Visualization modes

- *Processing* has several visualization modes: JAVA2D, P2D, P3D and OPENGL
- There are other possibilities using specific libraries (for example, ray-tracing rendering)
- The visualization mode has to be defined as the third argument of the function `size()`. The default mode, if nothing is indicated, is JAVA2D
- JAVA2D and P2D modes allow bidimensional representations. P2D allows better performance (hardware acceleration) but it doesn't have implemented all JAVA2D functions yet
- We'll use the default mode, JAVA2D, since it allows us to use all 2D functions

The 2D coordinate system

- Window size is defined using the function `size()`, generally one of the first actions carried out in the function `setup()`

- The `(0,0)` is top-left located, where positive `x` goes towards right and positive `y` goes towards bottom

(0,0)

x

y
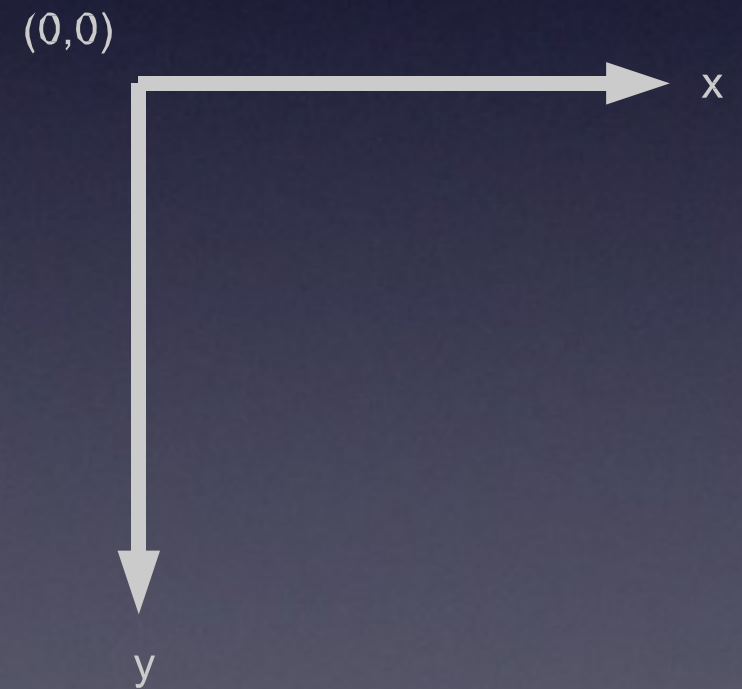
2D basic primitives in *processing*

- Points
- Lines
- Ellipses / Circles /Arcs
- Rectangles
- Triangles
- Quadrilaterals
- Curves (Bézier and Catmull-Rom)
- Shapes (free forms)

Color and options

In *processing* many functions provoke a state change => they set a parameter that will remain active until we change it. Example: `stroke()` => changes stroke colors and will affect any stroke until a new color is specified

- Stroke colors can be characterised using the function `stroke()`
  - `stroke(255)` => RGB(255,255,255) one parameter means a color within a 256 grey scale
  - `stroke(128, 0, 128)` =>Any RGB color
- Stroke thickness can be indicated with `strokeWeight()`
  - `strokeWeight(5)` =>A thickness of 5 pixels
- The fill color of a 2D figure can be indicated with `fill()`   `fill(128)` => RGB(128,128,128)
  - `fill(200, 120, 90)` => RGB(200,120,90)

Color and options

```
background()
```
- **Erases the window with the specified color**
- **Examples:** `background(0)`
  `background(128, 100, 128)`

```
noFill()
```
- **2D figures won't be filled**

```
noStroke()
```
- **2D figures won't have a external stroke (especially useful for closed figures, but it affects to all kind of figures, even lines)**

Points

`point(x, y)`
- Draws a point at (x,y) coordinates
- Color is specified using `stroke()` and its thickness (size) using `strokeWeight()`

`set(x, y, color)`
- Draws a point at (x,y) coordinates and with a specific color
- It is not affected by `stroke()` or `strokeWeight()`
- Example:
  - `set(50, 50, color(128, 120, 255))`

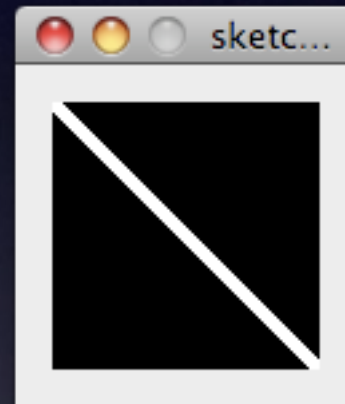Other uses of `set` (to be covered later on)
- Function `set` can be used to map an image to the coordinates (x,y)
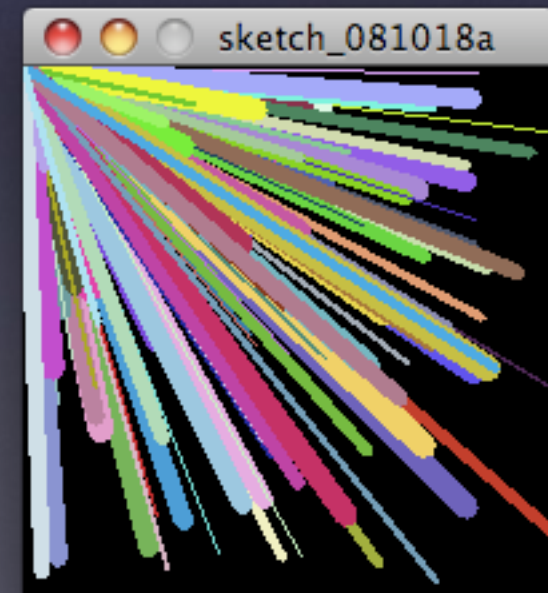- `set` can be executed over an image

Lines

```
line(x1, y1, x2, y2)
```
- Draws a line between the points (x1,y1) and (x2,y2)
- With the `stroke` functions we can specify its properties
- Example:

```
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));
}
```
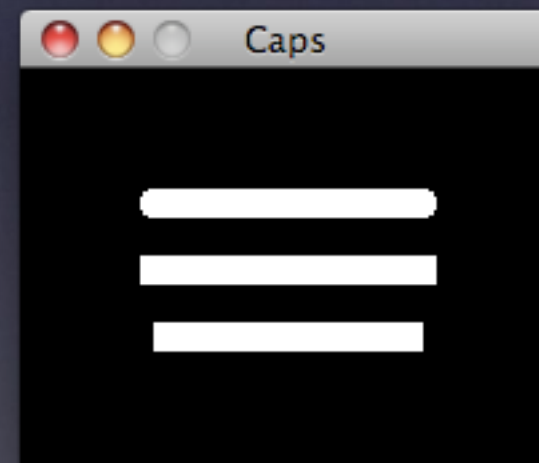
Lines

- Line endings
  - ROUND (rounded), PROJECT (projected, the line is extended depending on the stroke thickness), SQUARE (strictly squared)
  - Example:

```
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);
```

Ellipses and circles

`ellipse(x, y, width, height)`
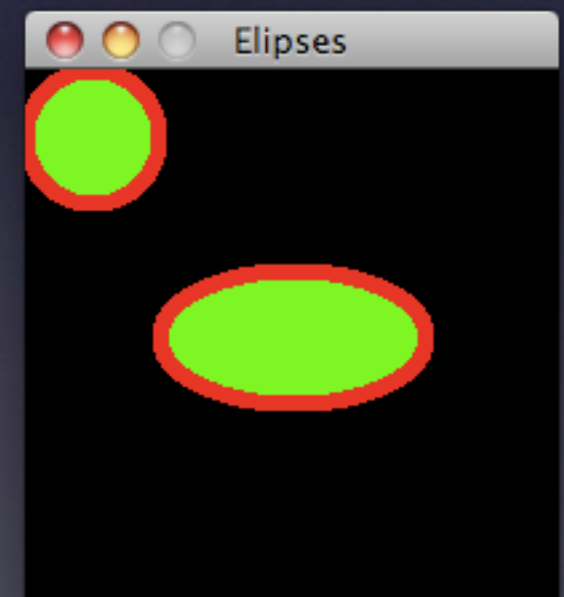- Draws a ellipse in the coordinates (x, y) and with the width and height specified

`ellipseMode()`
- Changes how ellipse parameters are interpreted
- `ellipseMode(CENTER)` => (x, y) are the ellipse center (default mode).
- `ellipseMode(RADIUS)` => same as previous, but width and height are radius, not diameters
- `ellipseMode(CORNER)` => (x, y) references the top-left corner of the ellipse bounding-box
- `ellipseMode(CORNERS)` => the four parameters indicate two opposite corners of the ellipse bounding-box

Ellipses and circles

- Example:

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

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

// (x, y) and diameters
ellipse(100, 100, 100, 50);

// 2 opposite corners
ellipseMode(CORNERS);
ellipse(0, 0, 50, 50);
```

Arcs

```
arc(x, y, width, height, start, end)
```

- Draws an arc as an ellipse sector with the coordinates (x,y) and with the specified width and height. This fragment or sector is defined by the angles indicated with *start* and *end* (radians by default) and following a clockwise direction
- Their parameters are also interpreted following `ellipseMode()`
- In *procesing*, filling figures is the default mode and it is applied even with non-closed figures. Function `noFill()` has to be explicitly called if filling is not desired.
- Example:

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

stroke(255, 0, 0); strokeWeight(5);
fill(0, 255, 0);
// (x, y) and diameters
arc(100, 100, 100, 100, 0, PI / 2.0);
// No fill
noFill();
arc(100, 100, 100, 100, PI, 3 * PI / 2.0);
```
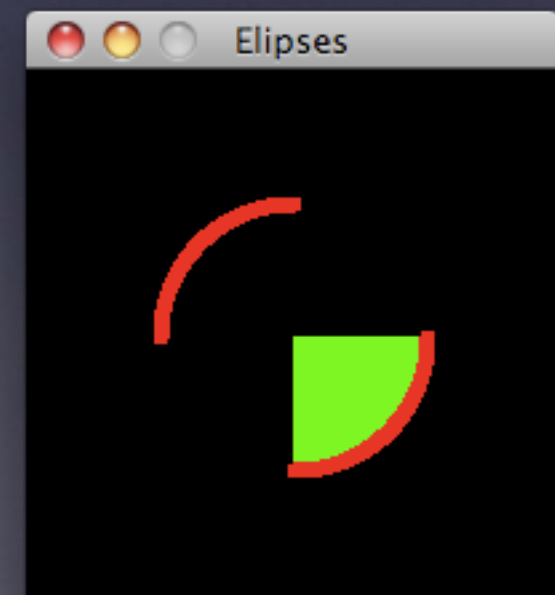
Rectangles

`rect(x, y, width, height)`
- Draws a rectangle

`rectMode()`
- Changes how rectangles parameters are interpreted
- The parameters are the same as for ellipses: `CENTER`, `RADIUS`, `CORNER` and `CORNERS`
- The default mode in this case is `CORNER` (x and y are the top-left corner)

Triangles and quadrilaterals

```
triangle(x1, y1, x2, y2, x3, y3)
```
- Draws a triangle following the three vertices
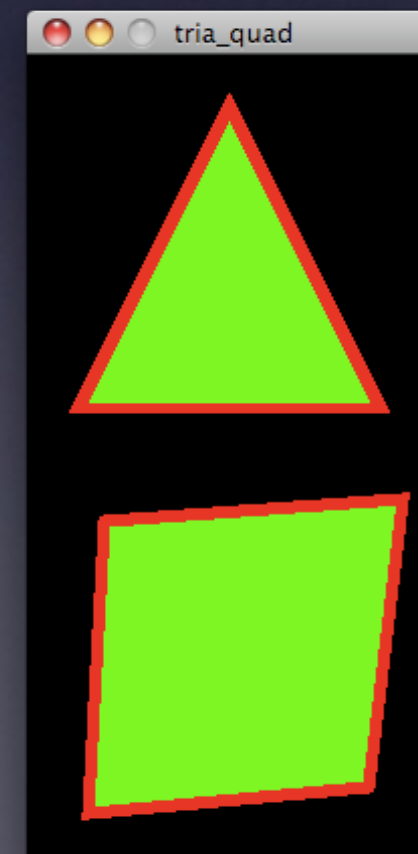
```
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```
- Draws a quadrilateral. The firs point is (x1,y1),the rest are the next 3 vertices specified either clockwise or counter-clockwise (one of them)

```
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);

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

Bézier curves

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

- They are cubic curves (allow an inflexion point)
- They were named after the mathematician Pierre Bézier
- 4 points are required to characterise them:
  - The first and the last points are the curve starting and ending points
  - Central points are control points and they attract the curve, modifying it without forcing it to pass through them

Bézier curves

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

  // We draw two Bézier curves
  MyBezier(100,140,50,40,350,40,300,140);
  MyBezier(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,     // Initial point
         cx1, cy1,   // Control point 1
         cx2, cy2,   // Control point 2
         x2, y2);    // Final point

  // We draw control points to
  // help to understand their effect
  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);
}
```
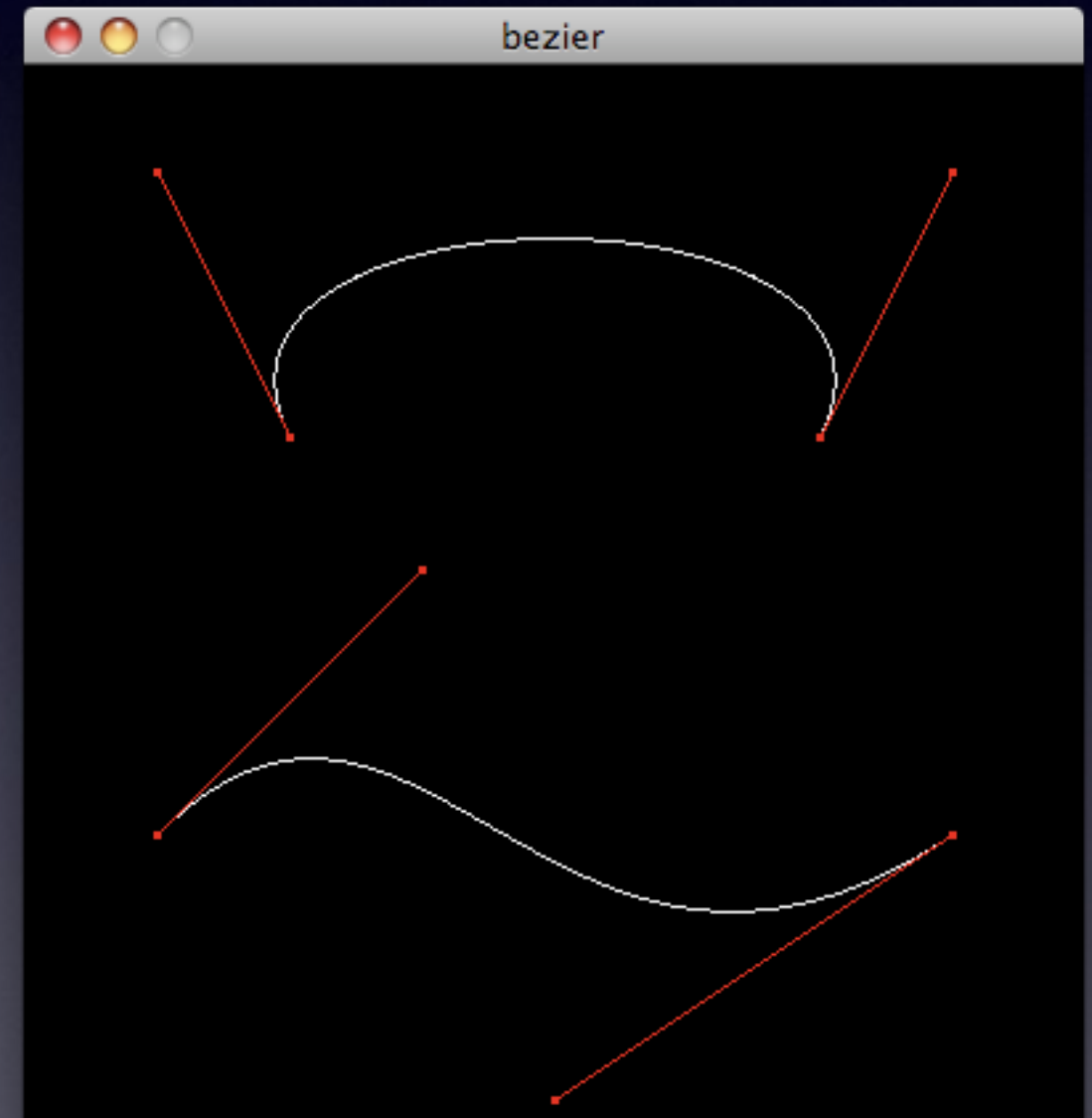
Catmull-Rom curves

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

- Formulated by Edwin Catmull and Raphie Rom
- Useful to achieve a curve that interpolated a set of points, and very useful for computer graphics (animation using keyframes, for instance)
- `curve` draws a curve from (x1,y1) to (x2,y2). The first control point defines the curvature at (x1,y1). The last control point defines the curvature at (x2,y2).
- It will be especially interesting inside 'shapes', free forms, where collections of vertices can be defined
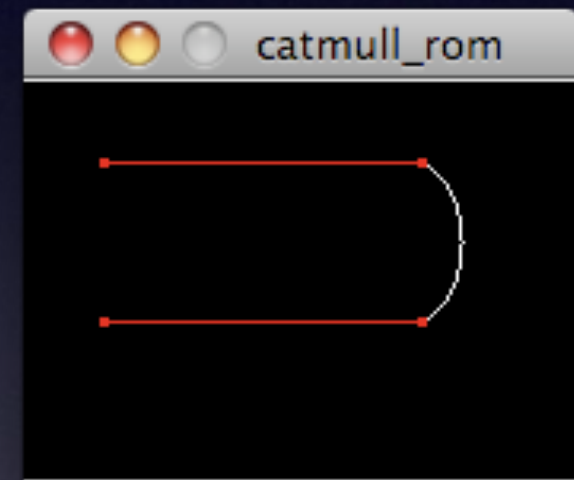
Catmull-Rom curves

```
size(175, 125);

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

curve(25, 25,   // Control point 1
      125, 25, // (x1, y1)
      125, 75, // (x2, y1)
      25, 75); // Control point 2

// We draw all the points to
// better understand how it works
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);
```
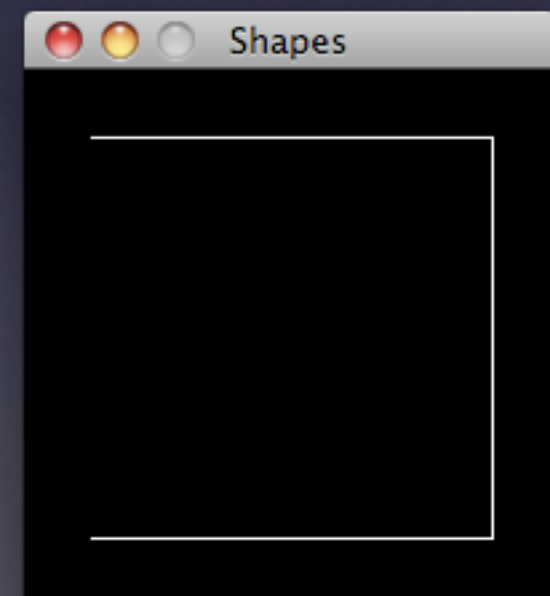
Shapes

- *shapes* in *processing* allow to draw free and complex forms by enumerating their vertices
- The main primitive is `vertex(x, y)`, that defines one of the vertices of the shape
- To begin a shape, the function `beginShape()` has to be invoked. Afterwards, their vertices can be defined using `vertex()`. The shape can be finished by invoking `endShape()`.
- Great versatility: between the `beginShape()` and `endShape()` calls, any code can be specified (function calls, loops etc.) in order to manage the vertices definition.
- However, not every function can be used inside the `beginShape()` and `endShape()` pair of functions, such as `rotate()`, `scale()` and `translate()`

Shapes

- When `beginShape()` has no arguments, it allows to define polylines
- If the parameter `CLOSE` is specified inside `endShape()`, the polyline will be automatically closed by joining the last vertex with the first
- Example:

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

// Shapes can also be characterised by
// using the functions fill and stroke.
// By default, shapes are filled, and
// noFill() has to be used otherwise.
noFill();
stroke(255);

// Polyline
beginShape();

// Now we can define  vertices ...
vertex(25, 25);
vertex(175, 25);
vertex(175, 175);
vertex(25, 175);

// End of shape
endShape();
```

Shapes

- With CLOSE, shape will be closed:

```
// Polyline
beginShape();

// Now we can define the vertices ...
vertex(25, 25);
vertex(175, 25);
vertex(175, 175);
vertex(25, 175);

// End of shape
endShape(CLOSE);
```
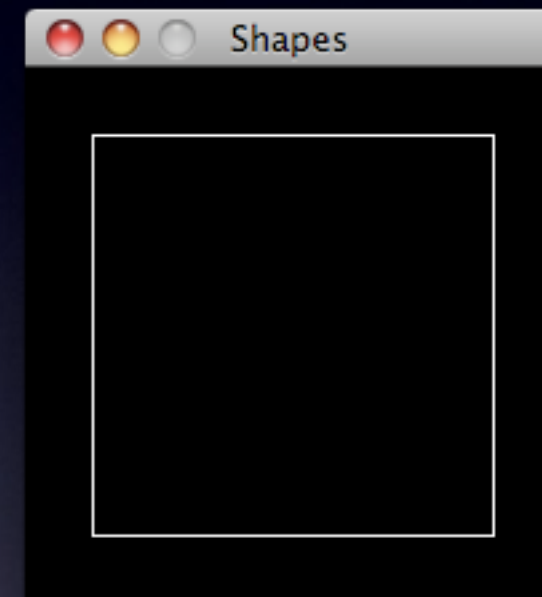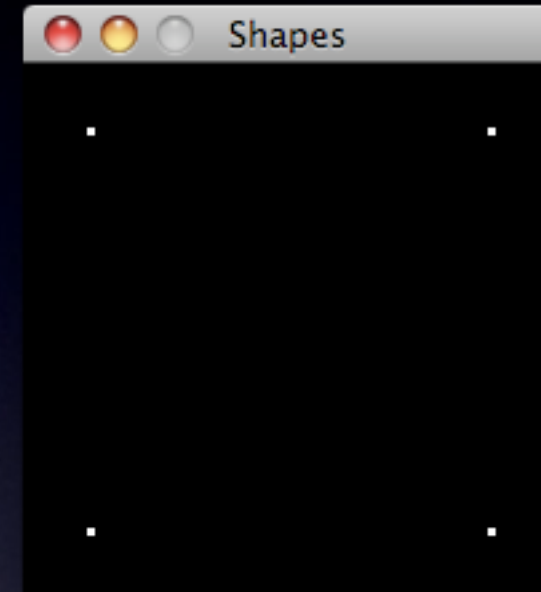
Shapes

- There is also the possibility of specifying an argument to the `beginShape()` function and allow another interpretation for the vertices:
    - `POINTS`. The vertices draw a set of points.
    - `LINES`. The vertices, in pairs, draw lines.
    - `TRIANGLES`. The vertices, in groups of three, draw triangles.
    - `TRIANGLE_STRIP`. A triangle strip.
    - `TRIANGLE_FAN`. A triangle fan.
    - `QUADS`. The vertices, in groups of four, draw quadrilaterals.
    - `QUAD_STRIP`. A quadrilateral strip.
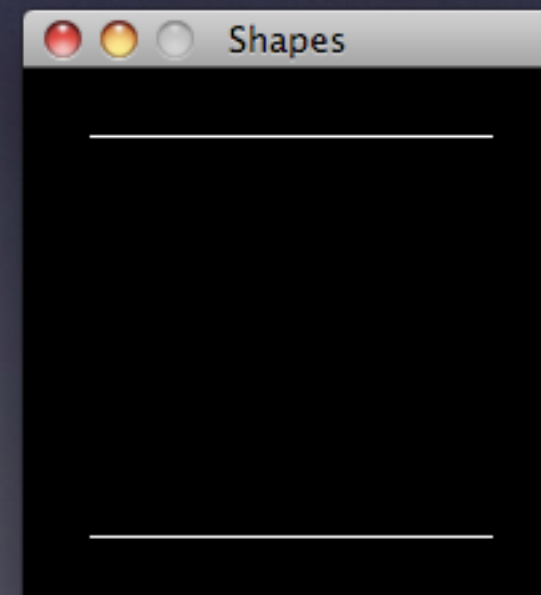
Shapes

```
// Points
beginShape(POINTS);

// Now we define the vertices ...
vertex(25, 25);
vertex(175, 25);
vertex(175, 175);
vertex(25, 175);

// End of shape
endShape();
```



```
// Lines
beginShape(LINES);

// Now we define the vertices ...
vertex(25, 25);
vertex(175, 25);
vertex(175, 175);
vertex(25, 175);
```
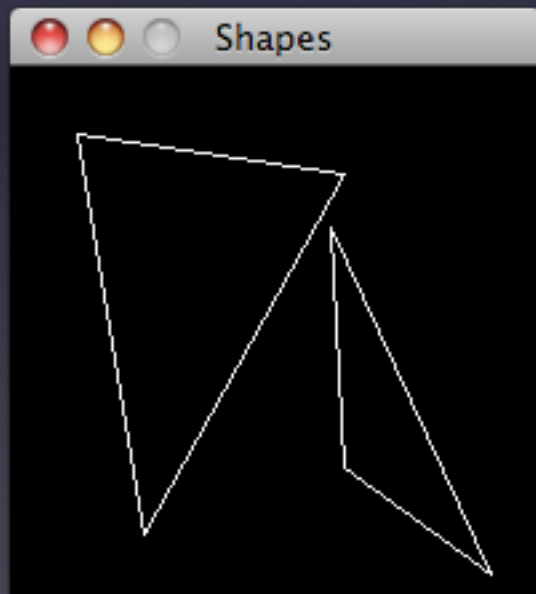
Shapes

```
// Triangles
beginShape(TRIANGLES);

// Now we can define the vertices ...
vertex(25, 25);
vertex(50, 175);
vertex(125, 40);

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

// End of shape
endShape();
```
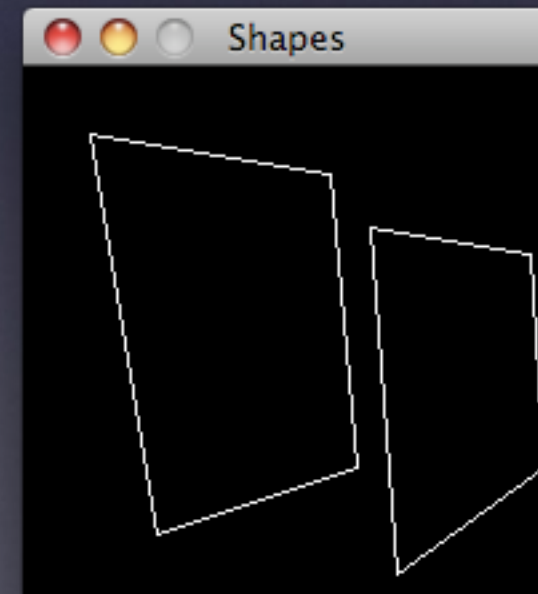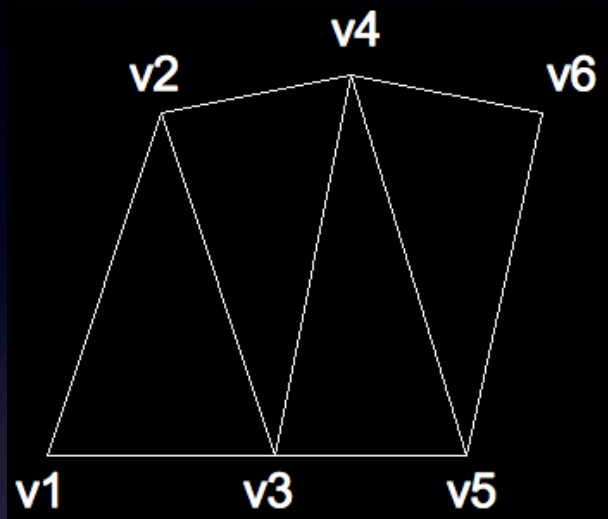
```
// Quadrilaterals
beginShape(QUADS);

// Now we can define the vertices ...
vertex(25, 25);
vertex(50, 175);
vertex(125, 150);
vertex(115, 40);

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

// End of shape
endShape();
```
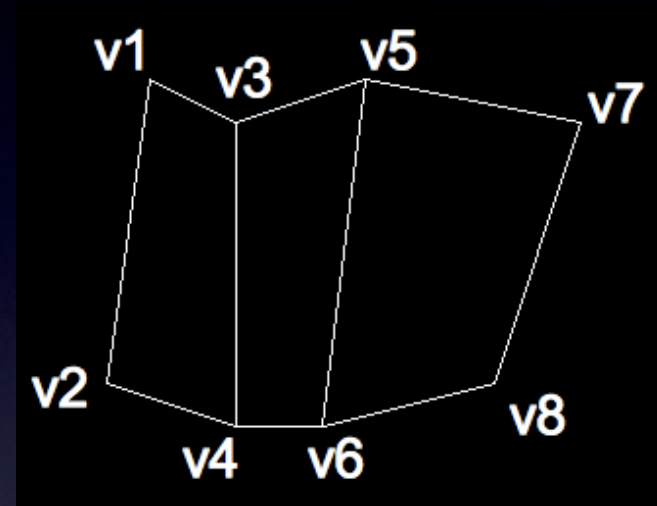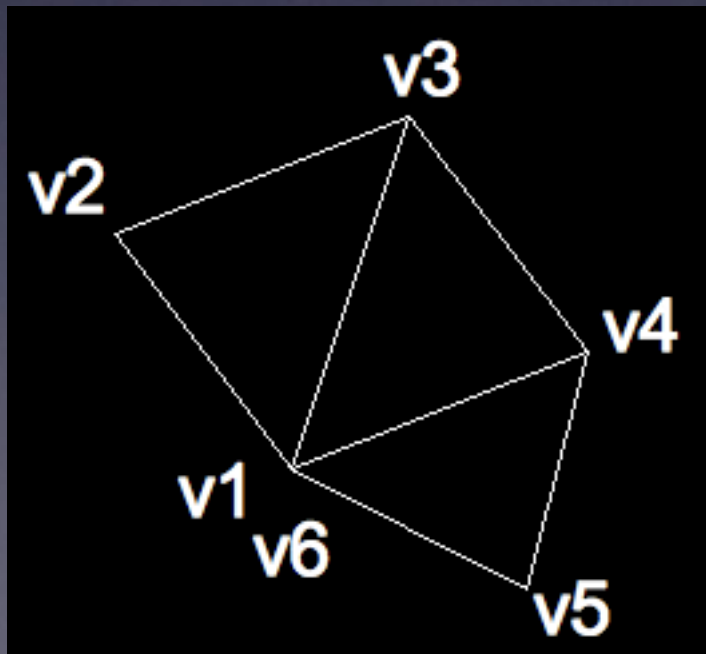
Shapes

TRIANGLE_STRIP

QUAD_STRIP

TRIANGLE_FUN
(Last point must be repeated)

Shapes

- In addition to specify the vertices with `vertex()`, *processing* allows the use of the functions `curveVertex()` and `bezierVertex()`, generating curves instead of straight lines
- These functions only work with the non-parameter version of `beginShape()`
- With these functions, we can create a chain of bézier or Catmull-Rom cubic curves either with `bezierVertex()` or `curveVertex()`
- They can be combined with the `vertex()` function and, consequently, generate complex shapes

Shapes

`curveVertex(x, y)`

- With `curveVertex()` a Catmull-Rom curve can be generated able to interpolate any set of points
- The first and last defined points with `curveVertex()` will act as control points and, thus, they will define the initial and ending curvature of the chain of curves
- The interior points will be the ones interpolated by a series of cubic curves
- A minimum of 4 vertices are compulsory, using `curveVertex()`, inside the `beginShape()` / `endShape()`

Shapes

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

// Polyline
beginShape();

// Now we can define the vertices ...
curveVertex(0, 0);

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

  curveVertex(x, y);      // Curve point
  rect(x-2, y-2, 4, 4); // We draw them to better appreciate
}

curveVertex(width-1, height-1);

// End of shape
endShape();
```
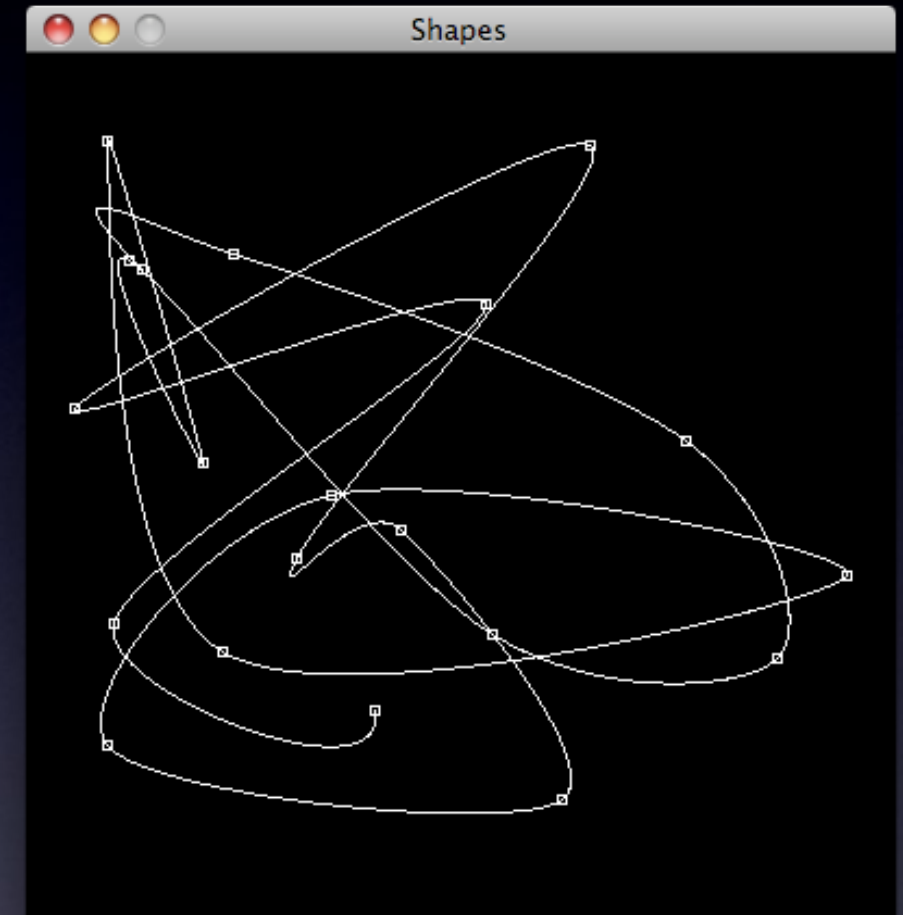
Shapes

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

- In a chain of bézier curves, the first point of the curve must be specified using `vertex()`
- Each `bezierVertex()` executed after this first vertex will specify two new control points and a destiny point
- The destiny point will be the initial point in front of new calls to `bezierVertex()`
- The last point in a bézier will be the first point of the next one. We can ensure continuity in the curvature between two curves by forcing linearity among: the last control point of the first curve, the first control point of the second curve and the last and first points of the curves (which are actually the same)

Shapes

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

// Polyline
beginShape();

vertex(10,100);  // First point of the curve
bezierVertex(70, 140, 160, 140, 150, 100);
bezierVertex(140, 60, 210, 60, 290, 100);

// End of shape
endShape();

// We draw the points
// to better see the result
strokeWeight(5);

// Points of the curve     100); point(290, 100);
stroke(255);
point(10, 100); point(150,

// Control points           140);
stroke(255, 0, 0);
point(70, 140); point(160,

point(140, 60); point(210, 60);

// Control points in a line ...
strokeWeight(1);
line(160, 140, 140, 60);
```
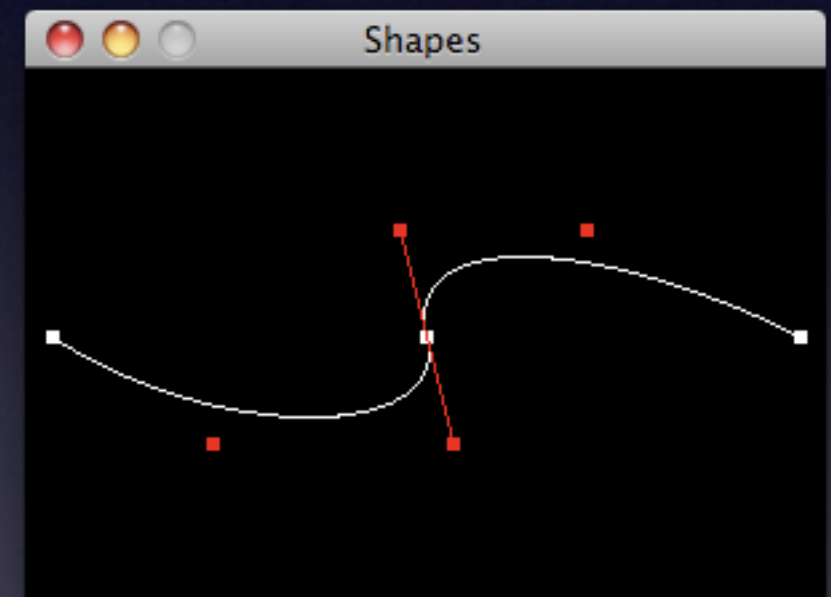
Shapes

```
// Complex shapes creation
// by combining linear and
// curve vertices.
size(400, 300);
background(0);
stroke(255);
noFill();

// Polyline
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();
```
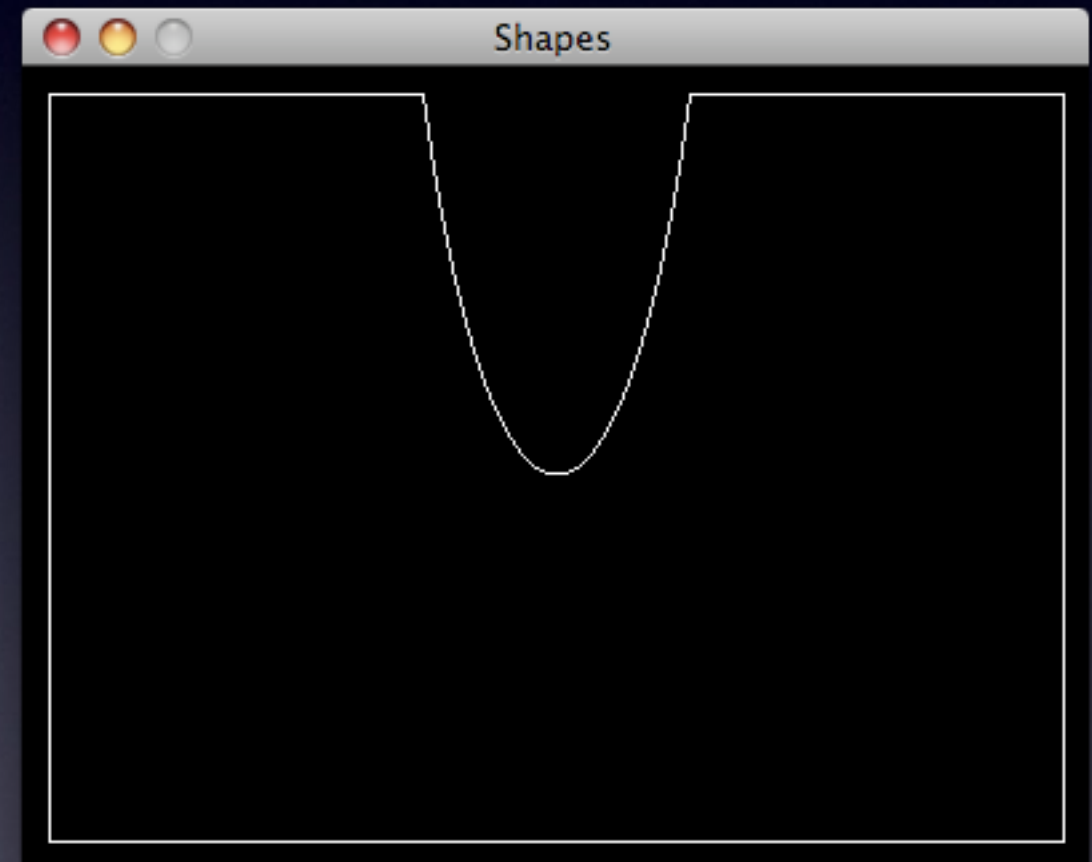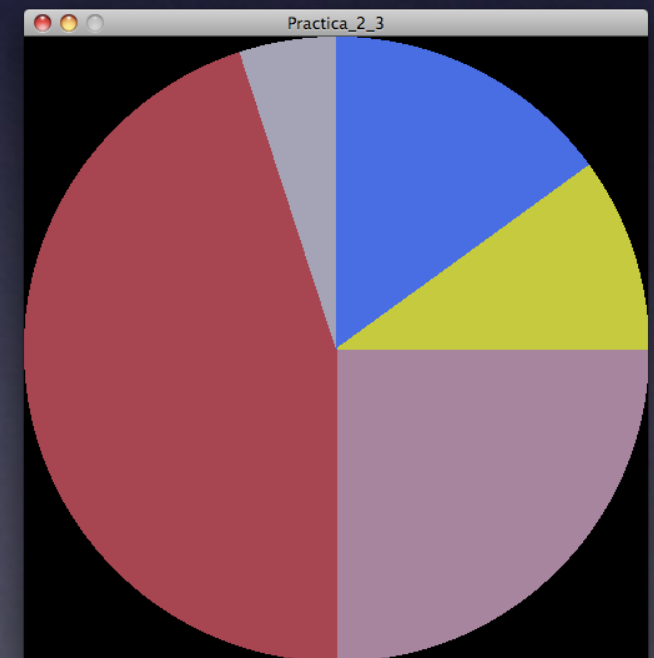
Practice 2-2

- Rewrite the practice 1-1 code in order to use a shape to draw the n-sided polygon

Practice 2-3

- In a statistical application it is necessary to represent a pie chart for a set of percentage values
- The elements are in an array and their sum is equal to 100%
- Example:

```
float[] values = {25.0, 45.0, 5.0, 15.0, 10.0};
```

- Implement a function to draw this information in a pie chart receiving this array as input
- The function will also receive as input the center and radius of the chart
- This function can know the number of values of the array by using the `length` attribute
- Choose a random color for each sector

Practice 2-4

- Develop an analogous function of practice 2-3 but now with the objective of drawing a 2D bar chart
- Take the most of the available window sizes and draw the coordinates axes
- Choose the bar width in function of the total number of values and the total window width; and the bar height in function of its particular value and the maximum value of all the elements, taking into account that the height of the bar of the element of maximum value is going to be the height of the window