam.height,
            0, 0, img.width, img.height);
        img.copyTo();

        //Imagen de grises
        Mat gris = img.getGrey();

        //Imagen de entrada
        image(img, 0, 0);

        //Detección y pintado de contenedores
        FaceDetect(ggris);

        gris.release();
    }
}

void FaceDetect(Mat grey)
```

```
{
    Mat auxroi;

    //Detección de rostros
    MatOfRect faces = new MatOfRect();
    face.detectMultiScale(grey, faces, 1.15, 3,
        Objdetect.CASCADE_SCALE_IMAGE,
        new Size(60, 60), new Size(200, 200));
    Rect [] facesArr = faces.toArray();

    //Dibuja contenedores
    noFill();
    stroke(255,0,0);
    strokeWeight(4);
    for (Rect r : facesArr) {
        rect(r.x, r.y, r.width, r.height);
    }

    //Búsqueda de ojos
    MatOfRect leyes, reyes;
    for (Rect r : facesArr) {
        //Izquierdo (en la imagen)
        leyes = new MatOfRect();
        Rect roi=new Rect(r.x,r.y,(int)(r.width*0.7),(int)(r.height*0.6));
        auxroi= new Mat(grey, roi);

        //Detecta
        leye.detectMultiScale(auxroi, leyes, 1.15, 3,
            Objdetect.CASCADE_SCALE_IMAGE,
            new Size(30, 30), new Size(200, 200));
        Rect [] leyesArr = leyes.toArray();

        //Dibuja
        stroke(0,255,0);
        for (Rect rl : leyesArr) {
            rect(rl.x+r.x, rl.y+r.y, rl.height, rl.width); //Strange dimenions change
        }
        leyes.release();
        auxroi.release();

        //Derecho (en la imagen)
        reyes = new MatOfRect();
        roi=new Rect(r.x+(int)(r.width*0.3),r.y,(int)(r.width*0.7),(int)(r.height*0.6));
        auxroi= new Mat(grey, roi);

        //Detecta
        reye.detectMultiScale(auxroi, reyes, 1.15, 3,
            Objdetect.CASCADE_SCALE_IMAGE,
            new Size(30, 30), new Size(200, 200));
        Rect [] reyesArr = reyes.toArray();

        //Dibuja
        stroke(0,0,255);
    }
}
```

```
for (Rect rl : reyesArr) {
    rect(rl.x+r.x+(int)(r.width*0.3), rl.y+r.y, rl.height, rl.width); //Strange dimenions change
}
reyes.release();
auxroi.release();
}

faces.release();
}
```

Una variante más costosa, tras detectar el rostro hace uso de un modelo de sus elementos para encajarlo en la imagen. El listado 6.11 basado en el ejemplo de Bryan Chung¹ con el modelo proporcionado por OpenCV².

Listado 6.9: Detección del modelo del rostro

```
import java.lang.*;
import processing.video.*;
import cvimage.*;
import org.opencv.core.*;
//Detectores
import org.opencv.objdetect.CascadeClassifier;
//Máscara del rostro
import org.opencv.face.Face;
import org.opencv.face.FaceMark;

Capture cam;
CVImage img;

//Detectores
CascadeClassifier face;
//Máscara del rostro
FaceMark fm;
//Nombres
String faceFile, modelFile;

void setup() {
    size(640, 480);
    //Cámara
    cam = new Capture(this, width, height);
    cam.start();

    //OpenCV
    //Carga biblioteca core de OpenCV
    System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
    println(Core.VERSION);
    img = new CVImage(cam.width, cam.height);

    //Detectores
```

¹<http://www.magicandlove.com/blog/2018/08/19/face-landmark-detection-in-opencv-face-module-with-processing/>

²https://github.com/opencv/opencv_3rdparty/tree/contrib_face_alignment_20170818

```
faceFile = "haarcascade_frontalface_default.xml";
//Modelo de máscara
modelFile = "face_landmark_model.dat";
fm = Face.createFacemarkKazemi();
fm.loadModel(dataPath(modelFile));
}

void draw() {
  if (cam.available()) {
    background(0);
    cam.read();

    //Get image from cam
    img.copy(cam, 0, 0, cam.width, cam.height,
             0, 0, img.width, img.height);
    img.copyTo();

    //Imagen de entrada
    image(img,0,0);
    //Detección de puntos fiduciales
    ArrayList<MatOfPoint2f> shapes = detectFacemarks(cam);
    PVector origin = new PVector(0, 0);
    for (MatOfPoint2f sh : shapes) {
      Point [] pts = sh.toArray();
      drawFacemarks(pts, origin);
    }
  }
}

private ArrayList<MatOfPoint2f> detectFacemarks(PImage i) {
  ArrayList<MatOfPoint2f> shapes = new ArrayList<MatOfPoint2f>();
  CVImage im = new CVImage(i.width, i.height);
  im.copyTo(i);
  MatOfRect faces = new MatOfRect();
  Face.getFacesHAAR(im.getBGR(), faces, dataPath(faceFile));
  if (!faces.empty()) {
    fm.fit(im.getBGR(), faces, shapes);
  }
  return shapes;
}

private void drawFacemarks(Point [] p, PVector o) {
  pushStyle();
  noStroke();
  fill(255);
  for (Point pt : p) {
    ellipse((float)pt.x+o.x, (float)pt.y+o.y, 3, 3);
  }
  popStyle();
}
```

6.3.2. Personas

6.3.2.1. Kinect

Para la versión 1 del sensor, no hemos logrado ejecutar en un sistema operativo que no sea Windows. Desde Processing ha sido necesario realizar previamente los siguientes pasos:

- Instalar el SDK para Kinect v1.8. Disponemos de copia, si bien está disponible en este [enlace](#).
- Conectar el dispositivo.
- Desde Processing acceder al menú *Herramientas* → *Añadir herramientas* → *Libraries*, buscar Kinect. Instalamos *Kinect4WinSDK*.

A partir del ejemplo base de Bryan Chung [Chung \[Accedido Marzo 2019c\]](#), incluido en la galería de ejemplos tras instalar la biblioteca, hemos adaptado listado [6.10](#) para además de los esqueletos, mostrar el modo de dibujar en la mano derecha del primero de ellos.

Listado 6.10: Esqueletos detectados con Kinect v1, con círculo en la mano derecha del primero

```
import kinect4WinSDK.Kinect;
import kinect4WinSDK.SkeletonData;

Kinect kinect;
ArrayList <SkeletonData> bodies;

void setup()
{
  size(640, 480);
  background(0);
  //Inicializaciones relacionadas con la Kinect
  kinect = new Kinect(this);
  smooth();
  bodies = new ArrayList<SkeletonData>();
}

void draw()
{
  background(0);
  //Pinta las imágenes de entrada, profundidad y máscara
  image(kinect.GetImage(), 320, 0, 320, 240);
  image(kinect.GetDepth(), 320, 240, 320, 240);
  image(kinect.GetMask(), 0, 240, 320, 240);
  //Dibuja esqueletos
  for (int i=0; i<bodies.size (); i++)
  {
    drawSkeleton(bodies.get(i));
  }
}
```

```
drawPosition(bodies.get(i));

//Circunferencia mano derecha pimer esqueleto
if (i==0)
{
    pushStyle();
    fill(255,0,0,50);
    //Detectada
    if (bodies.get(i).skeletonPositionTrackingState[Kinect.NUI_SKELETON_POSITION_HAND_RIGHT]!=Kinect.
        NUI_SKELETON_POSITION_NOT_TRACKED)
    {
        ellipse(bodies.get(i).skeletonPositions[Kinect.NUI_SKELETON_POSITION_HAND_RIGHT].x*width/2,
            bodies.get(i).skeletonPositions[Kinect.NUI_SKELETON_POSITION_HAND_RIGHT].y*height/2,30,30);
    }
    popStyle();
}
}

void drawPosition(SkeletonData _s)
{
    noStroke();
    fill(0, 100, 255);
    String s1 = str(_s.dwTrackingID);
    text(s1, _s.position.x*width/2, _s.position.y*height/2);
}

//Interfaz para dibujar el esqueleto
void drawSkeleton(SkeletonData _s)
{
    // Cuerpo
    DrawBone(_s,
        Kinect.NUI_SKELETON_POSITION_HEAD,
        Kinect.NUI_SKELETON_POSITION_SHOULDER_CENTER);
    DrawBone(_s,
        Kinect.NUI_SKELETON_POSITION_SHOULDER_CENTER,
        Kinect.NUI_SKELETON_POSITION_SHOULDER_LEFT);
    DrawBone(_s,
        Kinect.NUI_SKELETON_POSITION_SHOULDER_CENTER,
        Kinect.NUI_SKELETON_POSITION_SHOULDER_RIGHT);
    DrawBone(_s,
        Kinect.NUI_SKELETON_POSITION_SHOULDER_CENTER,
        Kinect.NUI_SKELETON_POSITION_SPINE);
    DrawBone(_s,
        Kinect.NUI_SKELETON_POSITION_SHOULDER_LEFT,
        Kinect.NUI_SKELETON_POSITION_SPINE);
    DrawBone(_s,
        Kinect.NUI_SKELETON_POSITION_SHOULDER_RIGHT,
        Kinect.NUI_SKELETON_POSITION_SPINE);
    DrawBone(_s,
        Kinect.NUI_SKELETON_POSITION_SPINE,
        Kinect.NUI_SKELETON_POSITION_HIP_CENTER);
    DrawBone(_s,
        Kinect.NUI_SKELETON_POSITION_HIP_CENTER,
```

```
Kinect.NUI_SKELETON_POSITION_HIP_LEFT);
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_HIP_CENTER,
Kinect.NUI_SKELETON_POSITION_HIP_RIGHT);
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_HIP_LEFT,
Kinect.NUI_SKELETON_POSITION_HIP_RIGHT);

// Brazo izquierdo
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_SHOULDER_LEFT,
Kinect.NUI_SKELETON_POSITION_ELBOW_LEFT);
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_ELBOW_LEFT,
Kinect.NUI_SKELETON_POSITION_WRIST_LEFT);
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_WRIST_LEFT,
Kinect.NUI_SKELETON_POSITION_HAND_LEFT);

// Brazo derecho
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_SHOULDER_RIGHT,
Kinect.NUI_SKELETON_POSITION_ELBOW_RIGHT);
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_ELBOW_RIGHT,
Kinect.NUI_SKELETON_POSITION_WRIST_RIGHT);
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_WRIST_RIGHT,
Kinect.NUI_SKELETON_POSITION_HAND_RIGHT);

// Pierna izquierda
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_HIP_LEFT,
Kinect.NUI_SKELETON_POSITION_KNEE_LEFT);
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_KNEE_LEFT,
Kinect.NUI_SKELETON_POSITION_ANKLE_LEFT);
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_ANKLE_LEFT,
Kinect.NUI_SKELETON_POSITION_FOOT_LEFT);

// Pierna derecha
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_HIP_RIGHT,
Kinect.NUI_SKELETON_POSITION_KNEE_RIGHT);
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_KNEE_RIGHT,
Kinect.NUI_SKELETON_POSITION_ANKLE_RIGHT);
DrawBone(_s,
Kinect.NUI_SKELETON_POSITION_ANKLE_RIGHT,
Kinect.NUI_SKELETON_POSITION_FOOT_RIGHT);
}

//Interfaz para dibujar un hueso
```

```
void DrawBone(SkeletonData _s, int _j1, int _j2)
{
    noFill();
    stroke(255, 255, 0);
    //Comprueba validez del dato
    if (_s.skeletonPositionTrackingState[_j1] != Kinect.NUI_SKELETON_POSITION_NOT_TRACKED &&
        _s.skeletonPositionTrackingState[_j2] != Kinect.NUI_SKELETON_POSITION_NOT_TRACKED) {
        line(_s.skeletonPositions[_j1].x*width/2,
            _s.skeletonPositions[_j1].y*height/2,
            _s.skeletonPositions[_j2].x*width/2,
            _s.skeletonPositions[_j2].y*height/2);
    }
}

void appearEvent(SkeletonData _s)
{
    if (_s.trackingState == Kinect.NUI_SKELETON_NOT_TRACKED)
    {
        return;
    }
    synchronized(bodies) {
        bodies.add(_s);
    }
}

void disappearEvent(SkeletonData _s)
{
    synchronized(bodies) {
        for (int i=bodies.size()-1; i>=0; i--)
        {
            if (_s.dwTrackingID == bodies.get(i).dwTrackingID)
            {
                bodies.remove(i);
            }
        }
    }
}

void moveEvent(SkeletonData _b, SkeletonData _a)
{
    if (_a.trackingState == Kinect.NUI_SKELETON_NOT_TRACKED)
    {
        return;
    }
    synchronized(bodies) {
        for (int i=bodies.size()-1; i>=0; i--)
        {
            if (_b.dwTrackingID == bodies.get(i).dwTrackingID)
            {
                bodies.get(i).copy(_a);
                break;
            }
        }
    }
}
```



```
}
```

La segunda versión del sensor proporciona mayor resolución y calidad tanto en la imagen en color, como en los datos de profundidad obtenidos. La biblioteca para Processing se debe a Thomas Sanchez Lengeling [Lengeling \[Accedido Marzo 2019\]](#). Su instalación es posible a través de la opción de menú *Añadir Herramientas*, y dentro de la pestaña *Libraries* buscar la biblioteca *Kinect v2 for Processing*. Previamente es necesario instalar el *Kinect SDK v2*³, además de contar con un equipo con USB 3.0 y 64 bits. Con estas restricciones y disponiendo de un único sensor de estas características, no nos permite verlo, si bien se han incluido las pautas para su instalación, que posibilitan la posterior ejecución de ejemplos de la galería de ejemplos de la biblioteca instalada. Similar al de la versión previa del sensor puede ser *SkeletonMaskDepth*, y sin GPU puede ser también interesante ejecutar *PointCloudOGL*.

6.3.2.2. Openpose

De nuevo basado en el ejemplo de Bryan Chung para un individuo⁴, el listado 6.11 aplica el detector basado en Openpose [Cao et al. \[2017\]](#) para determinar el esqueleto de una persona. Los requisitos computacionales suben significativamente, no habiendo probado su ejecución en un equipo con GPU.

Listado 6.11: Detección de esqueleto en 2D

```
//Adaptado de fuente original http://www.magicandlove.com/blog/2018/08/06/openpose-in-processing-and-opencv-dnn/
import processing.video.*;
import cvimage.*;
import org.opencv.core.*;
import org.opencv.core.Core.MinMaxLocResult;
import org.opencv.dnn.*;
import java.util.*;

final int CELL = 46;
final int W = 368, H = 368;
final float THRESH = 0.1f;
static int pairs[][] = {
  {1, 2}, // left shoulder
  {1, 5}, // right shoulder
  {2, 3}, // left arm
  {3, 4}, // left forearm
  {5, 6}, // right arm
  {6, 7}, // right forearm
  {1, 8}, // left body
  {8, 9}, // left thigh
  {9, 10}, // left calf
}
```

³<https://www.microsoft.com/en-us/download/details.aspx?id=44561>

⁴<http://www.magicandlove.com/blog/2018/08/06/openpose-in-processing-and-opencv-dnn/>

```
{1, 11}, // right body
{11, 12}, // right thigh
{12, 13}, // right calf
{1, 0}, // neck
{0, 14}, // left nose
{14, 16}, // left eye
{0, 15}, // right nose
{15, 17} // right eye
};
private float xRatio, yRatio;
private CVImage img;
private Net net;
private Capture cam;

public void setup() {
    size(320, 240);
    //Cámara
    cam = new Capture(this, width, height);
    cam.start();

    //OpenCV
    //Carga biblioteca core de OpenCV
    System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
    println(Core.VERSION);
    img = new CVImage(W, H);

    //Carga modelos
    net = Dnn.readNetFromCaffe(dataPath("openpose_pose_coco.prototxt"),
        dataPath("pose_iter_440000.caffemodel"));
    //net.setPreferableBackend(Dnn.DNN_BACKEND_DEFAULT);
    //net.setPreferableTarget(Dnn.DNN_TARGET_OPENCL);

    //Relación cámara ventana
    xRatio = (float)width / W;
    yRatio = (float)height / H;
}

public void draw() {
    if (cam.available()) {
        //Lectura del sensor
        cam.read();
        background(0);
        //Muestra imagen capturada
        image(cam, 0, 0);

        //Obtiene imagen para procesamiento
        img.copy(cam, 0, 0, cam.width, cam.height,
            0, 0, img.width, img.height);
        img.copyTo();
        //Procesa
        Mat blob = Dnn.blobFromImage(img.getBGR(), 1.0/255,
            new Size(img.width, img.height),
```

```
    new Scalar(0, 0, 0), false, false);
net.setInput(blob);
Mat result = net.forward().reshape(1, 19);

ArrayList<Point> points = new ArrayList<Point>();
for (int i=0; i<18; i++) {
    Mat heatmap = result.row(i).reshape(1, CELL);
    MinMaxLocResult mm = Core.minMaxLoc(heatmap);
    Point p = new Point();
    if (mm.maxVal > THRESH) {
        p = mm.maxLoc;
    }
    heatmap.release();
    points.add(p);
}

//Dibuja esqueleto
float sx = (float)img.width*xRatio/CELL;
float sy = (float)img.height*yRatio/CELL;
pushStyle();
noFill();
strokeWeight(2);
stroke(255, 255, 0);
for (int n=0; n<pairs.length; n++) {
    Point a = points.get(pairs[n][0]).clone();
    Point b = points.get(pairs[n][1]).clone();
    if (a.x <= 0 ||
        a.y <= 0 ||
        b.x <= 0 ||
        b.y <= 0)
        continue;
    a.x *= sx;
    a.y *= sy;
    b.x *= sx;
    b.y *= sy;
    line((float)a.x, (float)a.y,
        (float)b.x, (float)b.y);
}
popStyle();
blob.release();
result.release();
}
}
```

6.4. GALERÍA

Una breve selección de utilización de imágenes para interacción:

- *Discrete figures* Manabe [[Accedido Marzo 2019](#)]

- *Más que la cara* Lieberman [Accedido Marzo 2019]
- *Messa di voce* Golan Levin [Accedido Marzo 2019]
- *My little piece or privacy* Roy [Accedido Marzo 2019]
- *Starfield* Lab212 [Accedido Marzo 2019]

6.5. TAREA

Proponer un concepto y su prototipo de combinación de salida gráfica en respuesta a una captura de vídeo. Una sugerencia puede ser inspirarse en trabajos ya existentes, como los de la galería.

La entrega se debe realizar a través del campus virtual, remitiendo una memoria o el enlace a un *screencast* que además de describir las decisiones adoptadas, identifique al remitente, las referencias y herramientas utilizadas, cuidando el formato y estilo. Como material adicional debe incluirse el enlace al código desarrollado (p.e. github), con su correspondiente *README*, y un gif animado, de al menos 5 segundos, que ilustre ejecución del prototipo.

BIBLIOGRAFÍA

Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheik. Realtime multi-person 2d pose estimation using part affinity fields. In *CVPR*, 2017.

Modesto Castrillón, Oscar Déniz, Daniel Hernández, and Javier Lorenzo. A comparison of face and facial feature detectors based on the violajones general object detection framework. *Machine Vision and Applications*, 22(3):481–494, 2011.

Bryan Chung. OpenCV 4.0.0 Java Built and CVImage library, Accedido Marzo 2019a. URL <http://www.magicandlove.com/blog/2018/11/22/opencv-4-0-0-java-built-and-cvimage-library/>.

Bryan Chung. Magic and love interactive, Accedido Marzo 2019b. URL <http://www.magicandlove.com/>.

Bryan Chung. Kinect for processing library, Accedido Marzo 2019c. URL <http://www.magicandlove.com/blog/research/kinect-for-processing-library/>.

- Zachary Lieberman Golan Levin. *Messa di voce*, Accedido Marzo 2019. URL <https://zkm.de/en/artwork/messa-di-voce>.
- Lab212. *Starfield*, Accedido Marzo 2019. URL <https://www.creativeapplications.net/openframeworks/starfield-by-lab212-interactive-galaxy-the-swing-and-kinect/>.
- Thomas Sanchez Lengeling. *Kinect v2 Processing library for Windows*, Accedido Marzo 2019. URL <http://codigogenerativo.com/kinectpv2/>.
- Zach Lieberman. *Más que la cara*, Accedido Marzo 2019. URL <https://medium.com/@zachlieberman/más-que-la-cara-overview-48331a0202c0>.
- Rainer Lienhart and Jochen Maydt. An extended set of Haar-like features for rapid object detection. In *IEEE ICIP 2002*, volume 1, pages 900–903, September 2002.
- Daito Manabe. *Discrete figures*, Accedido Marzo 2019. URL https://www.agolpedeefecto.com/teatro_2018/teatro-discrete-figures.html.
- OpenCV team. *OpenCV library*, Accedido Marzo 2019. URL <https://opencv.org/>.
- Niklas Roy. *My little piece of privacy*, Accedido Marzo 2019. URL <https://www.niklasroy.com/project/88/my-little-piece-of-privacy>.
- Daniel Shiffman. *Images and pixels*, Accedido Marzo 2019. URL <https://processing.org/tutorials/pixels/>.
- Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2010. URL <http://szeliski.org/Book/>.
- Paul Viola and Michael J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):151–173, May 2004.

Práctica 7

Introducción a la síntesis y procesamiento de audio

Como describe el tutorial disponible en la web de Processing [DuBois and Thoben \[Accedido Marzo 2019\]](#), han sido significativos los avances en reproducción y transmisión de la señal de sonido desde finales del siglo XIX.

El sonido se propaga como onda a través de un medio, comprimiendo y descomprimiendo la materia que encuentra a su paso. La cantidad de desplazamiento, refleja la amplitud del sonido, que en el mundo natural se corresponde con la combinación de diversas componentes discretas. Medir dicho desplazamiento define la intensidad del sonido, percibiendo el oído humano frecuencias entre 20 y 20000 Hz. El audio es la interpretación por un sistema del sonido. El esquema de digitalización más frecuente se denomina PCM (*pulse-code modulation*), aplicando un muestreo, que determina la mayor frecuencia medible, además de una resolución numérica, un número de bits, del valor registrado.

En esta práctica se describen algunas nociones básicas para la síntesis y análisis de sonido con Processing.

7.1. SÍNTESIS

La carga de ficheros de audio, tipo wav o aiff, se trató en la sección [1.4.4](#), haciendo uso de la biblioteca *Sound*¹, que además de las posibilidades descritas a continuación, permite llevar a cabo una configuración global.

La biblioteca *Sound* cuenta con varias posibilidades de creación de osciladores simples basados en patrones de ondas sinusoidales, pulsos, etc., que permiten producir ondas de

¹<https://processing.org/reference/libraries/sound/index.html>

sonido *puras*. El listado 7.1 crea un oscilador por repetición de pulsos. Se repite un tono constante, por la repetición continua de la onda a una determinada frecuencia.

Listado 7.1: Oscilador de pulso

```
//Carga biblioteca
import processing.sound.*;

Pulse pulso;

void setup() {
  size(600, 400);
  background(255);

  // Crea un oscilador de tipo pulso
  pulso = new Pulse(this);

  //Lanza el oscilador
  pulso.play();
}

void draw() {
}
```

El ejemplo del listado 7.1 lanza un oscilador con la amplitud por defecto. Además de la creación y lanzamiento del oscilador, las variables de tipo *Pulse* cuentan con métodos para definir no sólo su amplitud, sino también su ancho, en el caso del tipo *Pulse*, frecuencia, etc. Utilizando la función *map* que mapea el valor en un rango de una variable en otro, mostramos varios ejemplos de uso de los métodos *amp*, *freq* y *width* en los listados 7.2, 7.3 y 7.4.

Listado 7.2: Oscilador de pulso con control de volumen

```
import processing.sound.*;

Pulse pulso;

void setup() {
  size(600, 400);
  background(255);

  // Crea un oscilador de tipo pulso
  pulso = new Pulse(this);

  //Lanza el oscilador
  pulso.play();
}

void draw() {
  //Ajusta amplitud en función de la posición y del puntero
  pulso.amp(map(mouseY,0, height,0,1));
}
```


Listado 7.3: Oscilador con control de volumen y frecuencia

```
import processing.sound.*;

Pulse pulso;

void setup() {
  size(600, 400);
  background(255);

  // Crea un oscilador de tipo pulso
  pulso = new Pulse(this);

  //Lanza el oscilador
  pulso.play();
}

void draw() {
  //Ajusta amplitud en función de la posición y del puntero
  pulso.amp(map(mouseY,0,height,0,1));

  //Ajusta la frecuencia en función de la posición x del puntero
  pulso.freq(map(mouseX, 0, width, 20.0, 500.0));
}
```

Listado 7.4: Oscilador con control de frecuencia y ancho del pulso

```
import processing.sound.*;

Pulse pulso;

void setup() {
  size(600, 400);
  background(255);

  // Crea un oscilador de tipo pulso
  pulso = new Pulse(this);

  //Lanza el oscilador
  pulso.play();
}

void draw() {
  //Ajusta el ancho en función de la posición y del puntero
  pulso.width(map(mouseY,0,height,0,1));

  //Ajusta la frecuencia en función de la posición x del puntero
  pulso.freq(map(mouseX, 0, width, 20.0, 500.0));
}
```

Otros osciladores con similar repertorio de métodos (excepto para el ancho del pulso) son *SawOsc*, *SqrOsc*, *TriOsc* y *SinOsc*. El listado 7.5 permite alternar entre ellos con las teclas del cursor, con similares alteraciones de la amplitud y frecuencia como en el ejemplo previo

para el oscilador basado en pulsos.

Listado 7.5: Varios osciladores con variación de frecuencia y volumen

```
import processing.sound.*;
Pulse pulso;
SinOsc sinu;
SawOsc sier;
SqrOsc cuad;
TriOsc tria;

int tipo=1;

void setup() {
  size(600, 400);
  background(255);

  // Crea los osciladores
  pulso = new Pulse(this);
  sinu = new SinOsc(this);
  sier = new SawOsc(this);
  cuad = new SqrOsc(this);
  tria = new TriOsc(this);

  //Inicialmente comienza con la de tipo pulso
  //Lanza el oscilador
  pulso.play();
}

void draw() {
  background(255);

  switch (tipo)
  {
    case 1:
      //Ajusta el volumen en función de la posición y del puntero
      pulso.amp(map(mouseY,0,height,0,1));
      //Ajusta la frecuencia en función de la posición x del puntero
      pulso.freq(map(mouseX, 0, width, 20.0, 500.0));
      break;
    case 2:
      //Ajusta el volumen en función de la posición y del puntero
      sinu.amp(map(mouseY,0,height,0,1));
      //Ajusta la frecuencia en función de la posición x del puntero
      sinu.freq(map(mouseX, 0, width, 20.0, 500.0));
      break;
    case 3:
      //Ajusta el volumen en función de la posición y del puntero
      sier.amp(map(mouseY,0,height,0,1));
      //Ajusta la frecuencia en función de la posición x del puntero
      sier.freq(map(mouseX, 0, width, 20.0, 500.0));
      break;
    case 4:
```

```
//Ajusta el volumen en función de la posición y del puntero
cuad.amp(map(mouseY,0,height,0,1));
//Ajusta la frecuencia en función de la posición x del puntero
cuad.freq(map(mouseX, 0, width, 20.0, 500.0));
break;
case 5:
//Ajusta el volumen en función de la posición y del puntero
tria.amp(map(mouseY,0,height,0,1));
//Ajusta la frecuencia en función de la posición x del puntero
tria.freq(map(mouseX, 0, width, 20.0, 500.0));
break;
default:
break;
}
//
ellipse(mouseX,mouseY,map(mouseX, 0, width, 1.0, 50.0), map(mouseY,0,height,0,50));
}

void keyPressed() {
  if (key == CODED) {
    if (keyCode == UP || keyCode == DOWN)
    {
      //Detiene oscilador anteriormente activo
      switch (tipo)
      {
        case 1:
          pulso.stop();
          break;
        case 2:
          sinu.stop();
          break;
        case 3:
          sier.stop();
          break;
        case 4:
          cuad.stop();
          break;
        case 5:
          tria.stop();
          break;
        default:
          break;
      }

      if (keyCode == UP) {
        tipo=tipo+1;
        if (tipo>5) tipo=1;
      }
      else {
        tipo=tipo-1;
        if (tipo<1) tipo=5;
      }
    }
  }
}
```

```
//Lanza nuevo oscilador
switch (tipo)
{
  case 1:
    pulso.play();
    println("PULSO");
    break;
  case 2:
    sinu.play();
    println("SINUSOIDAL");
    break;
  case 3:
    sier.play();
    println("SIERRA");
    break;
  case 4:
    cuad.play();
    println("CUADRADA");
    break;
  case 5:
    tria.play();
    println("TRIANGULAR");
    break;
  default:
    break;
}
}
```

La combinación de osciladores básicos permite obtener ondas sonoras de mayor complejidad. El listado 7.6 combina hasta cinco osciladores sinusoidales. A partir de la primera frecuencia, mapeada con la posición en x del puntero, se incorporan con las teclas del cursor osciladores que doblan frecuencia, a mitad de amplitud.

Listado 7.6: Composición de ondas sinusoidales a distintas frecuencias y amplitud

```
import processing.sound.*;

SinOsc[] ondas;

int nondas=1;
int maxondas=5;

void setup() {
  size(500, 100);
  background(255);

  // Crea los osciladores
  ondas = new SinOsc[maxondas];
```

```
for (int i = 0; i < maxondas; i++)
{
    // Osciladores sinusoidales
    ondas[i] = new SinOsc(this);
    //Inicialmente lanza únicamente el primero
    if (i==0)
    {
        ondas[i].play();
        //frecuencia y volumen de el primero
        ondas[i].freq(20);
        ondas[i].amp(0.5);
    }
}
//Muestra el número de osciladores
println(nondas);
}

void draw() {
    background(255);

    //Frecuencia de la menor, relacionada con la posición del ratón
    float freq0 = map(mouseX, 0, width, 20.0, 500.0);

    for (int i = 0; i < nondas; i++)
    {
        //Frecuencia doble que la previa
        ondas[i].freq(freq0 * pow(2,i));
        //Volumen total no debe superar 1.0
        //A mayor frecuencia, asociamos menor volumen, mitad que frecuencia anterior
        ondas[i].amp((1.0 / pow(2,i+1)));
    }
}

//Incluye o elimina osciladores
void keyPressed() {
    if (key == CODED) {
        if (keyCode == UP || keyCode == DOWN)
        {
            if (keyCode == UP) {
                //Actualiza el número de osciladores
                nondas=nondas+1;
                //Controla no salirse de los límites
                if (nondas>maxondas)
                    nondas=maxondas;
                else // si no se ha salido lanza la siguiente frecuencia más alta
                    ondas[nondas-1].play();
            } else {
                //Detiene la más alta activa
                if (nondas>0) ondas[nondas-1].stop();
                //Actualiza el número de osciladores
                nondas=nondas-1;
                //Controla no salirse de los límites
```

```
        if (nondas<1)
            nondas=0;
        }
        //Muestra el número de osciladores
        println(nondas);
    }
}
```

La envolvente de un señal oscilatoria se utiliza para delimitar sus valores extremos. El listado 7.7 lanza un oscilador con su envolvente, que limitada en el tiempo reproduce un sonido al realizar clic con el ratón. Observar la diferencia comentando la envolvente. Se definen los tiempos de subida, sostenido y bajada, además del volumen del sostenido.

Listado 7.7: Oscilador sinusoidal con envolvente

```
import processing.sound.*;

SinOsc osc;
Env env;

float tsubida = 0.001;
float tsostenido = 0.004;
float vsostenido = 0.5;
float tbajada = 0.4;

void setup() {
    size(640, 360);
    background(255);

    // Oscilador sinusoidal
    osc = new SinOsc(this);

    // Envolvente
    env = new Env(this);
}

void draw() {
}

void mousePressed() {
    osc.play();
    env.play(osc, tsubida, tsostenido, vsostenido, tbajada);
}
```

En base al ejemplo precedente, el listado 7.8 divide la ventana en trece zonas lanzando una envolvente asociada con un oscilador sinusoidal, asociado a notas MIDI [Wikipedia](#) [Accedido Marzo 2019], que previamente son convertidas en frecuencias.

Listado 7.8: Teclado con envolventes

```
import processing.sound.*;

SinOsc osc;
Env env;

// Notas MIDI
int[] midiSequence = { 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72 };

//Envolvente
float tsubida = 0.001;
float tsostenido = 0.004;
float vsostenido = 0.5;
float tbajada = 0.4;

void setup() {
  size(650, 100);
  background(255);

  // Oscilador sinusoidal
  osc = new SinOsc(this);

  // Envolvente
  env = new Env(this);

  noStroke();
  fill(0);
}

void draw() {
  //Dibujamos las celdas/teclas
  for (int i=0;i<6;i++){
    rect(i*100+50,0,50,100);
  }
}

void mousePressed() {
  //Nota en función del valore de mouseX
  int tecla=(int)(mouseX/50);
  println(tecla);

  osc.play(midiToFreq(midiSequence[tecla]), 0.5);
  env.play(osc, tsubida, tsostenido, vsostenido, tbajada);
}

// Conversor de nota MIDI A frecuencia, del ejemplo Envelopes de la biblioteca Sound
float midiToFreq(int nota) {
  return (pow(2, ((nota-69)/12.0))) * 440;
}
```

7.2. ANÁLISIS

Para ilustrar brevemente posibilidades de análisis de la señal sonora, se analiza la amplitud y frecuencias presentes en la señal capturada. Como primer paso, el listado 7.9 hace uso de las utilidades de la biblioteca *Sound* para la captura de sonido, modificando el ancho del rectángulo dibujado en base a la amplitud de la señal de entrada.

Listado 7.9: Volumen de la señal de audio

```
import processing.sound.*;

AudioIn IN;
Amplitude nivel;

void setup() {
  size(500, 100);
  background(255);

  // Entrada de audio, toma primer canal
  IN = new AudioIn(this, 0);

  //Lanza captura
  IN.start();

  // Analizador de amplitud
  nivel = new Amplitude(this);

  // Asocia entrada y analizador
  nivel.input(IN);

  noStroke();
  //Tono de relleno con transparencia
  fill(255,0,0,50);
}

void draw() {
  background(255);

  //Obtiene valor entre 0 y 1 en base al nivel
  float volumen = nivel.analyze();

  //Asocia ancho de rectángulo al nivel del volumen
  int ancho = int(map(volumen, 0, 1, 1, 500));
  rect(0,0,ancho,100);
}
```

Las frecuencias presentes en una señal sonora se obtienen a partir de la Transformada rápida de Fourier (FFT). El listado 7.10 muestra tanto los cambios de amplitud como de frecuencias de la señal de entrada.

Listado 7.10: Amplitud y frecuencias de la señal de audio


```
import processing.sound.*;

AudioIn IN;
FFT fft;
Amplitude nivel;
int bandas=512;
float [] spectrum = new float [bandas];

void setup() {
  size(512, 200);
  background(255);

  // Entreda de audioo, toma primer canal
  IN = new AudioIn(this, 0);

  //Lanza captura
  IN.start();

  // Analizador de amplitud
  nivel = new Amplitude(this);

  //Analizador frecuencias
  fft = new FFT(this, bandas);

  // Asocia entrada y analizadores
  nivel.input(IN);
  fft.input(IN);

  fill(255,0,0,50);
}

void draw() {
  background(255);

  //Nivel
  //Obtiene valor entre 0 y 1 en base al nivel
  float volumen = nivel.analyze();
  //Asocia ancho de rectángulo al nivel del volumen
  int ancho = int(map(volumen, 0, 1, 1, width));

  pushStyle();
  noStroke();
  rect(0,0,ancho,height/2);
  popStyle();

  //FFT
  fft.analyze(spectrum);

  for(int i = 0; i < bandas; i++){
    // Resultado de FFT normalizado
    // Línea por banda de frecuencia, considerando amplitud hasta 5
    line(i, height, i, height - spectrum[i]*height/2*5 );
  }
}
```

```
}  
}
```

7.3. MINIM

Los ejemplos anteriores se basan exclusivamente en la biblioteca *Sound*, que cuenta con más ejemplos a través de *Archivo->Ejemplos->Bibliotecas principales->Sound*. Sin embargo, existen otras bibliotecas, destacamos las posibilidades que ofrece *Minim Compartmental* [[Accedido Marzo 2019](#)], como evidencia la extensa galería de ejemplos disponible tras su instalación. Para empezar, visualizamos las ondas de la señal de entrada en el ejemplo del listado 7.11.

Listado 7.11: Visualiza la señal de entrada

```
//Carga biblioteca  
import ddf.minim.*;  
  
Minim minim;  
  
//Entrada  
AudioInput IN;  
  
void setup() {  
  size(500, 200);  
  background(255);  
  
  minim = new Minim(this);  
  
  // Línea estéreo de entrada, 44100 Hz 16 bits  
  IN = minim.getLineIn(Minim.STEREO, 2048);  
}  
  
void draw() {  
  background(0);  
  stroke(255);  
  // Dibuja ondas  
  // Valores entre -1 y 1, se escalan y desplazan  
  for(int i = 0; i < IN.left.size()-1; i++)  
  {  
    line(i, height/2 + IN.left.get(i)*height/2, i+1, height/2 + IN.left.get(i+1)*height/2);  
    line(i, 3*height/2 + IN.right.get(i)*height/2, i+1, 3*height/2 + IN.right.get(i+1)*height/2);  
  }  
}  
r  
void stop()  
{  
  //Cerrar Minim antes de finalizar  
  IN.close();  
}
```

```

    minim.stop();
    super.stop();
}

```

Minim permite grabar la señal de entrada. El listado 7.12 se basa en el ejemplo *Basics->RecordAudioInput* permitiendo definir el intervalo de grabación, y salvarla una vez durante la visualización.

Listado 7.12: Grabación de micrófono

```

//Basado en ejemplo de Minim Basics->RecordAudioInput
import ddf.minim.*;
import ddf.minim.ugens.*;

Minim minim;

//Entrada
AudioInput IN;
//Grabación
AudioRecorder recorder;
boolean recorded;
//Reproducción
AudioOutput OUT;
FilePlayer player;

void setup() {
  size(500, 200);
  background(255);

  minim = new Minim(this);

  // Línea estéreo de entrada, 44100 Hz 16 bits
  IN = minim.getLineIn(Minim.STEREO, 2048);

  // Define el nombre del archivo a salvar
  recorder = minim.createRecorder(IN, "sonido.wav");

  // Canal de salida para la reproducción
  OUT = minim.getLineOut( Minim.STEREO );
}

void draw() {
  background(0);
  stroke(255);
  // Dibuja ondas
  // Valores entre -1 y 1, se escalan y desplazan
  for(int i = 0; i < IN.left.size()-1; i++)
  {
    line(i, height/2 + IN.left.get(i)*height/2, i+1, height/2 + IN.left.get(i+1)*height/2);
    line(i, 3*height/2 + IN.right.get(i)*height/2, i+1, 3*height/2 + IN.right.get(i+1)*height/2);
  }

  if ( recorder.isRecording() )

```

```
{
    text("Grabando, 'pulsar r para detener", 5, 15);
}
else
{
    if ( !recorded )
    {
        text("Pulsar r para grabar", 5, 15);
    }
}
}

void keyReleased()
{
    if ( key == 'r' && !recorded )
    {
        if ( recorder.isRecording() )
        {
            recorder.endRecord();
            recorded=true;
            //Salva y reproduce
            recorder.save();
            if ( player != null )
            {
                player.unpatch( OUT );
                player.close();
            }
            player = new FilePlayer( recorder.save() );
            player.patch( OUT );
            player.play();
        }
        else
        {
            recorder.beginRecord();
        }
    }
}

void stop()
{
    //Cerrar Minim antes de finalizar
    IN.close();
    if ( player != null )
    {
        player.close();
    }
    minim.stop();

    super.stop();
}
```

7.3.1. Efectos

Como se comenta anteriormente, *Sound* ofrece algunas posibilidades de efectos y análisis, si bien *Minim* ofrece más flexibilidad. El ejemplo del listado 7.13 aplica filtros paso bajo o alto, a elección del usuario, a la señal de audio previamente capturada.

Listado 7.13: Registra y repite sonido aplicando filtrado de frecuencias

```
import ddf.minim.*;
import ddf.minim.effects.*;
import ddf.minim.ugens.*;

Minim minim;

//Entrada
AudioInput IN;
//Grabación
AudioRecorder recorder;
boolean recorded;
//Reproducción
AudioOutput OUT;
FilePlayer player;

// Filtros
LowPassSP lpf;
HighPassSP hpf;

int tipofiltro=1;
int maxfiltros=2;

void setup() {
  size(500, 200);
  background(255);

  minim = new Minim(this);

  // Línea estéreo de entrada, 44100 Hz 16 bits
  IN = minim.getLineIn(Minim.STEREO, 2048);

  // Define el nombre del archivo a salvar
  recorder = minim.createRecorder(IN, "sonido.wav");

  // Canal de salida para la reproducción
  OUT = minim.getLineOut( Minim.STEREO );
}

void draw() {
  background(0);
  stroke(255);
  // Dibuja ondas
  // Valores entre -1 y 1, se escalan y desplazan
  if (!recorded)
  {
```

```
for(int i = 0; i < IN.left.size()-1; i++)
{
    line(i, height/2 + IN.left.get(i)*height/2, i+1, height/2 + IN.left.get(i+1)*height/2);
    line(i, 3*height/2 + IN.right.get(i)*height/2, i+1, 3*height/2 + IN.right.get(i+1)*height/2);
}
}
else
{
    for(int i = 0; i < OUT.left.size()-1; i++)
    {
        line(i, height/2 + OUT.left.get(i)*height/2, i+1, height/2 + OUT.left.get(i+1)*height/2);
        line(i, 3*height/2 + OUT.right.get(i)*height/2, i+1, 3*height/2 + OUT.right.get(i+1)*height/2);
    }
}

if ( recorder.isRecording() )
{
    text("Grabando, 'pulsar r para detener", 5, 15);
}
else
{
    if ( !recorded )
    {
        text("Pulsar r para grabar", 5, 15);
    }
    else
    {
        text("Mueve el ratón para filtrar , cursor para cambiar de filtro", 5, 15);
    }
}
}

void keyReleased()
{
    if ( key == 'r' && !recorded )
    {
        if ( recorder.isRecording() )
        {
            recorder.endRecord();
            recorded=true;
            //Salva y reproduce
            recorder.save();
            if ( player != null )
            {
                player.unpatch( OUT );
                player.close();
            }
            player = new FilePlayer( recorder.save() );

            //Creación de filtros
            lpf = new LowPassSP(100, player.sampleRate());
            hpf = new HighPassSP(1000, player.sampleRate());

            //Asocia filtro por defecto
```

```
player.patch(lpf).patch( OUT );

//En bucle
player.loop();
}
else
{
    recorder.beginRecord();
}
}

//Escoge filtro a aplicar
if (key == CODED) {
    if (keyCode == UP || keyCode == DOWN)
    {
        switch (tipofiltro)
        {
            case 1:
                player.unpatch(lpf);
                break;
            case 2:
                player.unpatch(hpf);
                break;
            default:
                break;
        }

        if (keyCode == UP) {

            //Actualiza el filtro
            tipofiltro=tipofiltro+1;
            //Controla no salirse de los límites
            if (tipofiltro>maxfiltros)
                tipofiltro=1;

        } else {
            //Actualiza el número de osciladores
            tipofiltro=tipofiltro-1;
            //Controla no salirse de los límites
            if (tipofiltro<1)
                tipofiltro=maxfiltros;
        }

        switch (tipofiltro)
        {
            case 1:
                player.patch(lpf).patch(OUT);
                break;
            case 2:
                player.patch(hpf).patch(OUT);
                break;
            default:
                break;
        }
    }
}
```

```

        //Muestra el tipo de filtro
        println(tipofiltro);
    }
}

void mouseMoved()
{
    if (recorded)
    {
        float cutoff;

        // Mapea puntero a rango de frecuencias en base a cada filtro
        switch (tipofiltro){
            case 1:
                cutoff = map(mouseX, 0, width, 60, 2000);
                lpf.setFreq(cutoff);
                break;
            case 2:
                cutoff = map(mouseX, 0, width, 1000, 14000);
                hpf.setFreq(cutoff);
            default:

                break;

        }
    }
}

void stop()
{
    //Cerrar Minim antes de finalizar
    IN.close();
    if ( player != null )
    {
        player.close();
    }
    minim.stop();

    super.stop();
}

```

Muchas más opciones pueden explicarse a través de la batería de ejemplos de *Minim*. Como ejemplo final se presenta una adaptación del ejemplo *Basics->CreateAnInstrument* en el listado 7.14, que reproduce notas musicales a través de un *teclado*. La nomenclatura de especificación de las notas se ha tomado del siguiente [enlace²](https://es.wikipedia.org/wiki/Frecuencias_de_afinaci%C3%B3n_del_piano).

Listado 7.14: Teclado con notas musicales

```

//Basado en el ejemplo de Minim CreateAnInstrument
import ddf.minim.*;

```

²https://es.wikipedia.org/wiki/Frecuencias_de_afinaci%C3%B3n_del_piano


```
import ddf.minim.ugens.*;

Minim minim;
AudioOutput out;

//Notas musicales en notación anglosajona
String [] notas={"A3", "B3", "C4", "D4", "E4", "F4", "G4", "A4", "B4", "C5", "D5", "E5", "F5"};

// Clase que describe la interfaz del instrumento, idéntica al ejemplo
//Modificar para nuevos instrumentos
class SineInstrument implements Instrument
{
    Oscil wave;
    Line ampEnv;

    SineInstrument( float frequency )
    {
        // Oscilador sinusoidal con envolvente
        wave = new Oscil( frequency, 0, Waves.SINE );
        ampEnv = new Line();
        ampEnv.patch( wave.amplitude );
    }

    // Secuenciador de notas
    void noteOn( float duration )
    {
        // Amplitud de la envolvente
        ampEnv.activate( duration, 0.5f, 0 );
        // asocia el oscilador a la salida
        wave.patch( out );
    }

    // Final de la nota
    void noteOff()
    {
        wave.unpatch( out );
    }
}

void setup()
{
    size(650, 100);

    minim = new Minim(this);

    // Línea de salida
    out = minim.getLineOut();
}

void draw() {
    //Dibujamos las celdas/teclas
    for (int i=0;i<6;i++){
        rect(i*100+50,0,50,100);
    }
}
```

```
}  
  
void mousePressed() {  
    //Nota en función del valor de mouseX  
    int tecla=(int)(mouseX/50);  
    println(tecla);  
  
    //Primeros dos parámetros, tiempo y duración  
    out.playNote( 0.0, 0.9, new SineInstrument( Frequency.ofPitch( notas[tecla] ).asHz() ) );  
}
```

7.4. TAREA

Realizar una propuesta de prototipo integrando al menos gráficos y síntesis de sonido. Se acepta la modificación de alguna de las prácticas precedentes.

La entrega se debe realizar a través del campus virtual, remitiendo una memoria o el enlace a un *screencast* que además de describir las decisiones adoptadas, identifique al remitente, las referencias y herramientas utilizadas, cuidando el formato y estilo. Como material adicional debe incluirse el enlace al código desarrollado (p.e. github), con su correspondiente *README*.

BIBLIOGRAFÍA

Compartmental. Minim library, Accedido Marzo 2019. URL <http://code.compartmental.net/tools/minim/>.

R. Luke DuBois and Wilm Thoben. Sound, Accedido Marzo 2019. URL <https://processing.org/tutorials/sound/>.

Wikipedia. MIDI, Accedido Marzo 2019. URL <https://es.wikipedia.org/wiki/MIDI>.

Práctica 8

Introducción a p5.js

8.1. P5.JS

p5.js McCarthy [Accedido Marzo 2019] es una biblioteca JavaScript que comparte con Processing el objetivo de proporcionar herramientas de programación para fines creativos, con la salvedad de que su concepción y realización se orientan a la web. Como cambio de concepto, no se limita a posibilitar el dibujo sobre el lienzo, sino que considera todo el navegador, permitiendo interactuar con otros objetos HTML5. A diferencia de Processing.js Resig [Accedido Marzo 2019] que es un puerto JavaScript para Processing, p5.js no es un puerto sino una nueva interpretación.

Trabajar con p5.js *offline* además de descargar la biblioteca, requiere un editor, un servidor web, y un navegador. Dicha opción es probablemente la mejor opción para código de cierta envergadura. Sin embargo, para los ejemplos básicos que se presentan a continuación, se propone utilizar un editor *online*, como el proporcionado en la web de p5.js¹, o a través de OpenProcessing². Para diversas operaciones, como almacenar y descargar el trabajo realizado, será necesario registrarse.

Abrir el editor nos presenta un esqueleto de programación en modo continuo (también es posible el modo básico) con las funciones *setup* y *draw*, si bien probablemente observes dos diferencias: 1) para los métodos se utiliza *function* en lugar de *void*, y 2) al definir el tamaño de la zona de dibujo, la función *size* se sustituye por *createCanvas*. Como punto de partida para descubrir otras variaciones, se adaptan alguno de los ejemplos de las prácticas previas realizados en Processing.

El listado 8.1 se basa en el listado 1.26 dibujando líneas aleatorias desde la posición del

¹<https://editor.p5js.org>

²<https://www.openprocessing.org/sketch/create>

puntero. Además de los ejemplos proporcionados en la web de p5.js³, gracias a Stefano Presti, están disponibles algunos más con similitudes a los presentados en la primera práctica [Presti \[Accedido Marzo 2019\]](#).

Listado 8.1: Dibujo de líneas aleatorias desde el puntero

```
function setup() {
  createCanvas(400, 400);
  background(0);
}

function draw() {
  background(220);
  stroke(0, random(255), 0);
  line(mouseX, mouseY, random(width), random(height));
}
```

También la definición de variables difiere, ya que en estos ejemplos se hace uso de la palabra reservada *var*. El listado 8.2 adapta el código de 1.32, mostrando además la función *print* como la alternativa a *println* para la salida de texto.

Listado 8.2: Desplazamiento del círculo

```
var Radio = 50;
var cuenta = 0;

function setup()
{
  createCanvas(400,400);
  background(0);
}

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

Para finalizar con los ejemplos básicos, el listado 8.3 desplaza el círculo modificando su tamaño de forma aleatoria.

Listado 8.3: Desplazamiento del círculo con *latido*

```
var cirx=0;

function setup() {
```

³<https://p5js.org/es/examples/>

```
createCanvas(400,400);
noStroke();
}

function draw() {
  background(0);

  var cirtam=50;

  if (random(10)>9) {
    cirtam=60;
  }

  fill(193,255,62);
  ellipse(cirx,50,cirtam,cirtam);
  cirx=cirx+1;
  if(cirx>400){
    cirx=0;
  }
}
```

8.1.1. Eventos

El manejo de eventos de ratón y teclado tiene un comportamiento similar. Para teclado se activa la variable *keyIsPressed*, pudiendo obtener información de la tecla pulsada a través de la variable *keyCode*. El listado 8.4 ilustra en un ejemplo mínimo el modo de mostrar el código ASCII de una tecla pulsada, o detectar si es una tecla de control.

Listado 8.4: Detección de eventos de teclado

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  if ( keyIsPressed ) {

    if (keyCode != UP_ARROW && keyCode != DOWN_ARROW) {
      var x=keyCode;
      text(key+" ASCII "+ x , 20 , 20 );
    }
    else{
      if ( keyCode == UP_ARROW) text(key, 20 , 20 );
      else
        text(key, 20 , 20 );
    }
  }
}
```

Para el ratón, el acceso a la posición del puntero está integrado en el listado 8.1, sin evidenciar cambios. El manejo de eventos permite hacer uso de las funciones *mousePressed*, *mouseMoved*, *mouseDragged* o la variable *mouseIsPressed*. Como ejemplo sencillo, el listado 8.5 modifica su comportamiento si está pulsado el botón del ratón.

Listado 8.5: Variando el comportamiento con el ratón

```
var dragX, dragY, moveX, moveY;

function setup() {
  createCanvas(400, 400);
  smooth();
  noStroke();
}

function draw() {
  background(220);
  fill(0);
  ellipse(dragX, dragY, 30, 30);
  fill(150);
  ellipse(moveX, moveY, 30, 30);
}

function mouseMoved() {
  moveX=mouseX;
  moveY=mouseY;
}

function mouseDragged() {
  dragX=mouseX;
  dragY=mouseY;
}
```

8.1.2. 3D

Los gráficos en tres dimensiones presentan variaciones. Por una parte, la llamada a *createCanvas* añade *WEBGL* como modo de reproducción, y destacan las funciones para manejo de matrices de transformación, que pasan a denominarse *push* y *pop*. El listado 8.6 adapta el ejemplo previo mostrado en el listado 3.12 para dibujar un *planeta* con *satélite*. Observar que no requiere el traslado inicial de coordenadas al centro del lienzo.

Listado 8.6: Dibujando una esfera levemente rotada con movimiento

```
var ang;
var angS;

function setup() {
  createCanvas(400, 400, WEBGL);
  stroke(0);
}
```

```
// Inicializa
ang=0;
angS=0;
}

function draw() {
  background(200);

  // Esfera
  rotateX(radians(-45));

  // Planeta
  push();
  rotateY(radians(ang));
  sphere(100);
  pop();

  // Resetea tras giro completo
  ang=ang+0.25;
  if (ang>360)
    ang=0;

  // Objeto
  push();
  rotateZ(radians(angS));
  translate(-width*0.3,0,0);
  box(10);
  pop();

  // Resetea tras giro completo
  angS=angS+0.25;
  if (angS>360)
    angS=0;
}
```

Además de las primitivas 3D disponibles, pueden definirse formas con el par *beginShape/endShape*. El listado 8.7 adapta el listado previo 2.8 para dibujar una pirámide. Comparar ambos listados evidencia diversas diferencias.

Listado 8.7: Dibujando una pirámide

```
function setup() {
  createCanvas(400, 400,WEBGL);
}

function draw() {
  background(220);

  translate(mouseX-width/2, mouseY-height/2);
  beginShape();
  noFill();
  // Puntos de la forma
```

```

vertex(-100, -100, -100);
vertex( 100, -100, -100);
vertex(  0,   0,  100);

vertex( 100, -100, -100);
vertex( 100,  100, -100);
vertex(  0,   0,  100);

vertex( 100, 100, -100);
vertex(-100, 100, -100);
vertex(  0,   0,  100);

vertex(-100, 100, -100);
vertex(-100, -100, -100);
vertex(  0,   0,  100);
endShape();
}

```

La ejecución advierte que no se ha indicado la textura de la forma 3D. Integramos la textura en el listado 8.8 requiriendo la precarga de la imagen, además de su adición previa al proyecto a través de *Sketch->Add file*.

Listado 8.8: Dibujando una pirámide con textura

```

var img;

function preload() {
  img=loadImage ('logoulpgc.png') ;
}

function setup() {
  createCanvas(400, 400, WEBGL);
  //Asignamos coordenadas de tetxura en rango 0,1
  textureMode(NORMAL);
}

function draw() {
  background(0);

  translate(mouseX - width / 2, mouseY - height / 2);

  //Asigna textura
  texture(img);
  beginShape(TRIANGLES);
  // Vértices de la forma con coodenadas de textura
  vertex(-100, -100, -100,0,0);
  vertex( 100, -100, -100,1,0);
  vertex(  0,   0,  100,1,1);

  vertex( 100, -100, -100,1,0);
  vertex( 100,  100, -100,0,0);
  vertex(  0,   0,  100,1,1);
}

```



```

vertex( 100, 100, -100,0,0);
vertex(-100, 100, -100,1,0);
vertex( 0, 0, 100,1,1);

vertex(-100, 100, -100,1,0);
vertex(-100, -100, -100,0,0);
vertex( 0, 0, 100,1,1);
endShape();
}

```

La gestión de proyecciones y cámara recuerda a Processing, al igual que las luces. Se sugiere acudir a los ejemplos para detalles concretos⁴.

Es también posible crear más de un lienzo con la función *createGraphics*. Esto permite dibujar en lienzos fuera de la pantalla y su posterior utilización, por ejemplo como textura en el listado 8.9.

Listado 8.9: Dibujando una textura en un lienzo fuera de pantalla

```

var lienzo;

function setup() {
  createCanvas(400, 400,WEBGL);
  //Creamos lienzo
  graphics= createGraphics(100 ,100) ;
  graphics.background(255) ;
}

function draw() {
  background(220);

  //Dibuja círculos de color aleatorio en el segundo lienzo en base a la posición del ratón
  graphics.fill( random (0,255) , random (0,255) , random (0,255) ) ;
  graphics.ellipse (100*mouseX/width , 100*mouseY/height , 20 ) ;
  //Rotación del cubo
  rotateX(frameCount*0.03 ) ;
  rotateY(frameCount*0.03 ) ;
  rotateZ(frameCount*0.03 ) ;
  //Asignamos textura al cubo
  texture( graphics ) ;
  box (100) ;
}

```

8.1.3. Imágenes

Como se ha observado con las texturas en disco, la carga de imágenes requiere usar la función *preload*, que se llama antes de *setup*. En el caso de utilizar el editor web, previamente

⁴<https://p5js.org/es/examples/>

debe subirse la imagen al proyecto, como ocurre con cualquier tipo de datos, a través del menú *Sketch->Add File*. Un ejemplo mínimo se presenta en el listado 8.10.

Listado 8.10: Dibujo de líneas aleatorias desde el puntero

```
function preload( ) {
  img=loadImage ("basu.png" );
}

function setup() {
  createCanvas(400, 400);
}

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

8.1.4. Sonido

De forma similar, la reproducción de sonido requiere la precarga del contenido. El listado 8.11, adapta el código del listado 1.58 a p5.js.

Listado 8.11: Pelota con sonido de rebote

```
var pos=0;
var mov=5;
var sonido;

function preload( ) {
  sonido = loadSound("Bass-Drum-1.wav");
}

function setup() {
  createCanvas(400, 400);
}

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

  pos=pos+mov;

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

8.1.5. Cámara

El listado 8.12 replica un ejemplo básico de acceso a píxeles. Al ejecutar solicita permiso para acceder a la cámara, siendo costoso con el editor en línea, que además requiere añadir `//noproduct` antes del bucle para evitar el control de bucle infinito.

Listado 8.12: Captura de cámara

```
var capture;

function setup() {
  createCanvas(400,400) ;
  capture = createCapture(VIDEO) ;
}

function draw() {
  var dim=capture.width*capture.height;
  loadPixels( ) ;

  //noproduct
  for (var i =1; i<dim/2 ; i++){
    var suma= capture.get( i ) ;
    if ( suma<255*1.5){
      capture.set(i ,color(0,0,0));
    }
    else{
      capture.set( i , color(255,255,255)) ;
    }
  }
  updatePixels( ) ;
  capture.show( ) ;
  image(capture,0,0 ) ;
}
```

8.1.6. No todo es lienzo

Bibliotecas como *p5.dom* y *p5.sound* hacen posible la creación e interacción con otros elementos HTML. Si bien queda fuera de los objetivos de la práctica, se han mostrado usos básicos de imagen, sonido y vídeo. Otras bibliotecas disponibles pueden consultarse en la web de p5.js⁵.

⁵<https://p5js.org/es/libraries/>

8.2. OTRAS HERRAMIENTAS

La creación de contenidos con fines creativos en la web muestra actualmente una creciente actividad. Para contenidos gráficos animados mencionar `three.js`⁶. En el ámbito de la visión por computador, la conocida `OpenCV` cuenta para programación web con `OpenCV.js`⁷, si bien el material disponible está orientado a `OpenCV 3.x`, cuando ya la versión de la biblioteca es la 4.1.0.

El buen momento del aprendizaje automático ha dado pie a diversas iniciativas para facilitar el acceso a herramientas y recursos. Destacamos `ml4a` Kogan [Accedido Marzo 2019], que surge con el propósito de proporcionar recursos educativos gratuitos sobre aprendizaje automático, analizando herramientas disponibles como `ml5js` NYU ITP [Accedido Marzo 2019], `tensorflow.js` Google [Accedido Marzo 2019], `Wekinator` Fiebrink [Accedido Marzo 2019] y la próxima aparición como la aún en versión beta `Runway` Valenzuela [Accedido Marzo 2019].

8.3. TAREA

Crear un *Paint* utilizando `p5.js` que permita como mínimo modificar el color y grosor del pincel. La entrega se debe realizar a través del campus virtual, remitiendo una memoria o el enlace a un *screencast* que además de describir las decisiones adoptadas, identifique al remitente, las referencias y herramientas utilizadas, cuidando el formato y estilo. Como material adicional debe incluirse el enlace al código desarrollado (p.e. github), con su correspondiente *README*.

BIBLIOGRAFÍA

Rebecca Fiebrink. `Wekinator`, Accedido Marzo 2019. URL <http://www.wekinator.org/>.

Google. `Tensorflow.js`, Accedido Marzo 2019. URL <https://www.tensorflow.org/js>.

Gene Kogan. `Machine Learning for Artists`, Accedido Marzo 2019. URL <http://ml4a.github.io/>.

Lauren McCarthy. `p5.js`, Accedido Marzo 2019. URL <https://p5js.org>.

⁶<https://threejs.org/>

⁷https://docs.opencv.org/3.4/d5/d10/tutorial_js_root.html

NYU ITP. Friendly machine learning for the web, Accedido Marzo 2019. URL <https://ml5js.org>.

Stefano Presti. Introduction to p5.js programming examples, Accedido Marzo 2019. URL <https://github.com/steffopresto/p5.js->.

John Resig. processing.js, Accedido Marzo 2019. URL <https://processingjs.org>.

Cristóbal Valenzuela. Artificial intelligence for augmented creativity, Accedido Marzo 2019. URL <https://runwayapp.ai>.

Práctica 9

Introducción a la programación con Arduino

9.1. ARDUINO

Arduino [Arduino community \[Accedido Marzo 2019b\]](#) es una plataforma abierta de hardware/software para facilitar el desarrollo de proyectos que impliquen interacción física con el entorno.

9.1.1. Hardware

El elemento hardware característico es una tarjeta controladora con capacidades de entrada/salida a nivel de puertos analógicos, digitales y comunicaciones seriales.

La unidad básica disponible inicialmente fue el Arduino UNO [Arduino community \[Accedido Marzo 2019d\]](#), que se muestra en la figura 9.1. Se trata de una tarjeta microcontroladora basada en el chip ATmega328P (RISC, 8 bit, flash memory 32 Kb, 1 Kb EEPROM) y que cuenta con los siguientes elementos:

- Microcontrolador
- 14 pines digitales input/output (6 configurables como salidas PWM)
- 6 entradas analógicas
- Oscilador de 16 MHz
- Conexión USB

- Otros: interfaz serial, alimentación, reset, ICSP

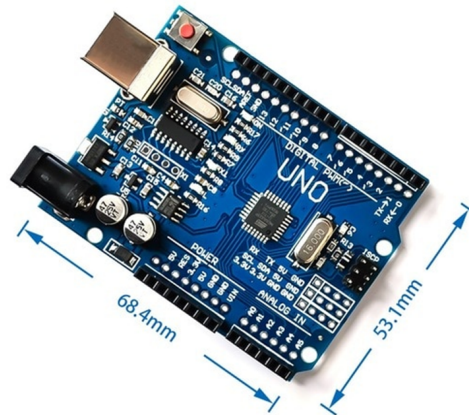


Figura 9.1: Tarjeta Arduino UNO.

La familia de tarjetas disponibles ha ido creciendo paulatinamente, de manera que ahora es posible acceder a multitud de alternativas, con una amplia variedad de prestaciones y características [Arduino community](#) [Accedido Marzo 2019a].

- 101
- Gemma
- LilyPad, LilyPad SimpleSnap, LilyPad USB
- Mega 2560, Mega ADK
- MKR1000, MKRZero
- Pro, Pro Mini
- Uno, Zero, Due
- Esplora
- Ethernet
- Leonardo
- Mini, Micro, Nano
- Yún

- Arduino Robot

Junto a estos componentes básicos, existen elementos auxiliares que permiten expandir el sistema con nuevas opciones de comunicación e interacción con una amplia gama de dispositivos sensores/actuadores. Por ejemplo, es posible añadir tarjetas auxiliares para disponer de conexión WiFi, buses industriales, control de motores, etc.

9.1.2. Software

La programación de la tarjeta puede realizarse desde diferentes entornos de desarrollo. Las dos opciones más habituales son la utilización del Arduino Desktop IDE y la versión online. En la figura 9.2 se muestra el aspecto del entorno de desarrollo en su versión *Desktop*.

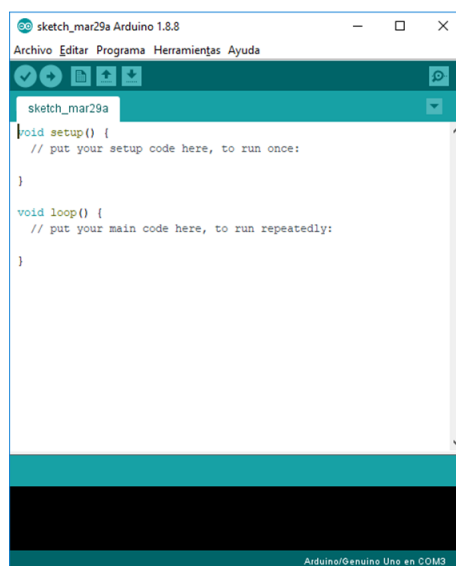


Figura 9.2: IDE Arduino.

El lenguaje de programación estándar es C/C++, aunque existe la posibilidad de utilizar otros lenguajes como Python, Java, Lisp, etc. A nivel de comunicación, es posible interactuar con la tarjeta desde cualquier lenguaje, puesto que simplemente será necesario enviar/recibir datos a través del puerto serial.

En la página [Arduino community](#) [Accedido Marzo 2019c] pueden encontrar la referencia a las instrucciones y funciones disponibles en C/C++.

9.1.3. Instalación

La instalación por defecto requiere permisos de administración, y simplemente precisa descargar el fichero adecuado y seguir los pasos indicados. También es posible realizar una instalación a nivel de usuario, aunque en ese caso los *drivers* tienen que instalarse separadamente.

Una vez conectada la tarjeta a través del cable USB, en el IDE debe aparecer identificada con su tipo y número de puerto serial.

9.2. PROGRAMACIÓN

Un programa en Arduino se denomina *sketch* (bosquejo, esquema), y consta de dos funciones principales:

- *setup()* de inicialización, que se ejecuta una única vez al lanzar el programa
- *loop()* de procesamiento, que se ejecuta por defecto de forma repetitiva.

El ejemplo del listado 9.1 puede considerarse el "Hola Mundo" de Arduino. El código cambia el estado del LED integrado en la tarjeta alternativamente entre niveles alto y bajo, consiguiendo un efecto de parpadeo con un periodo de un segundo.

Listado 9.1: Ejemplo de LED parpadeante

```
/*
  This example code is in the public domain.
  http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000); // wait for a second
  digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW
  delay(1000); // wait for a second
}
```

Para ejecutar el código deberá realizarse la compilación del código y la transferencia a la tarjeta previamente enlazada.

9.3. ALGUNOS RECURSOS ÚTILES

9.3.1. Control del tiempo

Un aspecto importante de la interacción con el usuario es el control del tiempo. En Arduino disponemos de una serie de funciones que permiten medir el tiempo transcurrido y definir pausas en la ejecución.

- *delay()*
- *delayMicroseconds()*
- *micros()*
- *millis()*

9.3.2. Interrupciones

En determinadas aplicaciones en las que un programa debe responder con rapidez a un evento determinado, puede ser necesario programar interrupciones. De esta manera se minimiza el riesgo de que algún cambio de estado no sea detectado por el código.

En el ejemplo del listado 9.2 se configura una función como rutina de servicio para atender el cambio de estado de un pin de entrada de la tarjeta.

Listado 9.2: Ejemplo de programación de interrupciones

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}

void loop() {
  digitalWrite(ledPin, state);
}

void blink() {
  state = !state;
}
```

9.3.3. Funciones matemáticas

Operaciones matemáticas básicas:

- *abs()*
- *constrain()*
- *map()*
- *max(), min()*
- *pow(), sq(), sqrt()*

Trigonometría:

- *cos()*
- *sin()*
- *tan()*

9.3.4. Generación de números aleatorios

En Arduino se puede acceder a un generador de números aleatorios a través de las siguientes funciones:

- *random()*
- *randomSeed()*

9.3.5. Procesamiento de texto

- *Characters*
- *isAlpha(), isAlphaNumeric()*
- *isAscii()*
- *isControl()*
- *isDigit(), isHexadecimalDigit()*
- *isGraph()*

- *isLowerCase(), isUpperCase()*
- *isPrintable()*
- *isPunct()*
- *isSpace(), isWhitespace()*

9.4. ALGUNAS FUENTES/EJEMPLOS ADICIONALES

- *Arduino motor party* *Levin* [Accedido Mayo 2019]
- *Openframeworks and arduino* *OF* [Accedido Mayo 2019]
- *IoT Moné* [Accedido Mayo 2019]

9.5. TAREA

Programar el Arduino de manera que se genere una pulsación de frecuencia variable en el LED integrado en la placa. Deberá producirse una señal senoidal que definirá la envolvente, de manera que cuando dicha señal alcance su valor máximo el LED parpadeará a una cierta frecuencia *freqMax*, mientras que cuando alcance el valor mínimo parpadeará a una frecuencia mínima *freqMin*.

La entrega se debe realizar a través del campus virtual, remitiendo una memoria o el enlace a un *screencast* que además de describir las decisiones adoptadas, identifique al remitente, las referencias y herramientas utilizadas, cuidando el formato y estilo. Como material adicional debe incluirse el enlace al código desarrollado (p.e. github), con su correspondiente *README*.

BIBLIOGRAFÍA

Arduino community. Arduino cards, Accedido Marzo 2019a. URL <https://www.arduino.cc/en/Products/Compare>.

Arduino community. Arduino official site, Accedido Marzo 2019b. URL <https://www.arduino.cc/>.

Arduino community. Arduino language reference, Accedido Marzo 2019c. URL <https://www.arduino.cc/reference/en/>.

Arduino community. Arduino uno, Accedido Marzo 2019d. URL <https://store.arduino.cc/arduino-uno-rev3>.

Golan Levin. Arduino motor party, Accedido Mayo 2019. URL <http://cmuems.com/2018/60212f/daily-notes/oct-12/arduino/>.

Lesla Moné. Iot devices, sensors, and actuators explained, Accedido Mayo 2019. URL <https://blog.leanix.net/en/iot-devices-sensors-and-actuators-explained>.

OF. Openframeworks and arduino, Accedido Mayo 2019. URL <http://openframeworks.cc/documentation/communication/ofArduino/>.

Práctica 10

Arduino y Processing

10.1. COMUNICACIONES

La forma más simple de interactuar con Arduino es a través de las funciones de comunicación serial. El conjunto de funciones que se pueden utilizar es el siguiente:

- *if(Serial)*
- *available(), availableForWrite()*
- *begin(), end()*
- *find(), findUntil()*
- *flush()*
- *parseFloat(), parseInt()*
- *peek()*
- *print(), println()*
- *read(), readBytes(), readBytesUntil(), readString(), readStringUntil()*
- *setTimeout()*
- *write()*
- *serialEvent()*

El listado 10.1 muestra el envío de un mensaje simple a través del puerto serial. El resultado se puede visualizar utilizando el monitor serial que proporciona el IDE de Arduino.

Listado 10.1: Ejemplo de envío de mensaje por el puerto serial con Arduino

```
int n;

void setup()
{
  //initialize serial communications at a 9600 baud rate
  Serial.begin(9600);
}
void loop()
{
  n = n+1;

  //send 'Hello n' over the serial port
  Serial.print("Hello ");
  Serial.println(n);
  //wait 1 second
  delay(1000);
}
```

10.2. LECTURA DE SENSORES EN ARDUINO

10.2.1. Conversión analógica/digital

La tarjeta dispone de entradas en voltaje que pueden ser convertidas a valores digitales por medio de un conversor A/D de 10 bits. También es posible generar valores analógicos de salida en forma de salida PWM (ciclo de trabajo variable). Las funciones que permiten operar con estas señales son las siguientes:

- *analogRead()*
- *analogReference()*
- *analogWrite()*

En el listado 10.2 se presenta un ejemplo de utilización de estas funciones para modificar la intensidad de un LED dependiendo del valor fijado en un divisor de tensión.

Listado 10.2: Ejemplo de manejo de señales analógicas

```
int ledPin = 9;    // LED connected to digital pin 9
```



```
int analogPin = 3; // potentiometer connected to analog pin 3
int val = 0;      // variable to store the read value

void setup() {
  pinMode(ledPin, OUTPUT); // sets the pin as output
}

void loop() {
  val = analogRead(analogPin); // read the input pin
  analogWrite(ledPin, val / 4); // analogRead values go from 0 to 1023, analogWrite values from 0 to 255
}
```

10.2.2. Sensor de luz

Puede construirse un esquema similar sustituyendo el potenciómetro por una fotorresistencia en serie con una resistencia limitadora fija. El divisor de tensión resultante permite medir la cantidad de luz recibida.

10.2.3. Sensor de distancia

Cualquier sensor de distancia que proporcione una salida analógica se puede integrar con facilidad. La salida debe estar dentro del rango de voltajes que admite el conversor analógico/digital del Arduino (0-5v).

10.2.4. Giróscopos, acelerómetros, magnetómetros

En el caso de sensores más sofisticados, es necesario utilizar librerías específicas.

10.3. COMUNICACIÓN ENTRE ARDUINO Y PROCESSING

Pueden encontrarse diversos tutoriales [Sparkfun \[Accedido Abril 2019\]](#) que indican cómo establecer una comunicación a través del puerto serial entre Arduino y Processing.

Partiendo del ejemplo [10.1](#) anterior, el código para recibir el mensaje con Processing sería el que se recoge en el listado [10.3](#).

Listado 10.3: Ejemplo de recepción de mensaje por el puerto serial con Processing

```
import processing.serial.*;

Serial myPort; // Create object from Serial class
String val;    // Data received from the serial port
```

```
void setup()
{
  String portName = Serial.list()[0]; //change the 0 to a 1 or 2 etc. to match your port
  myPort = new Serial(this, portName, 9600);
}

void draw()
{
  if ( myPort.available() > 0)
  { // If data is available ,
    val = myPort.readStringUntil('\n'); // read it and store it in val
  }
  println(val); //print it out in the console
}
```

La comunicación en sentido inverso se puede establecer tal y como se recoge en los ejemplos [10.4](#) y [10.5](#).

Listado 10.4: Ejemplo de envío de mensaje por el puerto serial con Processing

```
import processing.serial.*;

Serial myPort; // Create object from Serial class

void setup()
{
  size(200,200); //make our canvas 200 x 200 pixels big
  String portName = Serial.list()[0]; //change the 0 to a 1 or 2 etc. to match your port
  myPort = new Serial(this, portName, 9600);
}

void draw() {
  if (mousePressed == true)
  { //if we clicked in the window
    myPort.write('1'); //send a 1
    println("1");
  } else
  { //otherwise
    myPort.write('0'); //send a 0
  }
}
```

Listado 10.5: Ejemplo de recepción de mensaje por el puerto serial con Arduino

```
char val; // Data received from the serial port
int ledPin = 13; // Set the pin to digital I/O 13

void setup() {
  pinMode(ledPin, OUTPUT); // Set pin as OUTPUT
  Serial.begin(9600); // Start serial communication at 9600 bps
}
```

```
void loop() {
  if (Serial.available())
  { // If data is available to read,
    val = Serial.read(); // read it and store it in val
  }
  if (val == '1')
  { // If 1 was received
    digitalWrite(ledPin, HIGH); // turn the LED on
  } else {
    digitalWrite(ledPin, LOW); // otherwise turn it off
  }
  delay(10); // Wait 10 milliseconds for next reading
}
```

10.4. ALGUNAS FUENTES/EJEMPLOS

- *Face tracking and Arduino* [Aswinth \[Accedido Mayo 2019\]](#), [Barragán and Reas \[Accedido Mayo 2019\]](#)
- *Arduino and Processing* [Playground Arduino \[Accedido Mayo 2019\]](#), [Ben \[Accedido Mayo 2019\]](#)

10.5. TAREA

Programar una interfaz que utilice la información de distancia suministrada por el sensor Sharp GP2D12, leída a través del Arduino, para controlar el movimiento del juego Pong implementado con Processing. Debe ponerse especial cuidado en el conexionado de cada cable del sensor de distancia a las señales que correspondan en la tarjeta: rojo = 5v, negro = GND y amarillo = AI0.

La entrega se debe realizar a través del campus virtual, remitiendo una memoria o el enlace a un *screencast* que además de describir las decisiones adoptadas, identifique al remitente, las referencias y herramientas utilizadas, cuidando el formato y estilo. Como material adicional debe incluirse el enlace al código desarrollado (p.e. github), con su correspondiente *README*.

BIBLIOGRAFÍA

Raj Aswinth. Real time face detection and tracking robot using arduino, Accedido Mayo 2019. URL <https://forum.processing.org/two/discussion/23461/real-time-face-detection-and-tracking-robot-using-arduino>.

Hernando Barragán and Casey Reas. Electronics, Accedido Mayo 2019. URL <https://processing.org/tutorials/electronics/>.

Ben. Connecting arduino to processing, Accedido Mayo 2019. URL <https://learn.sparkfun.com/tutorials/connecting-arduino-to-processing>.

Playground Arduino. Arduino and processing, Accedido Mayo 2019. URL <https://playground.arduino.cc/Interfacing/Processing/>.

Sparkfun. Tutorial communication arduino-processing, Accedido Abril 2019. URL <https://learn.sparkfun.com/tutorials/connecting-arduino-to-processing/introduction>.