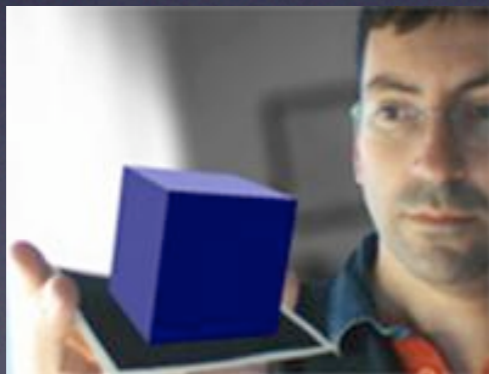


Gráficos por Computador

Primitivas básicas 2D con *processing*



Jordi Linares i Pellicer

Escola Politècnica Superior d'Alcoi

Dep. de Sistemes Informàtics i Computació

jlinares@dsic.upv.es

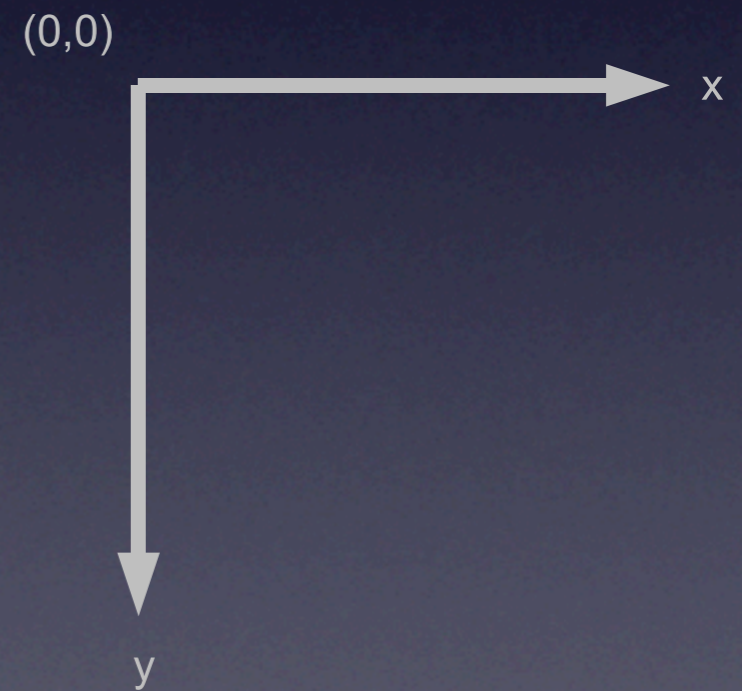
<http://www.dsic.upv.es/~jlinares>

Modos de visualización

- *Processing* dispone de varios modos de visualización: JAVA2D, P2D, P3D y OPENGL
- Hay otras posibilidades mediante el uso de librerías específicas (por ejemplo, renderizado basado en trazado de rayos)
- El modo de visualización se debe especificar como tercer argumento en la función `size()`. El modo por defecto, si no se especifica nada, es JAVA2D
- Los modos JAVA2D y P2D permiten abordar representaciones bidimensionales. P2D implica obtener un mayor rendimiento (aceleración por hardware) pero aún no tiene implementadas todas las funciones que permite JAVA2D
- Usaremos el modo por defecto, JAVA2D, que nos permite el uso de todas las funciones 2D

El sistema de coordenadas 2D

- El tamaño de la ventana se establece mediante la función `size()`, habitualmente una de las primeras acciones llevadas a cabo en la función `setup()`
- El $(0, 0)$ se encuentra situado en la esquina superior izquierda, donde las x positivas evolucionan a la izquierda y las y positivas evolucionan hacia abajo



Primitivas básicas 2D

en *processing*

- Puntos
- Líneas
- Elipses / Círculos / Arcos
- Rectángulos
- Triángulos
- Cuadriláteros
- Curvas (Bézier y Catmull-Rom)
- Shapes (formas libres)

Color y opciones

En *processing* muchas funciones llevan a cabo un cambio de estado => establecen un parámetro que permanecerá activo mientras no se cambie. Ejemplo: `stroke()` => cambia el color de los trazos y afectará a cualquier trazo hasta que se indique otro color

- El color de los trazos se puede caracterizar con la función `stroke()`
 - `stroke(255)` => RGB(255, 255, 255) un parámetro, utilizado para especificar un valor entre una escala de 256 niveles de gris
 - `stroke(128, 0, 128)` => Cualquier valor RGB
- El grosor de los trazos se puede caracterizar con `strokeWeight()`
 - `strokeWeight(5)` => Grosor 5
- El color de relleno de figuras 2D se especifica mediante la función `fill()`
 - `fill(128)` => RGB(128, 128, 128)
 - `fill(200, 120, 90)` => RGB(200, 120, 90)

Color y opciones

`background()`

- **Borra la ventana con el color especificado**
- **Ejemplos:** `background(0)`
`background(128, 100, 128)`

`noFill()`

- **Las figuras 2D se dibujaran sin relleno**

`noStroke()`

- **Las figuras 2D se dibujaran sin trazo**
(especialmente útil en figuras cerradas, pero afecta a todas, incluso a líneas)

Puntos

`point(x, y)`

- Dibuja un punto en la coordenadas (x, y) especificadas
- El color se establece mediante `stroke()` y su grosor con `strokeWeight()`

`set(x, y, color)`

- Dibuja un punto en las coordenadas (x, y) y un color concreto
- No queda afectado por `stroke()` o `strokeWeight()`
- Ejemplo:
 - `set(50, 50, color(128, 120, 255))`

Otros usos de `set` (estudio posterior)

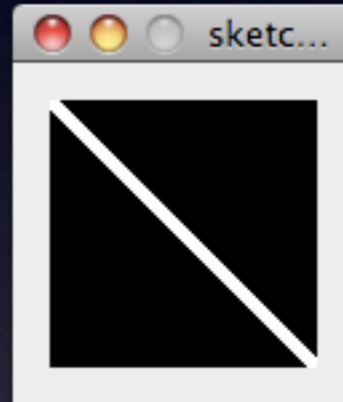
- La función `set` puede utilizarse también para ubicar en las coordenadas (x, y) una imagen
- `set` puede ejecutarse también sobre una imagen

Líneas

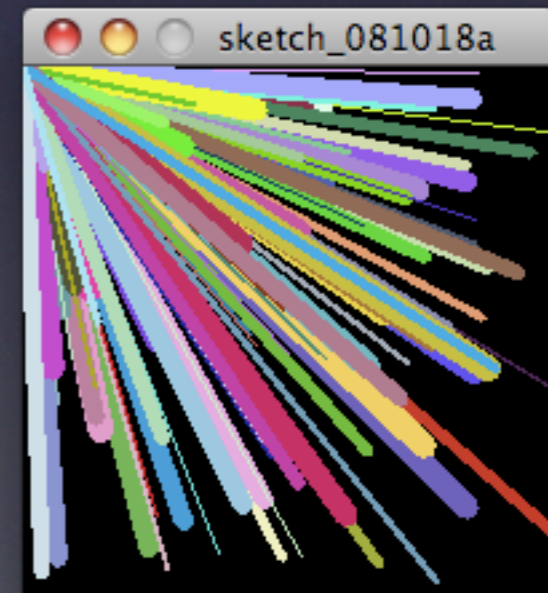
`line(x1, y1, x2, y2)`

- Dibuja una línea recta entre los puntos $(x1, y1)$ y $(x2, y2)$
- Con las funciones `stroke` indicamos sus propiedades
- Ejemplo:

```
size(100, 100);  
background(0);  
stroke(255);  
strokeWeight(5);  
line(0, 0, 99, 99);
```



```
size(200, 200);  
background(0);  
for (int i=0; i<100; i++) {  
  stroke(random(255), random(255), random(255));  
  strokeWeight(random(10));  
  line(0, 0, random(200), random(200));  
}
```



Líneas

- Terminaciones de líneas

- ROUND (redondeado), PROJECT (proyectado, se extiende en función del grosor del trazo), SQUARE (cuadrado estricto)

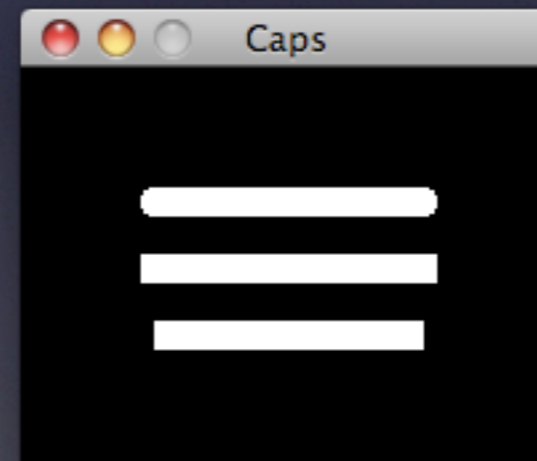
- Ejemplo:

```
size(100, 100);  
background(0);  
stroke(255);  
strokeWeight(10);
```

```
strokeCap(ROUND);  
line(50, 50, 150, 50);
```

```
strokeCap(PROJECT);  
line(50, 75, 150, 75);
```

```
strokeCap(SQUARE);  
line(50, 100, 150, 100);
```



Elipses y círculos

`ellipse(x, y, ancho, alto)`

- **Dibuja una elipse en las coordenadas (x, y) y el ancho y alto suministrados**

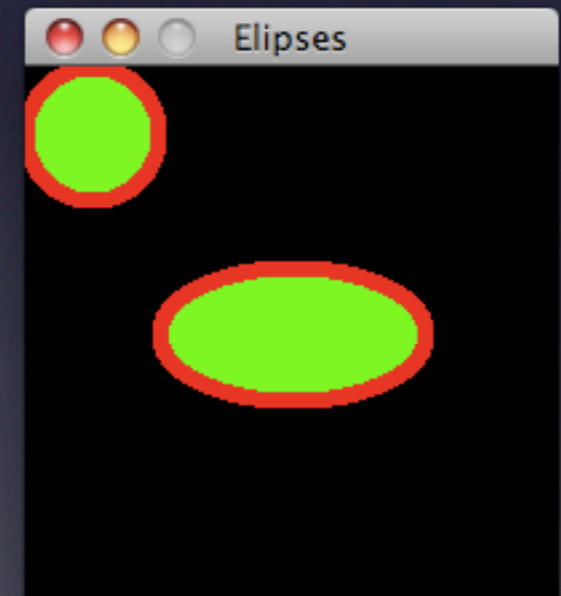
`ellipseMode()`

- **Cambia el modo en el que los parámetros de la elipse son interpretados**
- `ellipseMode(CENTER)` => (x, y) es el centro de la elipse (es el modo por defecto).
- `ellipseMode(RADIUS)` => igual que el anterior, pero ancho y alto son radios y no diámetros
- `ellipseMode(CORNER)` => (x, y) hace referencia a la esquina superior izquierda del rectángulo envolvente de la elipse
- `ellipseMode(CORNERS)` => los cuatro parámetros de la elipse hacen referencia a dos puntos opuestos del rectángulo envolvente de la elipse

Elipses y círculos

- Ejemplo:

```
size(200, 200);  
background(0);  
  
stroke(255, 0, 0);  
strokeWeight(5);  
fill(0, 255, 0);  
  
// (x, y) y diámetros  
ellipse(100, 100, 100, 50);  
  
// 2 esquinas opuestas  
ellipseMode(CORNERS);  
ellipse(0, 0, 50, 50);
```



Arcos

`arc(x, y, ancho, alto, inicio, fin)`

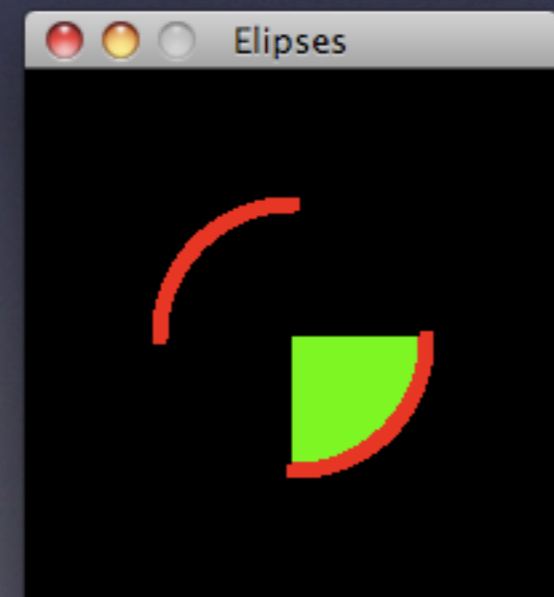
- Dibuja un arco como un sector de una elipse en las coordenadas (x, y) y el ancho y alto suministrados. El fragmento o sector de la elipse dibujado es el comprendido entre el ángulo *inicio* y el ángulo *fin* (radianes por defecto) en el sentido de las agujas del reloj
- Sus parámetros son igualmente interpretados en función del modo indicado con `ellipseMode()`
- En procesing, el modo por defecto al dibujar figuras es con relleno, y se aplica incluso ante figuras no cerradas. La función `noFill()` ha de ser llamada de forma explícita si no se quiere relleno.
- Ejemplo:

```
size(200, 200);
background(0);

stroke(255, 0, 0);
strokeWeight(5);
fill(0, 255, 0);

// (x, y) y diámetros
arc(100, 100, 100, 100, 0, PI / 2.0);

// Sin relleno
noFill();
arc(100, 100, 100, 100, PI, 3 * PI / 2.0);
```



Rectángulos

`rect(x, y, ancho, alto)`

- Dibuja un rectángulo

`rectMode()`

- Cambia el modo en el que los parámetros del rectángulo son interpretados
- Los parámetros son los mismos que para la elipse: `CENTER`, `RADIUS`, `CORNER` y `CORNERS`
- El modo por defecto en este caso es el `CORNER` (x e y son la esquina superior izquierda)

Triángulos y cuadriláteros

```
triangle(x1, y1, x2, y2, x3, y3)
```

- Dibuja un triángulo a partir de sus tres vértices

```
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```

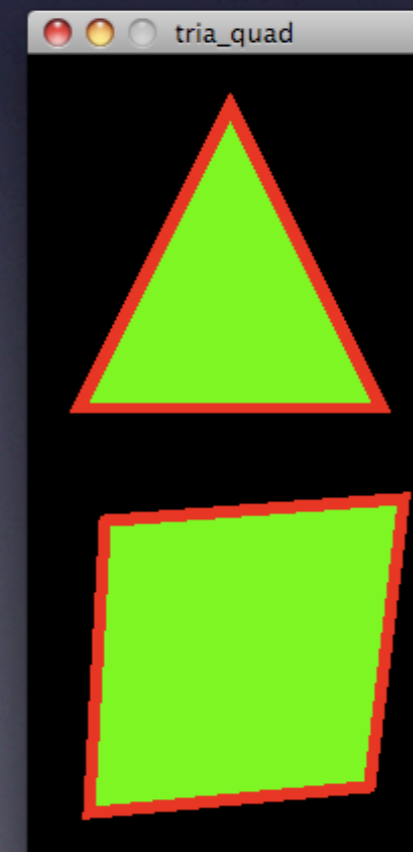
- Dibuja un cuadrilátero. El primer punto del mismo es (x1, y1), el resto deben ser los 3 vértices restantes debiéndose especificar en sentido horario o anti-horario (en uno de los dos)

```
size(200, 400);  
background(0);
```

```
stroke(255, 0, 0);  
strokeWeight(5);  
fill(0, 255, 0);
```

```
// (100,25) - (25,175) - (175,175)  
triangle(100,25,25,175,175,175);
```

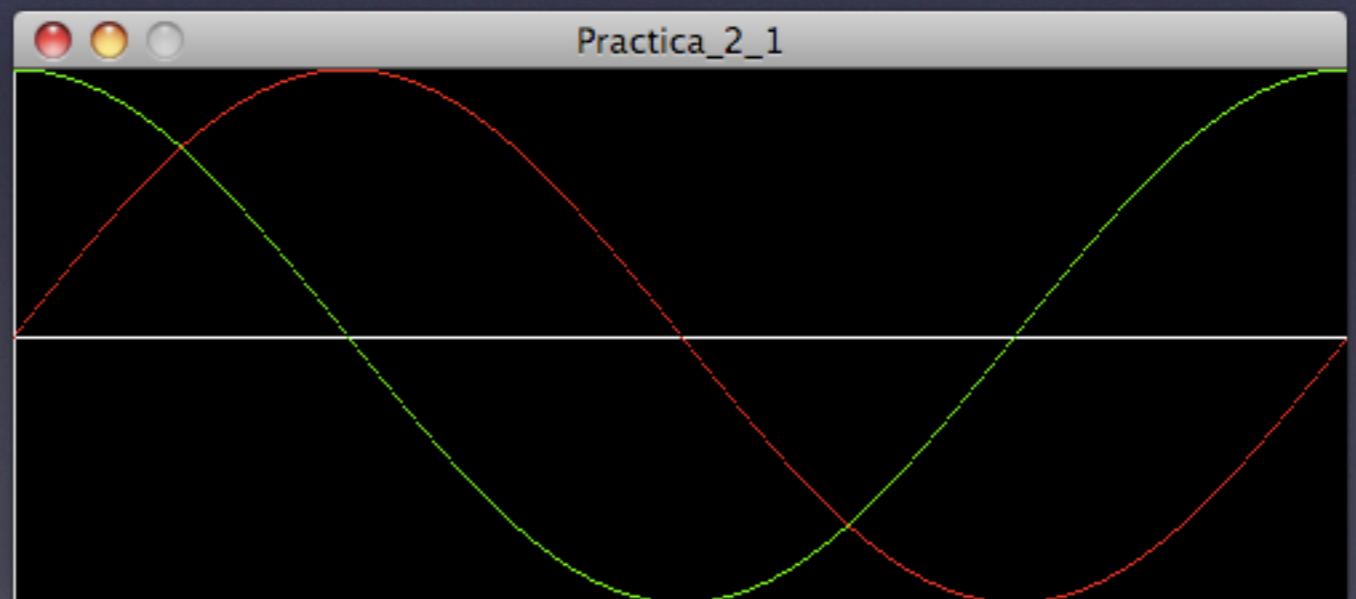
```
// Sentido horario  
// (38,231) - (186,220) - (169,363) - (30,376)  
quad(38, 231, 186, 220, 169, 363, 30, 376);
```



Práctica 2-1

Representación de funciones trigonométricas

- Representar la función seno y coseno, para los valores comprendidos entre 0 y 2π radianes
- El (0,0) de las coordenadas cartesianas deberá estar situado en $x=0$ e $y=\text{width}/2$ de las coordenadas de ventana (pegado al margen izquierdo, a la mitad de altura de la ventana)
- El recorrido de 0 a 2π debe aprovechar todo el ancho de la ventana
- Implementar una función que dibuje el seno, otra el coseno, y cuyos valores de entrada sean el color y grosor del trazo y el ancho y alto de la ventana
- Dibujar como base los ejes cartesianos



Curvas de Bézier

`bezier(x1, y1, cx1, cy2, cx2, cy2, x2, y2)`

- Son curvas cúbicas (permiten un punto de inflexión)
- Reciben su nombre del matemático que las formuló, Pierre Bézier
- Para su caracterización se precisa de 4 puntos:
 - El primero y el último marcan el inicio y fin de la curva
 - Los dos puntos centrales son los puntos de control y actúan como “atractores”, modificando la curvatura sin que la curva pase por ellos

Curvas de Bézier

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

  // Dibujamos dos curvas Bézier
  dibujaBezier(100,140,50,40,350,40,300,140);
  dibujaBezier(50,290,150,190,200,390,350,290);
}

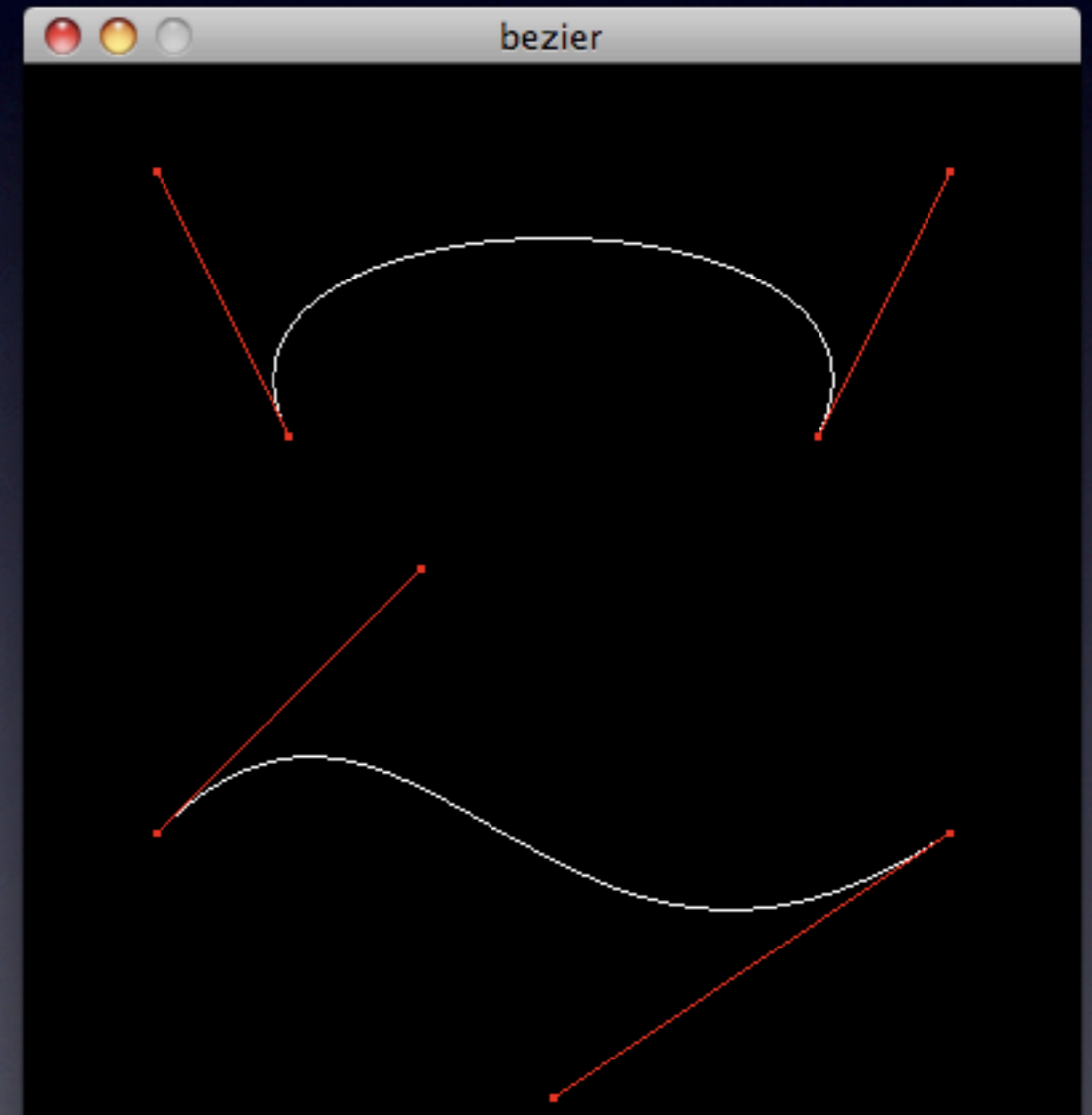
void dibujaBezier(int x1, int y1,
                 int cx1, int cy1,
                 int cx2, int cy2,
                 int x2, int y2)
{
  noFill();
  stroke(255);

  // bézier
  bezier(x1, y1, // Inicio
        cx1, cy1, // Punto control 1
        cx2, cy2, // Punto control 2
        x2, y2); // Fin

  // Dibujamos puntos de control para
  // ayudar a entender su efecto
  strokeWeight(3);
  stroke(255,0,0);

  point(x1, y1); point(x2, y2);
  point(cx1, cy1); point(cx2, cy2);

  strokeWeight(1);
  line(x1, y1, cx1, cy1);
  line(x2, y2, cx2, cy2);
}
```



Curvas de Catmull-Rom

`curve(cx1, cy1, x1, y1, x2, y2, cx2, cy2)`

- Formuladas por Edwin Catmull y Raphie Rom
- Útiles para conseguir una curva que interpole una colección de puntos, muy útil en gráficos por computador (animación por keyframes, por ejemplo)
- `curve` traza una curva de $(x1, y1)$ a $(x2, y2)$. El primer punto de control marcará la curvatura con la que se inicia la curva en $(x1, y1)$. El segundo punto de control marcará la curvatura con la que la curva finalizará en $(x2, y2)$
- Será especialmente útil dentro de 'shapes', formas libres, donde se podrán lanzar colecciones de vértices

Curvas de Catmull-Rom

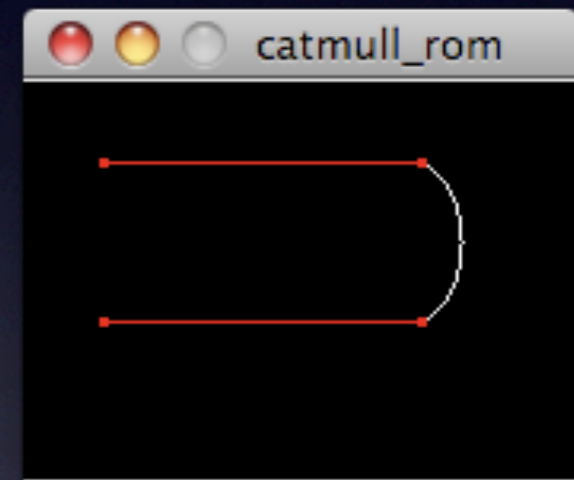
```
size(175, 125);

background(0);
noFill();
stroke(255);

curve(25, 25, // Punto de control 1
      125, 25, // (x1, y1)
      125, 75, // (x2, y1)
      25, 75); // Punto de control 2

// Dibujamos puntos para entender
// mejor cómo funciona una catmull-rom
strokeWeight(3);
stroke(255, 0, 0);
point(125, 25); point(125, 75);
point(25, 25); point(25, 75);

strokeWeight(1);
line(25, 25, 125, 25);
line(125, 75, 25, 75);
```



Shapes

- Las *shapes* en *processing* permiten el dibujo de formas complejas y libres mediante la enumeración de sus vértices
- La primitiva principal es `vertex(x, y)`, que define uno de los vértices de la figura
- Para iniciar una forma libre hay que invocar a la función `beginShape()`. A continuación se lanzan los vértices mediante `vertex()`. La figura se finaliza con la invocación de `endShape()`.
- **Gran versatilidad:** entre la llamada `beginShape()` y `endShape()` puede intercalarse cualquier código (llamadas a funciones, bucles etc.) que gestione el lanzamiento de los vértices, e incluso las primitivas de dibujo anteriores.
- Sin embargo no se puede hacer uso de otras entre el `beginShape()` y el `endShape()`, como `rotate()`, `scale()` y `translate()`, que se estudiarán más adelante.

Shapes

- Cuando no lleva argumentos `beginShape()` permite definir una polilínea
- Si se especifica el parámetro `CLOSE` en `endShape()`, la polilínea se cierra uniendo el último punto lanzado con el primero
- Ejemplo:

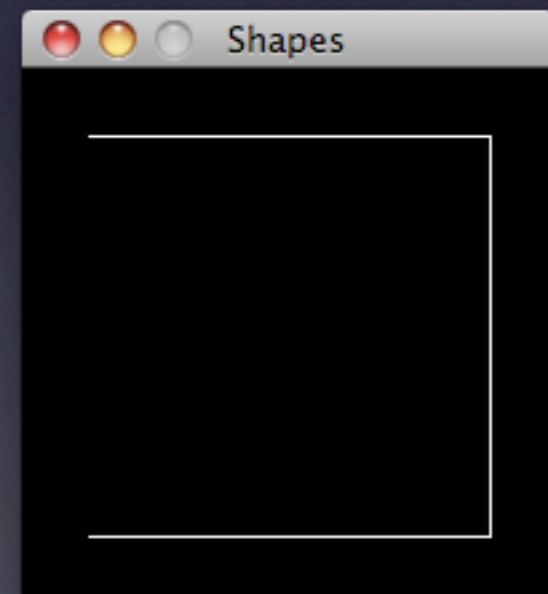
```
size(200, 200);
background(0);

// Las shape también pueden caracterizarse
// usando las funciones fill y stroke
// Por defecto, siempre rellenas, hay que
// llamar de forma explícita a noFill() si
// no lo deseamos.
noFill();
stroke(255);

// Polilínea
beginShape();

// Ahora ya podemos lanzar vértices ...
vertex(25, 25);
vertex(175, 25);
vertex(175, 175);
vertex(25, 175);

// Fin de la figura
endShape();
```



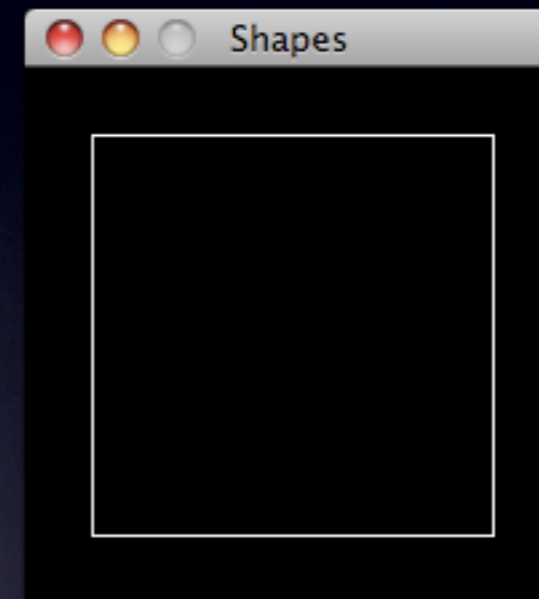
Shapes

- Con CLOSE cerramos la figura:

```
// Polilínea
beginShape ();

// Ahora ya podemos lanzar vértices ...
vertex (25, 25);
vertex (175, 25);
vertex (175, 175);
vertex (25, 175);

// Fin de la figura
endShape (CLOSE);
```



Shapes

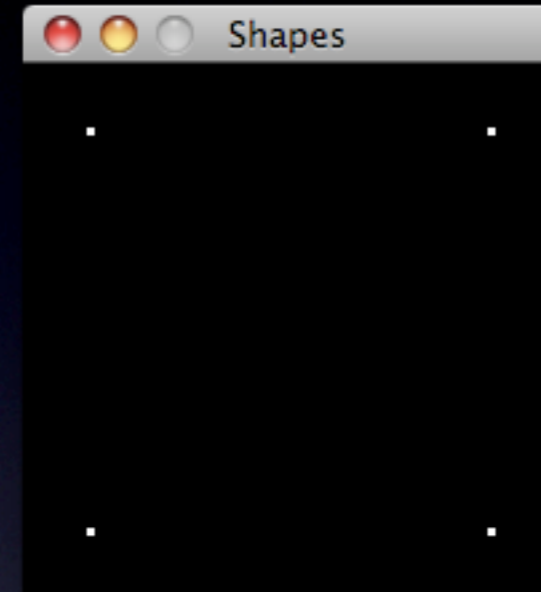
- Existe la posibilidad de pasar un argumento a `beginShape()` y permitir otra interpretación de los vértices:
 - `POINTS`. Los vértices trazan un conjunto de puntos.
 - `LINES`. Los vértices, por pares, dibujan líneas.
 - `TRIANGLES`. Los vértices, por tríos, dibujan triángulos.
 - `TRIANGLE_STRIP`. Una tira de triángulos.
 - `TRIANGLE_FAN`. Un abanico de triángulos.
 - `QUADS`. Los vértices, por series de cuatro, dibujan cuadriláteros.
 - `QUAD_STRIP`. Una tira de cuadriláteros

Shapes

```
// Puntos
beginShape (POINTS) ;

// Ahora ya podemos lanzar vértices ...
vertex(25, 25) ;
vertex(175, 25) ;
vertex(175, 175) ;
vertex(25, 175) ;

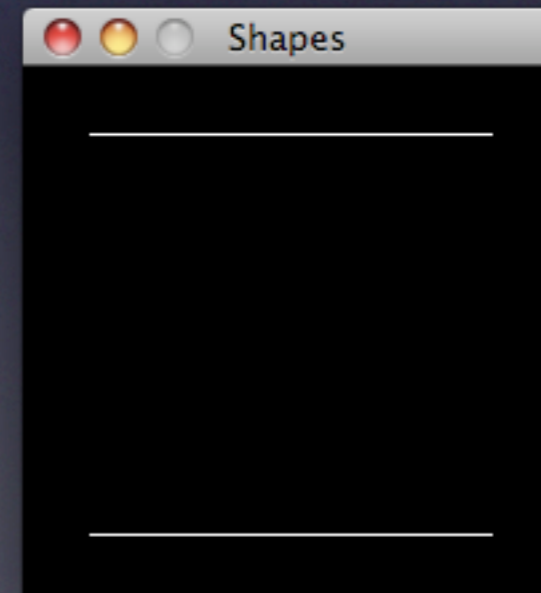
// Fin de la figura
endShape () ;
```



```
// Líneas
beginShape (LINES) ;

// Ahora ya podemos lanzar vértices ...
vertex(25, 25) ;
vertex(175, 25) ;
vertex(175, 175) ;
vertex(25, 175) ;

// Fin de la figura
endShape () ;
```



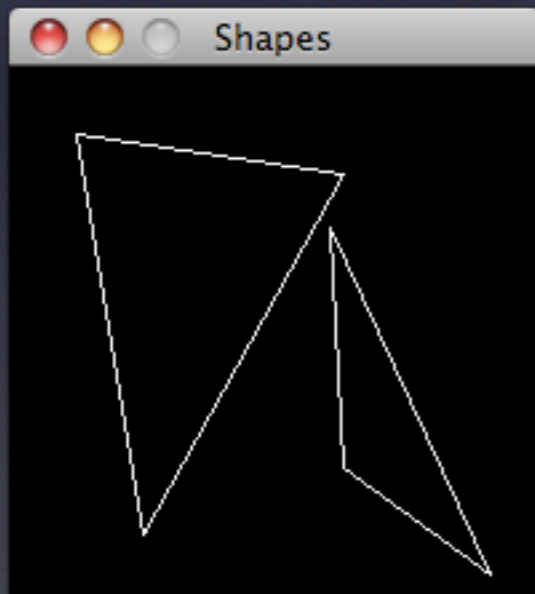
Shapes

```
// Triángulos
beginShape (TRIANGLES);

// Ahora ya podemos lanzar vértices ...
vertex (25, 25);
vertex (50, 175);
vertex (125, 40);

vertex (125, 150);
vertex (120, 60);
vertex (180, 190);

// Fin de la figura
endShape ();
```

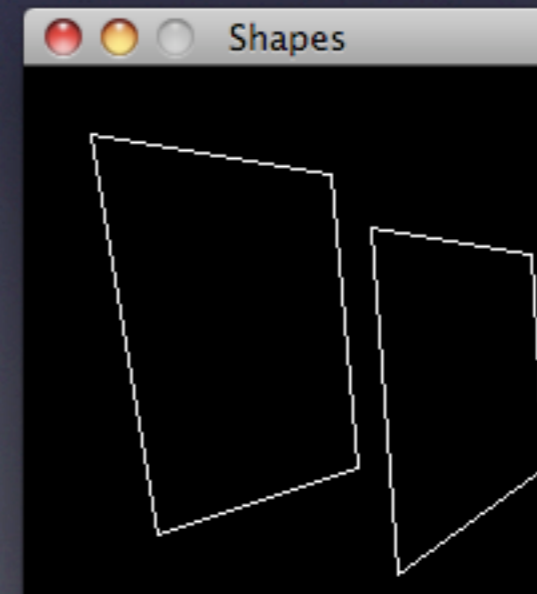


```
// Cuadriláteros
beginShape (QUADS);

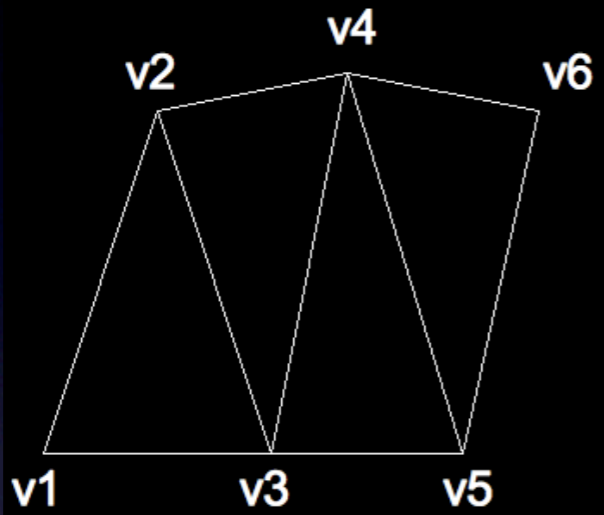
// Ahora ya podemos lanzar vértices ...
vertex (25, 25);
vertex (50, 175);
vertex (125, 150);
vertex (115, 40);

vertex (130, 60);
vertex (190, 70);
vertex (195, 150);
vertex (140, 190);

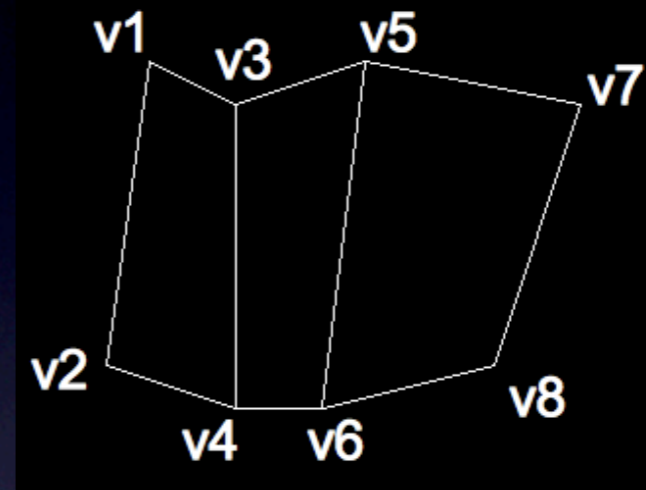
// Fin de la figura
endShape ();
```



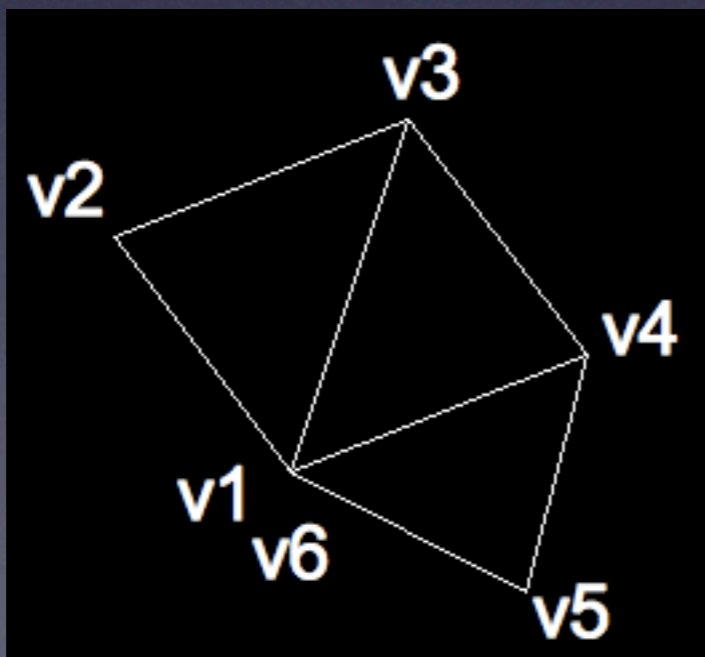
Shapes



TRIANGLE_STRIP



QUAD_STRIP



TRIANGLE_FAN
(Repetir primer punto)

Shapes

- Además de especificar los vértices con `vertex()`, *processing* permite especificar vértices mediante las funciones `curveVertex()` y `bezierVertex()`, generando curvas en lugar de líneas
- Estas funciones únicamente funcionan en la versión sin parámetros de `beginShape()`
- El uso de estas funciones nos permite crear cadenas de curvas cúbicas bézier, `bezierVertex()`, o de Catmull-Rom, `curveVertex()`
- Su uso se puede combinar con la función `vertex()` y, de esta forma, poder generar complejas figuras

Shapes

`curveVertex(x, y)`

- Con `curveVertex()` se puede generar una curva de Catmull-Rom capaz de interpolar cualquier conjunto de puntos
- El primer y último punto lanzado con `curveVertex()` actuarán como puntos de control y, por tanto, definirán la curvatura inicial y final de la cadena de curvas
- Los puntos interiores serán interpolados por una sucesión de curvas cúbicas
- Se exige un mínimo de 4 vértices, usando `curveVertex()`, en el interior del `beginShape()` / `endShape()`

Shapes

```
size(400, 400);
background(0);
noFill();
stroke(255);

// Polilínea
beginShape();

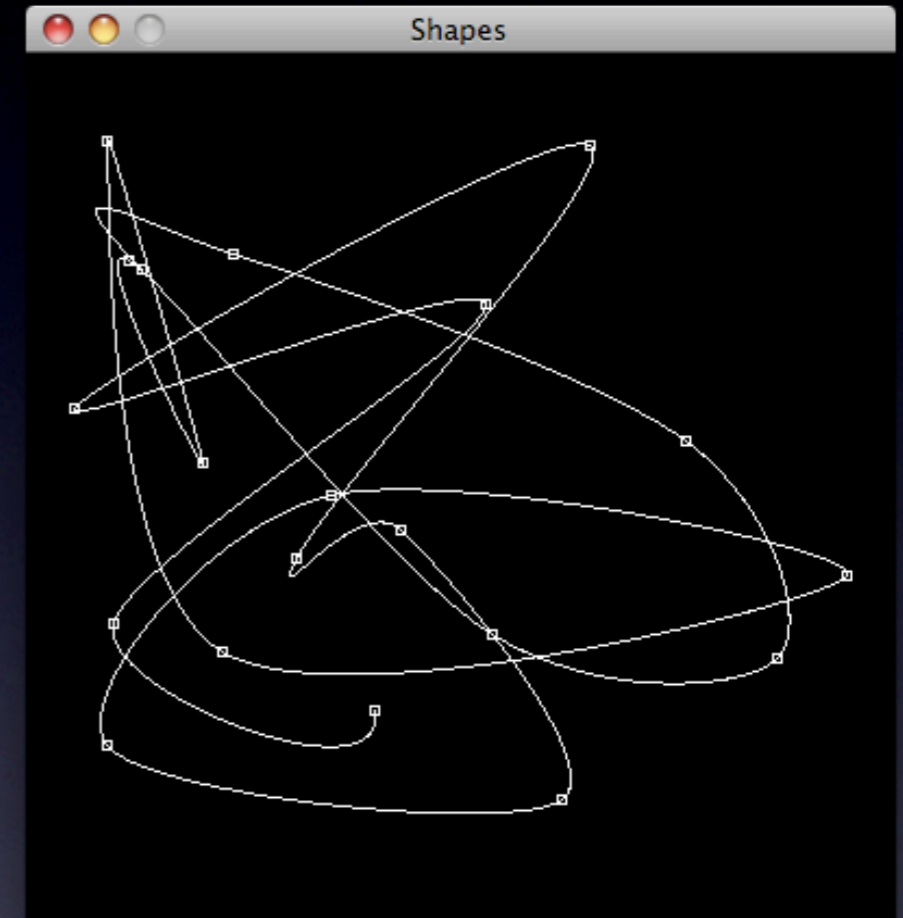
// Ahora ya podemos lanzar vértices ...
curveVertex(0, 0);

for (int i = 0; i < 20; i++)
{
  int x = (int)random(400), y = (int)random(400);

  curveVertex(x, y); // Lanzamos punto de curva
  rect(x-2, y-2, 4, 4); // Dibujamos el punto por claridad
}

curveVertex(width-1, height-1);

// Fin de la figura
endShape();
```



Shapes

`bezierVertex(cx1, cy1, cx2, cy2, x, y)`

- Para generar una cadena de curvas Bézier, el primer punto (inicio de la curva) debe lanzarse mediante `vertex()`
- Cada `bezierVertex()` que se lance a continuación especificará dos nuevos puntos de control y el punto de destino
- El punto de destino actuará como punto de inicio ante nuevos lanzamientos a `bezierVertex()`
- El último punto de una bézier se convierte en el primer punto de la siguiente. Si nos aseguramos además de que el último punto de control de una bézier está en línea con el último punto de la curva (primero de la siguiente) y con el primer punto de control de la siguiente, la curvatura se mantendrá en la unión

Shapes

```
size(300, 200);
background(0);
noFill();
stroke(255);

// Polilínea
beginShape();

vertex(10,100); // Primer punto de la curva
bezierVertex(70, 140, 160, 140, 150, 100);
bezierVertex(140, 60, 210, 60, 290, 100);

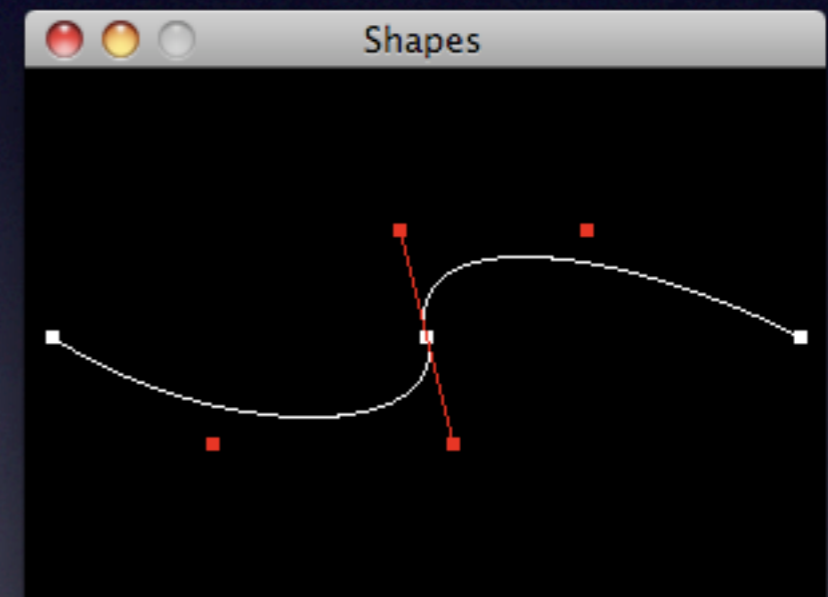
// Fin de la figura
endShape();

// Dibujamos puntos
// para mayor claridad
strokeWeight(5);

// Puntos de curva
stroke(255);
point(10, 100); point(150, 100); point(290, 100);

// Puntos de control
stroke(255, 0, 0);
point(70, 140); point(160, 140);
point(140, 60); point(210, 60);

// Puntos de control alineados ...
strokeWeight(1);
line(160, 140, 140, 60);
```



Shapes

```
// Creación de formas complejas
// mediante el uso combinado de
// vértices lineales y curvos
size(400, 300);
background(0);
stroke(255);
noFill();

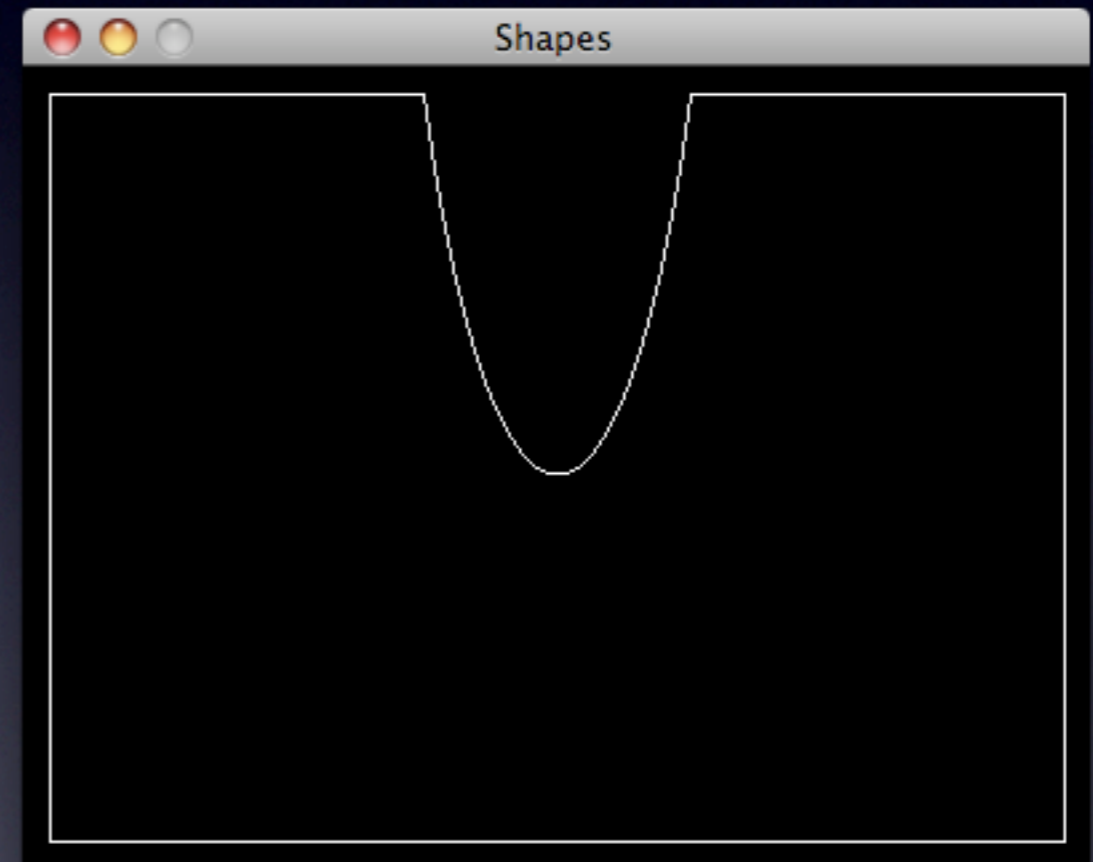
// Polilínea
beginShape();

vertex(10, 10);
vertex(150, 10);

bezierVertex(175, 200, 225, 200, 250, 10);

vertex(390, 10);
vertex(390, 290);
vertex(10, 290);
vertex(10, 10);

endShape();
```



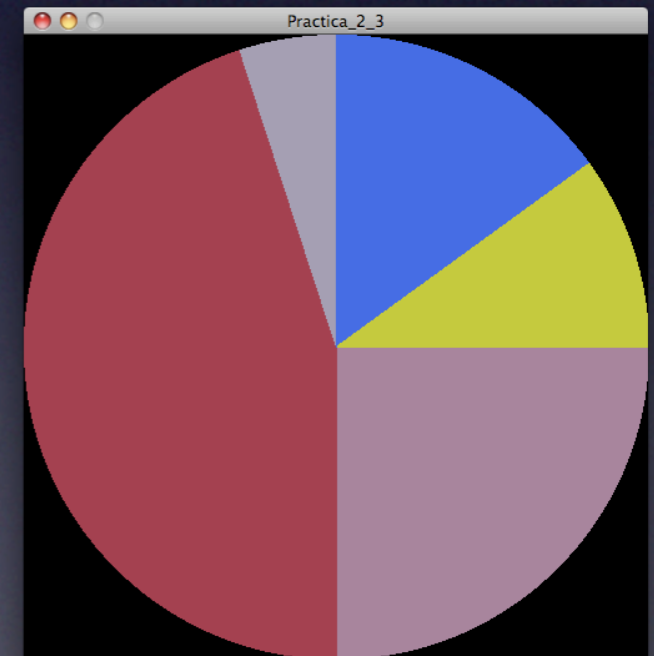
Práctica 2-2

- Reescribir el código de la práctica 1-1 de forma que se utilice una forma libre (shape) para el dibujo del polígono de n lados

Práctica 2-3

- En una aplicación estadística se necesita representar un gráfico de tarta capaz de visualizar un conjunto de elementos porcentuales
- Los elementos vienen como un vector y la suma de todos ellos totaliza el 100%
- Ejemplo:

```
float[] valores = {25.0, 45.0, 5.0, 15.0, 10.0};
```
- Implementar una función que a partir de este vector, dibuje un gráfico de tarta que represente gráficamente estos porcentajes
- La función debe recibir centro y radio del gráfico
- La función puede conocer el número de valores del vector recibido mediante el atributo `length`
- Elegir un color aleatorio para cada segmento



Práctica 2-4

- Desarrollar una función análoga a la práctica 2-3 pero esta vez con el objetivo de dibujar un gráfico de barras 2D
- Aprovechar toda la ventana disponible trazando también los ejes de coordenadas
- Elegir el ancho de las barras en función del número de valores representados y el ancho total de la pantalla; y el alto de las barras en función del valor percentual correspondiente y teniendo en cuenta que la barra de máximo valor ocupará la totalidad de la altura de la pantalla

