

CREANDO INTERFACES DE USUARIO

GUIÓN DE PRÁCTICAS
2019/20

18 de septiembre de 2020

Modesto Castrillón Santana, 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	14
1.3.3. Tipos de datos	15
1.3.4. Repeticiones	17
1.3.5. Dibujar un tablero de ajedrez	18
1.4. Programación en modo continuo	21
1.4.1. El bucle infinito de ejecución	21
1.4.2. Control	26
1.4.3. Interacción	33
1.4.3.1. Teclado	33
1.4.3.2. Ratón	36
1.4.4. Múltiples lienzos	40
1.4.5. Sonido	41
1.4.6. Exportando la pantalla	42
1.5. Otras referencias y fuentes	43
1.6. Tarea	44
2. Superficie de revolución	47
2.1. PShape	47
2.2. P3D	51
2.3. Sólido de revolución	56
2.3.1. Rotación de un punto 3D	57
2.4. Tarea	58

3. Transformaciones	61
3.1. Transformación básicas 2D	61
3.1.1. Traslación	63
3.1.2. Rotaciones	66
3.1.3. Escalado	68
3.2. Concatenación de transformaciones	68
3.3. Transformaciones básicas 3D	69
3.4. Objetos de archivo	73
3.5. Texto e imágenes	74
3.6. Tarea	76
4. Modelos cámara	79
4.1. Proyecciones	79
4.1.1. Ortográfica	80
4.1.2. Perspectiva	83
4.2. La cámara	85
4.3. Oclusión	89
4.4. Tarea	91
5. Iluminación y texturas	93
5.1. Iluminación	93
5.1.1. Luces	94
5.1.2. Materiales	101
5.2. Texturas	103
5.3. Shaders	106
5.4. Tarea	107
6. Procesamiento de imagen y vídeo	109
6.1. Imagen	109
6.2. Vídeo	111
6.3. Operaciones básicas	112
6.3.1. Píxeles	112
6.3.2. OpenCV	113
6.3.3. Grises	113
6.3.4. Umbralizado	115
6.3.5. Bordes	116
6.3.6. Diferencias	120

6.4. Detección	122
6.4.1. Caras	122
6.4.2. Personas	126
6.4.2.1. Kinect	126
6.4.2.2. Openpose	131
6.4.3. Manos	133
6.5. Galería	134
6.6. Tarea	134
7. Introducción a la síntesis y procesamiento de audio	137
7.1. Síntesis	137
7.2. Análisis	146
7.3. Minim	148
7.3.1. Efectos	151
7.4. Galería	156
7.5. Tarea	156
8. Introducción a p5.js	159
8.1. p5.js	159
8.1.1. Eventos	161
8.1.2. 3D	162
8.1.3. Imágenes	166
8.1.4. Sonido	166
8.1.5. Cámara	167
8.1.6. No todo es lienzo	167
8.2. Otras herramientas	168
8.3. Tarea	168
9. Introducción a los <i>shaders</i> en Processing	171
9.1. Introducción	171
9.2. <i>Shaders</i> de fragmentos	172
9.2.1. Hola mundo	173
9.2.2. Dibujando con algoritmos	176
9.2.3. Generativos	191
9.2.4. Imágenes	199
9.3. <i>Shaders</i> de vértices	203
9.4. Recursos adicionales	215

9.5. Tarea	215
10.Introducción a la programación con Arduino	217
10.1.Arduino	217
10.1.1. Hardware	217
10.1.2. Software	219
10.1.3. Instalación	220
10.2.Programación	220
10.3.Algunos recursos útiles	221
10.3.1. Control del tiempo	221
10.3.2. Interrupciones	221
10.3.3. Funciones matemáticas	222
10.3.4. Generación de números aleatorios	222
10.3.5. Procesamiento de texto	222
10.4.Algunas fuentes/ejemplos adicionales	223
10.5.Tarea	223
11.Arduino y Processing	225
11.1.Comunicaciones	225
11.2.Lectura de sensores en Arduino	226
11.2.1. Conversión analógica/digital	226
11.2.2. Sensor de luz	227
11.2.3. Sensor de distancia	227
11.2.4. Giróscopos, acelerómetros, magnetómetros	227
11.3.Comunicación entre Arduino y Processing	227
11.4.Algunas fuentes/ejemplos	229
11.5.Tarea [sin entrega]	229

Práctica 1

Introducción a Processing

1.1. INTRODUCCIÓN

Processing [Processing Foundation \[Accedido Enero 2020\]](#) es un proyecto de código abierto basado en el lenguaje Java, que tiene como objetivo facilitar cualquier desarrollo con fines creativos. Se concibe 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 de programación básica [Pazos \[Accedido Enero 2019\]](#), [Nyhoff and Nyhoff \[2017a\]](#), 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, en enero de 2020 última versión disponible es la 3.5.4, a través del mencionado [enlace](#), y descomprimir. Una vez instalado, al

¹<http://processing.org/>

lanzar la aplicación se presenta la interfaz del entorno de desarrollo de Processing (PDE), ver figura 1.1. En caso de querer modificar el lenguaje de la interfaz, puede escogerse a través del menú con *File* → *Preferences*.

El mencionado entorno de desarrollo, 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. Antes de comenzar a codificar, destacar que es posible acceder a diversos ejemplos a través de la barra de menú *Archivo* → *Ejemplos*. Cuando se ejecuta un programa, se abre en una nueva ventana denominada ventana de visualización (*display window*).

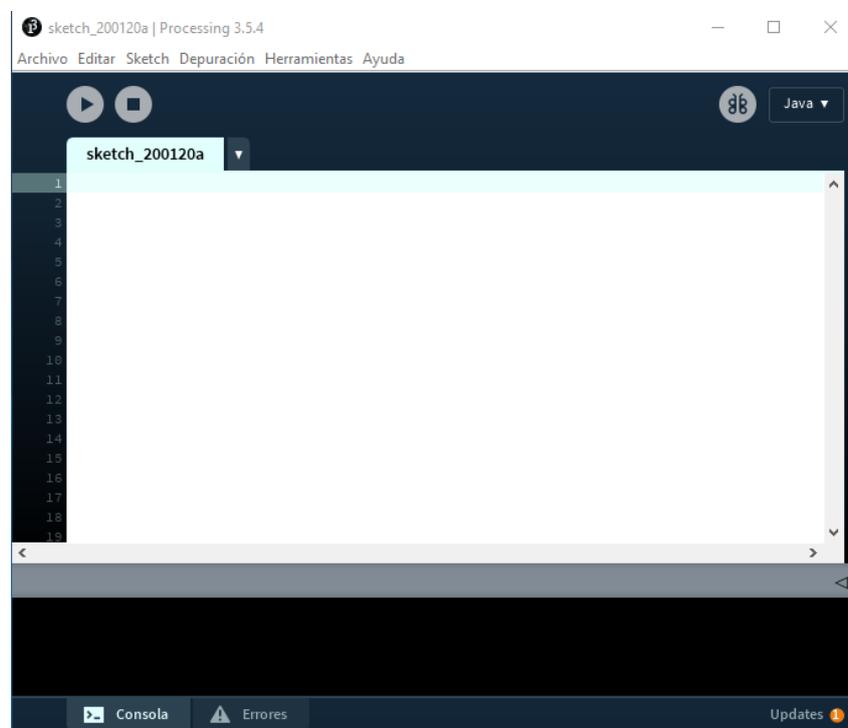


Figura 1.1: Imagen del entorno de programación, PDE, de Processing.

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: Los iconos de ejecución y parada.

	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 a continuación brevemente ambos. El modo básico permite la elaboración de imágenes estáticas, es decir que no se modifican. De forma sucinta, líneas simples de código tienen una representación directa en la ventana de visualización.

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.

Un siguiente ejemplo dibuja 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](#) puede practicarse con el espacio de color RGB (rojo, verde y azul) modificando

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

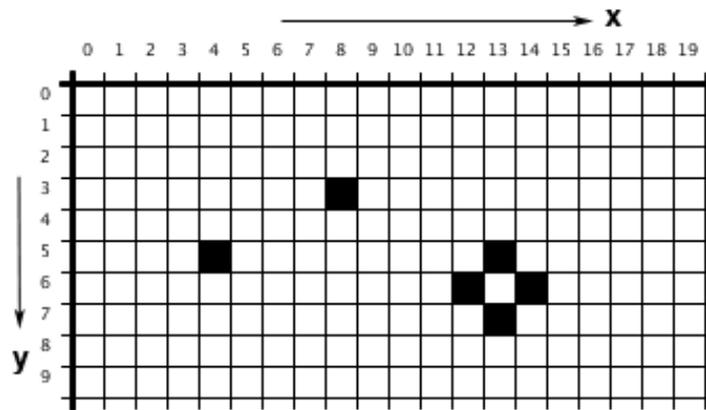


Figura 1.2: Sistema de coordenadas de la pantalla.

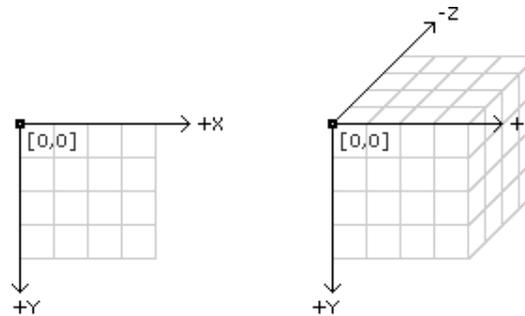


Figura 1.3: Sistema de coordenadas en 2 y 3 dimensiones (imagen de processing.org).

los valores de cada canal. RGB es el modelo de color por defecto, si bien puede adoptarse otro con *colormode*. El resultado de la ejecución de dicho código se muestra en la figura 1.4.

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 cuarto valor para especificar la transparencia del pincel a partir de dicho momento. Al contrario, si se expresara un único 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)*;

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 1.3. El valor entre paréntesis fija el valor máximo admisible. Cabe destacar en dicho listado 1.3 la especificación del tamaño de la ventana de visualización con el comando *size*.

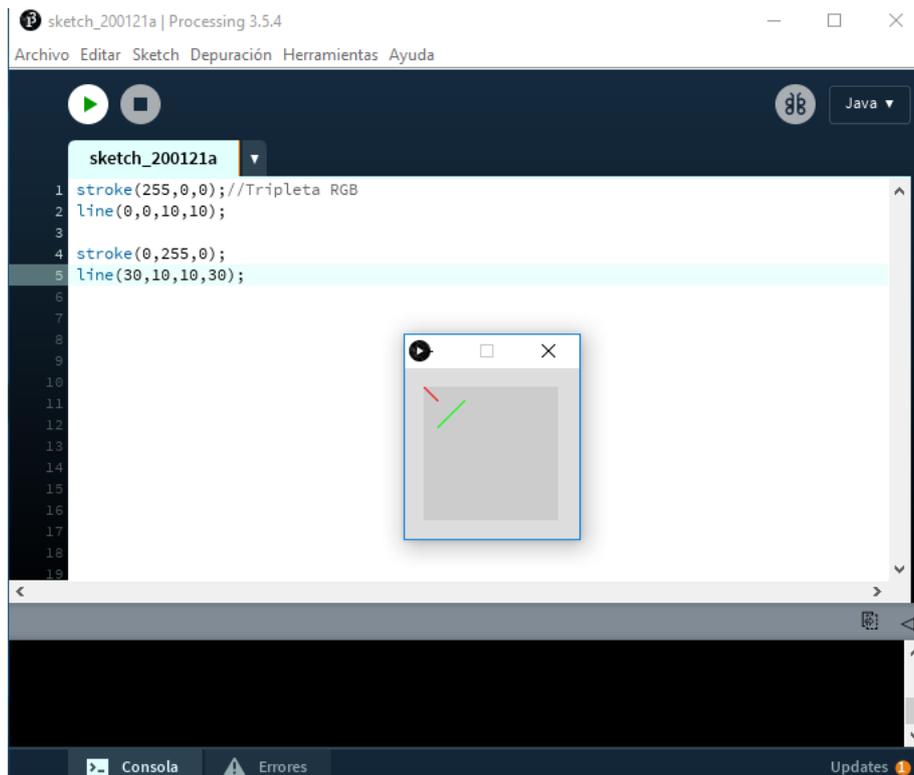


Figura 1.4: Entorno con el código del listado 1.2, y salida correspondiente.

Listado 1.3: 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 1.4 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 1.4: 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);
```

Sin embargo, como es habitual existen comandos que facilitan el dibujo de primitivas sencillas, el listado 1.5 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 1.5: 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 1.6. 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 1.6: Dibujo de una cuadrado sólido

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

A modo de resumen, el listado 1.7 muestra el uso de varias primitivas 2D, mostrando su salida en la figura 1.5.

Listado 1.7: Usando varias primitivas 2D

```
size (450,450);  
  
stroke (128);  
fill (128);  
  
ellipse (200,300,120,120); //Por defecto modo con coordenadas del centro y ejes  
  
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);
```

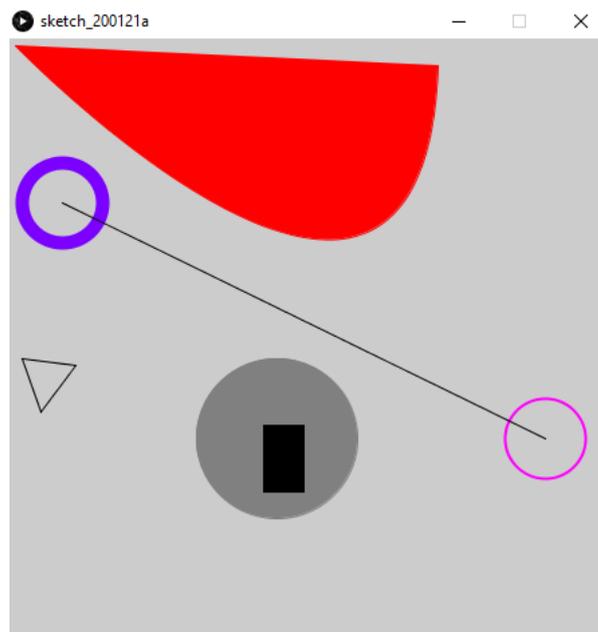


Figura 1.5: Ventada de visualización del listado 1.7.

Con los comandos ya conocidos, sería posible componer un dibujo estático combinando varias primitivas y colores (*rect*, *ellipse*, *line*, ...). Realmente no sería complicado reproducir el Mondrian de la figura 1.6.

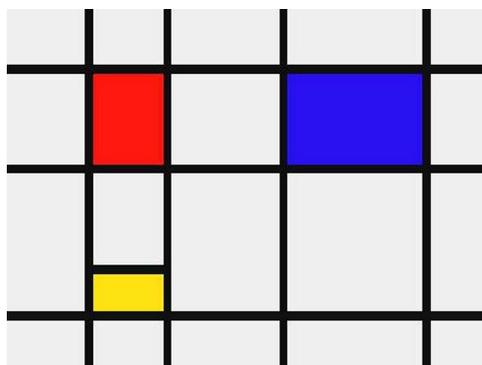


Figura 1.6: Piet Mondrian, *Composición con rojo, amarillo y azul* (1930).

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, almacenadas respectivamente en las variables *width* y *height*. Dichas variables se utilizan en el listado 1.8, para pintar una estrella simple que se coloca en el centro de la ventana, independientemente de las dimensiones fijadas con *size()*, ver figura 1.7.

Listado 1.8: 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);
```

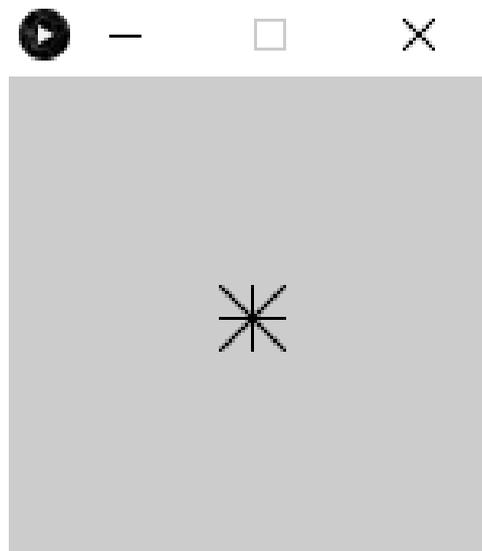


Figura 1.7: Ventana de visualización del listado 1.8 con unas dimensiones de la ventana 150 × 150.

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 1.9 utiliza la variable **I** para establecer el tamaño de la estrella.

Listado 1.9: Dibujo de una estrella variable

```
int l=10;

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

Para finalizar este apartado, el ejemplo del listado 1.10 dibuja una línea y un círculo de un determinado radio, haciendo uso de la función *ellipse* definiendo previamente el color de relleno.

Listado 1.10: 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. Varios tipos de variables se muestran en el listado 1.11. Tener presente que las variables se deben declarar explícitamente y asignarles valor antes de llamar o de realizar una operación con ellas.

Listado 1.11: 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 1.12 incluye ejemplos de accesos a vectores.

Listado 1.12: 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 1.13.

Listado 1.13: 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 1.14 crea una ventana de dimensiones 800×800 en la que pintamos líneas verticales de arriba a abajo separadas entre ellas 100 píxeles, ver figura 1.8. Recordar que la coordenada y de la parte superior es 0, y la inferior es 800 o *height* si usamos la variable correspondiente.

Listado 1.14: 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 1.15. 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 1.15: 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);
}
```

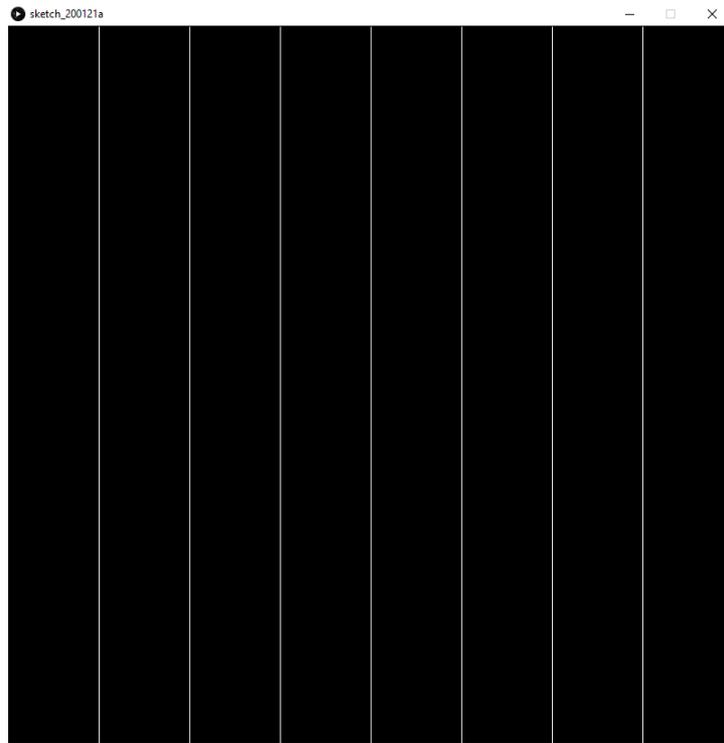


Figura 1.8: Ventana de visualización del listado 1.14.

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 1.9. El listado 1.16 fija el fondo a blanco, dibujando los cuatro recuadros negros de la primera fila del tablero.

Listado 1.16: 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 1.17 aplica un bucle *for* para pintar esas cuatro casillas negras.

Listado 1.17: Dibujo de las casillas de la primera fila con un bucle

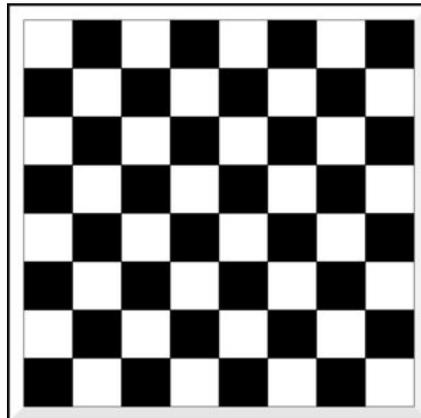


Figura 1.9: Rejilla estilo tablero de ajedrez.

```
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 1.18 da el salto para dibujar cuatro filas, cada una con su bucle particular.

Listado 1.18: 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 1.19 anida bucles, quedando más compacto.

Listado 1.19: 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 1.20.

Listado 1.20: 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 1.21.

Listado 1.21: 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.



Figura 1.10: Ventana de visualización del listado 1.22.

El ejemplo el listado 1.22 usa ambos métodos, delimitando con llaves las instrucciones asociadas a cada uno de ellos, ver figura 1.10. En concreto el método *setup* realiza la inicialización, fijando 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 veces por segundo.

Listado 1.22: 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 [1.23](#).

Listado 1.23: 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 [1.24](#) 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 1.24: 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 [1.25](#)

dibuja líneas desde un punto fijo, con el otro extremo aleatorio, variando también el color de forma aleatoria.

Listado 1.25: 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 1.26 se fuerza a que sean líneas aleatorias, pero verticales y blancas.

Listado 1.26: 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 1.27.

Listado 1.27: 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 1.28 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 1.28: 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 1.29 varía la tasa de refresco en base al número de ejecuciones y el evento de pulsado de un botón del ratón.

Listado 1.29: 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 1.30 ilustra un ejemplo de llamada a *redraw*.

Listado 1.30: 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 1.31 provoca el desplazamiento de un círculo por la ventana, ver figura 1.11.

Listado 1.31: Desplazamiento del círculo

```
int Radio = 50;
int cuenta = 0;

void setup()
{
  size(500,500);
  background(0);
}

void draw()
{
  background(0);
  stroke(175);
  fill(175);
  println("Iteraciones: " + cuenta);
  ellipse(20+cuenta, height/2, Radio, Radio);
  cuenta++;
}
```

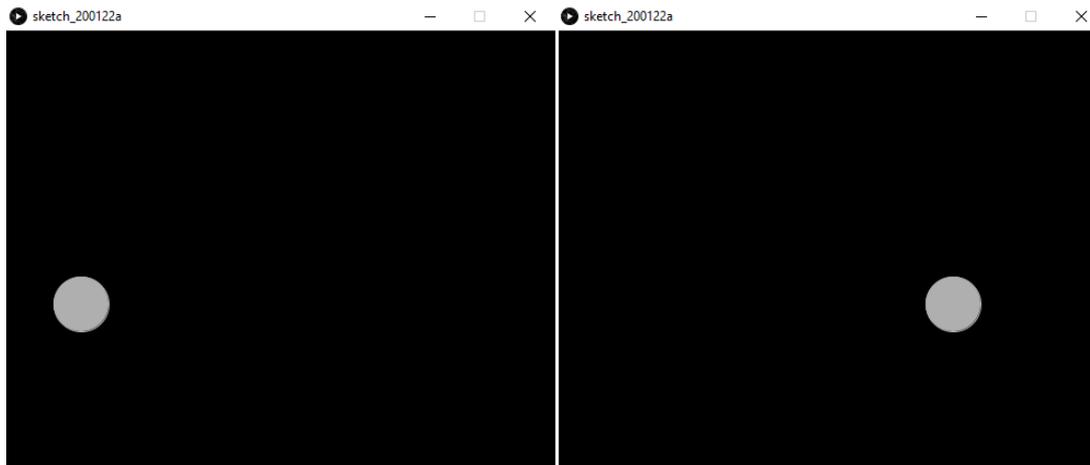


Figura 1.11: Dos instantes de la ventana de visualización del listado 1.31.

Finalizamos el apartado señalando que además de dibujar elementos geométricos, el listado 1.32 muestra que es posible escribir texto.

Listado 1.32: 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 1.31 se desplaza un círculo por la ventana haciendo uso de una variable. El listado 1.33 presenta una variante algo más reducida.

Listado 1.33: 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 1.34 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 1.34: 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 1.35 integra dos círculos a distintas velocidades.

Listado 1.35: 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 1.36 modifica la forma del círculo más lento de forma aleatoria cuando se dan ciertas circunstancias.

Listado 1.36: 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 1.37 lo modifica con una aparición más ajustada.

Listado 1.37: 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 1.38 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 1.38: 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 1.39 integra un *jugador*, que se desplaza en vertical acompañando al ratón, y también puede alterar el movimiento del círculo cuando haya choque, ver figura 1.12.

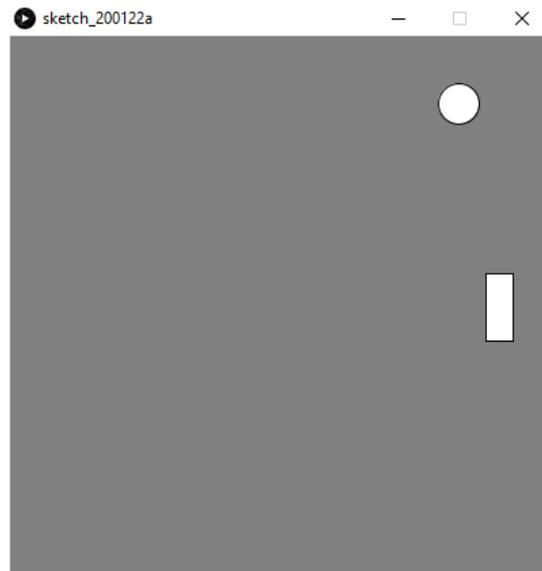


Figura 1.12: Ventana de visualización del listado 1.39.

Listado 1.39: 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 1.40 introducimos una variable contador, inicialmente a cero, que modifica su valor cada vez que haya un choque con la pared de la derecha.

Listado 1.40: 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 1.41.

Listado 1.41: 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 1.42 lo integra, manteniendo el rebote en las paredes, ahora también inferior y superior.

Listado 1.42: 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 1.43 hace uso de dicha variable para desplazar una línea.

Listado 1.43: 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 1.44 se muestra la tecla pulsada.

Listado 1.44: 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 1.45.

Listado 1.45: 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 1.46 utiliza las teclas de las flechas cambie la posición de una figura dentro del lienzo.

Listado 1.46: 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 1.47 realiza una acción al pulsar la tecla t.

Listado 1.47: 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 1.48.

Listado 1.48: 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 1.49 que modifica el color del fondo.

Listado 1.49: 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 1.50 emplea el evento de pulsado para pasar del modo básico al continuo.

Listado 1.50: 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 1.51.

Listado 1.51: 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 1.52.

Listado 1.52: 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 1.53.

Listado 1.53: 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 1.54.

Listado 1.54: Pintado cuando se pulsa

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

  background(128);
}

void draw() {

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

El listado 1.55 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 1.55: 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 1.56 el radio del pincel dependa del valor de x o y del ratón.

Listado 1.56: 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. Múltiples lienzos

En los ejemplos previos se trabaja con un único lienzo, la función *createGraphics*, permite crear un área de dibujo que no sea la pantalla, y que posteriormente pueda ser mostrada en la ventana de visualización. El listado 1.57, crea una ventana de 400 × 400 píxeles, integrando en ella otra zona de 100 × 100 píxeles en la que se dibujan círculos de color aleatorio en el segundo lienzo en base a la posición del ratón. Dicho lienzo se coloca en una zona de la ventana de visualización. ¿Te atreves a probar una posición no fija para su visualización?

Listado 1.57: Dibujando una textura en un lienzo fuera de pantalla

```
PGraphics lienzo;

void setup() {
  size(400, 400);
  //Creamos lienzo
  lienzo= createGraphics(100 ,100);
  lienzo.beginDraw();
  lienzo.background(100);
  lienzo.endDraw();
}

void draw() {
  background(220);

  //Dibuja círculos de color aleatorio en el segundo lienzo en base a la posición del ratón
  lienzo.beginDraw();
  lienzo.fill( random (0,255) , random (0,255) , random (0,255) );
  lienzo.ellipse(100*mouseX/width , 100*mouseY/height , 20, 20 );
}
```

```
lienzo.endDraw();  
  
image(lienzo, 200, 200);  
  
}  
}
```

1.4.5. 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 en formato *wav* es el [enlace](#)³. 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");  
}  
  
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 dependiendo de 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.

³<http://freewavesamples.com/>

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 ( ) ;
}
```

1.4.6. Exportando la pantalla

Para salvar la pantalla en cada iteración, 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");
}
```

Si interesara exportar a un archivo con formato gif animado, en versiones previas de Processing eras necesario previamente instalar *GifAnimation* a través del menú *Herramientas* > *Añadir herramientas* y buscando *gif* en la pestaña *Libraries*. El listado 1.61 presenta un pequeño ejemplo que salva lo que ocurra en pantalla hasta que pulsemos un botón del ratón. Sin embargo, para versiones más recientes suele ser necesario realizar la instalación manualmente, más información en este [enlace](#)⁴.

Listado 1.61: Exportando un gif animado

```
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

Reiterar 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

⁴<https://github.com/extrapixel/gif-animation>

a la programación con Processing Pazos [Accedido Enero 2019], como también lo hace el libro Nyhoff and Nyhoff [2017b]. 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 2019]. 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. Tener también en cuenta los recursos educativos a través del portal de la *Processing Foundation* Processing Foundation.

1.6. TAREA

Realizar de forma individual, un juego similar al Pong para dos jugadores, ver figura 1.13. La propuesta realizada debe incluir al menos: rebote, marcador, sonido, movimiento inicial aleatorio, admitiendo aportaciones propias de cada estudiante.

La entrega se debe realizar a través del campus virtual, remitiendo un enlace a un proyecto github, cuyo README sirva de memoria, por lo que se espera que el README:

- identifique al autor,
- describa el trabajo realizado,
- argumente decisiones adoptadas para la solución propuesta,
- incluya referencias y herramientas utilizadas,
- muestre el resultado con un gif animado.

Se sugiere cuidar el formato y estilo, una referencia interesante puede ser este enlace⁵.

BIBLIOGRAFÍA

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

⁵https://www.fast.ai/2020/01/20/blog_overview/

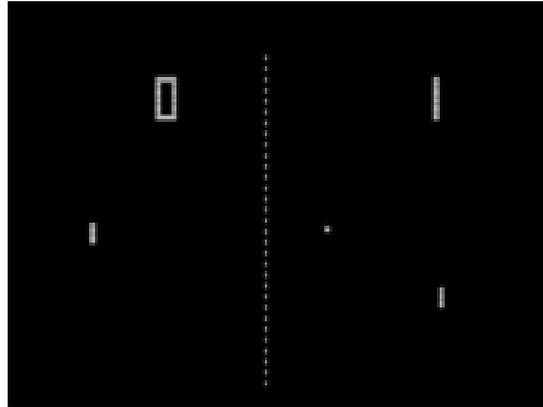


Figura 1.13: Interfaz clásica de Pong⁶.

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

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 Nyhoff and Larry R Nyhoff. *Processing: an introduction to programming*. CRC Press, 2017a.

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

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

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

Processing Foundation. processing foundation education portal.

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

Processing Foundation. Processing, Accedido Enero 2020. URL <http://processing.org/>.

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

A partir de esta práctica, muchos de los listados mostrados en el guion se proporcionan a través del [enlace github¹](#), en cuyo caso se indica en la cabecera del listado indicando el nombre del proyecto processing.

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 2020b](#)], 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 descritas en la primera práctica, 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* dentro del método *draw*, si bien requiere el uso de la traslación 2D con la función *translate* para desplazar el objeto *PShape* creado.

¹<https://github.com/otsedom/CIU>

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 tono de gris de relleno de la forma según la coordenada x del puntero.

Listado 2.3: Modificando el color de relleno de una variable *PShape* ((P2_formas3d))

```
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 \[Accedido Enero 2020b\]](#) incluye una adaptación del ejemplo de una estrella (ver original en *Archivo->Ejemplos->Topics->Create Shapes->PolygonPShape*), 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 forzar el cierre de la línea poligonal.

Listado 2.4: Polígono arbitrario, una estrella, con *PShape* (P2_estrella)

```

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, estrella en este ejemplo, 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 mostrados en el apartado previo trabajan en 2D. Processing ofrece diversos modos de reproducción como son: SVG, PDF, P2D y P3D. Los dos últimos, P2D y P3D, hacen uso de hardware compatible con OpenGL, permitiendo mayor rendimiento en la salida gráfica. 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 \[Accedido Enero 2020a\]](#). 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 el listado [2.7](#).

Listado 2.7: Cubo y esfera (P2_formas3d)

```
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, la mayor potencia y flexibilidad, como ya mencionamos, viene dada por poder definir objetos a través de vértices arbitrarios. Como primer ejemplo ilustrativo, 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, tras desplazar el origen de coordenadas al centro de la ventana.

Listado 2.8: Pirámide de cuatro lados (P2_piramide)

```
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);
}
```

Cuando la pareja *beginShape-endShape* no tiene argumentos, se 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 en este [enlace](#)²):

- *beginShape-endShape(CLOSE)*: Cierra la línea poligonal, uniendo el último vértice con el primero, aplicando el color de relleno, como en el listado 2.8.

²https://processing.org/reference/beginShape_.html

- *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, aplicando relleno.
- *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*: Similar al anterior, si bien 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* (*P2_trianglestrip*)

```
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);
}
```

Si en algún momento fuera necesario recuperar los valores de los vértices almacenados en la estructura *PShape*, podremos hacer uso de la función *getVertex*, mientras que *setVertex*

permite realizar modificaciones. El listado 2.11 modifica uno previo, el 2.9, para que en cada iteración se muevan aleatoriamente, de forma leve, los vértices haciendo uso de ambas funciones.

Listado 2.11: Ejemplo de uso de *TRIANGLE_STRIP* con *PShape* y posterior modificación de sus vértices (*P2_trianglestrip_random*)

```
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);
  //Desplaza aleatoriamente los puntos
  for (int i = 0; i < obj.getVertexCount(); i++) {
    PVector v = obj.getVertex(i);
    v.x += random(-1, 1);
    v.y += random(-1, 1);
    v.z += random(-1, 1);
    obj.setVertex(i, v);
  }
  //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 línea poligonal que bien puede aproximar 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.

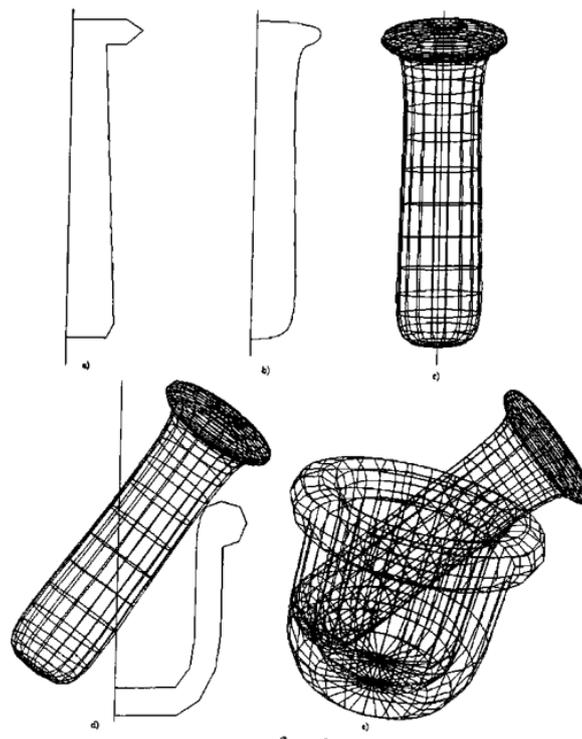


Figura 2.1: Creando una escena con superficies de revolución

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 para la mencionada malla pueden crearse de forma sistemática subdividiendo cada polígono, ver

figura 2.2 derecha.

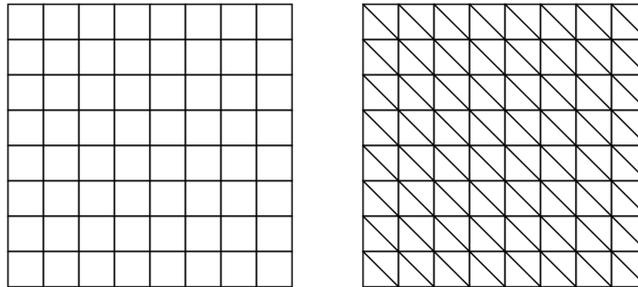


Figura 2.2: Ilustración de malla antes de tras triangularizar

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

$$\begin{aligned} x_2 &= x_1 \cdot \cos\theta - y_1 \cdot \sen\theta \\ y_2 &= x_1 \cdot \sen\theta + y_1 \cdot \cos\theta \end{aligned} \quad (2.1)$$

O su equivalente en forma matricial con premultiplicación:

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

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:

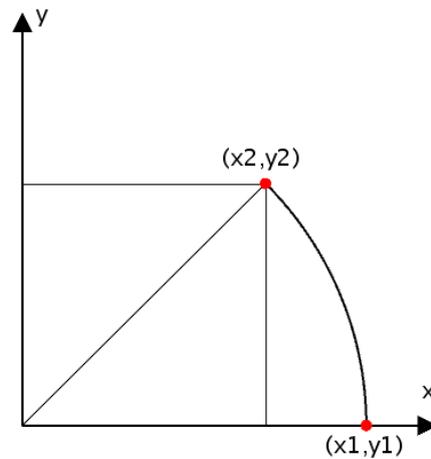


Figura 2.3: Rotación 2D de un ángulo θ

$$x_2 = x_1 \cdot \cos\theta - z_1 \cdot \text{sen}\theta$$

$$y_2 = y_1$$

$$z_2 = x_1 \cdot \text{sen}\theta + z_1 \cdot \cos\theta$$

(2.3)

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

Una vez que se ha definido el perfil de un sólido de revolución, la obtención de los vértices de dicho sólido requiere repetir, un determinado número de veces, la rotación de los puntos de dicho perfil, para obtener los vértices 3D de la malla del objeto. Una correcta conexión de dichos vértices, permite visualizar el volumen del objeto sobre la pantalla.

2.4. TAREA

Crear un prototipo que recoja puntos de un perfil del sólido de revolución al hacer clic con el ratón sobre la pantalla. Dicho perfil será utilizado por el prototipo para 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 únicamente se asumirá necesario almacenar el último definido.

La entrega se debe realizar a través del campus virtual, remitiendo un enlace a un proyecto github, cuyo README sirva de memoria, por lo que se espera que el README:

- identifique al autor,
- describa el trabajo realizado,
- argumente decisiones adoptadas para la solución propuesta,
- incluya referencias y herramientas utilizadas,
- muestre el resultado con un gif animado.

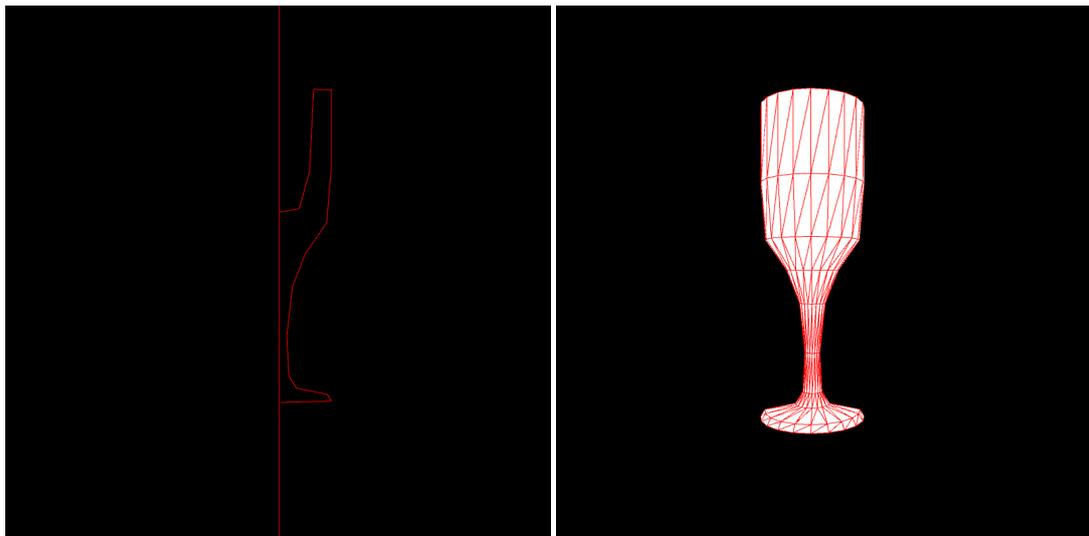


Figura 2.4: Perfil y sólido de revolución resultante

BIBLIOGRAFÍA

Daniel Shiffman. P3D tutorial, Accedido Enero 2020a. URL <https://processing.org/tutorials/p3d/>.

Daniel Shiffman. PShape tutorial, Accedido Enero 2020b. URL <https://processing.org/tutorials/pshape/>.

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 (P3_transf2D)

```
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* (P3_transf2Dpushpop)

```
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.

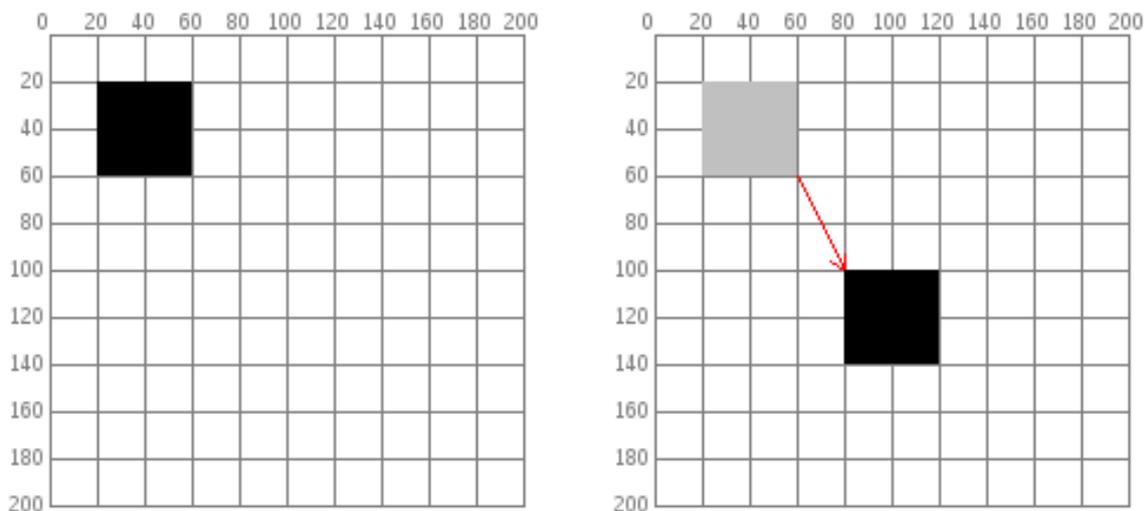


Figura 3.1: Cuadrado antes y después trasladar (fuente imagen Eisenberg [Accedido Febrero 2019]).

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.

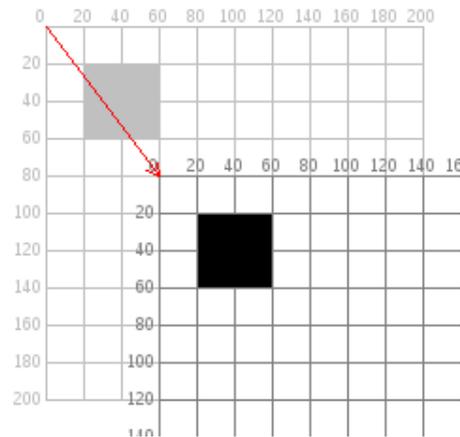


Figura 3.2: Traslación del sistema de referencia (fuente imagen Eisenberg [Accedido Febrero 2019]).

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 translación 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.

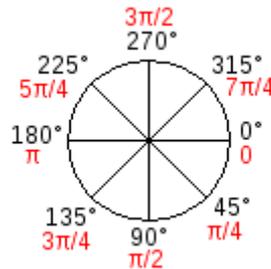


Figura 3.3: Ángulos de rotación en grados (negro) y radianes (rojo) (fuente imagen Eisenberg [Accedido Febrero 2019])

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:

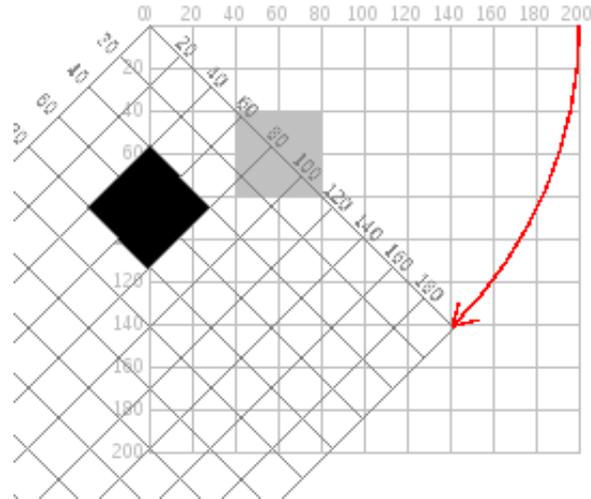


Figura 3.4: Resultados de la rotación del listado 3.5

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 (P3_planeta)

```
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 (P3_planetaR)

```
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 geoestacionaria, ver figura 3.5.

Listado 3.11: Dibujando una esfera levemente rotada con movimiento con satélite (P3_planetaysat)

```
float ang;
```

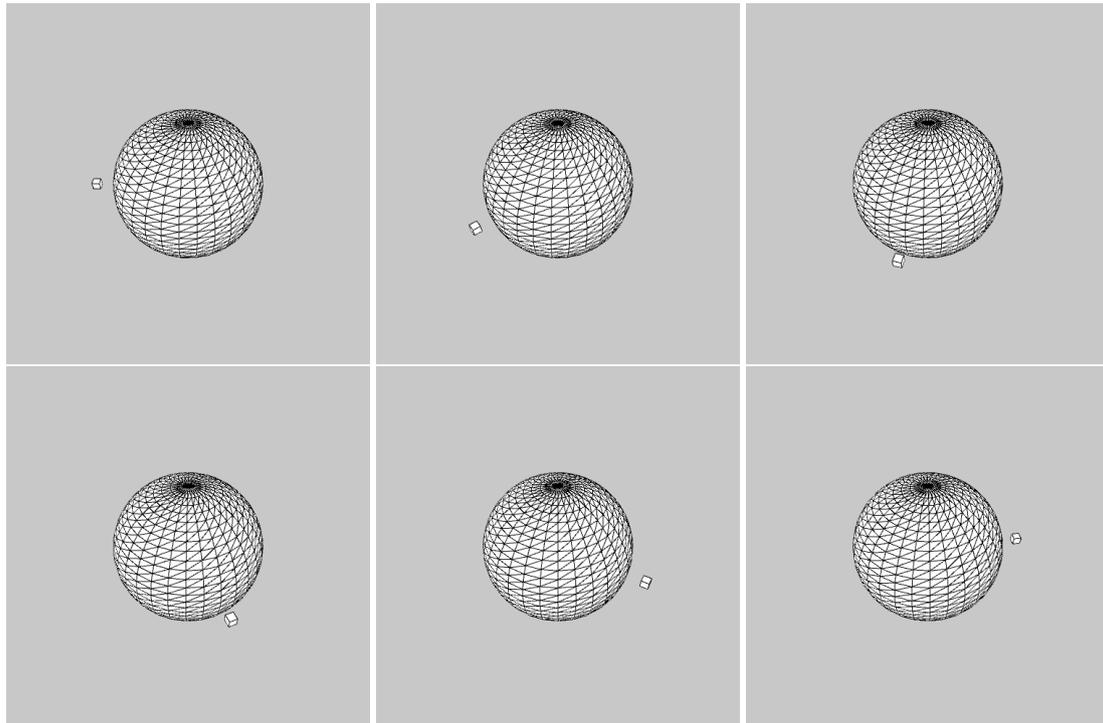


Figura 3.5: Instantáneas del *satélite* en órbita geoestacionaria

```
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, ver figura 3.6.

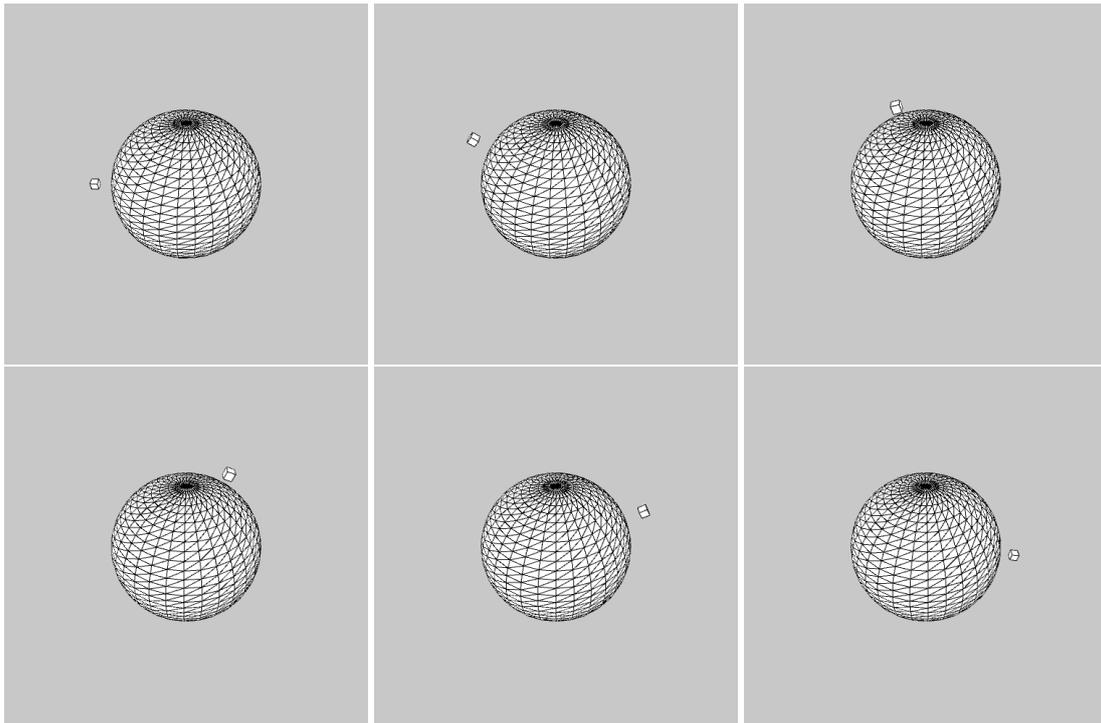


Figura 3.6: Instantáneas del *satélite* en órbita no geoestacionaria

Listado 3.12: Dibujando una esfera levemente rotada con movimiento (P3_planetaysatnogeo)

```
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 (p3_svg)

```
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 (p3_obj)

```
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 (p3_starwars)

```
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 (p3_imagen)

```
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).

La entrega se debe realizar a través del campus virtual, remitiendo un enlace a un proyecto github, cuyo README sirva de memoria, por lo que se espera que el README:

- identifique al autor,
- describa el trabajo realizado,
- argumente decisiones adoptadas para la solución propuesta,
- incluya referencias y herramientas utilizadas,
- muestre el resultado con un gif animado.

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 gráfica de una escena 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 \[Accedido Enero 2020\]](#): ortográfica y perspectiva.

Los mencionados dos tipos de proyecciones: ortográfica y perspectiva, se alternan con sus respectivas configuraciones por defecto en el listado 4.1. Dicho ejemplo dibuja un cubo en el centro de la ventana de visualización, hasta donde se ha trasladado el origen de coordenadas, permitiendo cambiar el modo de proyección entre ortográfica y perspectiva al hacer clic con el ratón, ver figura 4.1. Observar que la proyección perspectiva modifica el tamaño del objeto en función de su distancia, no así la ortográfica.

Las siguientes subsecciones describen las posibilidades ofrecidas para ambos tipos de proyección con mayor detalle.

Listado 4.1: Alternando entre proyección ortográfica y perspectiva (p4_orto_pers)

```
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. Ortográfica

Como se comenta anteriormente. en una proyección ortográfica dos objetos iguales aparecen del mismo tamaño independientemente de su distancia al observador. 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 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

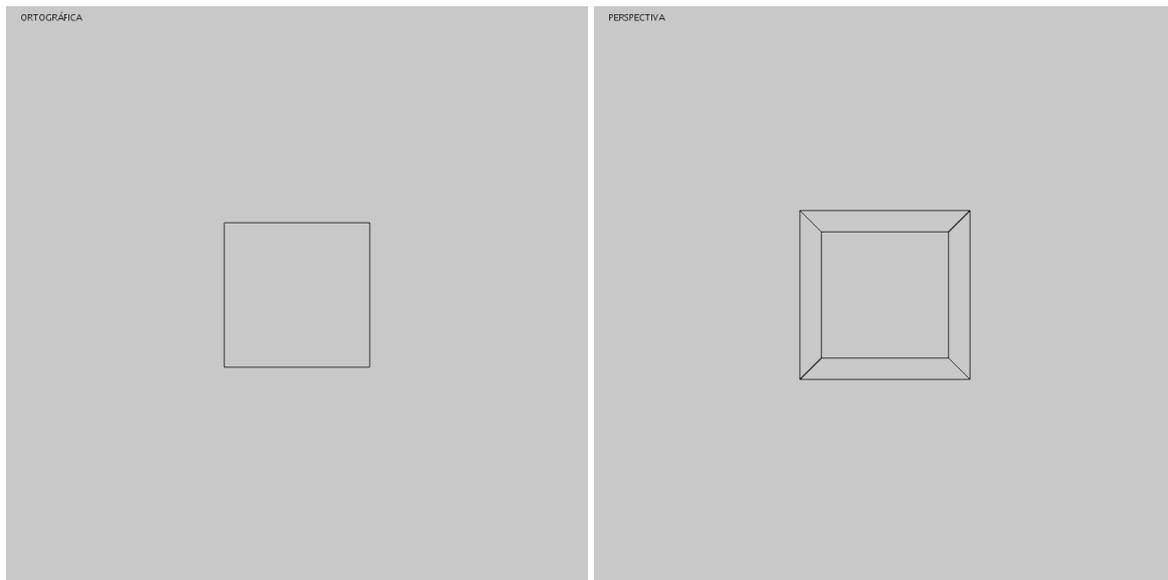


Figura 4.1: Proyección ortográfica y perspectiva de un cubo centrado en el origen

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 (p4_orto0)

```
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 4.3 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 para verlo en movimiento, pudiendo *intuirse* dos caras, lo cual permite apreciar su fondo.

Listado 4.4: Proyección ortográfica con movimiento del cubo (p4_orto1)

```
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 (p4_orto2)

```
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 bien 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 ejemplo mínimo de cubo proyectado con la perspectiva por defecto.

Listado 4.6: Proyección en perspectiva (p4_persp)

```

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 (p4_persp1)

```

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 (p4_persp2)

```

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 *frustrum*. Una llamada *frustrum* 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 menor 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 (p4_cam1)

```
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, ver figura 4.2.

Listado 4.10: Vista desde la cámara (p4_cam2)

```
int px,py;

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

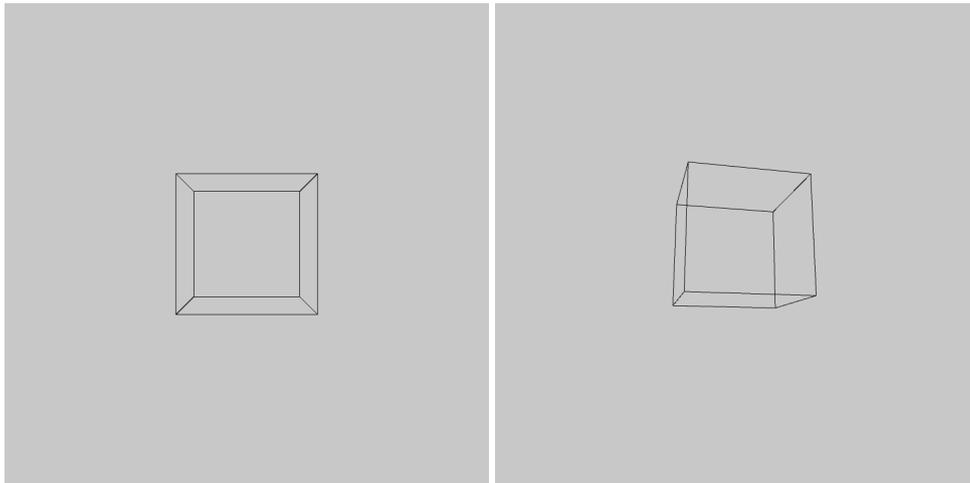


Figura 4.2: Cámara original y tras modificar su posición, según el listado 4.10

```
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, ver figura 4.3.

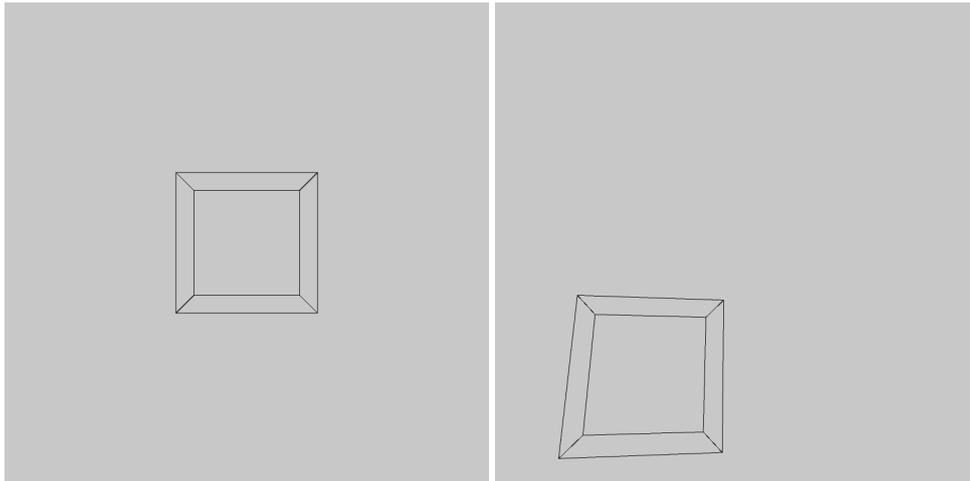


Figura 4.3: Cámara original y tras modificar el punto hacia el que mira, según el listado 4.11

Listado 4.11: Vista desde la cámara modificando hacia donde mira (p4_cam3)

```

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 (p4_cam4)

```
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=ang-360;
}
```

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 4.1.

4.3. OCLUSIÓN

Processing aplica por defecto el algoritmo de ocultación *z-buffer* a la hora de representar objetos poligonales con relleno. Es una característica que reduce ambigüedad en la reproducción, aportando realismo. 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 la opción está activada para el modo de reproducción *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 (p4_oculta)

```
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, como puede verse en la figura 4.4.

Listado 4.14: Un cubo (p4_oculta2)

```
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);
```

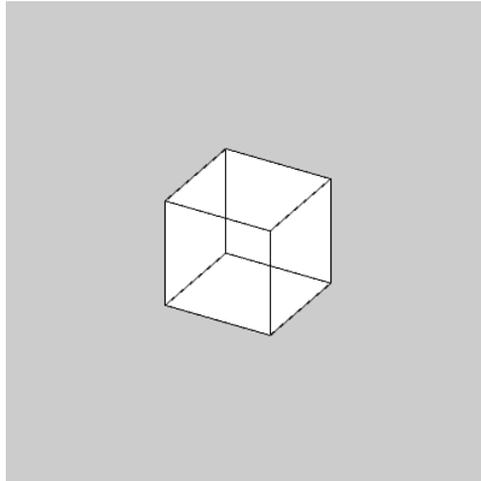


Figura 4.4: Resultado de no aplicar ocultación, según el listado 4.14

La utilización del z-buffer permite resolver situaciones con múltiples objetos, manejando correctamente 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, ve figura 4.5.

Listado 4.15: Dos cubos (p4_oculta3)

```
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*.

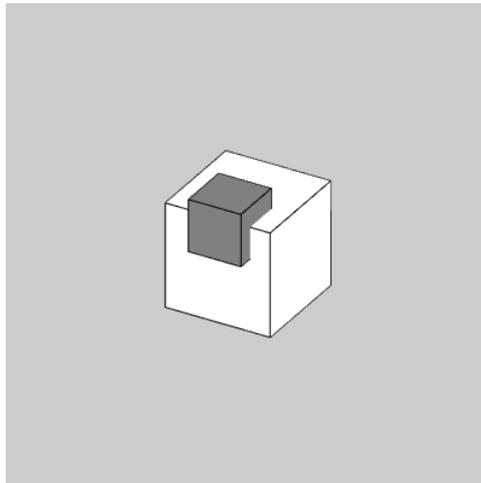


Figura 4.5: Z-buffer aplicado con dos cubos, listado 4.15

4.4. TAREA

La tarea anterior abordaba la definición de un sistema planetario. Para esta nueva entrega, se incluirá una *nave* que de forma interactiva podrá navegar por dicho sistema planetario. Dicha navegación podrá afectar no únicamente a la posición de la nave, sino también a la vertical y punto al que se mira desde ella, influyendo por tanto en su definición de cámara.

Si bien no hay obstáculo a experimentar con las cámaras existentes en la biblioteca de Processing, cada estudiante debe diseñar su propio mecanismo de interacción para inducir la navegación 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 un enlace a un proyecto github, cuyo README sirva de memoria, por lo que se espera que el README:

- identifique al autor,
- describa el trabajo realizado,
- argumente decisiones adoptadas para la solución propuesta,
- incluya referencias y herramientas utilizadas,
- muestre el resultado con un gif animado.

BIBLIOGRAFÍA

Daniel Shiffman. P3D tutorial, Accedido Enero 2020. 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 \[Accedido Enero 2020\]](#).

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 (p5_esfera)

```
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 (p5_lights1)

```
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 (p5_lightsambient)

```
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 (p5_lightsambientdir)

```
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 (p5_lightspot)

```

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 (p5_lightspoint)

```

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 (p5_lightspecular0)

```

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 (p5_lightmaterial)

```
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 (p5_textura)

```
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 con repetición (p5_texturarepeat)

```

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 (p5_texturapshape)

```

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 tira de triángulos (p5_texturatrianglestrip)

```
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. Indicar que este ejemplo requiere tener instalada la biblioteca de vídeo basada en *GStreamer*, para más detalles consultar la sección 6.2.

Listado 5.14: Entrada de vídeo como textura (p5_texturavid)

```
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.

5.3. 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 de elección personal, con objetos tridimensionales que incluya texturas, luces y movimiento de cámara (puede utilizarse una biblioteca para tal fin).

La entrega se debe realizar a través del campus virtual, remitiendo un enlace a un proyecto github, cuyo README sirva de memoria, por lo que se espera que el README:

- identifique al autor,
- describa el trabajo realizado,
- argumente decisiones adoptadas para la solución propuesta,
- incluya referencias y herramientas utilizadas,
- muestre el resultado con un gif animado.

BIBLIOGRAFÍA

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

Daniel Shiffman. P3D tutorial, Accedido Enero 2020. URL <https://processing.org/tutorials/p3d/>.

Práctica 6

Procesamiento de imagen y vídeo

6.1. IMAGEN

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 6.1.

Listado 6.1: Carga de imagen (p6_imashow)

```
PImage img;

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

  img=loadImage("moon.jpg");
}

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

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

Listado 6.2: Imagen que tiembla (p6_imashowrand)

```
PImage img;

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

  img=loadImage("moon.jpg");
}

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 6.3: Variando la intensidad de la imagen

```
PImage img;  
  
void setup () {  
  size (600,400);  
  
  img=loadImage ("moon.jpg");  
}  
  
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 6.4: Ciclo de animación (p6_imaanima)

```
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;  
  }  
}
```

6.2. 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 6.5 que en caso de hacer clic, desplaza de forma leve y aleatoria la posición de la imagen, similar al mostrado en la sección anterior con una imagen. Como en este ejemplo, una vez capturado el fotograma, podremos hacer con él cualquier acción aplicable a una imagen, ver sección 6.1.

Listado 6.5: Captura de cámara, con tembleque al hacer clic (p6_cam)

```
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);
  }
}
```

Las siguientes secciones pretenden mostrar algunas pinceladas de operaciones posibles con los fotogramas de un vídeo.

6.3. OPERACIONES BÁSICAS

6.3.1. Píxeles

Una imagen se compone de píxeles, en el listado 6.6 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 6.6: Captura de cámara mostrando la mitad superior umbralizada (p6_camumb)

```
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) ;
}
```

6.3.2. 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.

6.3.3. 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 6.7 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 6.7: Muestra imagen capturada en RGB y grises (p6_camgray)

```
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(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();
}
```

6.3.4. Umbralizado

El ejemplo de acceso a los píxeles, listado 6.6, aplicaba un umbralizado a la parte superior de la imagen capturada. El listado 6.8 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 en *x* del puntero. Las personas más curiosas pueden analizar las opciones de dicha función en la documentación.

Listado 6.8: Umbralizado dependiente de la posición del puntero (p6_camthreshold)

```
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();
}
```

6.3.5. 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 6.9. 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 6.9: Aplicación de Canny sobre una imagen (p6_camcanny)

```
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 6.10, 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 6.10: Resultado de aplicación del cálculo del gradiente (p6_camsobel)

```

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();
}
```

6.3.6. Diferencias

Para una cámara fija, el cálculo de diferencias con el modelo de fondo, o como en el listado 6.11, con el fotograma previo, indica las zonas de la imagen donde ha existido movimiento, mostrando el valor absoluto de dicha diferencia.

Listado 6.11: Diferencia entre imágenes sucesivas (p6_camdiff)

```
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();
}
```

6.4. DETECCIÓN

Ciertamente la Visión por Computador ofrece mucho más, otras operaciones con imágenes se introducen en el tutorial de Processing [Shiffman \[Accedido Enero 2020\]](#), 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*.

6.4.1. 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 6.12 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 6.12: Detección de caras y ojos (p6_camviola)

```
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(gris);

    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 6.15 basado en el ejemplo de Bryan Chung¹ con el

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

modelo proporcionado por OpenCV².

Listado 6.13: Detección del modelo del rostro (p6_camlandmarks)

```
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
  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,
```

with-processing/

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

```
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();
}
```

6.4.2. Personas

6.4.2.1. Kinect

Con la versión 1 del sensor (la segunda impone restricciones sobre el hardware como se indica más abajo), no hemos logrado ejecutar en un sistema operativo que no sea Windows. Asumiendo un sistema operativo Widows, 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 6.14 para además de los esqueletos, mostrar el modo de dibujar en la mano derecha del primero de ellos.

Listado 6.14: Esqueletos detectados con Kinect v1, con círculo en la mano derecha del primero (p6_kinectv1)

```
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*.

6.4.2.2. Openpose

De nuevo basado en el ejemplo de Bryan Chung para un único individuo⁴, el listado 6.15 aplica el detector basado en Openpose [Cao et al. \[2017\]](#) para determinar el esqueleto de una persona. En dicha referencia puedes además encontrar el enlace al modelo de *Caffe* necesario (en el listado se hace referencia al pesado *pose_iter_440000.caffemodel*), que no ha podido subirse al repositorio Github, y está disponible en el enlace⁵. Los requisitos computacionales suben significativamente, no habiendo probado su ejecución en un equipo con GPU.

Listado 6.15: Detección de esqueleto en 2D (p6_cam_openpose)

```
//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
  {1, 11}, // right body
  {11, 12}, // right thigh
  {12, 13}, // right calf
  {1, 0}, // neck
```

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/>

⁵http://posefs1.perception.cs.cmu.edu/OpenPose/models/pose/coco/pose_iter_440000.caffemodel

```
{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();
}
```

6.4.3. Manos

Incluimos en esta subsección la experiencia con el sensor *Leap Motion*⁶ que permite conocer la posición de todos los elementos a ambas manos, con diversas posibilidades para interacción.

Tras disponer de un dispositivo, para conseguir hacer uso del mismo bajo Windows 10 desde Processing, procedimos con los siguientes pasos:

⁶<https://www.ultraleap.com/tracking/>

- Descargar el *driver* desde el [enlace oficial](#)⁷, donde optamos por la versión para escritorio V2.
- Al ser Windows 10 tuvimos que realizar un *fix*, más información en este [enlace](#)⁸
- Instalamos *Leap Motion for Processing* siguiendo las instrucciones disponibles en el [enlace](#)⁹

Una vez finalizada la instalación pueden ejecutarse las aplicaciones con sus demos (*Leap Motion App Home*), así como los ejemplos básicos incluidos en Processing. El ejemplo *LM_1_Basics* muestra la visualización de ambas manos, mientras que el ejemplo *LM_2_Gestures* aborda el reconocimiento de gestos realizados.

6.5. GALERÍA

Una breve selección de utilización de imágenes para interacción:

- *Discrete figures* [Manabe](#) [[Accedido Marzo 2019](#)]
- *Más que la cara* [Lieberman](#) [[Accedido Marzo 2019](#)]. Interesantes los filtros que muestra en su perfil de [instagram](#)¹⁰.
- *Messa di voce* [Golan Levin](#) [[Accedido Marzo 2019](#)]
- *My little piece or privacy* [Roy](#) [[Accedido Marzo 2019](#)]
- *Starfield* [Lab212](#) [[Accedido Marzo 2019](#)]

6.6. TAREA

Proponer un concepto y su prototipo de combinación de salida gráfica e interacción 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 un enlace a un proyecto github, cuyo README sirva de memoria, por lo que se espera que el README:

⁷<https://developer.leapmotion.com/setup>

⁸<https://forums.leapmotion.com/t/resolved-windows-10-fall-creators-update-bugfix/6585>

⁹<https://github.com/nok/leap-motion-processing>

¹⁰<https://www.instagram.com/zach.lieberman>

- identifique al autor,
- describa el trabajo realizado,
- argumente decisiones adoptadas para la solución propuesta,
- incluya referencias y herramientas utilizadas,
- muestre el resultado con un gif animado.

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 Enero 2020. 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 del sonido por un sistema. 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 [1.4.5](#), 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. Aconsejo bajar el volumen antes de ejecutar.

Listado 7.1: Oscilador de pulso (p7_osc_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, asociados a la posición del puntero, 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 (p7_osc_pulsoamp)

```
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 (p7_osc_pulsoampfreq)

```
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 (p7_osc_pulsowidthfreq)

```
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 idéntico 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 (p7_osciladores)

```
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 (p7_osciladoresenoidales)

```
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 (p7_env)

```
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 (p7_env_midis)

```
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

Esta sección ilustra brevemente algunas de las 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 (p7_audioinvolumen)

```
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 (p7_audioinvolumenfft)

```
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 disponibles a través de *Archivo->Ejemplos->Bibliotecas principales->Sound*. Sin embargo, existen otras bibliotecas de audio, entre las que 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 (p7_minim_audiovis)

```
//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 también grabar la señal de entrada. El listado 7.12 se basa en el ejemplo *Basics->RecordAudioInput* que además de visualizar la señal, permite definir el intervalo de grabación, y salvar la señal capturada una vez.

Listado 7.12: Grabación de micrófono (p7_minim_audiovis_record)

```

//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 (p7_minim_audiovis_record_process)

```
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 [enlace](#)².

Listado 7.14: Teclado con notas musicales (p7_minim_instrumento)

²https://es.wikipedia.org/wiki/Frecuencias_de_afinaci%C3%B3n_del_piano

```
//Basado en el ejemplo de Minim CreateAnInstrument
import ddf.minim.*;
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. GALERÍA

Una breve selección de utilización de imágenes para interacción:

- Otra interesante biblioteca para Processing es [SoundCipher](#)³
- [Algo-Rhythm](#)⁴

7.5. 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 un enlace a un proyecto github, cuyo README sirva de memoria, por lo que se espera que el README:

- identifique al autor,
- describa el trabajo realizado,
- argumente decisiones adoptadas para la solución propuesta,
- incluya referencias y herramientas utilizadas,

³<http://explodingart.com/soundcipher/>

⁴<https://www.youtube.com/watch?v=6fB4K5fcOzo&feature=youtu.be>

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 2020] 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 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 alternativas a través de OpenProcessing², u otras propuestas como por ejemplo Stamper³ (para Chrome). Para diversas operaciones, como almacenar y descargar el trabajo realizado, habitualmente será necesario registrarse.

Abrir el editor de la web de p5.js, 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*.

En este documento, como punto de partida para descubrir otras variaciones, se adaptan

¹<https://editor.p5js.org>

²<https://www.openprocessing.org/sketch/create>

³<https://www.openprocessing.org/sketch/createhttps://p5stamper.com>

algunos de los ejemplos de las prácticas previas realizados en Processing.

El listado 8.1 se basa en el listado 1.25 dibujando líneas aleatorias desde la posición del puntero. Además de los ejemplos proporcionados en la web de p5.js⁴, 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 (p8_lines)

```
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 1.31, mostrando además la función *print* como la alternativa a *println* para la salida de texto.

Listado 8.2: Desplazamiento del círculo (p8_ball)

```
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* (p8_latido)

⁴<https://p5js.org/es/examples/>

```
var cirx=0;

function setup() {
  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 (p8_keyboard)

```
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 (p8_mouse)

```
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 (p8_planeta)

```
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 (p8_shape)

```
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 (p8_shapetex)

```

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 concretos⁵.

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 (p8_creategraphics)

```
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) ;
}
```

⁵<https://p5js.org/es/examples/>

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 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: Carga y visualización de una imagen (p8_imagen)

```
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 (p8_sonido)

```
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, en la que se han mostrado usos básicos de imagen, sonido y vídeo. Otras bibliotecas disponibles pueden consultarse en la web de p5.js⁶.

⁶<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`⁷. En el ámbito de la visión por computador, la conocida OpenCV cuenta para programación web con `OpenCV.js`⁸, si bien el material disponible está orientado a OpenCV 3.x, cuando ya la versión de la biblioteca es la 4.2.0.

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 2020], que surge con el propósito de proporcionar recursos educativos gratuitos sobre aprendizaje automático, analizando herramientas disponibles como `ml5js` NYU ITP [Accedido Marzo 2020], `tensorflow.js` Google [Accedido Marzo 2020b], *Teachable machine* Google [Accedido Marzo 2020a], `Wekinator` Fiebrink [Accedido Marzo 2019], o `RunwayML` que permite descargar la aún en versión beta Valenzuela [Accedido Marzo 2020].

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 un enlace al proyecto compartido para su ejecución en un navegador, además de un enlace al repositorio github, cuyo README sirva de memoria incluyendo:

- identifique al autor,
- describa el trabajo realizado,
- argumente decisiones adoptadas para la solución propuesta,
- incluya referencias y herramientas utilizadas,

BIBLIOGRAFÍA

Rebecca Fiebrink. `Wekinator`, Accedido Marzo 2019. URL <http://www.wekinator.org/>.

⁷<https://threejs.org/>

⁸https://docs.opencv.org/3.4/d5/d10/tutorial_js_root.html

Google. Teachable machine train a computer to recognize your own images, sounds, & poses.,
Accedido Marzo 2020a. URL <https://teachablemachine.withgoogle.com>.

Google. Tensorflow.js, Accedido Marzo 2020b. URL <https://www.tensorflow.org/js>.

Gene Kogan. Machine Learning for Artists, Accedido Marzo 2020. URL <http://ml4a.github.io/>.

Lauren McCarthy. p5.js, Accedido Marzo 2020. URL <https://p5js.org>.

NYU ITP. Friendly machine learning for the web, Accedido Marzo 2020. 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. Machine learning for creators, Accedido Marzo 2020. URL <https://runwayml.com>.

Práctica 9

Introducción a los *shaders* en Processing

9.1. INTRODUCCIÓN

Un *shader* se define como una pieza de código que se ejecuta sobre un conjunto de píxeles de forma simultánea, y diferente para cada píxel, en muchos casos aprovechando las capacidades de la GPU. A través de un *shader* se ofrece la posibilidad de manipular la imagen de salida antes de mostrarla en pantalla, permitiendo aplicar efectos de reproducción (*rendering*), modificando dicha imagen resultante según nuestras intenciones. En *The Book of Shaders* [Gonzalez Vivo and Lowe \[Accedido Marzo 2020\]](#), se sugiere la analogía con la invención de la imprenta moderna, que supuso un extraordinario incremento de velocidad a la hora de producir documentos.

En un primer momento, los *shaders* fueron concebidos para desarrollar modelos de iluminación y sombreado, por ello el nombre *shader*, si bien en la actualidad se aplican sobre todas las etapas de reproducción, distinguiéndose, los siguientes tipos:

- Píxel (DirectX) / Fragmento (OpenGL): Combinación de textura y color por píxel.
- Vértice: Modifica la posición del vértice, color, coordenadas de textura, e incidencia de la luz. No añade vértices.
- Geometría: Geometría generada procedimentalmente.

Entre los diversos lenguajes utilizados para el desarrollo de *shaders* (HLSL, GLSL, Cg) este guion se centra en GLSL por su vinculación con OpenGL. Processing, nuestro campo de juego, dispone de la clase *PShader* cubriendo las etapas de fragmentos y vértices. Las secciones a continuación se basan en las referencias [Gonzalez Vivo and Lowe \[Accedido Marzo 2020\]](#), [Colubri \[Accedido Febrero 2019\]](#). La primera de ellas, el mencionado *The Book of Shaders*,

declara ser *una guía paso a paso a través del abstracto y complejo universo de los Fragment Shaders* con pautas para Three.js, Processing u openFrameworks. La segunda de ellas es el tutorial de Processing para el desarrollo de *shaders*.

Señalar que Processing distingue tres clases de *shaders*: *POINT*, *LINE* y *TRIANGLE*. Los dos primeros permiten alterar el modo de dibujar puntos y líneas, mientras que el tercer se concibe para cualquier otro elemento gráfico, básicamente polígonos. Estos últimos al combinar luces y texturas, dan lugar e cuatro combinaciones según lo que esté presente en el proceso de reproducción. Por ese motivo Processing habla de un total de seis grupos diferentes de *shaders*. Processing realiza una autodetección, que se puede sobrescribir desde código por medio de una directiva *#define* con una de las siguientes seis posibilidades:

- *#define PROCESSING_POINT_SHADER*: *Shader* de puntos
- *#define PROCESSING_LINE_SHADER*: *Shader* de líneas
- *#define PROCESSING_COLOR_SHADER*: *Shader* de triángulos sin textura ni luces
- *#define PROCESSING_LIGHT_SHADER*: *Shader* de triángulos con luces
- *#define PROCESSING_TEXTURE_SHADER*: *Shader* de triángulos con textura
- *#define PROCESSING_TEXLIGHT_SHADER*: *Shader* de triángulos con textura y luces

Para el desarrollo de *shaders* no soportados en Processing, desde Processing puede escribirse directamente código a nivel bajo OpenGL¹, si bien queda fuera del objetivo de esta práctica. En las siguientes secciones no trataremos los dos primeros grupos, remitiendo a las personas interesadas al tutorial [Colubri \[Accedido Febrero 2019\]](#) para los relacionados con puntos y líneas.

Se incluyen a continuación en este capítulo dos secciones respectivamente introduciendo los *shaders* de fragmentos y vértices. Para este curso se plantea una única práctica, si bien probablemente en cursos sucesivos se plantearán al menos dos, permitiendo profundizar un poco más en cada uno de dichas secciones actuales.

9.2. *Shaders* DE FRAGMENTOS

En primer término obviamos el *shader* de vértices, centrando la atención en el *shader* de fragmentos, siguiendo el esquema propuesto en [Gonzalez Vivo and Lowe \[Accedido Marzo](#)

¹<https://github.com/processing/processing/wiki/Advanced-OpenGL>

2020]. Los ejemplos mostrados en esta sección, no especifican ningún *shader* de vértices, por lo que Processing adoptará el *shader* de vértices por defecto. Por dicha razón, no entraremos en muchos detalles en esta sección el *shader* de vértices, con excepción de una leve mención en la subsección dedicada a imágenes.

Un *shader* funciona como una función que recibe una localización, y devuelve un color. Para ejecutarse en paralelo, cada hilo o *thread* es independiente todos los demás, va ciego sin saber lo que hace el resto, al no haber comunicación posible entre ellos, evitando poner en riesgo la integridad de los datos.

9.2.1. Hola mundo

El código mostrado en el listado 9.1, dibuja una esfera centrada en la ventana con la reproducción por defecto o clásica de OpenGL. Manteniendo el botón del ratón pulsado, se carga un *shader* de fragmentos que altera la reproducción, asignando un color a todos los píxeles del objeto. El *shader* se describe en un archivo, en concreto *Colorea.glsl*, que debe estar presente en la misma carpeta o dentro de *data*. El código de dicho *shader* se muestra en listado 9.2.

Listado 9.1: Dibujo de esfera con o sin shader simple (p9_shader_holamundo)

```
PShader sh;

void setup() {
  size(640, 360, P3D);
  noStroke();
  fill(204);
  sh = loadShader("Colorea.glsl");
}

void draw() {
  if (mousePressed)
    shader(sh);
  else
    resetShader();

  background(0);
  float dirY = (mouseY / float(height) - 0.5) * 2;
  float dirX = (mouseX / float(width) - 0.5) * 2;
  directionalLight(204, 204, 204, -dirX, -dirY, -1);
  translate(width/2, height/2);
  sphere(120);
}
```

Listado 9.2: Código *shader* Colorea.glsl

```
#ifndef GL_ES
```

```
precision mediump float;
#endif

void main() {
    gl_FragColor = vec4(0.831,0.567,1.000,1.000);
}
```

Básicamente el *shader* tiene una línea de código en la que asigna un vector de cuatro valores a la variable reservada (*built-in*²) *gl_FragColor*. El código del *shader*, listado 9.1 evidencia diversos aspectos:

- Recuerda a C. Todo *shader* tiene un *main*, un *shader* de fragmento además devuelve al final el color del píxel en la variable global reservada *gl_FragColor*.
- Dicha variable *gl_FragColor* es de tipo *vec4*, siendo un color se refiere respectivamente a los canales rojo, verde, azul y alfa. Sus valores están normalizados entre 0.0 y 1.0.
- *vec2*, *vec3* y *vec4* son tipos añadidos de variables con respectivamente 2, 3 y 4 valores *float*.
- Se incluyen macros para el preprocesador (con #). Pueden definirse variables globales (con *#define*) y establecer condicionales (con *#ifdef* y *#endif*). El ejemplo mostrado verifica si *GL_ES* está definida, que suele estarlo para móviles y navegadores, de cara a bajar el nivel de precisión. Menor precisión, implica más velocidad, pero peor calidad. En este ejemplo concreto, la línea (*precision mediump float;*) ajusta todos los valores flotantes a una precisión media. Los otros niveles posibles son: *low* (*precision lowp float;*) y *high* (*precision highp float;*).
- Ojo, las especificaciones de GLSL no garantizan (depende del fabricante de la placa de vídeo) una conversión automática, por lo que se aconseja usar *.* para los flotantes; evita *gl_FragColor = vec4(1,0,0,1)*; o correrás el riesgo de tener un error complejo de localizar.

Si bien cada hilo no conoce lo que ocurre en el resto, es posible enviar valores de entrada desde la CPU. Estos valores serán iguales para todos los hilos, siendo variables *uniform*, pudiendo tener distintos tipos: *float*, *vec2*, *vec3*, *vec4*, *mat2*, *mat3*, *mat4*, *sampler2D* y *samplerCube*.

En un nuevo ejemplo, modificamos el código anterior, pasando información de la posición de ratón y resolución de la ventana de visualización, utilizando la función *set* de las variables *PShader*. El código Processing se incluye en el listado 9.3, y el *shader* en el listado 9.4.

²[https://www.khronos.org/opengl/wiki/Built-in_Variable_\(GLSL\)](https://www.khronos.org/opengl/wiki/Built-in_Variable_(GLSL))

En el código del *shader* se declaran tres variables *uniform* que recogen los valores enviados de resolución (*u_resolution*), posición del puntero (*u_mouse*), y segundos transcurridos (*u_time*). Al ejecutar, nuevamente al hacer clic el color de la esfera se ve modificado, si bien ahora dicho color varía, dependiendo de la posición del puntero, componentes roja y verde, y el tiempo pasado, componente azul.

Listado 9.3: Código Processing que envía valores al *shader* *ColoreaInput.glsl* en cada ejecución de *draw* (*p9_shader_setvalues*)

```
PShader sh;

void setup() {
  size(640, 360, P3D);
  noStroke();
  fill(204);
  sh = loadShader("ColoreaInput.glsl");
}

void draw() {
  if (mousePressed)
  {
    shader(sh);
    sh.set("u_resolution", float(width), float(height));
    sh.set("u_mouse", float(mouseX), float(mouseY));
    sh.set("u_time", float(millis()) / float(1000));
  }
  else
    resetShader();

  background(0);
  float dirY = (mouseY / float(height) - 0.5) * 2;
  float dirX = (mouseX / float(width) - 0.5) * 2;
  directionalLight(204, 204, 204, -dirX, -dirY, -1);
  translate(width/2, height/2);
  sphere(120);
}
```

Listado 9.4: Código *shader* *ColoreaInput.glsl*

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

void main() {
  vec2 mouse = u_mouse / u_resolution;
  gl_FragColor = vec4(mouse.x, mouse.y, abs(sin(u_time)), 1.000);
}
```

El código del *shader* incluye una llamada a la función *abs*. Las funciones del repertorio están diseñadas para aprovechar la potencia de la GPU. Además de *abs* hay otras funciones disponibles en GLSL como por ejemplo: *sin*, *cos*, *tan*, *asin*, *acos*, *atan*, *pow*, *exp*, *log*, *sqrt*, *abs*, *sign*, *floor*, *ceil*, *fract*, *mod*, *min*, *max* y *clamp*. Para una descripción exhaustiva, hacer uso de la documentación de referencia³.

Otra variable *vec4* reservada es *gl_FragCoord*, que para cada hilo contiene la coordenada del píxel. Si las variables globales a todos los píxeles son *uniform*, variables como *gl_FragCoord*, aquellas que para cada hilo contienen el valor particular del píxel, son *varying*. El *shader* mostrado en el listado 9.5 con el mismo *script* de Processing que el anterior colorea la esfera usando como componentes roja y verde, la normalización de la posición en la pantalla. ¿Qué pasará si usas *sin(u_time)* para la componente azul como en el anterior?

Listado 9.5: Código *shader* ColoreaInput.glsl (p9_shader_setvalues_coord)

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution;
    gl_FragColor = vec4(st.x, st.y, 0.0, 1.0);
}
```

9.2.2. Dibujando con algoritmos

El lenguaje GLSL, no dispone de funciones específicas de dibujo de primitivas gráficas. Esto quiere decir que no cuenta con funciones para dibujar puntos, líneas, circunferencias, etc. Por este motivo, será necesario utilizar técnicas procedimentales (*procedural*) para *dibujar*. En esta sección, se introducen algunos trucos simples con dicho propósito. El repertorio de trucos, crecerá con la experiencia y apoyado por la comunidad.

Como código Processing, utilizaremos un prototipo sencillo que dibuja un recuadro que cubre toda la ventana de visualización además de cargar el *shader*, ver el listado 9.6. La carga del *shader* de cada ejemplo mostrado a continuación en este apartado requiere modificar la línea *sh = loadShader("DibujaX.glsl")*; de forma adecuada. Una alternativa es hacer uso de un editor GLSL en línea como por ejemplo el proporcionado por [Gonzalez Vivo and Lowe \[Accedido Marzo 2020\]](#) disponible a través del [enlace](#)⁴, que permite *cacharrear* de forma

³<https://www.khronos.org/opengles/sdk/docs/manglsl/docbook4/>

⁴<https://thebookofshaders.com/edit.php>

interactiva con el código y ver su ejecución de forma inmediata.

Listado 9.6: (p9_shader_dibuja)

```
PShader sh;

void setup() {
  size(600, 600, P2D);
  noStroke();
  sh = loadShader("DibujaX.glsl");
}

void draw() {
  sh.set("u_resolution", float(width), float(height));
  sh.set("u_mouse", float(mouseX), float(mouseY));
  sh.set("u_time", millis() / 1000.0);
  shader(sh);
  rect(0,0,width,height);
}
```

El *shader Dibuja0.glsl* muestra una escala de grises de izquierda (negro) a derecha (blanco), ver el listado 9.7. Establece una relación entre el valor de *st.x*, la proporción de la coordenada x del píxel en la ventana, con el nivel de gris utilizado para pintar. En el código puede observarse la flexibilidad en el manejo de variables tipo *vec*, *gl_FragCoord.xy* hace referencia a los dos primeros valores, con *vec4(vec3(st.x),1.0)* se entiende que los tres primeros valores serán iguales.

Listado 9.7: Escala de grises (Dibuja0.glsl)

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

void main() {
  //Normalizamos coordenadas del píxel en base a la resolución
  vec2 st = gl_FragCoord.xy/u_resolution;

  //Escala de grises izquierda a derecha
  gl_FragColor = vec4(vec3(st.x),1.0);
}
```

El *shader Dibuja1.glsl*, ver listado 9.8, modifica el anterior, pintando en verde cuando *st.x* y *st.y* tienen valores próximos, es decir en la diagonal. La condición de proximidad viene definida por la variable *grosor*. El resultado es una línea, una salida gráfica.

Listado 9.8: Escala de grises y línea (Dibuja1.glsl)

```

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float grosor=0.1;

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;

    vec3 color = vec3(st.x); //Escala de grises izquierda a derecha;
    // Línea
    if (st.y-grosor<st.x && st.x<st.y+grosor)
        color = vec3(0.0,1.0,0.0); //Verde

    gl_FragColor = vec4(color,1.0);
}

```

La línea dibujada con el *shader Dibuja1.glsl*, muestra bordes *duros*, pudiendo producir el efecto de escalera (*aliasing*). El código del *shader Dibuja2.glsl*, ver listado 9.9, basado en los ejemplos de [Gonzalez Vivo and Lowe \[Accedido Marzo 2020\]](#), suaviza la transición de los bordes de la línea haciendo uso de *smoothstep*. La salida de la función *plot* proporciona un valor suavizado entre 0 y 1, dependiendo del valor de *grosor*. Dicha salida permite decidir en qué porcentaje cada píxel se pinta con el valor correspondiente de la escala de grises y verde.

Listado 9.9: Escala de grises con línea con transición suave (Dibuja2.glsl)

```

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float grosor=0.1;

float plot(vec2 st, float pct){
    //smoothstep da salida suave entre dos valores (Hermite)
    //Combina dos para crear el chichón
    return smoothstep( pct-grosor, pct, st.y) -
           smoothstep( pct, pct+grosor, st.y);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;

    vec3 color = vec3(st.x); //Escala de grises izquierda a derecha;

```

```

//Valor a tomar como referencia para bordes suaves
float val = st.x;

//Combina escala con línea según el valor de pct
float pct = plot(st, val);
// Valores altos verde, bajos escala de grises
color = (1.0-pct)*color+pct*vec3(0.0,1.0,0.0);

gl_FragColor = vec4(color,1.0);
}

```

Para dibujar las líneas se busca en los ejemplos previos proximidad entre los valores de $st.x$ y $st.y$. Si antes de verificar dicha proximidad, se altera uno de ellos, el resultado será una salida diferente a una línea. En el *shader Dibuja3.gls*, ver el listado, el 9.10, antes de comparar $st.x$ y $st.y$, se modifica el valor de la primera combinando la función *pow* y la posición del puntero. La forma resultante dependerá del puntero, obteniendo una mayor variedad de formas.

Listado 9.10: Modificando la forma resultante con el puntero (Dibuja3.gls)

```

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float grosor=0.1;

float plot(vec2 st, float pct){
    //smoothstep da salida suave entre dos valores (Hermite)
    return smoothstep( pct-grosor, pct, st.y) -
           smoothstep( pct, pct+grosor, st.y);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec2 mouse= u_mouse/u_resolution.xy;

    vec3 color = vec3(st.x); //Escala de grises izquierda a derecha;

    //Valor a tomar como referencia para bordes suaves
    float val = pow(st.x,1.0+mouse.x*10.0);

    //Combina escala con línea según el valor de pct
    float pct = plot(st, val);
    // Valores altos verde, bajos escala de grises
    color = (1.0-pct)*color+pct*vec3(0.0,1.0,0.0);
}

```

```
gl_FragColor = vec4(color,1.0);
}
```

En el *shader Dibuja4.glsl*, ver listado 9.11, se añade una alteración del grosor de la forma dependiendo el tiempo transcurrido.

Listado 9.11: Con grosor cambiante en función del tiempo (Dibuja4.glsl)

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float grosor=0.1;

float plot(vec2 st, float pct){
    //smoothstep da salida suave entre dos valores (Hermite)
    return smoothstep( pct-grosor, pct, st.y) -
           smoothstep( pct, pct+grosor, st.y);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec2 mouse= u_mouse/u_resolution.xy;

    //Grosors cambiante con el tiempo
    grosor=abs(sin(u_time));

    vec3 color = vec3(st.x); //Escala de grises izquierda a derecha;

    //Valor a tomar como referencia para bordes suaves
    float val = pow(st.x,1.0+mouse.x*10.0);

    //Combina escala con verde según el valor de pct
    float pct = plot(st,val);
    // Valores altos verde, bajos escala de grises
    color = (1.0-pct)*color+pct*vec3(0.0,1.0,0.0);

    gl_FragColor = vec4(color,1.0);
}
```

Los últimos dos ejemplos hacen uso de la función *pow*. El repertorio de funciones con las que jugar es más amplio. En el *shader Dibuja5.glsl*, ver listado 9.12 se muestra un abanico de ellas, basta comentar y descomentar para obtener distintos resultados en la ejecución.

Listado 9.12: Repertorio de funciones (Dibuja5.glsl)

```
#ifdef GL_ES
precision mediump float;
#endif
```

```

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float grosor=0.1;

float plot(vec2 st, float pct){
    //smoothstep da salida suave entre dos valores (Hermite)
    return smoothstep( pct-grosor, pct, st.y) -
           smoothstep( pct, pct+grosor, st.y);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec2 mouse= u_mouse/u_resolution.xy;

    vec3 color = vec3(st.x); //Escala de grises izquierda a derecha;

    //Valor a tomar como referencia para bordes suaves
    float val = smoothstep(0.1,0.9,st.x); // interpolación Hermite
    //float val = mod(st.x,0.5); // Módulo de 0.5
    //float val = fract(st.x); // Parte fraccionaria
    //float val = ceil(st.x); // Entero más cercano mayor o igual
    //float val = floor(st.x); // Entero más cercano menor o igual
    //float val = sign(st.x); // Signo
    //float val = abs(st.x); // Valor absoluto
    //float val = clamp(st.x,0.0,1.0); // Limitado entre 0 y 1
    //float val = min(0.0,st.x); // Valor mínimo
    //float val = max(0.0,st.x); // Valor máximo

    //Combina escala con línea según el valor de pct
    float pct = plot(st,val);
    // Valores altos verde, bajos escala de grises
    color = (1.0-pct)*color+pct*vec3(0.0,1.0,0.0);

    gl_FragColor = vec4(color,1.0);
}

```

La creación de transiciones con formas complejas a partir de funciones es campo de trabajo de la comunidad, en [Gonzalez Vivo and Lowe \[Accedido Marzo 2020\]](#) se sugiere revisar las propuestas de Golan Levin⁵ e Iñigo Quiles⁶.

Como el mundo no se limita a los grises, en el *shader Dibuja6.glsl*, ver el listado [9.13](#) muestra una escala entre dos colores, utilizando *st.x* como porcentaje de combinación entre ellos a través de la función *mix*.

Listado 9.13: Escala de color y línea verde (Dibuja6.glsl)

⁵<http://www.flong.com/>

⁶<https://thebookofshaders.com/05/?lan=es>

```

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float grosor=0.02;

vec3 colorA = vec3(0.149,0.141,0.912);
vec3 colorB = vec3(1.000,0.833,0.224);

float plot(vec2 st, float pct){
    //smoothstep da salida suave entre dos valores (Hermite)
    return smoothstep( pct-grosor, pct, st.y) -
           smoothstep( pct, pct+grosor, st.y);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec2 mouse= u_mouse/u_resolution.xy;

    //Valor a tomar como referencia para bordes suaves
    float val = st.x;

    vec3 color = mix(colorA, colorB, val); //Escala entre dos colores

    //Combina escala con línea según el valor de pct
    float pct = plot(st,val);
    // Mezcla en función val
    color = mix(color,vec3(0.0,1.0,0.0),pct);

    gl_FragColor = vec4(color,1.0);
}

```

Concluimos esta serie con el *shader Dibuja7.gsl*, ver el listado 9.14, que combina el uso de funciones, para obtener una salida no limitada a líneas, y la escala de colores.

Listado 9.14: Escala de color y formas (Dibuja7.gsl)

```

#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float grosor=0.02;

vec3 colorA = vec3(0.149,0.141,0.912);

```

```

vec3 colorB = vec3(1.000,0.833,0.224);

float plot(vec2 st, float pct){
    //smoothstep da salida suave entre dos valores (Hermite)
    return smoothstep( pct-grosor, pct, st.y) -
           smoothstep( pct, pct+grosor, st.y);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec2 mouse= u_mouse/u_resolution.xy;

    //Valor a tomar como referencia para bordes suaves
    vec3 pct = vec3(st.x);

    //Funciones por canal
    pct.r = smoothstep(0.0,1.0, st.x);
    pct.g = sin(st.x*PI);
    pct.b = pow(st.x,0.5);

    vec3 color = mix(colorA, colorB, pct); //Escala entre dos colores

    //Combina por cada canal
    color = mix(color,vec3(1.0,0.0,0.0),plot(st,pct.r));
    color = mix(color,vec3(0.0,1.0,0.0),plot(st,pct.g));
    color = mix(color,vec3(0.0,0.0,1.0),plot(st,pct.b));

    gl_FragColor = vec4(color,1.0);
}

```

El acceso a vectores en los ejemplos previos sugiere su uso como estructuras, p.e. *st.x*, con algunas particularidades *gl_FragCoord.xy*. Realmente es posible acceder a los valores como índice de un vector, además de alternativas para los nombres de los campos, de cara a facilitar la comprensión del código por conveniencia si trabajamos con posiciones, color o texturas como se muestra en el listado 9.15.

Listado 9.15: Acceso a posiciones de un vector

```

vec4 vector;
vector[0] = vector.r = vector.x = vector.s;
vector[1] = vector.g = vector.y = vector.t;
vector[2] = vector.b = vector.z = vector.p;
vector[3] = vector.a = vector.w = vector.q;

```

Estas posibilidades permiten mucha variedad a la hora de manipular valores, el trozo de código mostrado en el listado 9.16 juega con las asignaciones de los planos de color.

Listado 9.16: Acceso a posiciones de un vector

```

vec3 yellow, magenta, green;

// Define yellow

```

```

yellow.rg = vec2(1.0); // Asigna 1. a los canales rojo y verde
yellow[2] = 0.0;      // Asigna 0 al canal azul

// Define magenta
magenta = yellow.rgb; // Asigna
intercambiando los canales verde y azul

// Define green
green.rgb = yellow.bgb; // Asigna el canal azul de yellow a los canales rojo y azul

```

Para dibujar formas más complejas, dada la carencia de funciones de dibujado, la única opción es la mencionada forma procedimental. La función *step(th, val)* retorna 0 o 1 según si *val* es menor que *th* o no. El *shader Dibuja10.glsl*, ver listado 9.17, la aprovecha para crear un recuadro.

Listado 9.17: Dibujando un recuadro (Dibuja10.glsl)

```

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

void main() {
//Normalizamos coordenadas con la resolución
vec2 st = gl_FragCoord.xy/u_resolution;
vec3 color = vec3(0.0);

// step devuelve 1.0 (blanco) 0.0 (negro).
float left = step(0.1, st.x); // st.x>0.1
float bottom = step(0.1, st.y); // st.y>0.1

float right = step(0.1, 1.0 - st.x); // st.x<0.9
float top = step(0.1, 1.0 - st.y); // st.y<0.9

// Multiplicar equivale a AND
color = vec3( left * bottom * top * right );

gl_FragColor = vec4(color, 1.0);
}

```

Para círculos, la propuesta es calcular la distancia al centro de la ventana. En el *shader Dibuja11.glsl*, ver el listado 9.18, se hace uso de la función *distance* combinada con *step*. No utilizar *step* mostrará un campo de distancias como escala de grises.

Listado 9.18: Dibujando un círculo (Dibuja11.glsl)

```

#ifdef GL_ES
precision mediump float;

```

```

#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

void main() {
    //Normalizamos coordenadas con la resolución
    vec2 st = gl_FragCoord.xy/u_resolution;
    float pct = 0.0;

    // Distancia del píxel al centro
    pct = distance(st,vec2(0.5));

    // Blanco para píxeles con distancia menor que 0.4
    float circ = step(0.2,0.5-pct);

    vec3 color = vec3(circ);

    gl_FragColor = vec4( color, 1.0 );
}

```

Las transformaciones se consiguen modificando las coordenadas, es decir *st* de forma apropiada. La traslación de un círculo en función del tiempo, pintado no en blanco sino degradado, se muestra en el *shader Dibuja12.glsl*, ver el listado [9.19](#).

Listado 9.19: Dibujando un círculo que se traslada (Dibuja12.glsl)

```

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

void main() {
    //Normalizamos coordenadas con la resolución
    vec2 st = gl_FragCoord.xy/u_resolution;
    float pct = 0.0;

    // Mover la forma
    vec2 translate = vec2(cos(u_time),sin(u_time));
    st += translate*0.35;

    // Distancia del píxel al centro
    pct = distance(st,vec2(0.5));

    // Blanco para píxeles con distancia menor que 0.4
    float circ = step(0.3,0.5-pct)*st.x;

    vec3 color = vec3(circ);
}

```

```
gl_FragColor = vec4( color, 1.0 );
}
```

El degradado permite ver el efecto de la rotación del círculo, que como ya sabemos se basa en cálculo matricial. Una rotación 2D, dependiente del tiempo, se aplica en el *shader Dibuja13.gsl*, ver listado 9.20, que incluye la función *rotate2d* para el cálculo de la matriz de rotación.

Listado 9.20: Dibujando un círculo que rota (Dibuja13.gsl)

```
#ifndef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

mat2 rotate2d(float _angle){
    return mat2(cos(_angle),-sin(_angle),
               sin(_angle),cos(_angle));
}

void main() {
    //Normalizamos coordenadas con la resolución
    vec2 st = gl_FragCoord.xy/u_resolution;
    float pct = 0.0;

    // Mueve espacio al vec2(0.0)
    st -= vec2(0.5);
    // Rota el espacio, variando el ángulo en función del tiempo
    st = rotate2d( sin(u_time)*PI ) * st;
    // Recoloca el espacio
    st += vec2(0.5);

    // Círculo
    pct = distance(st,vec2(0.5));
    float circ = step(0.3,0.5-pct)*st.x;

    vec3 color = vec3(circ);

    gl_FragColor = vec4( color, 1.0 );
}
```

El escalado de forma similar, requiere una multiplicación de *st* por un valor, un ejemplo sencillo se presenta en el *shader Dibuja14.gsl*, ver listado 9.21.

Listado 9.21: Dibujando un círculo que cambia de tamaño (Dibuja14.gsl)

```
#ifndef GL_ES
```

```

precision mediump float;
#endif

#define PI 3.14159265359

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

mat2 scale(vec2 _scale){
    return mat2(_scale.x,0.0,
                0.0,_scale.y);
}

void main() {
    //Normalizamos coordenadas con la resolución
    vec2 st = gl_FragCoord.xy/u_resolution;
    float pct = 0.0;

    // Mueve espacio al vec2(0.0)
    st -= vec2(0.5);
    // Escala el espacio, variando el ángulo en función del tiempo
    st = scale( vec2(sin(u_time)+1.0) ) * st;
    // Recoloca el espacio
    st += vec2(0.5);

    // Círculo
    pct = distance(st,vec2(0.5));
    float circ = step(0.3,0.5-pct)*st.x;

    vec3 color = vec3(circ);

    gl_FragColor = vec4( color, 1.0 );
}

```

Finalizar indicando que otras formas requieren de la concepción de su generación procedural, en el listado 9.22 se propone la creación de una cruz, combinando recuadros tal y como se proponía anteriormente con la función *step*.

Listado 9.22: Dibujando una cruz (Dibuja15.glsl)

```

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float cuadro(vec2 _st, float th_w, float th_h){
    // step devuelve 1.0 (blanco) 0.0 (negro).
    float left = step(th_w,_st.x); // st.x>0.1
    float bottom = step(th_h,_st.y); // st.y>0.1
}

```

```

float right = step(th_w,1.0-_st.x); // st.x<0.9
float top = step(th_h,1.0-_st.y); // st.y<0.9

return left*bottom*right*top;
}

void main() {
//Normalizamos coordenadas con la resolución
vec2 st = gl_FragCoord.xy/u_resolution;
vec3 color = vec3(0.0);

// Multiplicar equivale a AND
color = vec3( cuadro(st,0.4,0.1) + cuadro(st,0.1,0.4) );

gl_FragColor = vec4(color,1.0);
}

```

En [Gonzalez Vivo and Lowe \[Accedido Marzo 2020\]](#) se propone el uso de las coordenadas angulares/polares para la creación del diversas formas. El código del listado 9.23 se debe a Andrew Baldwin⁷ y crea polígonos regulares definido N.

Listado 9.23: Polígonos regulares (Dibuja16.glsl)

```

#ifndef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359
#define TWO_PI 6.28318530718

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

// Reference to
// http://thndl.com/square-shaped-shaders.html

void main() {
vec2 st = gl_FragCoord.xy/u_resolution.xy;
st.x *= u_resolution.x/u_resolution.y;
vec3 color = vec3(0.0);
float d = 0.0;

// Remap the space to -1. to 1.
st = st *2.-1.;

// Number of sides of your shape
int N = 3;

// Angle and radius from the current pixel

```

⁷<http://thndl.com/square-shaped-shaders.html>

```

float a = atan(st.x, st.y)+PI;
float r = TWO_PI/float(N);

// Shaping function that modulate the distance
d = cos(floor(.5+a/r)*r-a)*length(st);

color = vec3(1.0-smoothstep(.4, .41, d));
// color = vec3(d);

gl_FragColor = vec4(color, 1.0);
}

```

Es un punto de partida para experimentar, prueba las propuestas de los listados [9.24](#), [s 9.25](#) y [s 9.26](#).

Listado 9.24: Sectores angulares que giran (Dibuja17.gls)

```

#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359
#define TWO_PI 6.28318530718

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

mat2 rotate2d(float _angle){
    return mat2(cos(_angle), -sin(_angle),
               sin(_angle), cos(_angle));
}

void main(){
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    st.x *= u_resolution.x/u_resolution.y;
    vec3 color = vec3(0.0);
    float d = 0.0;

    // Redimensiona al espacio -1,1
    st = st *2. -1.;

    // Rota el espacio, variando el ángulo en función del tiempo
    st = rotate2d( sin(u_time)*PI ) * st;

    // Número de sectores
    int N = 12;

    // Ángulo y radio del píxel actual
    float a = atan(st.x, st.y)+PI;
    float r = TWO_PI/float(N);

    // Color basado en módulo
    color = vec3(mod(a, r));
}

```

```

gl_FragColor = vec4(color,1.0);
}

```

Listado 9.25: Dibujando una cruz (Dibuja18.gls)

```

#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359
#define TWO_PI 6.28318530718

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

mat2 rotate2d(float _angle){
    return mat2(cos(_angle),-sin(_angle),
               sin(_angle),cos(_angle));
}

void main(){
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    st.x *= u_resolution.x/u_resolution.y;
    vec3 color = vec3(0.0);
    float d = 0.0;

    // Redimensiona al espacio -1,1
    st = st *2.-1.;

    // Rota el espacio, variando el ángulo en función del tiempo
    st = rotate2d( sin(u_time)*PI ) * st;

    // Número de sectores
    int N = 12;

    // Ángulo y radio del píxel actual
    float a = atan(st.x,st.y)+PI;
    float r = TWO_PI/float(N);

    // Distancia afectada por el módulo del sector
    d = cos(floor(.5+a/r)*r-a)*(length(st)+mod(a,r));

    color = vec3(1.0-smoothstep(.4,.41,d));

    gl_FragColor = vec4(color,1.0);
}

```

Listado 9.26: Polígono regular modificado (Dibuja19.gls)

```

#ifdef GL_ES
precision mediump float;

```

```
#endif

#define PI 3.14159265359
#define TWO_PI 6.28318530718

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

mat2 rotate2d(float _angle){
    return mat2(cos(_angle),-sin(_angle),
               sin(_angle),cos(_angle));
}

void main(){
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    st.x *= u_resolution.x/u_resolution.y;
    vec3 color = vec3(0.0);
    float d = 0.0;

    // Redimensiona al espacio -1,1
    st = st *2.-1.;

    // Rota el espacio, variando el ángulo en función del tiempo
    st = rotate2d( sin(u_time)*PI ) * st;

    // Número de sectores
    int N = 12;

    // Ángulo y radio del píxel actual
    float a = atan(st.x,st.y)+PI;
    float r = TWO_PI/float(N);

    // Afectamos de forma diferente sectores pares e impares
    if (floor(mod(a,2.))>=1.)
        d = cos(floor(.5+a/r)*r-a)*length(st);
    else
        d = cos(floor(.5+a/r)*r-a)*length(st)*2.;

    color = vec3(1.0-smoothstep(.4,.41,d));

    gl_FragColor = vec4(color,1.0);
}
```

9.2.3. Generativos

Al ejecutarse en paralelo para cada píxel en una GPU, el número de repeticiones no influye en el coste, siendo una potente herramienta para crear patrones. Tras la breve muestra de dibujo de formas gráficas con técnicas procedimentales del apartado anterior, el *shader*

Dibuja20.gls, ver el listado 9.27, aprovecha el escalado para replicar nueve veces un círculo (número de repeticiones configurable modificando el valor de *scale*). La función *fract* permite *move* entre celdas de la rejilla resultante. La función utilizada para dibujar el círculo, se basa en la propuesta en [Gonzalez Vivo and Lowe \[Accedido Marzo 2020\]](#) que evita el uso de la costosa *sqrt* en el cálculo de distancias, utilizando *dot*, y una transición suave con *smoothstep*.

Listado 9.27: Patrón basado en círculos (*Dibuja20.gls*)

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;

float scale = 3.0;

float circle(in vec2 _st, in float _radius){
    vec2 l = _st-vec2(0.5);
    return 1.-smoothstep(_radius-(radius*0.01),
                        _radius+(radius*0.01),
                        dot(l,l)*4.0);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution;
    vec3 color = vec3(0.0);

    //Escala en función de la escala adoptada
    st = fract(st*scale);

    // Dibuja un círculo en cada "celda"
    color = vec3(circle(st,0.5));

    gl_FragColor = vec4(color,1.0);
}
```

Se abre un mundo de posibilidades, ya que se dispone de mecanismos para diferenciar cada copia o celda, pudiendo aplicar transformaciones, o animaciones de su forma, color y posición. Una posibilidad es desplazar la posición de la forma en cada fila como se hace en el *shader Dibuja21.gls*, ver el listado 9.28, que hace uso de la función *mod* para determinar si la fila es par o no, además de animar el tamaño de cada círculo.

Listado 9.28: Patrón basado en círculos desplazados (*Dibuja21.gls*)

```
#ifdef GL_ES
precision mediump float;
#endif
```

```

uniform vec2 u_resolution;
uniform float u_time;

float scale = 3.0;

float circle(in vec2 _st, in float _radius){
    vec2 l = _st-vec2(0.5);
    return 1.-smoothstep(_radius-(radius*0.01),
                        _radius+(radius*0.01),
                        dot(l,l)*4.0);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution;
    vec3 color = vec3(0.0);

    // Traslación dependiente de st.y
    if (floor(mod(st.y*scale,2.0)) == 1. )
        st.x += 0.5;

    //Escala en función de la escala adoptada
    st = fract(st*scale);

    // Dibuja un círculo en cada "celda"
    float pct = max(-sin(u_time),sin(u_time));
    color = vec3(circle(st,0.5*pct));

    gl_FragColor = vec4(color,1.0);
}

```

En el *shader Dibuja22.gsl*, ver el listado 9.29, se introduce un efecto adicional de animación trasladando únicamente los círculos de las filas/columnas pares.

Listado 9.29: Patrón basado en círculos que se animan en filas/columnas pares (Dibuja22.gsl)

```

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;

float scale = 5.0;

float circle(in vec2 _st, in float _radius){
    vec2 l = _st-vec2(0.5);
    return 1.-smoothstep(_radius-(radius*0.01),
                        _radius+(radius*0.01),
                        dot(l,l)*4.0);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution;
    vec3 color = vec3(0.0);

```

```

// Traslación de filas pares
float off = sin(u_time);
if (sign(off)>0.)
{
    if ( floor(mod(st.y*scale,2.0)) == 1.)
        st.x += off;
}
else
{
    if ( floor(mod(st.x*scale,2.0)) == 1.)
        st.y += off;
}

//Escala en función de la escala adoptada
st = fract(st*scale);

// Dibuja un círculo en cada "celda"
float pct = max(-sin(u_time),sin(u_time));
color = vec3(circle(st,0.5*pct));

gl_FragColor = vec4(color,1.0);
}

```

Seguimos animando en el *shader Dibuja23.glsl*, ver el listado 9.30, moviendo el patrón a lo largo de los ejes x e y en función del tiempo.

Listado 9.30: Patrón basado en círculos animados de distinta escala y movimiento (Dibuja23.glsl)

```

#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.141592

uniform vec2 u_resolution;
uniform float u_time;

float scale = 3.0;

float circle(in vec2 _st, in float _radius){
    vec2 l = _st-vec2(0.5);
    return 1.-smoothstep(_radius-(radius*0.01),
        _radius+(radius*0.01),
        dot(l,l)*4.0);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution;
    vec3 color = vec3(0.0);

    st.x +=max(0., sin(u_time));
    st.y -=max(0., -sin(u_time));
}

```

```

//Escala en función de la escala adoptada
st = fract(st*scale);

// Dibuja un círculo en cada "celda"
float pct = max(-sin(u_time), sin(u_time));
color = vec3(circle(st,0.5*pct));

gl_FragColor = vec4(color,1.0);
}

```

Finalizamos esta serie con el *shader Dibuja24.gsl*, ver el listado 9.31, que modifica el tamaño del círculo en función de la columna, además de mantener la modificación de su tamaño en función del tiempo.

Listado 9.31: Patrón basado en círculos animados de distinta escala (Dibuja24.gsl)

```

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;

float scale = 3.0;

float circle(in vec2 _st, in float _radius){
    vec2 l = _st-vec2(0.5);
    return 1.-smoothstep(_radius-(radius*0.01),
        _radius+(radius*0.01),
        dot(l,l)*4.0);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution;
    vec3 color = vec3(0.0);

    float off = floor(st.x*scale);

    //Escala en función de la escala adoptada
    st = fract(st*scale*(scale-off));

    // Dibuja un círculo en cada "celda"
    float pct = max(-sin(u_time), sin(u_time));
    color = vec3(circle(st,0.5*pct));

    gl_FragColor = vec4(color,1.0);
}

```

Una interesante posibilidad es el uso de aleatoriedad a distintos niveles en la creación del contenido gráfico. En el *shader Dibuja30.gsl*, ver el listado 9.32, se replica la propuesta de [Gonzalez Vivo and Lowe \[Accedido Marzo 2020\]](#) para el uso de valores aleatorios en 2D,

dada la ausencia de funciones en GLSL.

Listado 9.32: Rejilla con tonos aleatorios (Dibuja30.gsl)

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float random (vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233)))*
                43758.5453123);
}

float res = 10.0;

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;

    st *= res; // Escalado
    vec2 ipos = floor(st); // Parte entera
    vec2 fpos = fract(st); // Parte fraccionaria

    // Aleatorio basado en parte entera
    vec3 color = vec3(random( ipos ));

    gl_FragColor = vec4(color,1.0);
}
```

La aplicación de la aleatoriedad es amplia, en el *shader Dibuja31.gsl*, ver el listado 9.33, tomado de la misma fuente, se crea un patrón con cuatro variantes de orientaciones posibles.

Listado 9.33: Patrón Truchet (Dibuja31.gsl)

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float random (vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233)))*
                43758.5453123);
}

vec2 truchetPattern(in vec2 _st, in float _index){
```

```

_index = fract (((_index-0.5)*2.0));
if (_index > 0.75) {
    _st = vec2(1.0) - _st;
} else if (_index > 0.5) {
    _st = vec2(1.0-_st.x,_st.y);
} else if (_index > 0.25) {
    _st = 1.0-vec2(1.0-_st.x,_st.y);
}
return _st;
}

float res = 10.0;

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;

    st *= res; // Escalado
    vec2 ipos = floor(st); // Parte entera
    vec2 fpos = fract(st); // Parte fraccionaria

    // Patrón aleatorio, cuatro opciones
    vec2 tile = truchetPattern(fpos, random( ipos ));

    // Triángulo según tipo
    float color = 0.0;
    color = step(tile.x, tile.y);

    gl_FragColor = vec4(vec3(color),1.0);
}

```

Los valores aleatorios presentan una frecuencia muy alta, siendo excesiva para diversos propósitos. Las funciones de ruido se adaptan mejor al suavizar las frecuencias presentes en la salida resultante. Destacar en primer lugar el ruido de Perlin aplicado en el *shader Dibuja32.gsl*, ver el listado 9.34, una vez disponible una rejilla, se interpola en posiciones intermedias.

Listado 9.34: Rejilla con tonos aleatorios (Dibuja32.gsl)

```

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

// Valor aleatorio en 2D
float random (in vec2 st) {
    return fract(sin(dot(st.xy,
                        vec2(12.9898,78.233)))
                * 43758.5453123);
}

```

```

// 2D Noise based on Morgan McGuire (morgan3d)
// https://www.shadertoy.com/view/4dS3Wd
float noise (in vec2 st) {
    vec2 i = floor(st);
    vec2 f = fract(st);

    // Cuatro esquinas de cada mosaico
    float a = random(i);
    float b = random(i + vec2(1.0, 0.0));
    float c = random(i + vec2(0.0, 1.0));
    float d = random(i + vec2(1.0, 1.0));

    // Interpolación con Hemite cúbico
    vec2 u = smoothstep(0.,1.,f);

    // Combina esquinas ponderadas
    return mix(a, b, u.x) +
           (c - a)* u.y * (1.0 - u.x) +
           (d - b) * u.x * u.y;
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;

    // Escalado
    vec2 pos = vec2(st*5.0);

    // Función de ruido
    float n = noise(pos);

    gl_FragColor = vec4(vec3(n), 1.0);
}

```

El previo es un ejemplo de ruido por valor, cuyo resultado sugiere la presencia de una rejilla. El *shader Dibuja33.gsl*, ver el listado 9.35, al utilizar ruido por gradiente mejora el resultado.

Listado 9.35: Rejilla con tonos aleatorios (Dibuja33.gsl)

```

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

// Valor aleatorio en 2D
vec2 random2(vec2 st){
    st = vec2( dot(st,vec2(127.1,311.7)),

```

```

        dot(st, vec2(269.5, 183.3)) );
    return -1.0 + 2.0*fract(sin(st)*43758.5453123);
}

// Gradient Noise by Inigo Quilez - iq/2013
// https://www.shadertoy.com/view/XdXGW8
float noise(vec2 st) {
    vec2 i = floor(st);
    vec2 f = fract(st);

    // Interpolación con Hermite cúbico
    //vec2 u = smoothstep(0., 1., f); // Equivalente
    vec2 u = f*f*(3.0-2.0*f);

    return mix( mix( dot( random2(i + vec2(0.0,0.0)) ), f - vec2(0.0,0.0) ),
                dot( random2(i + vec2(1.0,0.0)) ), f - vec2(1.0,0.0) ), u.x),
            mix( dot( random2(i + vec2(0.0,1.0)) ), f - vec2(0.0,1.0) ),
                dot( random2(i + vec2(1.0,1.0)) ), f - vec2(1.0,1.0) ), u.x), u.y);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(0.0);

    // Escalado
    vec2 pos = vec2(st*5.0);

    // Función de ruido
    color = vec3( noise(pos)*.5+.5 );

    gl_FragColor = vec4(color, 1.0);
}

```

Propuestas de funciones de ruido más recientes son el ruido celular (Steven Worley) y el simplex (Ken Perlin), ver para más detalles [Gonzalez Vivo and Lowe \[Accedido Marzo 2020\]](#).

9.2.4. Imágenes

Para la modificación de imágenes por medio de un *shader* se hace uso de texturas, que se reciben desde la CPU como variables *uniform sampler2D*, permitiendo acceder por coordenadas al color. En esta sección se presentan varios ejemplos de los incluidos en Processing para mostrar posibles alteraciones sobre imágenes.

Un primer ejemplo *Ejemplos->Topics->Shaders->Blurfilter*, aplica un filtro de desenfoque (*blur*) combinando en la operación los valores del propio píxel y los 8 vecinos en coordenadas de texturas. El código Processing se muestra en el listado 9.36.

Listado 9.36: Ejemplo *Blurfilter* (p9_blur)

```
/**
```

```

* Blur Filter
*
* Change the default shader to apply a simple, custom blur filter.
*
* Press the mouse to switch between the custom and default shader.
*/

PShader blur;

void setup() {
  size(640, 360, P2D);
  blur = loadShader("blur.glsl");
  stroke(255, 0, 0);
  rectMode(CENTER);
}

void draw() {
  filter(blur);
  rect(mouseX, mouseY, 150, 150);
  ellipse(mouseX, mouseY, 100, 100);
}

```

Destaca la utilización de la función *filter* que se aplican sobre la ventana de visualización haciendo uso de filtros predefinidos o *shaders* como en este caso. Al ejecutar, el resultado es que se emborronan recuadros previos, mostrando el último de forma nítida.

Antes de comprender el código del *shader*, se debe recordar que al igual que en ejemplos previos únicamente se define el *shader* de fragmentos, en dicha situación Processing adopta un *shader* de vértices por defecto de tipo textura (sin luces), debiendo el *shader* de fragmentos seguir los nombres de las variables *varying* definidas en dicho *shader* de vértices por defecto para el color del vértice y la coordenada de textura: respectivamente *vertColor* y *vertTexCoord*.

Tras esta aclaración, es el momento de observar el *shader* de fragmentos se muestra en el listado 9.37. Conocidas las coordenadas de textura de un fragmento *vertTexCoord*, se muestrean los vecinos (o texels) haciendo uso de la variable *uniform texOffset*, que especifica el desplazamiento para moverse entre vecinos en el mapa de textura. Es una variable fijada por Processing, con valores $(1/width, 1/height)$ siendo *width* y *height* la resolución de la textura. De esta forma $vertTexCoord.st + vec2(texOffset.s, 0)$ se refiere al texel colocado una posición a la derecha. De esta forma, en este ejemplo concreto, el color del fragmento se calcula a partir de la ponderación del kernel 3×3

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

que incluye al propio fragmento y sus 8 vecinos. Primeramente se obtienen las coordenadas para cada fragmento en la textura, y posteriormente se recupera el color en cada una de ellas,

para finalmente aplicar el kernel.

Listado 9.37: Código *shader* blur.glsl

```

#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif

#define PROCESSING_TEXTURE_SHADER

uniform sampler2D texture;
uniform vec2 texOffset;

varying vec4 vertColor;
varying vec4 vertTexCoord;

void main(void) {
    // Grouping texcoord variables in order to make it work in the GMA 950. See post #13
    // in this thread:
    // http://www.idevgames.com/forums/thread-3467.html
    vec2 tc0 = vertTexCoord.st + vec2(-texOffset.s, -texOffset.t);
    vec2 tc1 = vertTexCoord.st + vec2(0.0, -texOffset.t);
    vec2 tc2 = vertTexCoord.st + vec2(+texOffset.s, -texOffset.t);
    vec2 tc3 = vertTexCoord.st + vec2(-texOffset.s, 0.0);
    vec2 tc4 = vertTexCoord.st + vec2(0.0, 0.0);
    vec2 tc5 = vertTexCoord.st + vec2(+texOffset.s, 0.0);
    vec2 tc6 = vertTexCoord.st + vec2(-texOffset.s, +texOffset.t);
    vec2 tc7 = vertTexCoord.st + vec2(0.0, +texOffset.t);
    vec2 tc8 = vertTexCoord.st + vec2(+texOffset.s, +texOffset.t);

    vec4 col0 = texture2D(texture, tc0);
    vec4 col1 = texture2D(texture, tc1);
    vec4 col2 = texture2D(texture, tc2);
    vec4 col3 = texture2D(texture, tc3);
    vec4 col4 = texture2D(texture, tc4);
    vec4 col5 = texture2D(texture, tc5);
    vec4 col6 = texture2D(texture, tc6);
    vec4 col7 = texture2D(texture, tc7);
    vec4 col8 = texture2D(texture, tc8);

    vec4 sum = (1.0 * col0 + 2.0 * col1 + 1.0 * col2 +
                2.0 * col3 + 4.0 * col4 + 2.0 * col5 +
                1.0 * col6 + 2.0 * col7 + 1.0 * col8) / 16.0;
    gl_FragColor = vec4(sum.rgb, 1.0) * vertColor;
}

```

El listado 9.38 adapta el ejemplo incluido en Processing (*Ejemplos->Topics->Shaders->EdgeDetect*) para procesar la imagen captada por la webcam, aplicando en este caso una operación que muestra los bordes o contornos presentes en la imagen. La estructura del

shader de fragmentos es similar al ejemplo previo, alterando básicamente el kernel aplicado

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

como se muestra en el listado 9.39. En este caso no aplica la función *filter*, sino que activa el *shader*. Para utilizar la función *filter*, se lanzaría tras mostrar la imagen en la ventana.

Listado 9.38: Detección de bordes de la cámara (p9_bordes)

```
/**
 * Edge Detection
 *
 * Change the default shader to apply a simple, custom edge detection filter.
 *
 * Press the mouse to switch between the custom and default shader.
 */

PShader edges;
import processing.video.*;

Capture cam;
PImage img;
boolean enabled = true;

void setup() {
  size(640, 480, P2D);
  //Cámara
  cam = new Capture(this, width, height);
  cam.start();

  edges = loadShader("edges.glsl");
}

void draw() {
  if (enabled == true) {
    shader(edges);
  }

  if (cam.available()) {
    background(0);
    cam.read();
    image(cam, 0, 0);
  }
}

void mousePressed() {
  enabled = !enabled;
  if (!enabled == true) {
    resetShader();
  }
}
```

}

Listado 9.39: Código *shader* p9_bordes

```

#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif

#define PROCESSING_TEXTURE_SHADER

uniform sampler2D texture;
uniform vec2 texOffset;

varying vec4 vertColor;
varying vec4 vertTexCoord;

void main(void) {
    // Grouping texcoord variables in order to make it work in the GMA 950. See post #13
    // in this thread:
    // http://www.idevgames.com/forums/thread-3467.html
    vec2 tc0 = vertTexCoord.st + vec2(-texOffset.s, -texOffset.t);
    vec2 tc1 = vertTexCoord.st + vec2(0.0, -texOffset.t);
    vec2 tc2 = vertTexCoord.st + vec2(+texOffset.s, -texOffset.t);
    vec2 tc3 = vertTexCoord.st + vec2(-texOffset.s, 0.0);
    vec2 tc4 = vertTexCoord.st + vec2(0.0, 0.0);
    vec2 tc5 = vertTexCoord.st + vec2(+texOffset.s, 0.0);
    vec2 tc6 = vertTexCoord.st + vec2(-texOffset.s, +texOffset.t);
    vec2 tc7 = vertTexCoord.st + vec2(0.0, +texOffset.t);
    vec2 tc8 = vertTexCoord.st + vec2(+texOffset.s, +texOffset.t);

    vec4 col0 = texture2D(texture, tc0);
    vec4 col1 = texture2D(texture, tc1);
    vec4 col2 = texture2D(texture, tc2);
    vec4 col3 = texture2D(texture, tc3);
    vec4 col4 = texture2D(texture, tc4);
    vec4 col5 = texture2D(texture, tc5);
    vec4 col6 = texture2D(texture, tc6);
    vec4 col7 = texture2D(texture, tc7);
    vec4 col8 = texture2D(texture, tc8);

    vec4 sum = 8.0 * col4 - (col0 + col1 + col2 + col3 + col5 + col6 + col7 + col8);
    gl_FragColor = vec4(sum.rgb, 1.0) * vertColor;
}

```

9.3. *Shaders* DE VÉRTICES

Los ejemplos previos no hacen uso del *shader* de vértices en absoluto, por lo que Processing toma unel *shader* por defecto. Para dar algunas pinceladas sobre las posibilidades del

shader de vértices, en esta sección se adopta en primer término el ejemplo *File->Examples->Topics->Shaders->ToonShader*, cuyo código adaptado se muestra en el listado 9.40.

Listado 9.40: Código ejemplo *ToonShader* p9_bordes_toon

```
/**
 * Adaptado desde ejemplo Toon Shading.
 *
 * Example showing the use of a custom lighting shader in order
 * to apply a "toon" effect on the scene. Based on the glsl tutorial
 * from lighthouse 3D:
 * http://www.lighthouse3d.com/tutorials/glsl-tutorial/toon-shader-version-ii/
 */

PShader toon1, toon2;
int modo = 0, modomax = 2;

void setup() {
  size(640, 360, P3D);
  noStroke();
  fill(204);
  toon2 = loadShader("ToonFrag.glsl", "ToonVert.glsl");
  toon1 = loadShader("ToonFrag1.glsl", "ToonVert.glsl");
}

void draw() {
  if (modo == 1) {
    shader(toon1);
  }
  else {
    if (modo == 2) {
      shader(toon2);
    }
  }

  //Dibuja esfera
  noStroke();
  background(0);
  float dirY = (mouseY / float(height) - 0.5) * 2;
  float dirX = (mouseX / float(width) - 0.5) * 2;
  directionalLight(204, 204, 204, -dirX, -dirY, -1);
  translate(width/2, height/2);
  sphere(120);
}

void mousePressed() {
  //Si había un shader activo, lo resetea
  if (modo>0) resetShader();

  modo++;
  if (modo > modomax) modo = 0;
}
```

En el *setup* se observa que la llamada a *loadShader* hace referencia a dos archivos. el

primero de ellos es el *shader* de fragmentos, ver variantes en listados 9.41 y 9.42, mientras que el segundo es el *shader* de vértices, ver listado 9.43. El modo de visualización se cambia haciendo clic con el ratón.

Listado 9.41: Versión original del *shader* de fragmentos (ToonFrag.glsl)

```
#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif

varying vec3 vertNormal;
varying vec3 vertLightDir;

void main() {
    float intensity;
    vec4 color;
    // Producto escalar normal y vector hacia la fuente de luz
    intensity = max(0.0, dot(vertLightDir, vertNormal));

    // Cuatro tonos posibles de iluminación
    if (intensity > 0.95) {
        color = vec4(1.0, 0.5, 0.5, 1.0);
    } else if (intensity > 0.5) {
        color = vec4(0.6, 0.3, 0.3, 1.0);
    } else if (intensity > 0.25) {
        color = vec4(0.4, 0.2, 0.2, 1.0);
    } else {
        color = vec4(0.2, 0.1, 0.1, 1.0);
    }

    gl_FragColor = color;
}
```

Listado 9.42: Segunda versión del *shader* de fragmentos (ToonFrag1.glsl)

```
#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif

varying vec3 vertNormal;
varying vec3 vertLightDir;

void main() {
    float intensity;
    // Producto escalar normal y vector hacia la fuente de luz
    intensity = max(0.0, dot(vertLightDir, vertNormal));

    // Color con únicamente reflexión difusa
    gl_FragColor = vec4(vec3(intensity),1.);
}
```

Al observar los *shaders*, en el de fragmentos aparece la variable ya conocida *gl_FragColor*, que almacena la información del color de cada fragmento. En el de vértices está la variable reservada *gl_Position* que contiene la posición del vértice. Por otro lado, existen dos variables de tipo *varying* compartidas entre ambos *shaders*: *vertNormal* y *vertLightDir*. Básicamente el *shader* de vértices las pasa al de fragmentos que realiza el cálculo del nuevo color a asignar al píxel. Se presentan dos variantes del *shader* de fragmentos para mostrar dos posible usos de la información recibida. Para cambiar entre ellas debe hacerse clic con el ratón.

Listado 9.43: *Shader* de vértices (ToonVert.glsl)

```
// Toon shader using per-pixel lighting. Based on the glsl
// tutorial from lighthouse 3D:
// http://www.lighthouse3d.com/tutorials/glsl-tutorial/toon-shader-version-ii/

#define PROCESSING_LIGHT_SHADER

uniform mat4 modelview;
uniform mat4 transform;
uniform mat3 normalMatrix;

uniform vec3 lightNormal[8];

attribute vec4 vertex;
attribute vec3 normal;

varying vec3 vertNormal;
varying vec3 vertLightDir;

void main() {
    // Vertex in clip coordinates
    gl_Position = transform * vertex;

    // Normal vector in eye coordinates is passed
    // to the fragment shader
    vertNormal = normalize(normalMatrix * normal);

    // Assuming that there is only one directional light.
    // Its normal vector is passed to the fragment shader
    // in order to perform per-pixel lighting calculation.
    vertLightDir = -lightNormal[0];
}
```

En resumen, no hay grandes novedades tras definir el *shader* de vértices. Para mostrar más opciones del *shader* de vértices, los siguientes ejemplos reproducen el prototipo basado en el dibujado de un cilindro del tutorial Colubri [Accedido Febrero 2019]. Un código levemente modificado para esta sección es el presentado en el listado 9.44.

Listado 9.44: Dibujo de un cilindro (p9_shader_cilindro)

```
PShape can;
```

```
float angle;
PShader shd;
PImage img;

void setup() {
  size(640, 360, P3D);
  img = loadImage("logoulpgc.png");
  can = createCan(100, 200, 32);
  shd = loadShader("ColorFrag0.glsl", "ColorVert0.glsl");
  //shd = loadShader("ColorFrag1.glsl", "ColorVert1.glsl");
  /shd = loadShader("TexFrag0.glsl", "TexVert0.glsl");
  //shd = loadShader("TexFrag1.glsl", "TexVert1.glsl");
  //shd = loadShader("LightFrag0.glsl", "LightVert0.glsl");
  //shd = loadShader("LightFrag1.glsl", "LightVert1.glsl");
  //shd = loadShader("TexLightFrag0.glsl", "TexLightVert0.glsl");
  //shd = loadShader("TexLightFrag1.glsl", "TexLightVert1.glsl");
}

void draw() {
  background(0);

  shader(shd);
  shd.set("u_time", float(millis())/float(1000));

  pointLight(255, 255, 255, width/2, height, 200);

  translate(width/2, height/2);
  rotateY(angle);
  shape(can);
  angle += 0.01;
}

PShape createCan(float r, float h, int detail) {
  textureMode(NORMAL);
  PShape sh = createShape();
  sh.beginShape(QUAD_STRIP);
  sh.noStroke();
  sh.texture(img);
  for (int i = 0; i <= detail; i++) {
    float angle = TWO_PI / detail;
    float x = sin(i * angle);
    float z = cos(i * angle);
    float u = float(i) / detail;
    sh.normal(x, 0, z);
    sh.vertex(x * r, -h/2, z * r, u, 0);
    sh.vertex(x * r, +h/2, z * r, u, 1);
  }
  sh.endShape();
  return sh;
}
```

El código incluye la llamada a sus *shaders* de vértices y fragmentos. En el caso de utilizar *shader* de color (sin textura ni luces), un ejemplo mínimo con ambos *shaders* se muestra en

el listado 9.46.

Listado 9.45: Shader de color (ColorFrag0.glsl y ColorVert0.glsl)

```
ColorVert0.glsl:

uniform mat4 transform;

attribute vec4 position;
attribute vec4 color;

varying vec4 vertColor;

void main() {
    gl_Position = transform * position;
    vertColor = color;
}

ColorFrag0.glsl:

#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif

varying vec4 vertColor;

void main() {
    gl_FragColor = vertColor;
}
```

El *shader* de vértices calcula dos valores, posición y color de entrada, que recoge el *shader* de fragmentos sin realizar modificación alguna. La posición se calcula tras multiplicar por la única variable *uniform*, la matriz 4×4 *transform*, que combina las matrices de proyección y modelo. El producto por la coordenadas del mundo proporciona las coordenadas transformadas y proyectadas. Pese a que el código Processing, la forma tiene una textura asignada, no se muestra, dado que Processing detecta que el *shader* es de color (sin texturas ni luces).

Si enviamos información del tiempo transcurrido al *shader* descomentando en el *draw* la línea `shd.set("ü_time", float(millis())/float(1000));`, y adoptando los *shaders* del listado 9.46, variamos el color del objeto como ya hemos visto en secciones anteriores desde el *shader* de fragmentos.

Listado 9.46: Shader de color (ColorFrag0.glsl y ColorVert0.glsl)

```
ColorVert1.glsl:

uniform mat4 transform;

attribute vec4 position;
attribute vec4 color;
```

```
varying vec4 vertColor;

void main() {
    gl_Position = transform * position;
    vertColor = color;
}

ColorFrag1.glsl:

#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif

uniform float u_time;

varying vec4 vertColor;

void main() {
    if (sign(sin(u_time*10.)) == 1)
        // Copia color del objeto
        gl_FragColor = vertColor;
    else
        // Invierte el color
        gl_FragColor = vec4(vec3(1) - vertColor.xyz, 1);
}
```

Modificando la llamada a los *shaders* para adoptar un ejemplo mínimo de *shader* de textura (sin luces) mostrado en el listado 9.47.

Listado 9.47: *Shader* de textura (TexVert0.glsl y TexFrag0.glsl)

```
TexVert0.glsl:

uniform mat4 transform;
uniform mat4 texMatrix;

attribute vec4 position;
attribute vec4 color;
attribute vec2 texCoord;

varying vec4 vertColor;
varying vec4 vertTexCoord;

void main() {
    gl_Position = transform * position;

    vertColor = color;
    vertTexCoord = texMatrix * vec4(texCoord, 1.0, 1.0);
}

TexFrag0.glsl:
```

```

#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif

uniform sampler2D texture;

varying vec4 vertColor;
varying vec4 vertTexCoord;

void main() {
    // Combina color del objeto con la textura
    gl_FragColor = texture2D(texture, vertTexCoord.st) * vertColor;
}

```

La novedad es la variable *uniform texMatrix* que transforma convenientemente (escala e inversión de la coordenada y) la textura para cada vértice. El *shader* de fragmentos incluye el puntero a la imagen de la textura remitida desde Processing. Un efecto de pixelado sobre la textura se obtiene modificando el *shader* de fragmentos, ver el listado 9.48.

Listado 9.48: *Shader* de textura (TexVert1.glsl y TexFrag1.glsl)

```

TexVert0.glsl:

uniform mat4 transform;
uniform mat4 texMatrix;

attribute vec4 position;
attribute vec4 color;
attribute vec2 texCoord;

varying vec4 vertColor;
varying vec4 vertTexCoord;

void main() {
    gl_Position = transform * position;

    vertColor = color;
    vertTexCoord = texMatrix * vec4(texCoord, 1.0, 1.0);
}

TexFrag0.glsl:

#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif

uniform sampler2D texture;

varying vec4 vertColor;
varying vec4 vertTexCoord;

```

```

void main() {
    // Efecto de pixelado
    int si = int(vertTexCoord.s * 50.0);
    int sj = int(vertTexCoord.t * 50.0);
    gl_FragColor = texture2D(texture, vec2(float(si) / 50.0, float(sj) / 50.0)) * vertColor;
}vertTexCoord.st) * vertColor;
}

```

El siguiente ejemplo hace uso del *shader* de luces, que va a permitir especificar tu propio algoritmo de reproducción (*rendering*). Un modelo sencillo se muestra en el listado 9.48, cuyo *shader* se basa en el ángulo entre la normal del vértice y la luz. *ecPosition* tiene las posición del vértice en coordenadas de los ojos.

Listado 9.49: *Shader* de textura (LightVert0.glsl y LightFrag0.glsl)

```

LightVert0.glsl:

uniform mat4 modelview;
uniform mat4 transform;
uniform mat3 normalMatrix;

uniform vec4 lightPosition;

attribute vec4 position;
attribute vec4 color;
attribute vec3 normal;

varying vec4 vertColor;

void main() {
    gl_Position = transform * position;
    vec3 ecPosition = vec3(modelview * position);
    vec3 ecNormal = normalize(normalMatrix * normal);

    vec3 direction = normalize(lightPosition.xyz - ecPosition);
    float intensity = max(0.0, dot(direction, ecNormal));
    vertColor = vec4(intensity, intensity, intensity, 1) * color;
}

LightFrag0.glsl:

#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif

varying vec4 vertColor;

void main() {
    gl_FragColor = vertColor;
}

```

El anterior es un ejemplo con un única fuente de luz, si bien Processing admite pasar información de hasta 8 fuentes con todos sus parámetros.

En el algoritmo del ejemplo previo, se interpolaba color, en el listado 9.50 se incluyen los *shaders* correspondientes para llevar a cabo una interpolación de la normal, con la consiguiente mejora visual del resultado.

Listado 9.50: *Shader* de textura (LightVert1.glsl y LightFrag1.glsl)

```
LightVert1.glsl:

uniform mat4 modelview;
uniform mat4 transform;
uniform mat3 normalMatrix;

uniform vec4 lightPosition;
uniform vec3 lightNormal;

attribute vec4 position;
attribute vec4 color;
attribute vec3 normal;

varying vec4 vertColor;
varying vec3 ecNormal;
varying vec3 lightDir;

void main() {
    gl_Position = transform * position;
    vec3 ecPosition = vec3(modelview * position);

    ecNormal = normalize(normalMatrix * normal);
    lightDir = normalize(lightPosition.xyz - ecPosition);
    vertColor = color;
}

LightFrag1.glsl:

#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif

varying vec4 vertColor;
varying vec3 ecNormal;
varying vec3 lightDir;

void main() {
    vec3 direction = normalize(lightDir);
    vec3 normal = normalize(ecNormal);
    float intensity = max(0.0, dot(direction, normal));
    gl_FragColor = vec4(intensity, intensity, intensity, 1) * vertColor;
}
```

El último tipo de *shader* en Processing combina luz y textura, los listados 9.51 y 9.52, cierran esta sección, mostrando los *shaders* para realizar el cálculo por vértice o píxel de la iluminación.

Listado 9.51: *Shader* de textura (TexLightVert0.glsl y TexLightFrag0.glsl)

```
TexLightVert0.glsl:

uniform mat4 modelview;
uniform mat4 transform;
uniform mat3 normalMatrix;
uniform mat4 texMatrix;

uniform vec4 lightPosition;

attribute vec4 position;
attribute vec4 color;
attribute vec3 normal;
attribute vec2 texCoord;

varying vec4 vertColor;
varying vec4 vertTexCoord;

void main() {
    gl_Position = transform * position;
    vec3 ecPosition = vec3(modelview * position);
    vec3 ecNormal = normalize(normalMatrix * normal);

    vec3 direction = normalize(lightPosition.xyz - ecPosition);
    float intensity = max(0.0, dot(direction, ecNormal));
    vertColor = vec4(intensity, intensity, intensity, 1) * color;

    vertTexCoord = texMatrix * vec4(texCoord, 1.0, 1.0);
}

TexLightFrag0.glsl:

#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif

uniform sampler2D texture;

varying vec4 vertColor;
varying vec4 vertTexCoord;

void main() {
    gl_FragColor = texture2D(texture, vertTexCoord.st) * vertColor;
}
```

Listado 9.52: *Shader* de textura (TexLightVert1.glsl y TexLightFrag1.glsl)

```
TexLightVert1.glsl:
```

```
uniform mat4 modelview;
uniform mat4 transform;
uniform mat3 normalMatrix;
uniform mat4 texMatrix;

uniform vec4 lightPosition;

attribute vec4 position;
attribute vec4 color;
attribute vec3 normal;
attribute vec2 texCoord;

varying vec4 vertColor;
varying vec3 ecNormal;
varying vec3 lightDir;
varying vec4 vertTexCoord;

void main() {
    gl_Position = transform * position;
    vec3 ecPosition = vec3(modelview * position);

    ecNormal = normalize(normalMatrix * normal);
    lightDir = normalize(lightPosition.xyz - ecPosition);
    vertColor = color;

    vertTexCoord = texMatrix * vec4(texCoord, 1.0, 1.0);
}
}
TexLightFrag1.glsl:

#ifdef GL_ES
precision mediump float;
precision mediump int;
#endif

uniform sampler2D texture;

varying vec4 vertColor;
varying vec3 ecNormal;
varying vec3 lightDir;
varying vec4 vertTexCoord;

void main() {
    vec3 direction = normalize(lightDir);
    vec3 normal = normalize(ecNormal);
    float intensity = max(0.0, dot(direction, normal));
    vec4 tintColor = vec4(intensity, intensity, intensity, 1) * vertColor;
    gl_FragColor = texture2D(texture, vertTexCoord.st) * tintColor;
}
```

9.4. RECURSOS ADICIONALES

Este capítulo pretende mostrar una visión superficial sobre los *shaders*, no se han mostrado otras posibilidades como por ejemplo la carga de *shaders* distintos para cada objeto de la escena. Se incluyen en este apartado referencias a ejemplos, información y lugares de encuentro de la comunidad, referencia y ejemplos:

- Los mencionados ejemplos en Processing, ver *File->Examples->Topics->Shaders*
- [GLSL Sandbox](http://glslsandbox.com)⁸
- [Shadertoy](https://www.shadertoy.com)⁹
- [Vertex Shader Art](https://www.vertexshaderart.com)¹⁰
- [Codeanticode shader experiments](https://github.com/codeanticode/pshader-experiments)¹¹
- [OpenGL ES Shading Language Reference](http://shaderific.com/glsl/)¹²

9.5. TAREA

Realizar una propuesta de prototipo que haga uso al menos de un *shader* de fragmentos, sugiriendo que cree un diseño generativo o realice algún procesamiento sobre imagen. Será aceptable la combinación con cualquier elemento de prácticas precedentes.

La entrega se debe realizar a través del campus virtual, remitiendo un enlace a un proyecto github, cuyo README sirva de memoria, por lo que se espera que el README:

- identifique al autor,
- describa el trabajo realizado,
- argumente decisiones adoptadas para la solución propuesta,
- incluya referencias y herramientas utilizadas,
- muestre el resultado con un gif animado.

⁸<http://glslsandbox.com>

⁹<https://www.shadertoy.com>

¹⁰<https://www.vertexshaderart.com>

¹¹<https://github.com/codeanticode/pshader-experiments>

¹²<http://shaderific.com/glsl/>

BIBLIOGRAFÍA

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

Patricio Gonzalez Vivo and Jen Lowe. The book of shaders, Accedido Marzo 2020. URL <https://thebookofshaders.com/01/?lan=es>.

Práctica 10

Introducción a la programación con Arduino

10.1. 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.

NOTA: Las prácticas de esta parte van a estar condicionadas por la evolución del confinamiento.

10.1.1. 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 10.1. 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

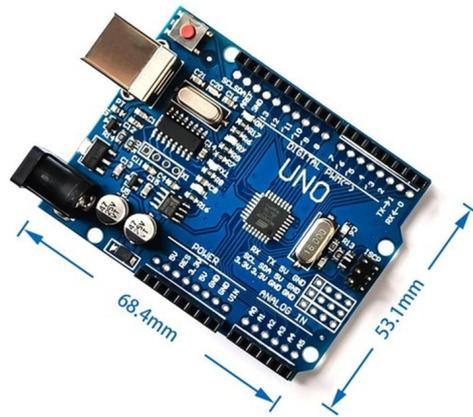


Figura 10.1: Tarjeta Arduino UNO.

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

- Yùn
- 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.

10.1.2. 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 10.2 se muestra el aspecto del entorno de desarrollo en su versión *Desktop*.

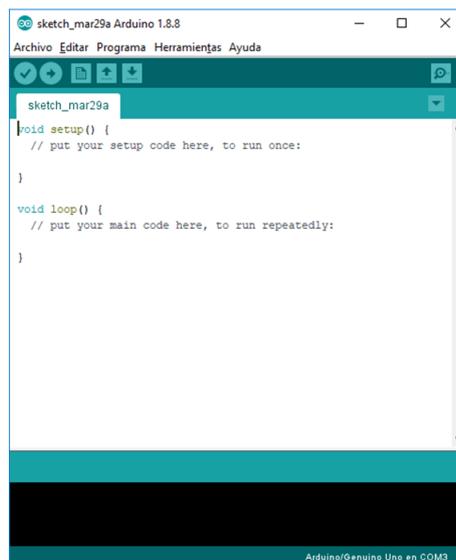


Figura 10.2: IDE Arduino.

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++.

10.1.3. 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.

10.2. 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 10.1 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 10.1: 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.

10.3. ALGUNOS RECURSOS ÚTILES

10.3.1. 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()*

10.3.2. 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 10.2 se configura una función como rutina de servicio para atender el cambio de estado de un pin de entrada de la tarjeta.

Listado 10.2: 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;
}
```

10.3.3. Funciones matemáticas

Operaciones matemáticas básicas:

- *abs()*
- *constrain()*
- *map()*
- *max(), min()*
- *pow(), sq(), sqrt()*

Trigonometría:

- *cos()*
- *sin()*
- *tan()*

10.3.4. Generación de números aleatorios

En Arduino se puede acceder a un generador de números aleatorios a través de las siguientes funciones:

- *random()*
- *randomSeed()*

10.3.5. Procesamiento de texto

- *Characters*
- *isAlpha(), isAlphaNumeric()*
- *isAscii()*
- *isControl()*
- *isDigit(), isHexadecimalDigit()*
- *isGraph()*

- *isLowerCase(), isUpperCase()*
- *isPrintable()*
- *isPunct()*
- *isSpace(), isWhitespace()*

10.4. ALGUNAS FUENTES/EJEMPLOS ADICIONALES

- *Arduino motor party* *Levin* [Accedido Mayo 2019]
- *Openframeworks and arduino* *OF* [Accedido Mayo 2019]
- *IoT Moné* [Accedido Mayo 2019]

10.5. 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 11

Arduino y Processing

11.1. 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 11.1 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 11.1: 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);
}
```

11.2. LECTURA DE SENSORES EN ARDUINO

11.2.1. 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 11.2 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 11.2: 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
}
```

11.2.2. 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.

11.2.3. 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).

11.2.4. Giróscopos, acelerómetros, magnetómetros

En el caso de sensores más sofisticados, es necesario utilizar librerías específicas.

11.3. 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 [11.1](#) anterior, el código para recibir el mensaje con Processing sería el que se recoge en el listado [11.3](#).

Listado 11.3: 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 [11.4](#) y [11.5](#).

Listado 11.4: 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 11.5: 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
}
```

11.4. 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 \[Accedido Mayo 2019\]](#)

11.5. TAREA [SIN ENTREGA]

NOTA: Solo a efectos informativo, se incluye a continuación la práctica de integración Arduino-Processing que estaba prevista. En las actuales circunstancias no va a ser posible realizarla y no tendrán que realizar ninguna entrega.

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>.