### Primitive Grafiche in Processing

Analizziamo gli elementi costruttivi base per disegnare con Processing, partendo da quelli a zero dimensioni, ossia i punti.

#### Punti (0D)

L’elemento grafico indispensabile per disegnare qualsiasi oggetto è il punto.
Per disegnare un punto si usa il metodo point(). Chiamato con due parametri disegna un punto nello spazio 2D; se chiamato con 3 parametri nello spazio 3D.
Come per tutti gli elementi grafici 2D, è possibile impostare lo spessore e il colore della linea con strokeWeight() e stroke().
Ad esempio:

stroke(180,0,0);

strokeWeight(20);

point(60,60);

disegna un punto rosso da 20 px di diametro.



#### Primitive lineari (1D)

**Linee rette**. Con il comando line() è possibile disegnare una linea tra due punti, sia nello spazio 2D che nello spazio 3D. Anche in questo caso, come per il punto, è possibile definire il colore e lo spessore della linea.

line(20, 20, 100, 100); //x1,y1, x2, y2

**Linee Curve**. In Processing è possibile disegnare due tipologie di linee curve: spline di Catmull-Rom e curve di Bezier [[3](http://www.mokabyte.it/2008/05/processing-2/#ref)].

In entrambi i casi, le funzioni per disegnarle accettano 4 punti come parametri:

Curve, disegna una spline: il secondo ed il terzo punto sono i punti d’inizio e fine, mentre il primo e l’ultimo sono punti di controllo. La curva passa per tutti e 4 i punti, ma viene disegnata solo tra il secondo ed il terzo.

curve(5, 26, 73, 24, 73, 61, 15, 80);



Figura 4 – Spline e i suoi punti di controllo

Bezier, per la curva di bezier, al contrario, il primo e l’ultimo sono l’inizio e la fine della curva mentre, il secondo e il terzo sono punti di controllo, ossia i punti finali delle linee tangenti all’inizio e alla fine della curva.

bezier(85, 20, 10, 10, 90, 90, 15, 80);



 Figura 5 – Curva bezier e sue tangenti

In entrambi i casi per avere curve più lunghe e con maggiori punti di controllo bisogna usare una serie di chiamate ai metodi curveVertex() e bezierVertex() racchiuse nel blocco beginShape()/endShape().

beginShape();

curveVertex(5,195);

curveVertex(84,91);

curveVertex(68,19);

curveVertex(21,17);

curveVertex(32,100);

curveVertex(150,100);

endShape();

#### Primitive 2D

**Triangoli**. Il triangolo è l’elemento costruttivo base nella grafica 3D e viene usato per simulare le superfici continue (mesh). Per questa ragione in Processing esiste una funzione che permette di disegnare con una sola istruzione un triangolo, senza doverlo costruire come sequenza di linee rette (ovvero come un poligono generico).

triangle(10,10,30,70,80,80); //i tre vertici del triangolo

**Quadrilateri**. Processing ha due funzioni per disegnare delle figure geometriche a 4 lati:

* rect() che disegna un quadrilatero regolare accettando come parametri il vertice in alto a sinistra, l’altezza e la larghezza;
* quad() che invece disegna una quadrilatero irregolare, definendo singolarmente i 4 vertici.

Per lasciare maggiore flessibilità è possibile cambiare il significato dei parametri della funzione rect() con il metodo rectMode().

//comportamento di default

rectMode(CORNER);

//il punto specificato è il centro del rettangolo e non il vertice

rectMode(CENTER);

//altezza e larghezza diventano la posizione del vertice in basso a destra

rectMode(CORNERS);



 Figura 6 – Rettangoli disegnati coi vari mode

 **Poligoni**. Lo abbiamo già visto parlando delle curve: includendo una serie di vertici nel blocco beginShape()/endShape() è possibile fare poligoni di un numero arbitrario di lati.

beginShape(POLYGON);

vertex(30, 20);

vertex(35, 15);

vertex(40, 25);

vertex(35, 30);

vertex(30, 27);

endShape(CLOSE);

**Figure curve**. La funzione per creare cerchi o ellissi è ellipse().

Il suo funzionamento è identico a quello dalla funzione rect(), con la differenza che invece che disegnare un quadrilatero, disegna una ellisse inscritta.
Anche in questo caso è possibile modificare il significato dei parametri della funzione con ellipseMode().



Figura 7 – Ellissi disegnate coi vari mode

#### Primitive 3D

In Processing esistono solo due primitive strettamente 3D: sfere e cubi.
La funzione box() crea un cubo quando invocata con un solo parametro, mentre crea un parallelepipedo quando chiamata con le tre dimensioni.
La funzione sphere() disegna un una sfera del raggio specificato. È possibile modificare la qualità della sfera disegnata cambiando il numero di vertici con il metodo sphereDetail().



Figura 8 – Cubi e sfere

### Trasformazioni di coordinate

Le trasformazioni di coordinate consentono di spostare l’origine degli assi o di ruotarli.
Abbiamo 3 funzioni per farlo:

1. rotateX(): ruota la vista corrente attorno all’asse X e intorno all’origine degli assi (lo stesso vale per rotateY() e rotateZ()). In uno scena 2D l’unica rotazione possibile è attorno all’asse Z, ma per evitare confusione esiste la funzione rotate() che funziona solo in 2D. Il parametro è l’angolo di rotazione in radianti.
2. translate(): trasla gli assi del valore specificato dai parametri (x e y nel 2D, x, y e z nel 3D)
3. scale(): aumenta o diminusce la dimensione degli oggetti. Per esempio scale(2.0) raddoppia le dimensioni di tutti gli oggetti.

Un comportamento comune a tutte le funzioni di trasformazione è che sono additive, cioè il loro effetto si somma. Ogni chiamata modifica la matrice di trasformazione [[4](http://www.mokabyte.it/2008/05/processing-2/#ref)] usata da Processing per calcolare la posizione di tutti i vertici disegnati sulla scena.

translate(50, 20);

translate(20, 20);

//equivale a

translate(70, 40);

Ultimo concetto da comprendere è lo stack delle trasformazioni: come ho detto poche righe fa, ogni volta che è chiamata una funzione di trasformazione la matrice viene cambiata. Come posso fare quindi a ruotare un figura in un verso e un’altra nel verso opposto? Potrei fare due rotazioni opposte, con la seconda di un angolo equivalente alla prima più l’effettiva rotazione del secondo oggetto. Questa soluzione va bene in casi semplici, ma in casi più complessi diventa difficile tenere traccia di tutte le trasformazioni fatte per poi poterle annullare.
Processing ci viene in aiuto con le funzioni pushMatrix() e popMatrix(): queste servono per “salvare” il sistema di coordinate correnti e, in seguito, per recuperarlo.
Usando queste funzioni, per risolvere il problema di prima, basta chiamare la funzione pushMatrix() prima di ruotare il primo oggetto, e poi chiamare la funzione popMatrix() per tornare al sistema di coordinate precedente.