

# References and Links

- SVG Essentials – O'reilly (2002)
- SVG – Andrew H. Watt – Mc Graw (Hill 2002)
- <http://www.adobe.com/svg/>
- <http://www.visionmoster.com>
- <http://www.html.it/svg>
- <http://www.w3schools.com/svg>
- <http://www.svg.org/>
- <http://svglbc.datenverdrahten.de>
- <http://burningpixel.com/svg>
- <http://www.kevlindev.com>

# Contenuti

- Cosa è SVG ? (Definizioni, esempi, applicazioni)
- Il sistema di coordinate.
- Forme Primitive (Linee, Rettangoli, Ellissi, Poligoni e Spezzate)
- Struttura di un documento SVG
- Trasformazioni nel piano (traslazioni e rotazioni)

# Contenuti 2

- Paths
- Patterns e Gradienti
- Testo
- Clipping e Masking
- Filtri
- Scripting e animazione (solo animazioni)

# Cosa è SVG ?

- SVG (Scalable Vector Graphics) è un'applicazione di XML per la rappresentazione di **oggetti grafici** in forma *compatta e portabile*
- (Esigenza) Esiste un forte interesse a creare uno standard comune di grafica vettoriale utilizzabile sul web

# Cosa è SVG (secondo W3)

- SVG è un linguaggio per descrivere oggetti grafici bidimensionali in XML.
- SVG ammette tre tipi di oggetti grafici:
  - Forme grafiche vettoriali (Paths formati da linee e curve)
  - Immagini (raster)
  - Testo
- Gli oggetti grafici possono:
  - Essere raggruppati
  - Avere uno stile (legami con CSS)
  - Essere trasformati (ruotati, ridimensionati, traslati)
- I testi possono
  - Essere strutturati in linguaggio XML (favorendo accessibilità e ricerca)
- I widgets possono essere trasformati, filtrati,
- I Disegni SVG possono essere dinamici e interattivi.
- Ciò avviene attraverso:
  - la gestione degli eventi di HTML
  - Un linguaggio di scripting (ECMA script = Javascript)
  - Il DOM

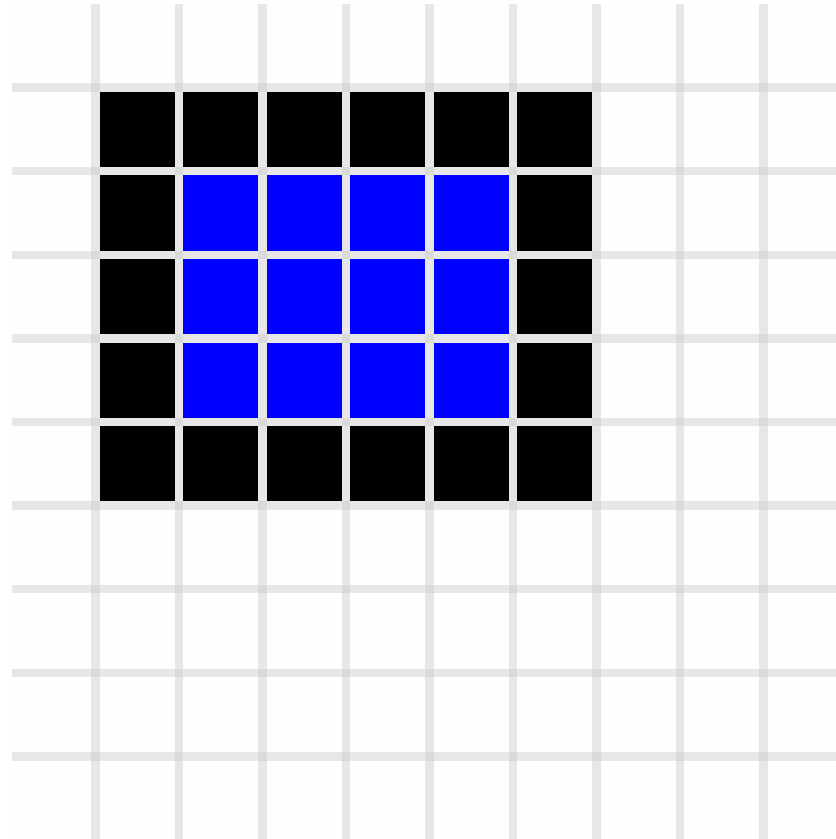
# Raster 1

- Nella grafica Raster un'immagine è rappresentata come una matrice i cui elementi sono detti *picture elements* (*pixel*).
- Ogni pixel è descritto da un colore RGB o da un indice ad una palette di colori

# Raster 2

- La matrice di pixel (chiamata bitmap) viene memorizzata in uno specifico formato (Jpeg, Gif, Window Bitmap, PNG, Tiff) in forma compressa.
- Un'immagine vale più di mille parole (ma richiede molto più spazio...)

# Esempio: Rettangolo Raster

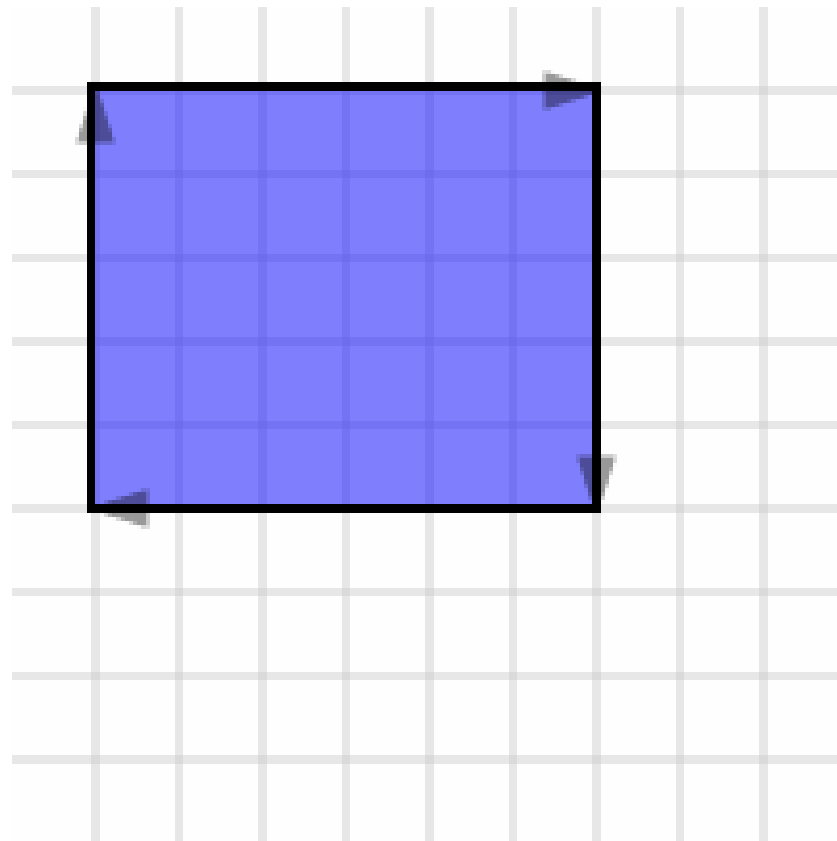




# Grafica Vettoriale

- Nella grafica vettoriale un'immagine è descritta come una serie di forme geometriche.
- Piuttosto che una serie “finita” di pixel, un visualizzatore di immagini vettoriali riceve informazioni su come disegnare l'immagine sul DISPLAY DEVICE in uno specifico sistema di riferimento.
- Le immagini vettoriali possono essere stampate con strumenti appositi (plotter)

# Esempio: Rettangolo Vettoriale



# Differenze

Raster	Vector
Quali puntini devo colorare ?	Quale linee devo tracciare ?
Disegno a mano libera	Disegno tecnico

# Confronto

## Raster

### *Pro*

- Fotorealismo
- Standard su Web

### *Contro:*

- Nessuna descrizione semantica.
- Grandi dimensioni

## Vector

### *Pro*

- Le trasformazioni sul piano sono semplici (Zooming, Scaling, Rotating)

### *Contro:*

- Non fotorealistico
- Formati vettoriali proprietari

# Utilizzo del vettoriale

- **CAD** (Computer Assisted Drawing) usa grafica vettoriale per misure precise, capacità di zoomare dentro i particolari dei progetti, ecc. (AutoCad,...)
- **Desktop Publishing & Design** (Adobe Illustrator, Macromedia Freehand, Publisher)
- Linguaggio di stampa **Postscript**
- **Animazioni** su web (Macromedia Flash)
- **GIS** (Geographical Information Systems): Arcview, Envi,...

# Vantaggi di SVG

- **Open Source:** I file sono scritti in un linguaggio comune a tutti (niente formati binari incomprensibili)
- **Web Based:** SVG apre il web alla grafica vettoriale (occupa poco spazio)
- **XML Based:** Integrazione con altri linguaggi XML (Xhtml, MathML, ecc.) ed estensibilità

# Un esempio concreto di scalabilità



Cat

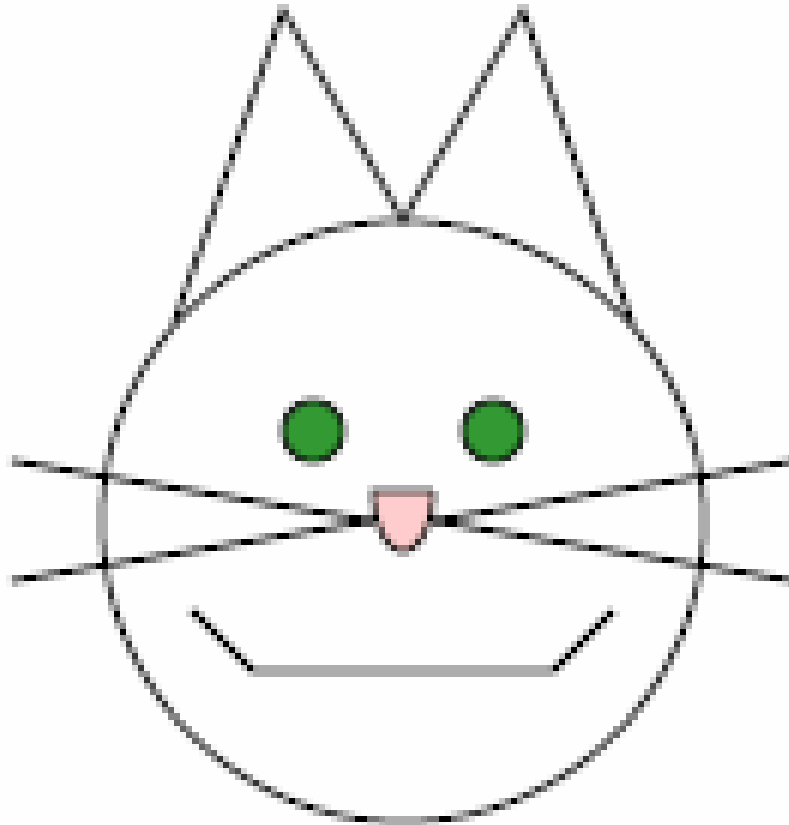
Raster  
(140x170)



Cat

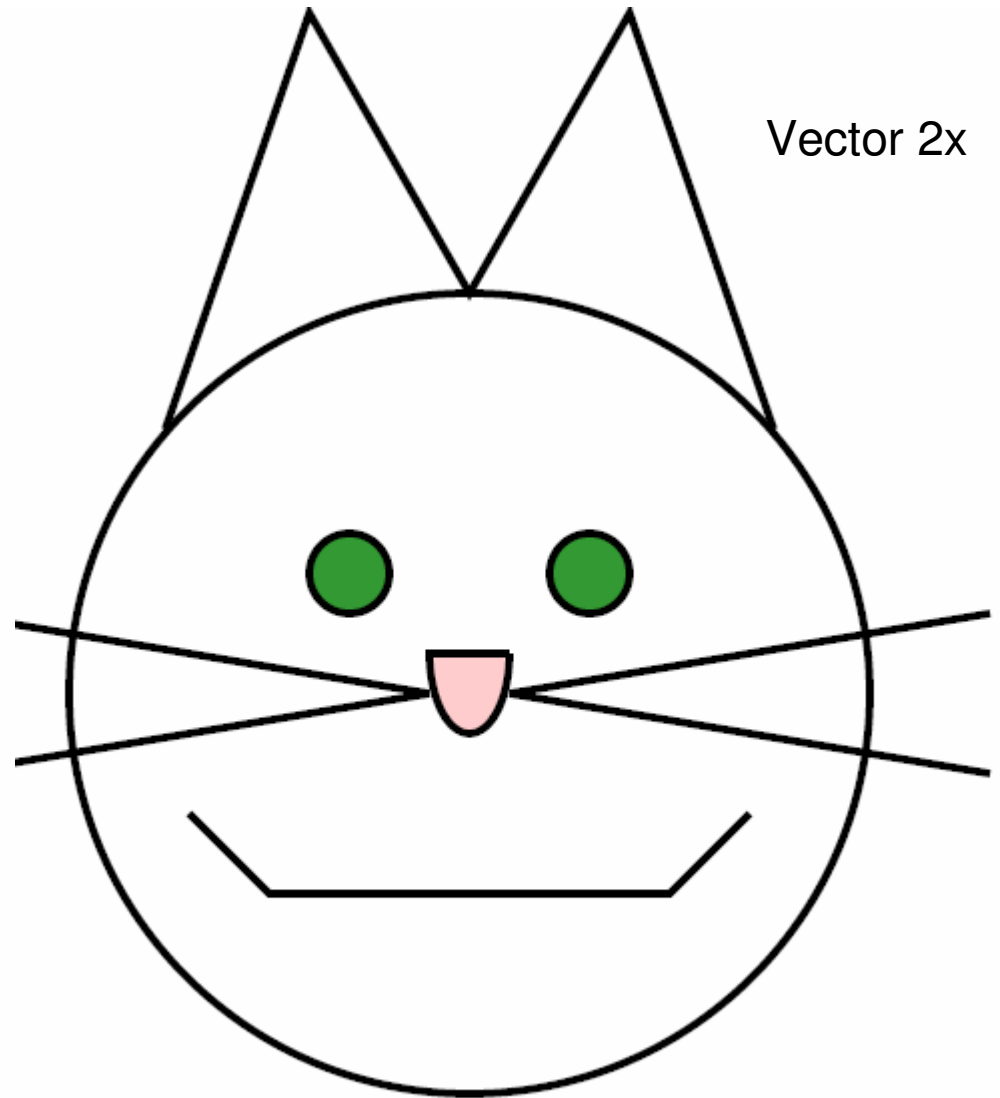
Vector  
(140x170)

Raster 2x



Cat

Vector 2x



Cat



# Struttura di un documento SVG

```
<?xml version="1.0"?>
```

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG  
1.0//EN" "http://www.w3.org/TR/2001/REC-SVG-  
20010904/DTD/svg10.dtd">
```

```
<svg width="140" height="170">
```

```
<title>Titolo</title>
```

```
<desc>Descrizione</desc>
```

```
</svg>
```

# Il sistema di riferimento

- Il “mondo” di SVG è un foglio infinito.
- Tuttavia occorre stabilire le dimensioni della tela (canvas) e il sistema di riferimento in quest’area.
- L’area di lavoro in SVG si chiama **viewport**.
- Le dimensioni del viewport sono settate dagli attributi *width* e *height* del tag di apertura <svg>
- Es. <svg width=“400” height=“400”>

# Unità di misura

- px – pixels
- cm – centimetri
- mm – millimetri
- in – inches (pollici)
- pt – punti ( $1/72$  di pollice)
- em – dimensione del font di default
- ex – altezza del carattere x

# Esempi

`<svg width="400" height="400">`

Equivale a `<svg width="400px" height="400px">` e  
crea un viewport di 400x400 pixels

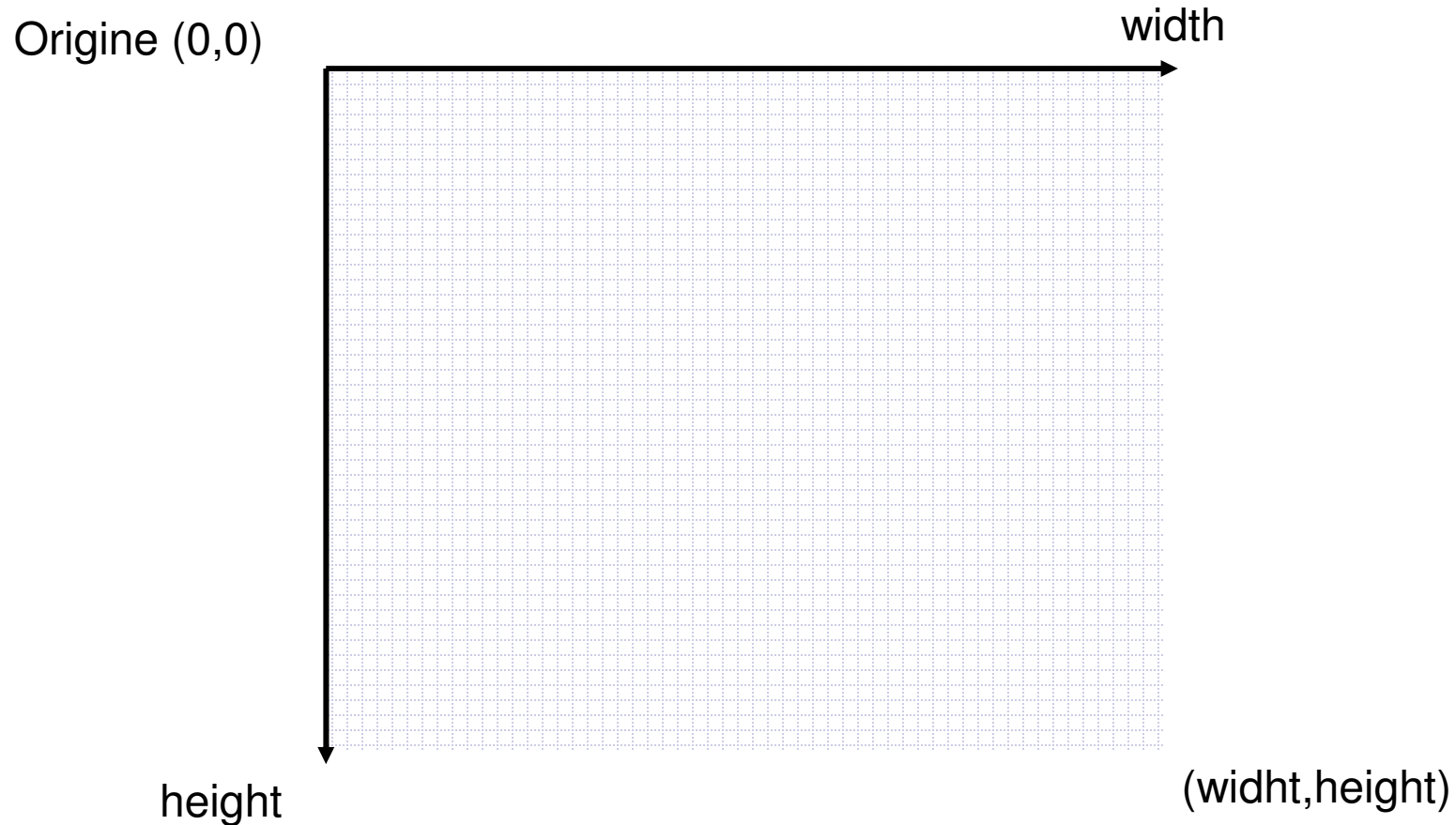
`<svg width="15cm" height="10cm">`

Tipiche dimensioni di una fotografia (in italia)

`<svg width="4in" height="3in">`

Tipiche dimensioni di una fotografia (in america)

# Il sistema di riferimento di default (in pixels)



# Esempio

```
<svg width="500" height="500">  
<rect x="0" y="0" width="200" height="200"  
  style="stroke:black;fill:none;"/>  
<rect x="0" y="0" width="150px" height="150px"  
  style="stroke:black;fill:none;"/>  
<rect x="0" y="0" width="7cm" height="7cm"  
  style="stroke:black;fill:none;"/>  
<rect x="0" y="0" width="140pt" height="140pt"  
  style="stroke:black;fill:none;"/>  
</svg>
```

# Sistema di coordinate di default

- Specificare l'unità di misura del viewport non cambia il sistema di coordinate che sarà sempre in pixels.

# Cambiare l'unità di misura di lavoro

È possibile ri-suddividere le dimensioni della griglia specificando il nuovo sistema di riferimento con l'attributo viewBox di svg.

```
<svg width="100" height="100"  
viewBox="0 0 50 50">  
<rect x="10" y="10" width="40" height="40">  
</svg>
```



# Perchè una viewBox ?

- Usando un viewBox, è possibile disegnare in un ambiente “generale”.
- Il luogo dove verrà visualizzato (=viewport) potrebbe variare a seconda l’uso che ne voglio fare (ad esempio per monitor con risoluzione con 1024x768 o 800x600 o 640x480 o su carta A4, A3 o ancora per stampare l’oggetto in più pagine – stile Corel Draw)
- In altre parole il viewBox è come un oggetto che viene poi incorporato dentro il luogo di visualizzazione.

# Esempi

Un cerchio di raggio 250 pixels dentro:

- Dimensioni Naturali
- Un desktop 1024x768
- Un desktop 800x600
- Un foglio di carta A4
- Negli esempi il cerchio viene “scalato” in modo da riempire il **viewport** e rispettando l’aspetto

# Preservare l'Aspect Ratio (AR)

Se il rapporto delle dimensioni tra viewport e viewBox è lo stesso non ci sono problemi.

SVG offre tre possibilità di controllo dell'AR:

- *meet*: riscalare l'oggetto grafico uniformemente rispetto alla dimensione più piccola;
- *slice*: riscalare l'oggetto grafico uniformemente rispetto alla dimensione più grande e tagliare le parti che eventualmente finiscono fuori dal viewport;
- *none*: il view box viene riscalato nelle dimensioni del view port producendo distorsione.

# L'attributo preserveAspectRatio

Permette di specificare l'allineamento dell'oggetto riscalato e se vuole "incontrare" i bordi oppure essere "tagliato".

Il modello dell'attributo è

**preserveAspectRatio="alignment [meet|slice]"**

Dove alignment è una coppia formata da

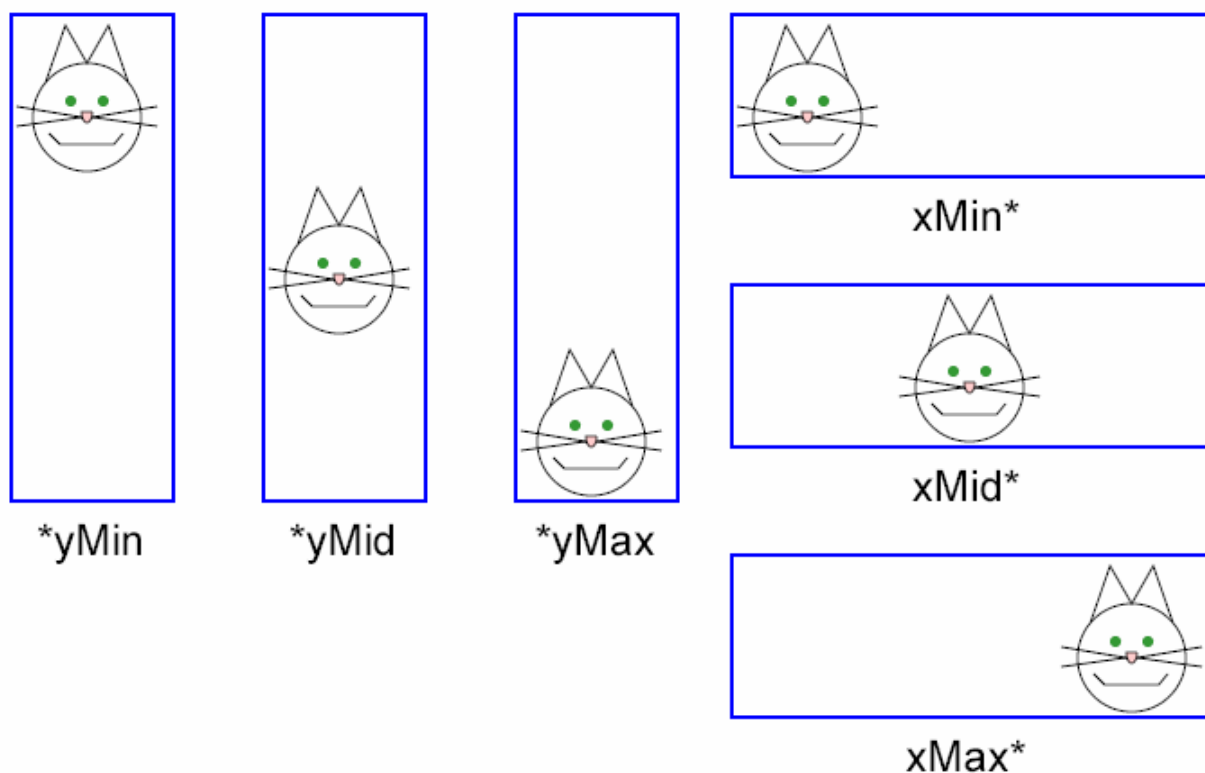
(xMin,xMid,XMax) x (YMin,YMid,YMax)

Esempio:

```
<svg width="45" height="120" viewBox="0 0 90 90"  
preserveAspectRatio="xMaxYMin meet">
```

# Esempio: meet

viewBox=90x90

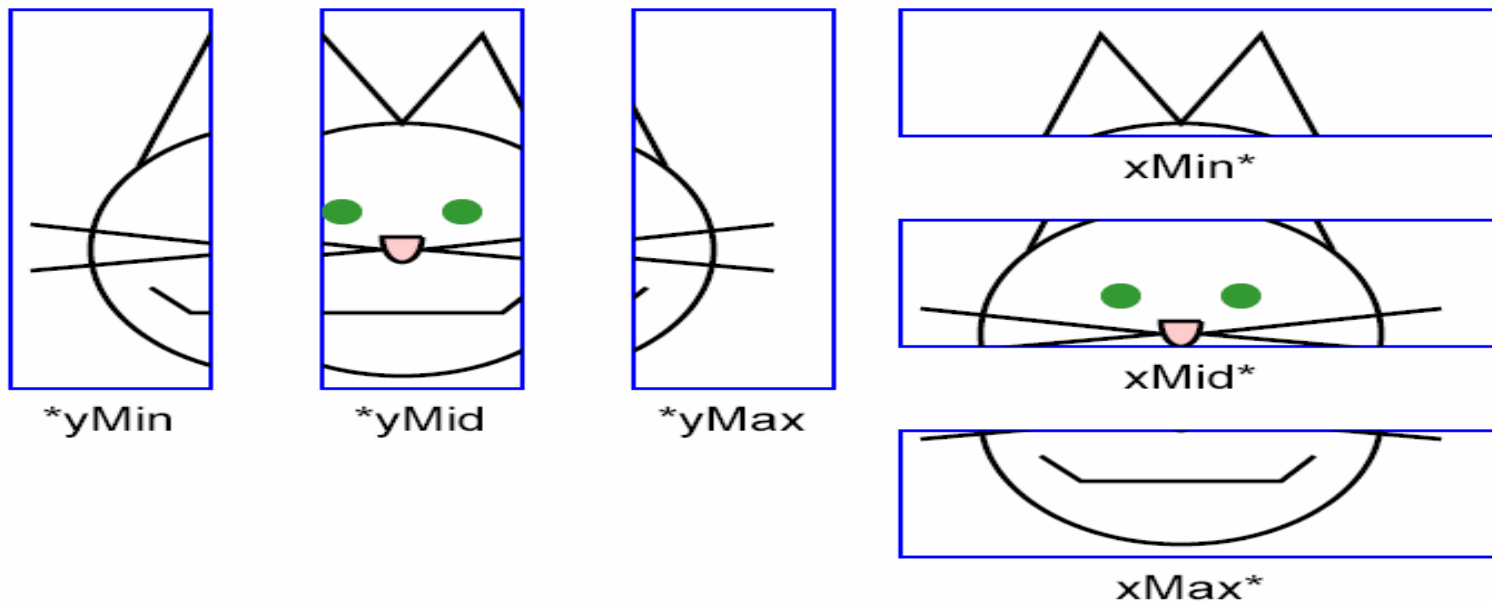


Viewport = 45x135

Viewport = 135x145

# Esempio: slice

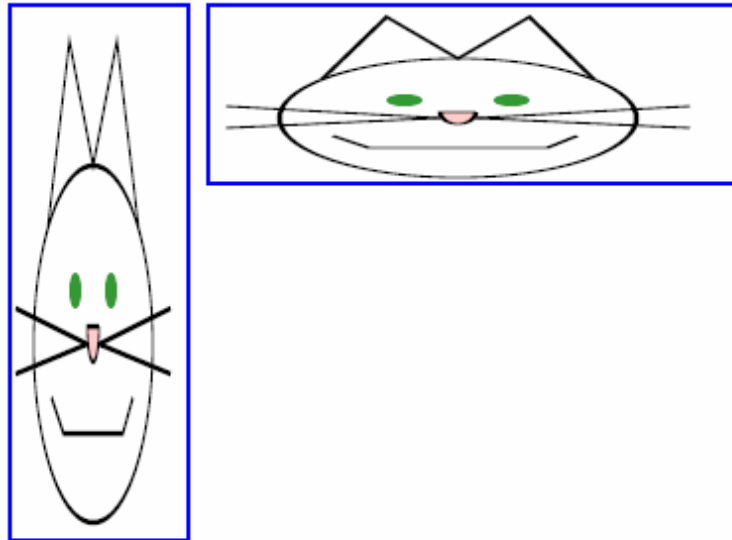
viewBox=90x90



Viewport = 45x135

Viewport = 135x145

# Esempio: none



# Annidare i viewport

Dentro un viewport, se ne possono definire di nuovi.

Cioè dentro un `<svg>` ...se ne possono creare tanti annidati.



# Forme elementari (primitive grafiche)

Le primitive grafiche di SVG sono:

- Linee
- Rettangoli
- Cerchi ed ellissi
- Poligoni
- Polylines

# Linee

- Una linea è un segmento avente per vertici  $(x1,y1)$  e  $(x2,y2)$  dove le coordinate sono espresse in qualche unità di misura ammissibile

- Sintassi

```
<line x1="startx" y1="starty" x2="startx"  
y2="starty"/>
```

# Attributi delle linee

- Vanno specificati tramite l'attributo style, e (come nei CSS) sono formati da coppie **proprietà:valore;**

**stroke-width:** intero (spessore linea)

**stroke:** colore (colore linea)

# Sui colori

I colori in SVG (come nei CSS) si possono rappresentare con:

- Keywords: acqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, tea, white e yellow;
- Forma esadecimale: #rrggbb;
- Forma esadecimale compatta: #rgb;
- Forma RGB: rgb(r,g,b) dove  $r,g,b \in [0,255]$

# Altri attributi di stile

■ **stroke-opacity**:  $x$  ( $x \in [0.0, 1]$ )

Indica l'opacità (e quindi la trasparenza) della linea.  
(0 trasparente, 1 opaco)

■ **stroke-dasharray**: lista di “trattino-spazio” in pixels

Es. `<line x1="0" y1="1" x2="150" y2="150" style="stroke-dasharray: 10,5,15,20;">`

Fa una linea in cui si ripetono trattino 10 spazio 5 trattino 15 spazio 20

Note: Se la lista è dispari, viene duplicata.

# Rettangoli

- Un rettangolo viene specificato tramite la coordinata dell'angolo in alto a sinistra, la sua grandezza e la sua altezza.

- Sintassi:

**<rect x="" y="" width="" height="" />**

# Proprietà di stile

- Per i rettangoli, così come per molte primitive bi-dimensionali, le proprietà di stile (quindi dentro `style=""`) sono:
- **stroke** (linea esterna: bordo) e varianti:
  - `stroke`, `stroke-width`, `stroke-opacity`, `stroke-dasharray`
- **fill** (riempimento) e varianti
  - `fill`, `fill-opacity`

# Rettangoli con bordi tondi

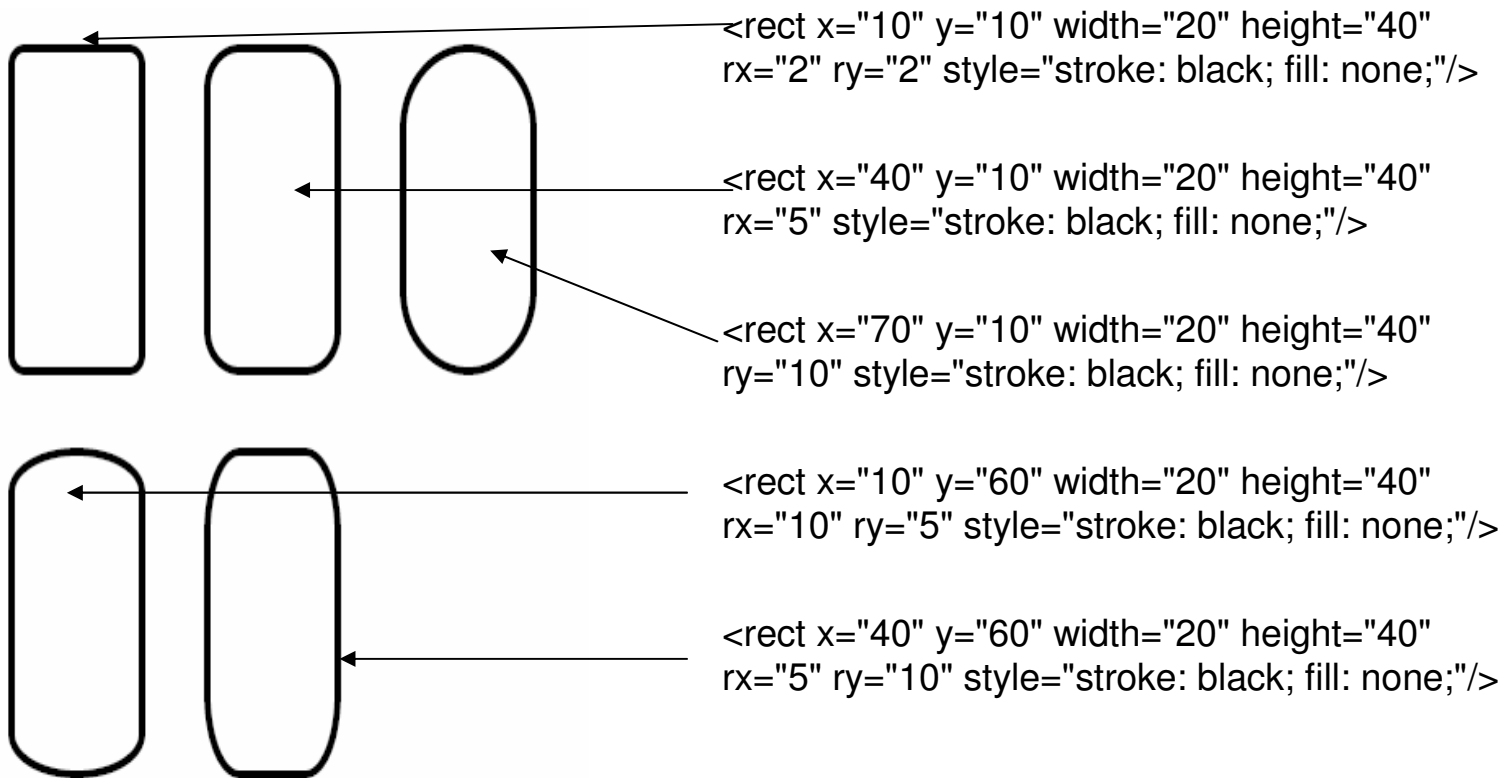
- È possibile specificare la curvatura del bordo del rettangolo tramite gli attributi

```
<rect ... rx="" ry="" />
```

Specificando uno solo dei due parametri l'altro verrà considerato uguale.



# Esempio



# Cerchi ed ellissi

- Un cerchio viene rappresentato da:

**<circle cx="" cy="" r="">**

Dove **(cx,cy)** sono le coordinate del centro ed **r** è il raggio (in pixels)

- Per creare un'ellisse la sintassi è

**<ellipse cx="" cy="" rx="" ry="">**

Sul bordo e il riempimento valgono gli stessi attributi di stile dei rettangoli.

# Poligoni

- I poligoni possono essere creati in SVG specificando la lista delle coordinate dei vertici (separati da virgole o spazi)

**<polygon points="x1,y1 x2,y2 ...xn,yn"/>**

Non si deve specificare nuovamente il vertice iniziale

Esempio:

**<polygon points="50,0 0,100, 100,100"/>**

# Poligoni: Riempimento

- Se i poligoni non hanno lati che intersecano, il riempimento avviene in maniera tradizionale con **fill**:...
- Nel caso che ci siano lati intersecanti SVG propone “dure” regole per determinare se un punto appartiene o meno al poligono.
- Queste regole vengono specificate dall'attributo *fill-rule*

# Fill-rule: nonzero

- **nonzero**: viene tracciata una linea *dal punto all'infinito* e vengono contate quante volte la linea interseca i lati del poligono seguendo la regola:
  - Se il lato va da destra a sinistra  $\text{cont} = \text{cont} + 1$
  - Se il lato va da sinistra a destra  $\text{cont} = \text{cont} - 1$
  - Se  $\text{cont} = 0$ , il punto è **fuori** dal poligono
  - Se  $\text{cont} \neq 0$ , il punto è dentro il poligono.

# Fill-rule: evenodd

- **evenodd**: viene tracciata una linea *dal punto all'infinito* e vengono contate quante volte la linea interseca i lati del poligono
- Se cont è dispari, il punto sta nel poligono
- Se cont è pari, il punto sta fuori.

# Polyline: spezzate

- `<polyline>` è un modo rapido di scrivere tante volte `<line>`
- Ha gli stessi attributi di `<polygon>`
- `<polyline points="" style="stroke:black;fill:none>`
- Conviene sempre settare il riempimento a ***none*** per evitare riempimenti strani

# Attributi delle linee.

- Fine linea: Vengono specificati per gli elementi `<line>` e `<polyline>` con l'attributo:  
**stroke-linecap**: [butt,round,shape]
- Line join: Specifica come unire due linee negli angoli e viene definito da:  
**stroke-linejoin**: [bevel, round, miter]



# Struttura e presentazione

- Così come l'HTML descrive il contenuto della pagina mentre la sua rappresentazione **dovrebbe** essere lasciata ai fogli di stile.
- Anche in SVG è stata prevista la possibilità di specificare lo stile dell'oggetto grafico **esternamente** alla definizione dell'oggetto stesso

# Gli stili in SVG

In SVG gli stili possono essere specificati in quattro modi (dal più importante al meno importante)

1. Inline styles
2. Internal stylesheets
3. External stylesheets
4. Attributi

# Stili *inline*

- Lo stile dell'oggetto viene definito nell'elemento tramite l'attributo *style*;
- Lo schema dello stile è:

**proprietà1:valore1; proprietà2:valore2;...**

# Stili interni

- Vengono specificati dentro l'elemento <svg> e dentro uno speciale elemento <defs> che vedremo in seguito.
- Lo stile viene definito da

```
<style type="text/css"><![CDATA [circle { fill:blue; fill-opacity: 0.5; }]]</style>
```

Tutti i cerchi definiti saranno riempiti di blu, e con opacità del 50%

# Fogli di stile esterni

- I fogli di stile esterni sono file di testo che raccolgono definizioni di stile che vanno applicate a più documenti SVG.
- L'estensione di un foglio di stile è .css (*Cascading Style Sheets*)
- Viene richiamato tra la definizione del documento .xml e il DOCTYPE

# Esempio

```
<?xml version="1.0"?>
<?xml-stylesheet href="ext_style.css" type="text/css"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
  "http://www.w3.org/TR/2001/REC-SVG-
    20010904/DTD/svg10.dtd">
<svg width="200px" height="200px" viewBox="0 0 200
  200"
  preserveAspectRatio="xMinYMin meet">

<line x1="10" y1="10" x2="40" y2="10"/>
<rect x="10" y="20" width="40" class="yellow" cx="70"
  cy="20" r="10"/>
<polygon class="thick" points="60 50, 60 80, 90 80"/>
<polygon height="30"/>
<circle class="thick semiblue"
  points="100 30, 150 30, 150 50, 130 50"/>
</svg>
```

```
{ fill:none; stroke: black; }
/* default for all elements */
rect { stroke-dasharray: 7 3;}
circle.yellow { fill: yellow; }
.thick
{ stroke-width: 5; }
.semiblue
{ fill:blue; fill-opacity: 0.5; }
```

# Presentation Attributes

- È possibile specificare le proprietà stile direttamente come attributo dell'elemento

Esempio:

```
<circle cx="" cy="" r="" fill="red"; stroke="yellow";/>
```

- Questo genere di rappresentazione “mischia” stili e contenuto. La sua priorità è l'ultima in scala.

# Raggruppare gli elementi: **g**

- Gli oggetti SVG possono essere raggruppati e richiamati.
- Il tag **<g id="nome\_gruppo">** apre la definizione degli elementi del gruppo.
- Dentro l'elemento **g** è possibile inserire i tag **<title>** e **<desc>**.
- È possibile stabilire uno stile che viene ereditato dagli elementi interni.



# Richiamare i gruppi: **use**

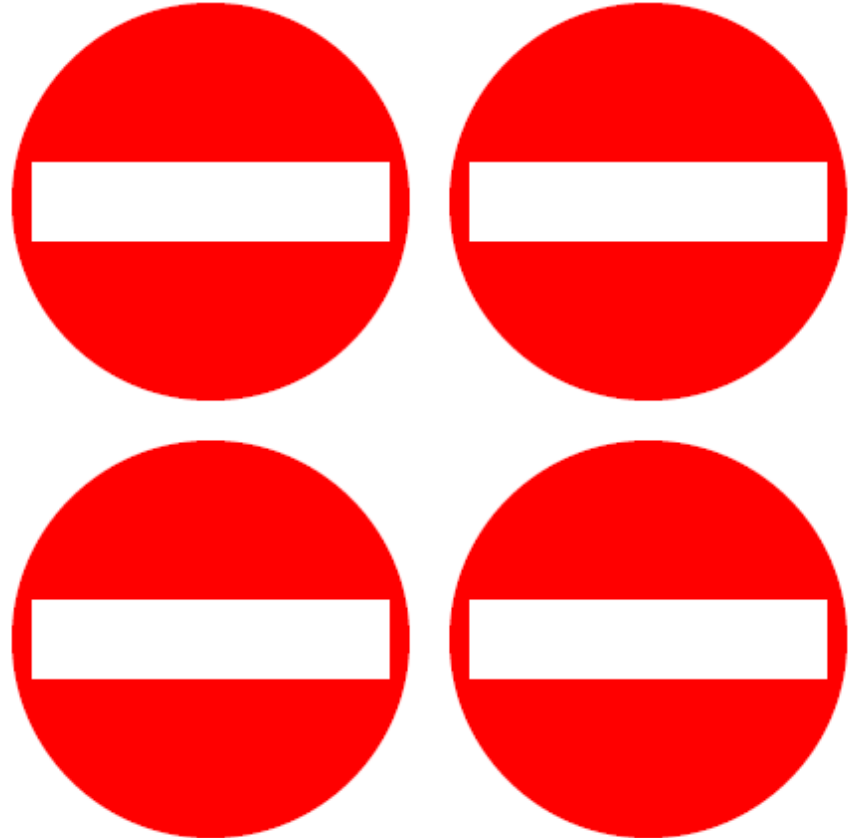
- Tramite il tag **<use>** definito da:

**<use xlink:href="#id\_elemento" x="" y="" />**

- Vengono ridisegnati tutti gli elementi di **<g>** nella posizione traslata rispetto all'originale di **x** pixels in orizzontale e **y** pixels in verticale (x, y possono assumere anche valori negativi)

# Esempio

```
<svg width="500" height="500">
<title>Divieto di accesso</title>
<desc>Un segnale stradale</desc>
<g id="divietoaccesso">
<circle cx="50" cy="50" r="50"
  style="fill:red"/>
<rect x="5" y="40" width="90" height="20"
  style="fill:white"/>
</g>
<use xlink:href="#divietoaccesso" x="110"
  y="0"/>
<use xlink:href="#divietoaccesso" x="0"
  y="110"/>
<use xlink:href="#divietoaccesso" x="110"
  y="110"/>
</svg>
```



# Definizioni: <defs>

È possibile definire degli elementi generici mettendoli tra **<defs>** e **</defs>**.

Tali elementi:

- Non vengono immediatamente disegnati;
- Possono “subire” futuri cambiamenti di stile;
- Riutilizzo più semplice.

# Vantaggi di **defs**

- L'elemento `<use>` può far riferimento a qualsiasi risorsa disponibile in rete.
- È possibile quindi creare un file svg con sole definizioni e poi richiamarli negli altri file svg.

# L'elemento **symbol**

- **<symbol>** permette di creare gruppi che non vengono visualizzati subito;
- Tuttavia è bene posizionarli nella zona delle definizioni per leggibilità del codice;
- **<symbol>** permette di definire un **viewBox** e l'attributo **preserveAspectRatio** da usare in combinazione con **<use ... width="" height="">**.

# Elemento **IMAGE**

- Con **<use>** è possibile riutilizzare una porzione di codice SVG interno.
- Con **<image>** è possibile includere:
  - Un file **.svg** intero
  - Una bitmap (JPEG o PNG)
- Sintassi:

**<image**     **xlink:href=**“”     **x=**“”     **y=**“”     **width=**“”  
          **height=**“”**>**

# Trasformazioni del sistema di coordinate

In SVG è possibile effettuare delle trasformazioni sul piano:

- Traslazioni;
- Scaling;
- Rotazioni;

Tutte queste azioni vengono eseguite tramite l'attributo **transform** da applicare ai relativi elementi (gruppi, primitive, ecc.).

# Traslazioni

- Le traslazioni le abbiamo già viste quando nell'elemento **<use>** abbiamo specificato gli attributi **x** e **y** che indicavano lo spostamento orizzontale e verticale dell'oggetto da riutilizzare.
- In generale la sintassi della traslazione è:

**<... transform="translate(x-value,y-value)"**

**x-value** e **y-value** rappresentano (in pixels) lo spostamento orizzontale e verticale del sistema di riferimento dell'elemento a cui è applicato.



# Scaling

- Lo **scaling** permette il ridimensionamento delle unità di misura del sistema di riferimento.
- La conseguenza di tale trasformazione è che l'oggetto risulta ingrandito o rimpicciolito o riflesso.

- La sintassi della trasformazione è:

**<.. transform="scale(r)">**

se si vogliono ridimensionare entrambi gli assi in modo eguale

**<.. transform="scale(rx,ry)">**

se il fattore di scala è diverso per i due assi.

# Rotazione

- La trasformata rotazione permette di ruotare il sistema di riferimento dell'oggetto secondo un centro di rotazione e un angolo.
- Se il centro di rotazione non viene espresso la rotazione viene effettuata rispetto all'origine.
- L'angolo viene espresso in gradi.

## Sintassi

**<... transform="rotate(angle,centerx,centery);"**

**<... transform="rotate(angle);"**

# Combinare le trasformazioni

- Dentro l'attributo **transform** è possibile effettuare qualsiasi combinazione di trasformazioni separandole con lo spazio

Esempio:

```
<... transform="rotate(30) translate(120,130)  
scale(2)">
```

Ovviamente l'ordine delle trasformazioni è determinante.

# Trasformazioni **skew** (obliquità)

- SVG permette di piegare gli assi di un certo angolo tramite le trasformate
- `skewX(angolo)` e `skewY(angolo)`

# PATHS

- Tutti le primitive descritte fino ad adesso sono *scorciatoie* per la nozione più generale di **<path>**
- L'elemento PATH crea forme tramite linee, curve e archi che possono essere adiacenti o partire da un punto arbitrario del canvas.
- L'elemento PATH ha un solo attributo (escluso lo stile) denominato **d** (che sta per data) che contiene una serie di:
  - Comandi (spostati\_la,disegna una linea, crea un arco)
  - Coordinate

# PATHS: M, L, Z.

- Il comando **M** x y (**moveto**) sposta il pennino sul punto di coordinate (x,y);
- Ogni PATH deve iniziare con un **moveto**;
- Il comando **L** x' y' (**lineto**) disegna una linea dal punto corrente fino a (x',y')
- Il comando **Z** (**closepath**) chiude il cammino unendo il punto corrente con quello iniziale.

# Esempio

```
<path d="
```

```
M 50,50
```

```
L 50,100
```

```
L 100,150
```

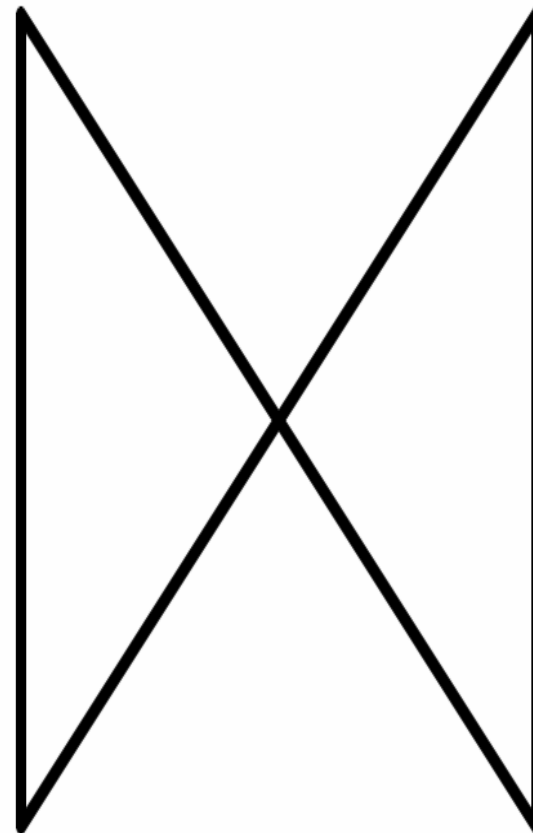
```
L 100,100
```

```
Z "
```

```
style="">
```

M 50,50

L 100,150



L 50,100

L 100,100

# PATH

- Inserendo una nuova **M** tra i comandi del **path**, si crea un nuovo sottocammino, cioè viene alzato il pennino e spostato nella nuova posizione pronto a ricevere altri comandi.

- Es (path2.svg)

```
<path d="M 50,50 L 150, 150 L 150,50 Z M 200,50  
L 300, 150 L 400,50 Z "  
style="stroke:black;fill:none;stroke-  
linejoin:round;"/>
```



# Esercizio:

- Scrivere il path generico relativo alle primitive grafiche:
  - ☐ Rect
  - ☐ Polyline
  - ☐ Polygon

# Soluzione: Rect

```
<rect x="startx" y="starty" width="larghezza"  
      height="altezza"/>
```

Diventa

```
<path d="M startx starty  
L startx+larghezza,starty  
L startx+larghezza,starty+altezza  
L startx,starty+altezza  
Z">
```

# Coordinate relative

- Esiste una versione “relativa” dei comandi **M** ed **L**. Per relativo si intende rispetto al punto precedente.
- I comandi “relativi” sono gli stessi ma in forma minuscola **m** ed **l**. Le coordinate dopo **m** e **l** sono espresse in pixels e possono anche assumere valori negativi.

# Il rettangolo “relativo”

```
<rect x="startx" y="starty" width="larghezza"  
      height="altezza"/>
```

Diventa:

```
<path d="M startx starty  
l larghezza,0  
l 0,altezza  
l -larghezza,0  
Z">
```

## *Shortcuts:* Scorciatoie per linee verticali ed orizzontali

- $H,(h) x'$  : fa una linea orizzontale dalla posizione corrente al punto  $(x',y\text{-attuale})$
- $V,(v) y'$ : equivalente verticale;

# Il rettangolo: parte terza

```
<rect x="startx" y="starty" width="larghezza"  
      height="altezza"/>
```

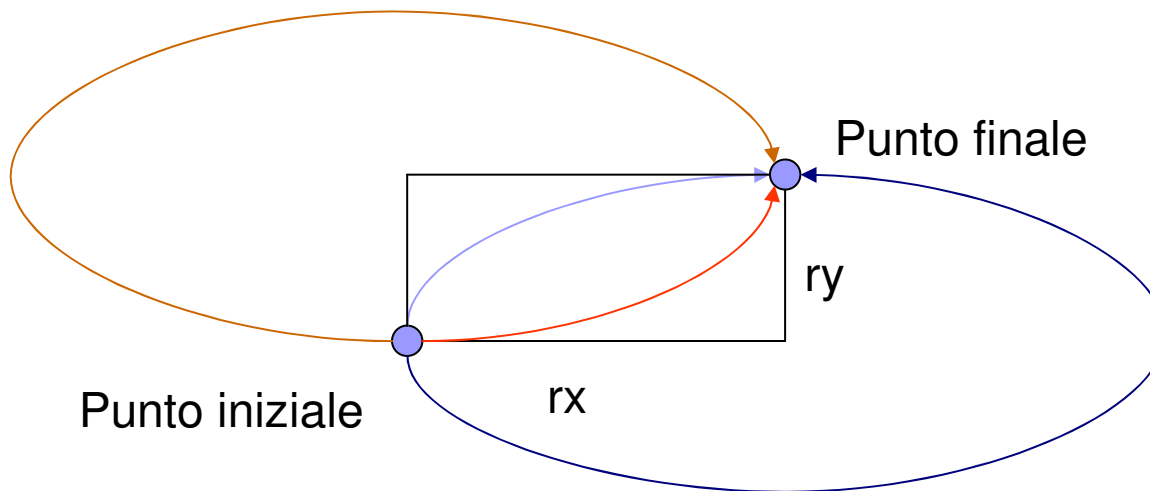
Diventa:

```
<path d="M startx starty  
h larghezza  
v altezza  
h -larghezza  
Z">
```

# PATH: Archi ellittici

- Gli archi rappresentano il tipo più semplice di curva. Vengono dichiarati con il comando A
- Un arco è definito da:
  - ☐ Raggio-orizzontale
  - ☐ Raggio-verticale
  - ☐ Large-arc-flag
  - ☐ Sweep-flag
  - ☐ Endx
  - ☐ Endy

# Archi: Teoria





# Curve di Bezier

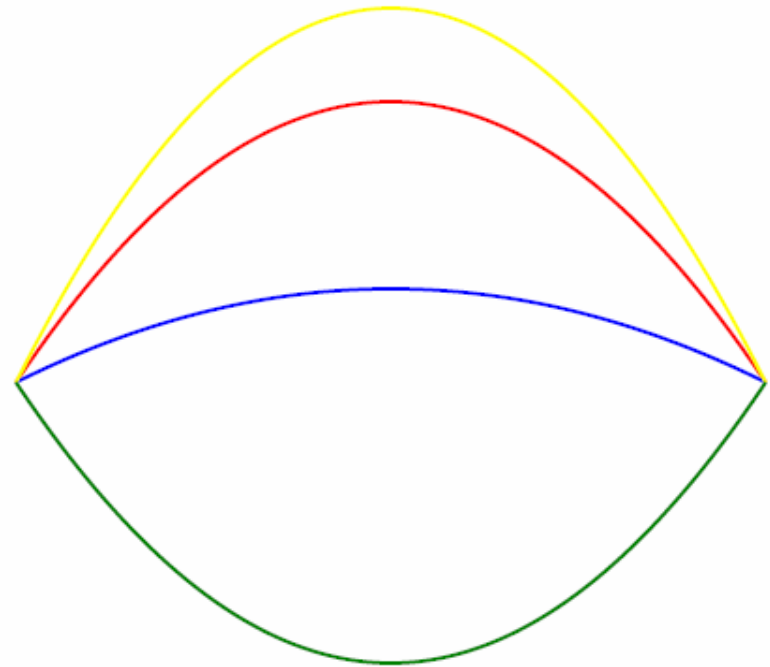
- Le curve di Bezier (da Pierre Bezier della Renault e indipendentemente anche da Paul de Casteljau della Citroen) introducono un modo computazionalmente conveniente di creare curve parametriche di secondo e terzo grado.
- Sono costituite dai punti di inizio e fine curva, più una serie di punti di controllo.
- Metaforicamente un punto di controllo è come un magnete che attira la linea (come se fosse deformabile) con + forza tanto quanto + il punto è vicino.

# Curve di Bezier quadratiche

- Sono caratterizzate da:
    - Punto iniziale
    - Punto finale
    - Un punto di controllo
  - La sintassi è (dentro un PATH)
- Q controlpointx controlpointy endx endy**
- Esiste anche la versione q (con coord. relative)

# Esempi

```
<path d="M 100 250 Q 200
100 300 250"
style="stroke:red;fill:none;"/>
<path d="M 100 250 Q 200
200 300 250"
style="stroke:blue;fill:none;"/>
<path d="M 100 250 Q 200 50
300 250"
style="stroke:yellow;fill:none;"/>
<path d="M 100 250 Q 200
400 300 250"
style="stroke:green;fill:none;"/>
```



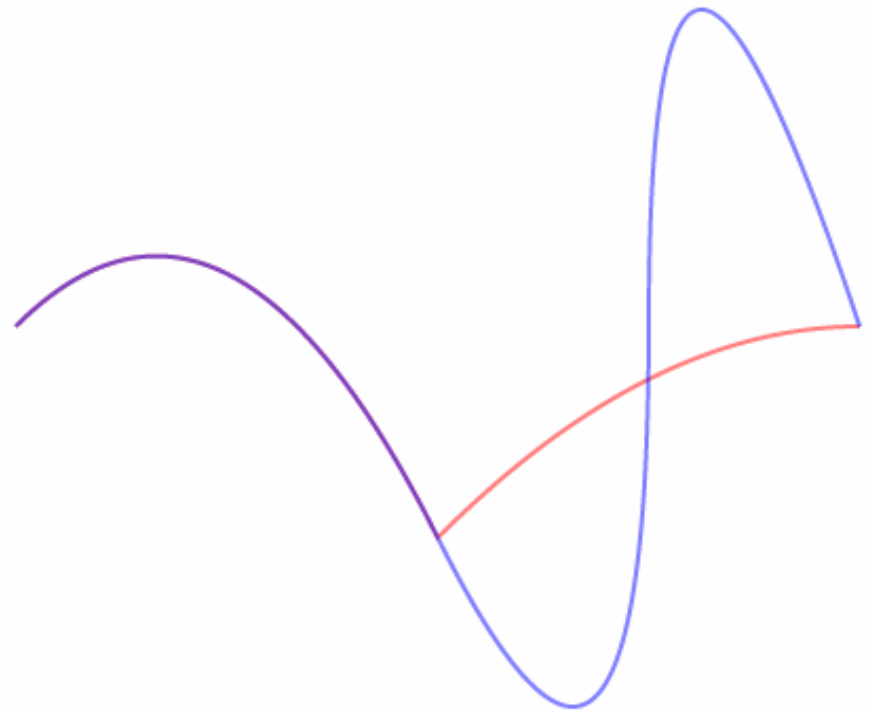
# Polybezier

- Si ottengono specificando altre coppie di control point e punto finale
- `<path... Q cpx1,cpy1 endx1, endx1  
cpx2,cpy2 endx2, endx2>`
- Se si vuole ottenere uno smoothing della curva si può usare T (o t) dopo la prima coppia.

# Polybezier con e senza smooth

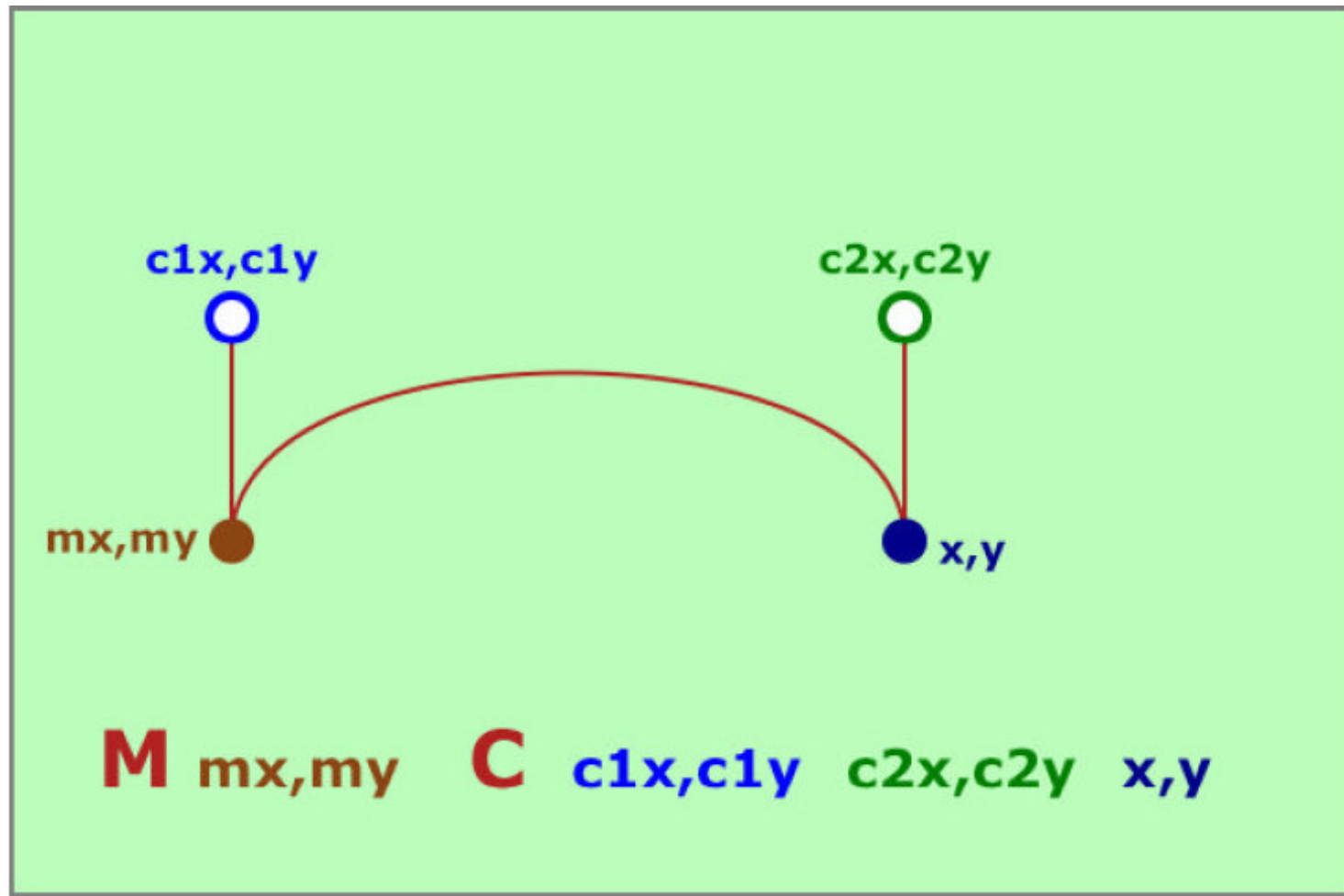
```
<path d="M 100 50 Q 150 0  
200 100 250 50 300 50"  
style="stroke:red;fill:none;  
stroke-opacity:0.5;"/>
```

```
<path d="M 100 50 Q 150 0  
200 100 T 250 50 300 50"  
style="stroke:blue;fill:none;s  
troke-opacity:0.5;"/>
```

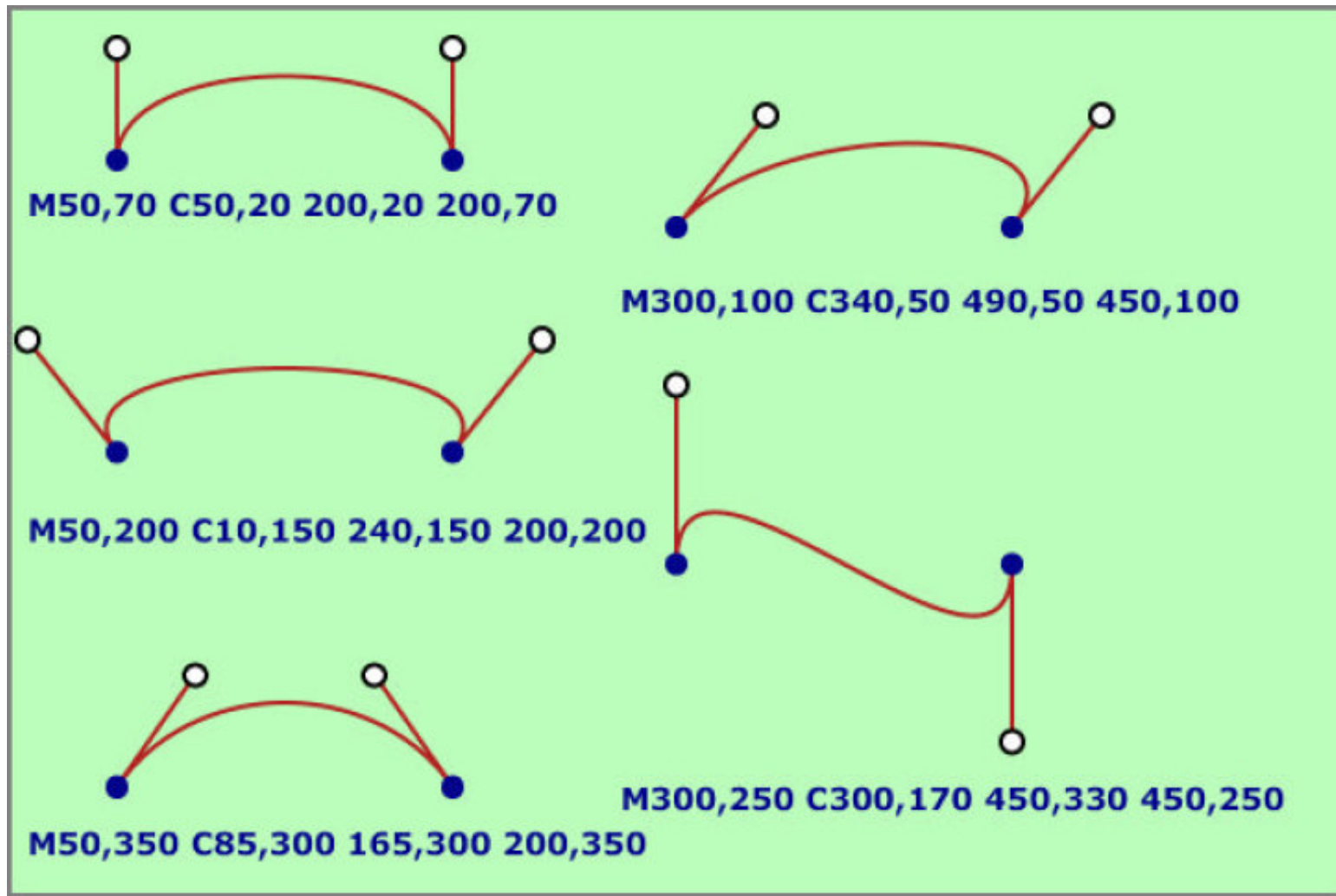


# Curve di Bezier cubiche

- Le curve di Bezier cubiche hanno due punti di controlli piuttosto che uno.
- La sintassi è:  
<... C CP1x CP1y CP2x CP2y Endx Endy>
- Esiste la versione relativa c
- Esistono le polybezier cubiche con e senza smooth (comando S)



# I Punti di controllo

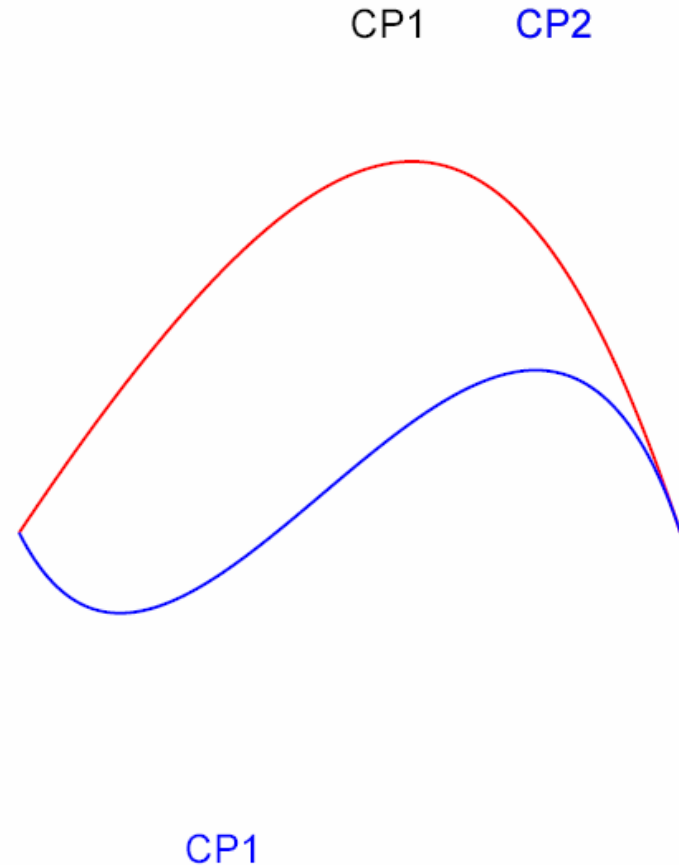




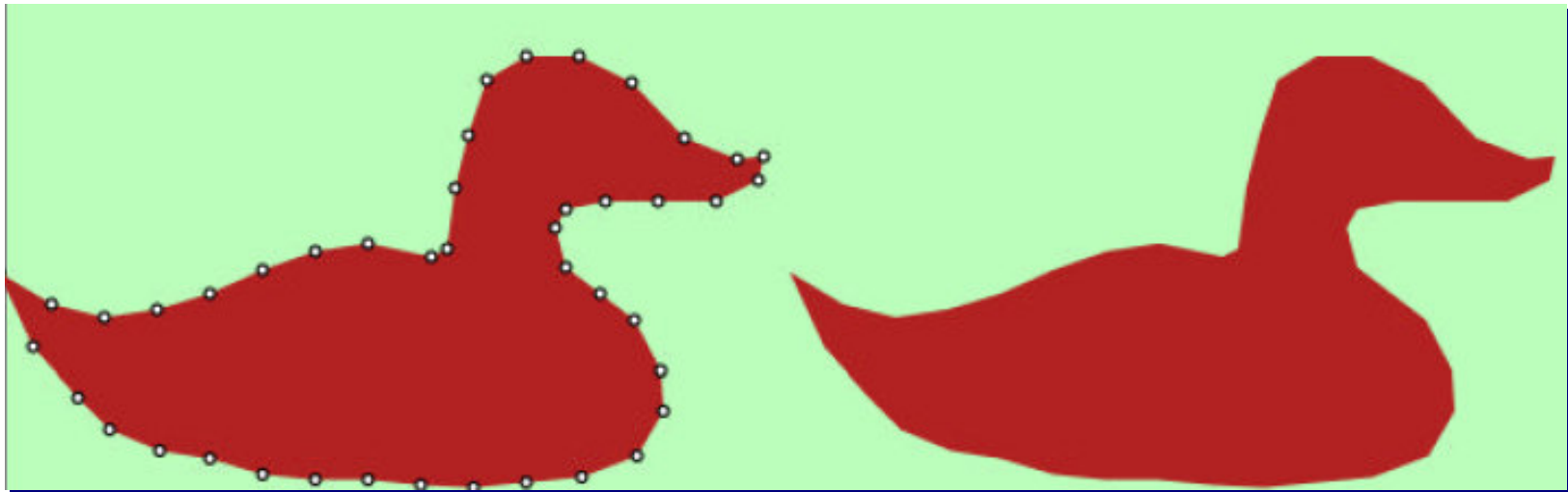
# Un esempio

```
<path d="M 100 250 C 200  
100 250 100 300 250"  
style="stroke:red;fill:none;"/>
```

```
<path d="M 100 250 C 150  
350 250 100 300 250"  
style="stroke:blue;fill:none;"/>
```

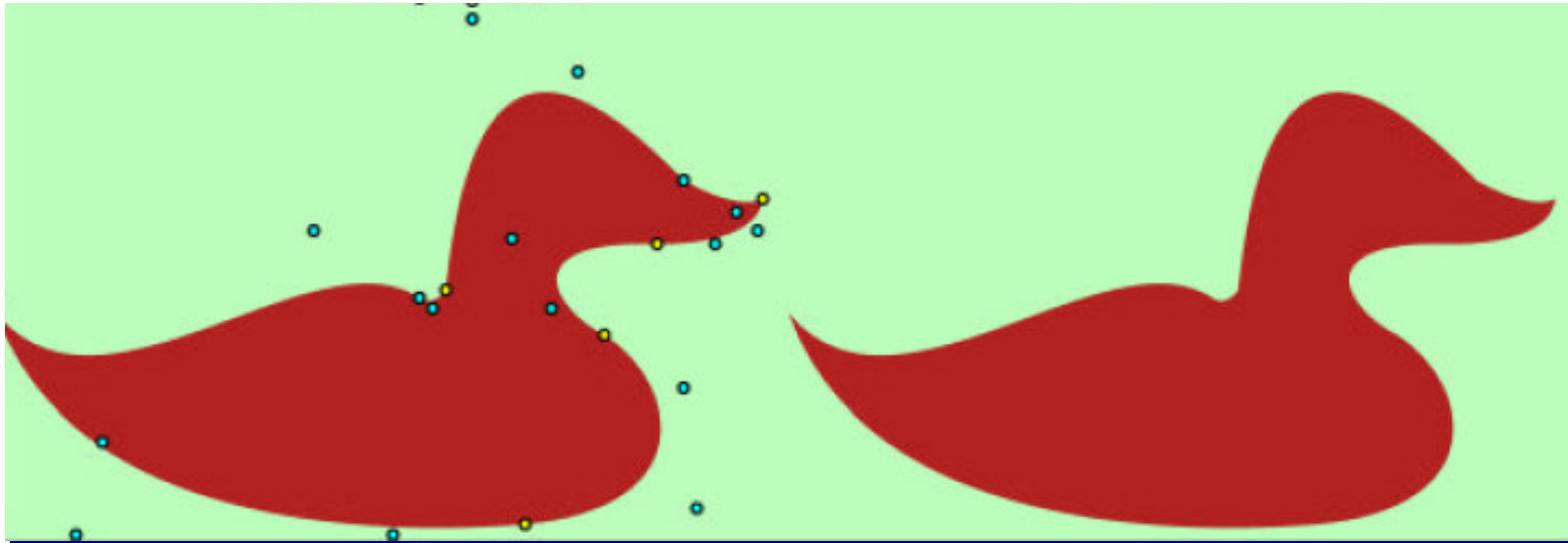


# Red Duck con polilinea



```
<path d="M 0 112 L 20 124 L 40 129 L 60 126 L 80 120 L 100 111 L 120  
104 L 140 101 L 164 106 L 170 103 L 173 80 L 178 60 L 185 39 L 200 30 L  
220 30 L 240 40 L 260 61 L 280 69 L 290 68 L 288 77 L 272 85 L 250 85 L  
230 85 L 215 88 L 211 95 L 215 110 L 228 120 L 241 130 L 251 149 L 252  
164 L 242 181 L 221 189 L 200 191 L 180 193 L 160 192 L 140 190 L 120  
190 L 100 188 L 80 182 L 61 179 L 42 171 L 30 159 L 13 140 Z"/>
```

# Red Duck con curve di Bezier)



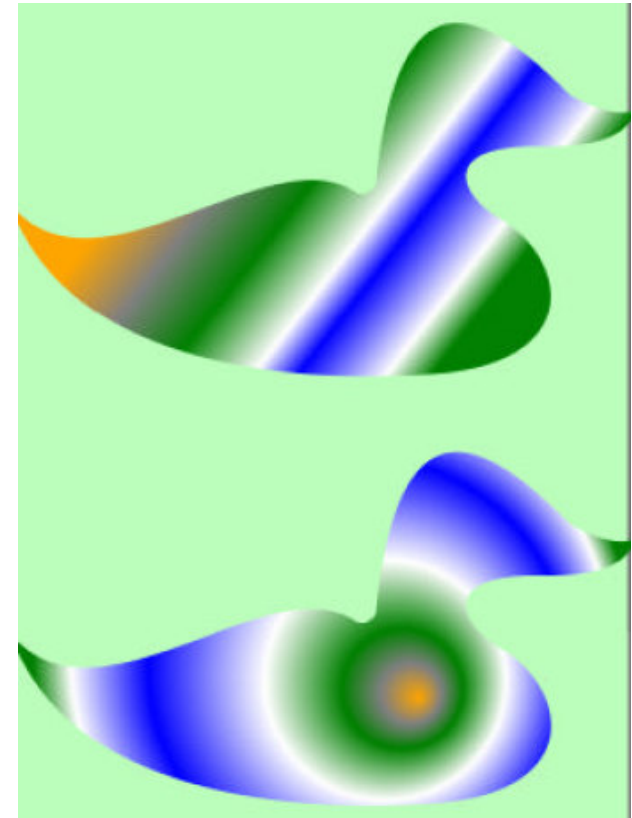
```
<path d="M 0 312  
C 40 360 120 280 160 306 C 160 306 165 310 170 303  
C 180 200 220 220 260 261 C 260 261 280 273 290 268  
C 288 280 272 285 250 285 C 195 283 210 310 230 320  
C 260 340 265 385 200 391 C 150 395 30 395 0 312 Z"/>
```

# Su: path e fill

- È possibile riempire un PATH con l'attributo di stile **fill:color**
- Valgono le stesse regole descritte per i poligoni tramite l'attributo fill-rule

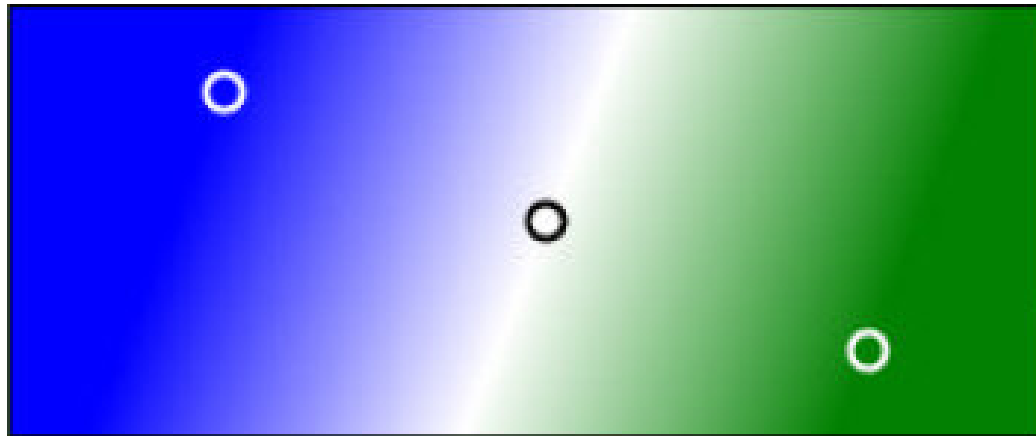
# I Color Gradient

- La proprietà **<fill>** permette inoltre di realizzare effetti colore particolari tramite l'utilizzo dei gradienti. E' possibile riempire un oggetto con delle gradazioni di colore tra due estremi in maniera sia ***lineare*** che ***radiale***.



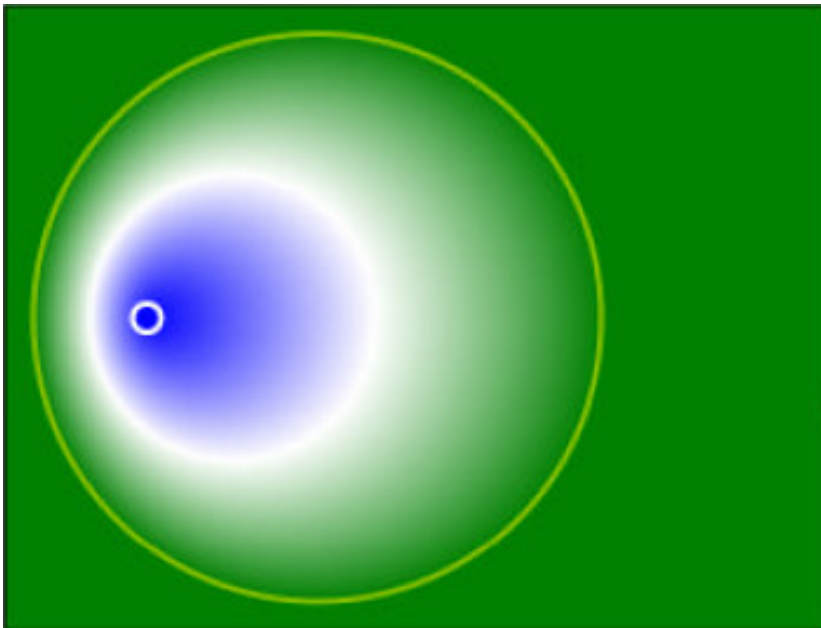
# <linearGradient>

```
<linearGradientid="MyGradient" gradientUnits="userSpaceOnUse"  
x1="80" y1="44" x2="260" y2="116">  
<stop offset="0" style="stop-color:blue"/>  
<stop offset="0.5" style="stop-color:white"/>  
<stop offset="1" style="stop-color:green"/>  
</linearGradient>  
<rect x="20" y="20" width="290" height="120" style="fill:url(#MyGradient)"/>
```



# <radialGradient>

```
<radialGradient id="MyGradient2" gradientUnits="userSpaceOnUse"
  cx="130" cy="270" r="100" fx="70" fy="270">
  <stop offset="0" style="stop-color:blue"/>
  <stop offset="0.5" style="stop-color:white"/>
  <stop offset="1" style="stop-color:green"/>
</radialGradient>
<rect x="20" y="160" width="290" height="220" style="fill:url(#MyGradient2)"/>
```



*La circonferenza di centro  $(cx, cy)$  e raggio  $r$  si riferisce all'offset=1*

*Il fuoco  $(fx, fy)$  individua il punto con offset=0*

# Markers

- I marker permettono di inserire oggetti grafici in un path:
  - ☐ All'inizio
  - ☐ Nel mezzo
  - ☐ Alla fine

Ad esempio:





# Markers

- I markers sono disegni “self-contained”;
- Vengono definiti da:

```
<marker id="nome_mark" markerWidth=""  
      markerHeight="">
```

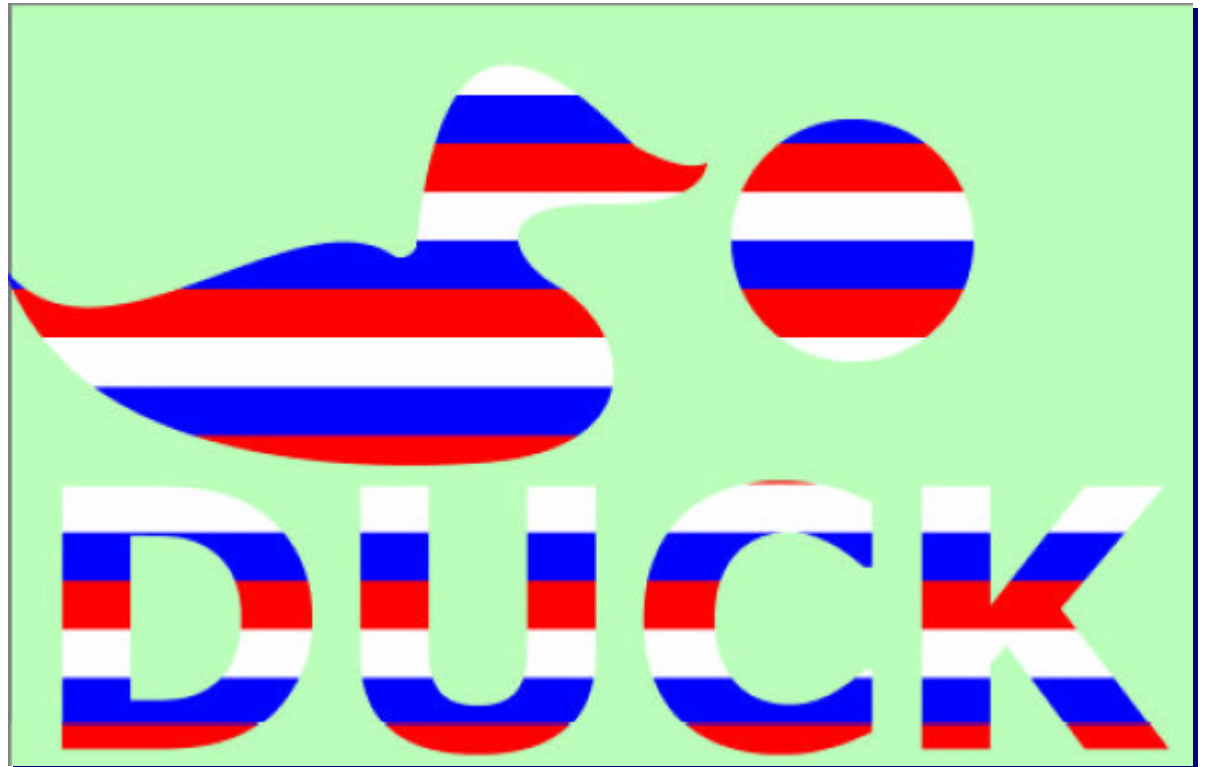
Oggetti...

```
</marker>
```

# Markers

- Una volta definiti vengono richiamati tramite l'attributo di stile:  
**marker-start: url(#nome\_mark); (an, mid, end)**
- Per default si appoggiano all'inizio del mark in basso a SX.
- E' possibile usare gli attributi refX e refY in <marker> per allineare il marker con l'oggetto.
- E' anche possibile usare l'attributo orient="auto" per far automaticamente orientare il marcatore.
- I marker si possono usare anche nei poligoni, ecc.

# Clipping



E' possibile clippare un “disegno” (un gruppo di elementi) rispetto ad un dato oggetto grafico mediante la primitiva **<clipPath>**

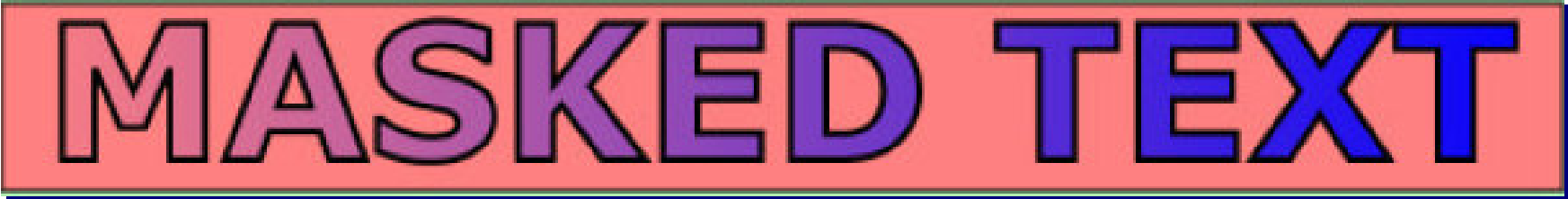
# Sintassi <clipPath>

```
<clipPath id="myClip">  
<circle cx="350" cy="100" r="50"/>  
</clipPath>
```

```
<g style="stroke:none;clip-path:url(#myClip)">  
<rect style="fill:red" x="0" y="0" width="500" height="20" />  
<rect style="fill:white" x="0" y="20" width="500" height="20" />  
<rect style="fill:blue" x="0" y="40" width="500" height="20" />  
<rect style="fill:red" x="0" y="60" width="500" height="20" />  
<rect style="fill:white" x="0" y="80" width="500" height="20" />  
<rect style="fill:blue" x="0" y="100" width="500" height="20" />  
<rect style="fill:white" x="0" y="120" width="500" height="20" />  
<rect style="fill:blue" x="0" y="140" width="500" height="20" />  
</g>
```

# Masking

- E' possibile specificare un valore di opacità che varia da punto a punto mediante un utilizzo combinato dei color gradient e della primitiva **<mask>**.



**MASKED TEXT**

# Sintassi <mask>

```
<linearGradient id="Gradient" x1="0" y1="0" x2="500" y2="0">  
<stop offset="0" style="stop-color:white; stop-opacity:0"/>  
<stop offset="1" style="stop-color:white; stop-opacity:1"/>  
</linearGradient>
```

```
<rect x="0" y="0" width="500" height="60" style="fill:#FF8080"/>  
<mask maskContentUnits="userSpaceOnUse" id="Mask">  
  <rect x="0" y="0" width="500" height="60"  
    style="fill:url(#Gradient)" />  
</mask>
```

```
<text x="250" y="50" style="font-family:Verdana; font-size:60;  
  textanchor:middle;fill:blue; mask:url(#Mask)"> MASKED TEXT </text>  
<text x="250" y="50" style="font-family:Verdana; font-size:60; textanchor:  
middle;fill:none; stroke:black; stroke-width:2">MASKED TEXT </text>
```

# Pattern

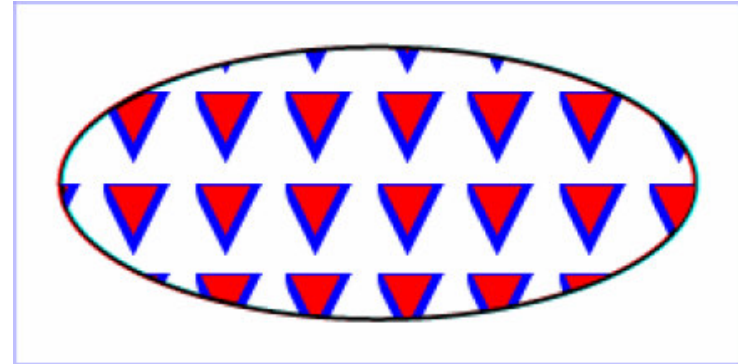
- SVG permette di riempire (fill) gli oggetti con *pattern* grafici. La definizione di un pattern segue la sintassi classica:

```
<pattern id="" x="", y="", width="", height="",  
  patternTransform="",  
  patternUnits="UserSpaceOnUse">
```

....

```
</pattern>
```

# Esempio

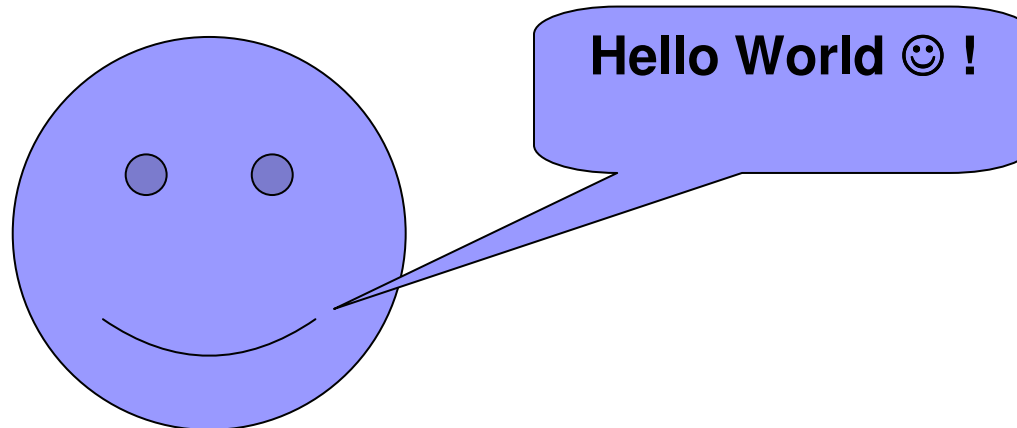


```
<defs>  
<pattern id="TrianglePattern" patternUnits="userSpaceOnUse"  
x="0" y="0" width="100" height="100" >  
<path d="M 0 0 L 7 0 L 3.5 7 z" fill="red" stroke="blue" />  
</pattern>  
</defs>  
  
<ellipse fill="url(#TrianglePattern)" stroke="black" stroke-width="5"  
cx="400" cy="200" rx="350" ry="150" />
```



# Aggiungere del testo a SVG

- Se è vero che ogni disegno racconta una storia, è perfettamente giusto usare le parole per aiutare a raccontare la storia.



# Attributi di stile

- fill:color (crea un font filled)
- stroke: color (crea un font outlined)
- font-family: (serif, sans-serif, monospace)
- font-size: x
- font-weight: (bold,normal)
- font-style: (italic, normal)
- text-decoration: (underline, overline, line-through)
- word-spacing: x
- letter-spacing: x

# Esempio

Testo normale

Testo con stroke black, width .5, fill:none

Testo con stroke black, width .5, fill:red

Testo Famiglia Serif

Testo Famiglia Sans-Serif

Testo Famiglia Monospace

Testo Font size 50

Testo font-weight:bold

Testo font-style:italic

Testo text-decoration:overline

Testo text-decoration:underline

Testo text-decoration:line-through

Testo con spaziatura caratteri 30

T e s t o c o n s p a z i a

# Allineamento del testo

- Il punto finale del testo inserito non è noto *a priori*.
- L'allineamento del testo viene eseguito intorno al punto iniziale tramite l'attributo di stile:

**text-anchor: start | middle | end**

Start  
Middle  
End

# Cambio di stile inline

- Tramite l'elemento `<tspan style="">` è possibile cambiare lo stile del testo in una sessione `<text ...> </text>`.
- Tspan è analogo di span in HTML

# Attributi di tspan

- dx, dy: spostano i caratteri interni di dx e dy pixel rispetto ai precedenti
- x,y: spostano i caratteri interni nella posizione x,y
- baseline-shift: sub | super

# Esempio

Questo e' un unico blocco di testo  
*un po' italico* o anche normale o **Grassetto**

c  
a  
d

e e ritorna normale

H<sub>2</sub>

```
<g font-size="24pt">  
<text x="0" y="24">  
Questo e' un unico blocco di testo  
<tspan x="0" y="48" style="font-  
style:italic">  
un po' italico</tspan>  
o anche normale o  
<tspan style="font-weight:bold">  
Grassetto</tspan>  
<tspan x="0" y="64" dy="10 20 30  
40">cade e ritorna normale</tspan>  
<tspan x="0" y="200">H  
<tspan style="baseline-  
shift:sub;">2</tspan></tspan>  
</text>
```

# Stabilire la lunghezza del testo

- Anche se non possibile sapere a priori la lunghezza del testo, è possibile specificare la lunghezza del testo tramite l'attributo `textLength` di `text`.
- Il testo occuperà lo spazio indicato in due possibili modi regolate dall'attributo `textLength`:
  - `spacing` (default): viene cambiato lo spazio tra i caratteri;
  - `spacingAndGlyphs`: oltre allo spazio vengono ridimensionati i caratteri.



# Testo verticale

- Il testo in verticale si può ottenere in 3 modi:
- Con una rotazione di 90 gradi  
`<text x="" y="" transform(rotate(90))`
- Con la proprietà di stile *writing-mode: tb*  
(effetto analogo al precedente)
- Se si vuole che il testo sia scritto in verticale occorre aggiungere la proprietà *glyph-orientation-vertical:0;*

# Testi che seguono un path

- I testi non devono essere necessariamente in orizzontale o verticale ma possono seguire una forma arbitraria.
- In SVG ciò si ottiene tramite l'elemento  
`<textPath xlink:href="#nomepath"> </textPath>`  
che punta un path definito precedentemente.
- Il testo avrà in ogni punto la baseline perpendicolare al path che sta seguendo.
- E' possibile aggiustare l'inizio del text aggiungendo l'attributo `startOffset` con un valore in percentuale.

# Esempio

```
<defs>
<path id="path1" d="M 100 100
C 200 50 300 150 400 100"/>
</defs>
<text style="font-size:24;">
<textPath xlink:href="#path1">
Testo in curva di Bezier
</textPath>
</text>
<text style="font-size:24;"
transform="translate(0,100)" >
<textPath xlink:href="#path1"
startOffset="20%">
Testo in curva di Bezier 20%
offset </textPath> </text>
```

*Testo in curva di Bezier*

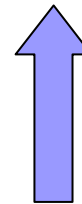
*Testo in curva di Bezier*

# Animazioni in SVG

- SVG permette di creare animazioni basandosi su SMIL2 (Synchronized Multimedia Integration Language Level2).
- In questo sistema vengono specificati
  - i valori iniziali e finali di attributi, colori e trasformazioni che si vogliono animare;
  - Il momento di inizio dell'animazione
  - La durata dell'animazione

# Esempio Basic

```
<rect x="10" y="10"  
  width="500" height="200"  
  style="stroke: black; fill:  
  none;">  
  <animate  
    attributeName="width"  
    attributeType="XML"  
    from="500" to="100"  
    begin="0s" dur="5s"  
    fill="freeze" />  
</rect>
```



# Animazioni...

## **Alias: Animazioni multiple su un singolo oggetto**

- In SVG è possibile creare animazioni multiple a diversi attributi dell'elemento.
- Le animazioni multiple si ottengono specificando dentro l'elemento la sequenza delle animazioni tramite il tag `<animate>`
- Gli attributi degli elementi possono essere di tipo:
  - XML (se si riferiscono alla natura di un oggetto)
  - CSS (se si riferiscono ad elementi di stile)

# Esempio: cerchio\_ellisse

```
<ellipse cx="250" cy="250" rx="250" ry="250"  
style="stroke:black; fill: blue;">  
<animate attributeName="rx" attributeType="XML"  
from="250" to="100" begin="0s" dur="8s" fill="freeze" />  
<animate attributeName="ry" attributeType="XML"  
from="250" to="200" begin="0s" dur="8s" fill="freeze" />  
<animate attributeName="fill-opacity" attributeType="CSS"  
from="0.1" to="1" begin="0s" dur="8s" fill="freeze" />  
</ellipse>
```

# Animazione di più oggetti

- Ogni “oggetto” di SVG può essere animato tramite l'attributo **<animate>** dentro l'elemento.
- Le animazioni avvengono “contemporaneamente” in base alle indicazioni temporali dell'animazione stessa.



# Esempio animazioni\_multiple

```
<circle cx="50" cy="50" r="50" style="fill:red">
<animate attributeName="cx" attributeType="XML" from="50" to="250"
begin="0s" dur="4s" fill="freeze" />
<animate attributeName="cy" attributeType="XML" from="50" to="250"
begin="4s" dur="4s" fill="freeze" /> </circle>
<rect x="250" y="250" width="100" height="100" style="fill:blue;fill-
opacity:0.5;">
<animate attributeName="x" attributeType="XML" from="250" to="200"
begin="6s" dur="2s" fill="freeze" />
<animate attributeName="y" attributeType="XML" from="250" to="200"
begin="8s" dur="2s" fill="freeze" />
<animate attributeName="fill-opacity" attributeType="CSS" from="0.5"
to="1" begin="8s" dur="2s" fill="freeze" />
<animate attributeName="rx" attributeType="XML" from="0" to="20"
begin="9s" dur="2s" fill="freeze" /> </rect>
```

# Il Tempo

- Il “cronometro” di SVG parte quando il documento è interamente caricato e si stacca quando la finestra del visualizzatore viene chiusa.
- Il tempo si esprime in:
  - ☐ hh:mm:ss
  - ☐ mm:ss
  - ☐ Valore\_numerico unità\_di\_tempo (1h, 3.5min)
  - ☐ Le unità di tempo sono: h, m, s, ms

# Sincronizzazione delle animazioni

- È possibile sincronizzare le animazioni specificando che l'inizio di una avviene alla fine di un'altra.

- In lingua SVG ciò si traduce:

```
<element><animate id="rif" begin=""  
dur=""/></element>
```

```
<element2><animate  
begin="rif.end" dur=""/></element2>
```

Analogo con rif.begin

## Sincro (2)

- È possibile aggiungere un offset **positivo** tra la fine di un'animazione e l'inizio di un'altra.

<element2><animate

begin="**ref.end +3sec**" dur="" /></element2>

- Attributo **end**
- Forza la fine di un'animazione, anche se la stessa non è giunta a compimento. (in altre parole può capitare che  $\text{begin} + \text{dur} > \text{end}$ )

# Repetita Iuvant: ripetere le animazioni

Le animazioni si ripetono tramite gli attributi di <animate>:

- repeatCount="numero" – indica in numero di volte in cui ripetere l'animazione (**indefinite = infinito**)
- repeatDur="tempo" – indica per quanto tempo ancora devo tenere l'animazione; (**indefinite = infinito**)

## Sincro (3)

- È possibile far partire un'animazione alla i-esima esecuzione di un'altra animazione;

- In lingua SVG ciò si traduce:

```
<element><animate id="rif" begin=""  
dur=""/></element>
```

```
<element2><animate  
begin="rif.repeat(i)" dur=""/></element2>
```

# E gli attributi non numerici ?

- Fino ad adesso abbiamo creato delle animazioni facendo variare i valori numerici di alcuni attributi degli elementi (XML, CSS).
- Esistono attributi che hanno valori alfanumerici (tipo: **visibility: hidden|visible**)
- Questi attributi vengono animati da:

```
<set attributeName="" attributeType="" to=""  
begin="" dur="" fill="">
```

# Esempio: Sincro1

```
<circle cx="50" cy="50" r="50" style="fill:red">
<animate id="animazione1" attributeName="cx"
attributeType="XML" from="50" to="200"
begin="0s" dur="3s" fill="freeze" />
</circle>
```

```
<circle cx="300" cy="50" r="50" style="fill:blue">
<animate id="animazione2" attributeName="cy"
attributeType="XML" from="50" to="200"
begin="animazione1.end" dur="3s" fill="freeze" />
</circle>
```

```
<circle cx="300" cy="300" r="50"
style="fill:orange">
<animate id="animazione3" attributeName="cx"
attributeType="XML" from="300" to="150"
begin="animazione2.end" dur="3s" fill="freeze" />
</circle>
```

```
<circle cx="50" cy="300" r="50"
style="fill:yellow">
<animate id="animazione4" attributeName="cy"
attributeType="XML" from="300" to="150"
begin="animazione3.end" dur="3s"
fill="freeze" />
```

```
</circle>
<text x="110" y="200" visibility="hidden"
style="font-size:40pt; font-family:serif;
stroke:blue; stroke-dasharray:3;fill:sienna">
<set attributeName="visibility"
attributeType="CSS" to="visible"
begin="animazione4.end" dur="1s"
fill="freeze"/>
Done!</text>
```



# Esempio: Sincro2

```
<circle cx="50" cy="50" r="50"
style="fill:red">
animate id="animazione1"
attributeName="cx" attributeType="XML"
from="50" to="450" begin="0s" dur="3s"
repeatCount="3" fill="freeze" />
</circle>
<circle cx="50" cy="200" r="50"
style="fill:blue">
<animate id="animazione2"
attributeName="cx" attributeType="XML"
from="50" to="450" begin="0s" dur="3s"
repeatDur="10s" fill="freeze" />
</circle>
```

```
<circle cx="50" cy="350" r="50"
style="fill:maroon">
<animate id="animazione3"
attributeName="cx"
attributeType="XML"
from="50" to="450"
begin="animazione1.repeat(1)"
dur="3s"
fill="freeze" />
</circle>
<path d="M 400 100 l 100 -100 m 0 100 l -
100 -100" style="stroke:black"
visibility="hidden">
<set attributeName="visibility" to="visibile"
begin="animazione1.end" fill="freeze"/>
</path>
```

# Animazioni: Colori

- L'elemento **<animate>** non funziona con i colori e quindi con gli attributi a valore colore.
- Questa animazione si esegue con l'elemento **<animateColor** attributeName="" begin="" dur="" from="red" to="yellow" fill="">

# Esempio:Sincro3

```
<rect x="50" y="50" width="250"  
  height="250" style="fill:red">  
<animateColor attributeName="fill"  
  attributeType="CSS" from="red" to="black"  
  begin="0s" dur="5s" fill="freeze"  
  repeatCount="5"/>  
</rect>
```

# Animazioni: Trasformazioni

- Similmente a quanto detto sui colori, le trasformazioni necessitano di un tag ad hoc, detto **<animateTransform>**
- **<animateTransform** attributeType="XML" attributeName="transform" type="scale" from="" to="" begin="" dur="" fill="">

# Esempio: Sincro4

```
<rect x="200" y="200" width="150" height="150" style="fill:red">  
<animateTransform attributeName="transform"  
attributeType="XML" type="rotate"  
from="0 200 200" to="359 200 200" begin="0s" dur="5s"  
fill="freeze" repeatCount="indefinite"/>  
<animate attributeName="rx"  
attributeType="XML" begin="0s" dur="5s"  
from="0" to="150" fill="freeze" repeatCount="indefinite"/>  
</rect>
```

# Composizione di animazioni di trasformazioni

- Se si vogliono comporre più trasformazioni occorre specificare l'attributo `additive="sum"` in `<animateTransform>`
- Di default l'attributo `additive="replace"`, cioè la trasformazione rimpiazza la precedente.

# Animazioni: Motion

- Fino ad adesso le animazioni “spaziali” sono state effettuate in linea retta (cambiando i parametri x,y o con `translate`)
- Il tag **<animateMotion>** permette di spostare oggetti lungo un PATH arbitrario
- Sintassi: `<animateMotion path="" dur="" fill="">`

# Rotazione automatica dell'oggetto

- È possibile specificare nell'elemento `<animateTransform>` l'attributo `rotate="auto"` che automaticamente ruota l'oggetto rispetto al path che segue.



# Usare path definiti in precedenza.

- È possibile richiamare dei path definiti precedentemente nella sezione <defs> in questo modo:

```
<animateMotion dur="" begin="" fill="">
```

```
<mpath xlink:href="#id_path"/>
```

```
</animateMotion>
```

# sincro5.svg

```
<defs>
<path id="cammino" d="M 100 250 C 150
350 250 100 300 250 350 100 400 150
500 100" style="stroke:blue;fill:none;"/>
<g id="man" style="stroke:black">
<line id="gambasx" x1="0" y1="100"
x2="20" y2="70">
<animate attributeName="x1"
attributeType="XML" from="0" to="40"
begin="0s" dur="2s"
repeatCount="indefinite"
end="moto.end+3s"
/> </line>

<line id="gambadx" x1="20" y1="70"
x2="40" y2="100">
<animate attributeName="x2"
attributeType="XML"
from="40" to="0" begin="0s" dur="2s"
repeatCount="indefinite"
end="moto.end+3s" /> </line>
```

```
<line id="corpo" x1="20" y1="70" x2="20"
y2="40"/>
<line id="bracciosx" x1="20" y1="40" x2="0"
y2="60"/>
<line id="bracciodx" x1="20" y1="40" x2="40"
y2="60"/>
<circle id="testa" cx="20" cy="30" r="10">
<animate attributeName="cx"
attributeType="XML"
from="15" to="25" begin="0s" dur="3s"
repeatCount="indefinite" end="moto.end+3s" />
</circle>
</g>
</defs>

<use xlink:href="#man" transform="translate(-
50,-100)">
<animateMotion id="moto" dur="10s" begin="0s"
fill="freeze">
<mpath xlink:href="#cammino"/>
</animateMotion>
</use>
<!-- Per far vedere la linea -->
<use xlink:href="#cammino"/>
```

# Animation Control

Abbiamo finora visto come in SVG sia possibile animare delle primitive grafiche specificando i valori iniziali e finali degli attributi coinvolti in un dato intervallo di tempo. L'animazione, per default, si realizza in modo *lineare*.

E' possibile specificare un modo diverso mediante la primitiva **<calcMode>**

## <calcMode>

- L'attributo **<calcMode>** prevede altre 3 modalità di funzionamento oltre quella di default (**linear**):

**discrete**

**paced**

**spline**

## <calcMode> e le Spline...

- E' possibile specificare una lista intermedia di valori, da associare ad una specifica locazione temporale. Gli attributi coinvolti sono:

**values=""**

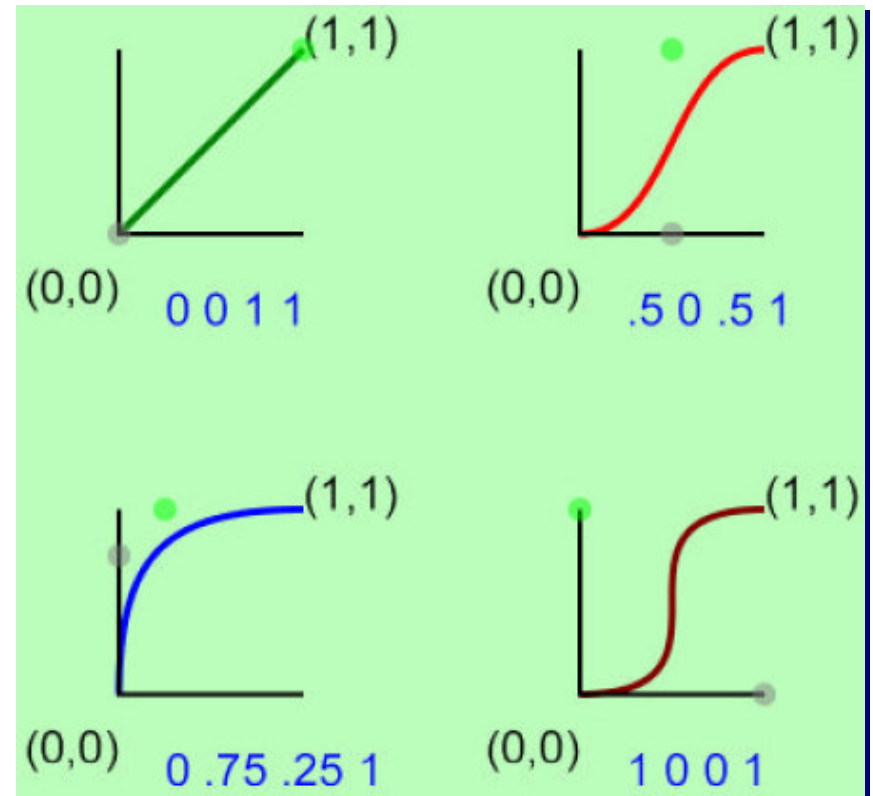
**calcMode="spline"**

**keySplines=""**

# <keySplines>

I parametri di <keySplines> specificano i 2 punti di controllo di una curva cubica di Bezièr, che va dall'origine (0,0) al punto (1,1).

L'asse X si riferisce al tempo, l'asse Y alla distanza percorsa.



# <keyTimes>

- E' possibile anche specificare gli istanti temporali in cui un'animazione deve assumere specifici valori, mediante l'uso combinato di **<keyTimes>** e **<values>**:

Esempio:

.....

**values**="12, 100, 200"

**keyTimes**="3s,7s,12s"

# Varianti di controllo

All'interno di un'animazione è possibile utilizzare:

**from="" by=""** (al posto di **to**)

Si cambia il valore a step prefissati.

Ancora è possibile specificare i valori intermedi di un attributo mediante:

**attributeName="x" values="5; 10; 31; 5"**



# Links in SVG

Qualsiasi oggetto SVG può diventare un link se è inserito dentro l'elemento

**`<a xlink:href="uri"> </a>`**

Es:

`<a xlink:href="URI">`

`<text x="" y="" style=""> Questo è un link </text>`

`</a>`

Crea un link testuale.

# Ancora links...

Se si vuole aprire un link in una nuova finestra bisogna impostare l'attributo **<xlink:show>** dell'elemento **<a>** su **new**.

L'attributo **<mailto>**, che permette di aprire l'applicazione predefinita per la posta elettronica, su un nuovo messaggio con l'indirizzo pronto, si utilizza sempre all'interno di un tag **<a>**:

# Barre di navigazione

- E' possibile utilizzare gli elementi visti finora per creare semplici barre di navigazione con o senza etichette, integrandole con semplici animazioni sul testo.

# Grafica SVG in pagine HTML

E' possibile inserire immagini SVG in pagine web HTML o XHTML.

I tag HTML da utilizzare a tale scopo sono **<embed>** (ufficialmente disapprovato dal W3C) oppure **<object>**.

# Il tag **<embed>**

## ■ La sintassi:

```
<embed      src="NestedSVG.svg"      width="500"  
height="400" type="image/svg+xml">
```

■ Per ottenere “artificiosamente” delle barre di navigazione laterali è possibile inserire **<embed>** all'interno di un tag **<body>**:

```
<body leftmargin="0" marginwidth="0" topmargin="0"  
      marginheight="0">
```

```
<embed.....
```

```
</body>
```

# Il tag <object>

## ■ La sintassi:

```
<object src="NestedSVG.svg" width="500" height="400"  
  type="image/svg+xml">  
</object>
```

Da un tag <**embed**> non è possibile visualizzare un messaggio di testo che indichi che deve essere scaricato un SVG viewer. Con il tag <**object**> ci sono due possibilità: visualizzare un'immagine o un testo utilizzando l'attributo alt del tag <**img**>.

```

```

# Interattività

In SVG si può fare riferimento ad eventi esterni, avviati dall'utente (es: il **click** del mouse, il **mouseover**, il **mouseout**, ecc.)

Gli eventi possono essere gestiti associandogli degli script (si utilizza il linguaggio ECMAScript) che vengono mandati in esecuzione allo scatenarsi dell'evento stesso.

# Lista Eventi

- Gli eventi principali gestiti da SVG sono:

**click**: evento scatenato dal click del mouse su di un elemento grafico;

**mousemove**: evento associato al movimento del mouse;

**mouseover**: evento scatenato dal passaggio del mouse su di un oggetto grafico;

**mouseout**: evento scatenato quando il puntatore del mouse abbandona l'area di un elemento grafico;

**mousedown**: evento associato alla pressione del tasto del mouse su di un elemento grafico;

**load**: evento scatenato quando il documento SVG viene caricato dal visualizzatore.



# Gli script



E' possibile utilizzare un vero e proprio linguaggio di scripting ECMAScript (standardizzato e basato su JavaScript) per la gestione degli eventi.

Esempio:

```
<script type="text/ecmascript"> <![CDATA[  
function circle_click(evt) {  
var circle = evt.target;  
var currentRadius = circle.getAttribute("r");  
if (currentRadius == 100)  
    circle.setAttribute("r", currentRadius*2);  
    else  
    circle.setAttribute("r", currentRadius*0.5);  
} ]]> </script>
```

# Evt, target

- **Evt** permette di identificare l'elemento grafico a cui abbiamo associato l'evento scatenato e tale informazione viene passata come parametro alle funzioni relative alla gestione degli eventi.
- Attraverso il metodo **target** siamo in grado di ottenere un riferimento all'elemento grafico a cui è associato l'evento.
- L'uso di **evt.target** ci permette quindi di memorizzare facilmente all'interno di una variabile, un riferimento all'oggetto su cui è stato scatenato l'evento.

# Scripting:qualche dettaglio

Come succede per un file XML o HTML, quando un documento SVG viene caricato dal visualizzatore, viene creata una struttura interna ad albero che rappresenta il documento.

Ad esempio:

```
<svg width="100" height="100">  
  <rect x="10" y="10" width="10" height="10"/>  
  <circle cx="50" cy="50" r="10"/>  
</svg>
```

Ad ogni tag SVG corrisponde un nodo della struttura. I nodi **<rect>** e **<circle>**, essendo definiti all'interno del tag **<svg>**, vengono chiamati nodi figli di **<svg>**, mentre **<svg>** stesso è detto nodo padre.

Inoltre il nodo principale del documento (**<svg>**) viene chiamato nodo root (radice).

SVG mette a disposizione una serie di metodi, che costituiscono l'interfaccia **DOM** (Document Object Model), per accedere e manipolare i nodi della struttura.

# DOM

I principali metodi sono:

**getElementById(nome\_id):** restituisce l'elemento grafico di cui abbiamo specificato l'identificatore;

**setAttribute(nome\_attributo, valore\_attributo):** consente di modificare il valore di un determinato attributo di un nodo;

**getAttribute(nome\_attributo):** permette di leggere il valore di un attributo di un elemento;

**createElement(nome\_elemento):** consente di creare un nuovo elemento grafico;

**appendChild(nome\_elemento):** consente di inserire un nuovo elemento come figlio del nodo a cui questa funzione è applicata.

# Un Esempio

```
<svg id="elementoRadice" width="300" height="300"
  onload="aggiungiRect()">
  <script type="text/ecmascript"><![CDATA[
```

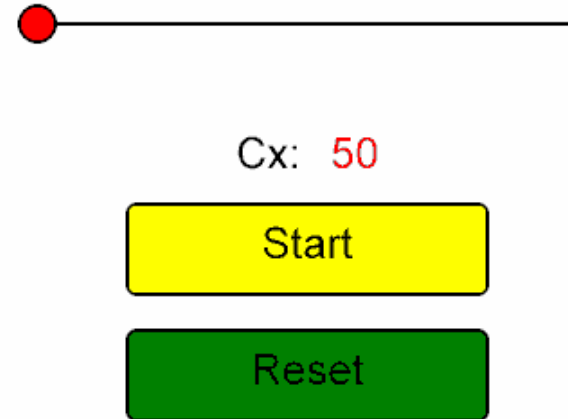
```
function aggiungiRect(){
  var svgdoc=document.getElementById("elementoRadice");
  var newrect=document.createElement("rect");
  newrect.setAttribute("x",10);
  newrect.setAttribute("y",150);
  newrect.setAttribute("width",250);
  newrect.setAttribute("height",100);
  newrect.setAttribute("style","fill:blue;stroke:black;stroke-width:2;");
  svgdoc.appendChild(newrect);
}
```

```
]]></script>
<rect x="10" y="10" width="250" height="100"
style="stroke:black;fill:red;stroke-width:2"/>
</svg>
```



# Un altro esempio

Cliccando sul pulsante "start" il cerchio rosso inizia a muoversi lungo la linea fino al termine della linea stessa e cliccando poi sul pulsante "reset" il cerchio torna alla posizione iniziale. La scritta presente al centro dell'immagine visualizza costantemente il valore dell'attributo *cx* dell'elemento circolare, per mettere meglio in evidenza come viene realizzata l'animazione.



# Uno sguardo al codice

```
<script type="text/ecmascript"><![CDATA[
    var elemento;      var scrittaCx; var intervallo=10;
    function startAnimazione(evt) {
        elemento=evt.target.ownerDocument.getElementById("cerchio");
        scrittaCx=evt.target.ownerDocument.getElementById("valoreCx");
        anima();          }

    function anima(){
        var posizionex=parseFloat(elemento.getAttribute("cx"));
        posizionex++;
        if (posizionex<351) { elemento.setAttribute("cx",posizionex);
                               scrittaCx.firstChild.nodeValue=posizionex;
                               setTimeout("anima()",intervallo);
                               }          }

    function resetAnimazione(evt){
        elemento=evt.target.ownerDocument.getElementById("cerchio");
        scrittaCx=evt.target.ownerDocument.getElementById("valoreCx");
        elemento.setAttribute("cx",50);
        scrittaCx.firstChild.nodeValue=50;
    }
}]></script>
```



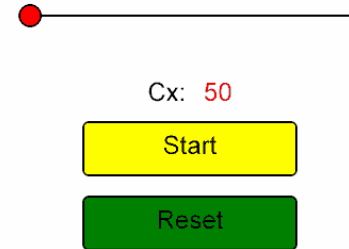
Cx: 50

Start

Reset

Sono i valori  
dei campi id

# Qualche commento..



La funzione **startAnimazione**, dopo aver memorizzato in due variabili i riferimenti agli oggetti che verranno modificati per realizzare l'animazione, manda in esecuzione la funzione **anima**. Questa funzione, finché il cerchio è all'interno della linea, va a modificare dinamicamente il valore dell'attributo **cx**. Inoltre aggiorna il valore dell'elemento testuale che visualizza sull'immagine il valore di **cx**.

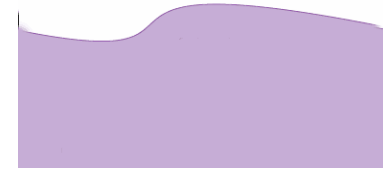
Successivamente viene lanciata ricorsivamente la stessa funzione **anima** con l'aiuto della funzione predefinita **setTimeout** che si occupa di lanciare **anima** dopo un certo intervallo di tempo (nel nostro caso 50 ms). La funzione viene lanciata dopo un certo intervallo di tempo perché altrimenti l'animazione sarebbe troppo rapida e verrebbe visualizzata male. La posizione del cerchio viene quindi spostata di un pixel verso destra ogni 50 ms.

Da notare nell'esempio un metodo alternativo per aver un riferimento all'elemento root del documento SVG: l'uso del metodo **ownerDocument** applicato al riferimento ad un elemento grafico.



# Onde....

Onde.svg



E' possibile animare una curva di Bezier, facendone variare in maniera pseudo-casuale i punti di controllo.

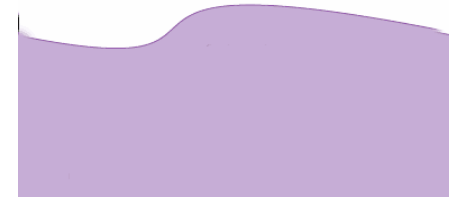
Nell'esempio il path iniziale da animare è:

```
<path id="line" style="fill:#905CA8;fill-opacity:0.5;stroke:#905CA8"
d="M 0,150 h 400"/>
```

L'animazione ha inizio al verificarsi dell'evento **onload** del tag **<svg>**. Viene quindi salvato un "puntatore" all'intero documento SVG in una variabile globale. Inoltre viene chiamata la funzione di sistema **setInterval**, che a sua volta si occupa di richiamare iterativamente la funzione **next\_frame**.

```
function on_load (event){
svgdoc = event.getCurrentNode().getOwnerDocument();
setInterval ('next_frame()', 100);}
```

# Il codice delle onde..1/3



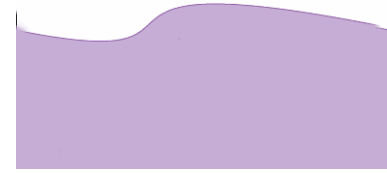
L'animazione è realizzata facendo variare i 2 punti di controllo tenendo comunque fermi i punti iniziali e finali. Una volta generati in valori "target" per i punti di controllo le coordinate correnti vengono gradualmente aumentate o diminuite fino a coincidere esattamente con essi.

Le coordinate correnti dei punti di controllo vengono indicate nel codice con le variabili (x0, y0) e (x1, y1) mentre i valori "target" vengono invece indicati con (tx0, ty0) e (tx1, ty1) .

La funzione **next\_frame** innanzitutto si occupa di recuperare, tramite i metodi DOM relativi, l'handler del path da animare:

```
var linenode = svgdoc.getElementById ('line');  
if (!linenode) return;
```

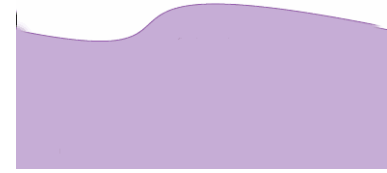
## Il codice delle onde..2/3



A questo punto se necessario vengono generati nuovi valori “target” per le coordinate dei punti di controllo. In particolare ciò sarà vero alla prima invocazione (valori “target” posti ad -1) oppure quando tutte le coordinate correnti hanno raggiunto i valori “target”

```
if (tx0 < 0 || (tx0 == x0 && ty0 == y0 && tx1 == x1 && ty1 == y1)) {  
    tx0 = Math.floor (400*Math.random());  
    ty0 = Math.floor (300*Math.random());  
    tx1 = Math.floor (400*Math.random());  
    ty1 = Math.floor (300*Math.random()); }
```

# Il codice delle onde..3/3



Le coordinate correnti vengono indirizzate verso i valori “target”, ad intervalli di +/- 10 pixel attraverso una ulteriore funzione **change\_coord()**:

```
x0 = change_coord (x0, tx0);  
y0 = change_coord (y0, ty0);  
x1 = change_coord (x1, tx1);  
y1 = change_coord (y1, ty1);
```

Possono quindi essere cambiati gli elementi dell’attributo “d” del path in questione utilizzando le nuove coordinate:

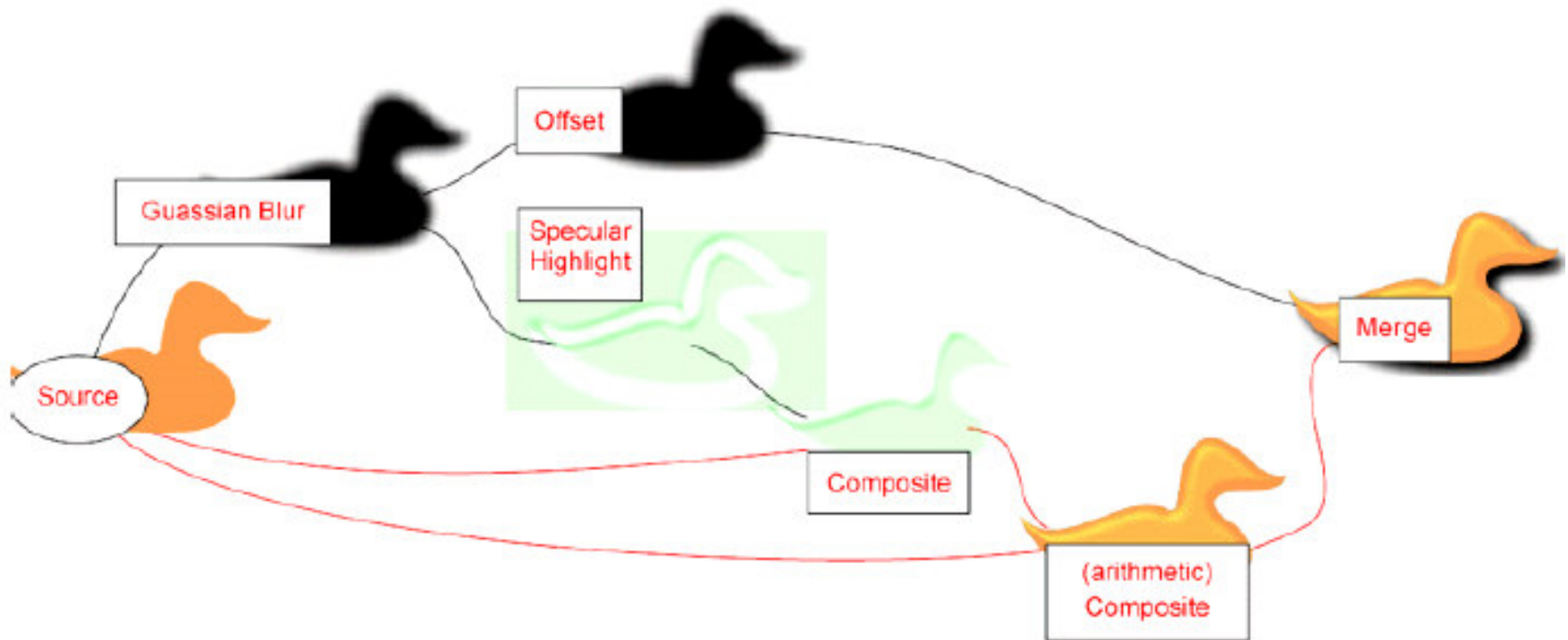
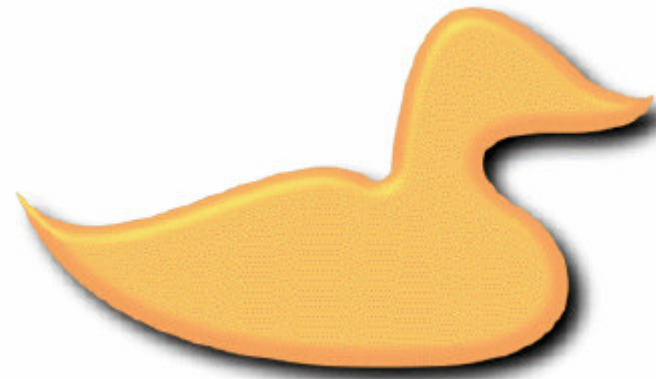
```
linenode.setAttribute('d', 'M 0, 300 L 0, 150 C'+x0+', '+y0+', '+x1+', '+y1+', 400,150 L  
400, 300 z');
```

# I Filtri in SVG



- In SVG è possibile utilizzare un insieme di *declarative feature's set* in grado di generare e descrivere *effetti grafici* anche complessi.
- La possibilità di applicare filtri ad elementi grafici di qualsiasi natura (anche testi) li rende uno strumento flessibile e potente.
- Un filtro per SVG è costituito da una serie di operazioni che applicate ad una data **sorgente grafica** la modificano, mostrando il risultato direttamente sul *device* finale.

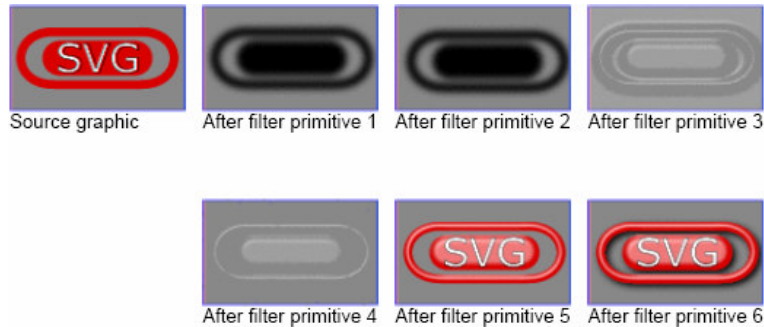
# Potenzialità



## <filter>

- I filtri si definiscono mediante la primitiva **<filter>**. Il loro utilizzo effettivo si ottiene mediante un **idfilter**.
- Ciascun elemento **<filter>** contiene poi delle primitive “figlie” che compiono le vere e proprie operazioni grafiche.
- Il grafico sorgente o l’output di un filtro può essere riutilizzato come input.

# Un primo esempio



```

1 <filter id="MyFilter" filterUnits="userSpaceOnUse" x="0" y="0" width="200" height="120">
2   <desc>Produces a 3D lighting effect.</desc>
3   <feGaussianBlur in="SourceAlpha" stdDeviation="4" result="blur"/>
4   <feOffset in="blur" dx="4" dy="4" result="offsetBlur"/>
5   <fespecularLighting in="blur" surfaceScale="5" specularConstant=".75"
6     specularExponent="20" lighting-color="#bbbbbb"
      result="specOut">
      <fePointLight x="-5000" y="-10000" z="20000"/>
    </fespecularLighting>
    <feComposite in="specOut" in2="SourceAlpha" operator="in" result="specOut"/>
    <feComposite in="SourceGraphic" in2="specOut" operator="arithmetic"
      k1="0" k2="1" k3="1" k4="0" result="litPaint"/>
    <feMerge>
      <feMergeNode in="offsetBlur"/>
      <feMergeNode in="litPaint"/>
    </feMerge>
  </filter>

```



# Attributi di **<filter>**

- La sintassi specifica di ***filter*** è la seguente:  
**<filter** id="...",  
**filterUnits**="*userSpaceOnUse|objectBoundingBox*",  
**primitiveUnits**="*userSpaceOnUse|objectBoundingBox*"  
**x**=", **y**=", **width**=", **height**=",  
**filterRes**=" **xlink:href**=""  
>  
**<fe..... >**  
**</filter>**

Il filtro verrà poi applicato ad un qualsiasi oggetto grafico mediante l'attributo di stile **filter:url(#.....)**

# Attributi comuni...

- La sintassi specifica di ciascun ***primitive filter*** o ***filter effect*** è la seguente:

<fe.....

**x=**“”,**y=**“”,

**width=**“”, **height=**“”,

**in=**“” **result=**“”

>

.....insieme ai singoli parametri che ciascuno di essi richiede.

# Filter effects: *Gaussian Blur*

Realizza la classica sfocatura regolare, utile ad esempio per creare l'ombreggiatura di primitive grafiche:

```
<feGaussianBlur in="SourceGraphic",  
stdDeviation="", result="">
```

Si può utilizzare il valore dell'attributo **result** come input per ulteriori filtri.

# <feOffset> & <feMerge>

- L'elemento **<feOffset>** trasla l'input:  
**<feOffset in="" dx="" dy="" result="">**
- L'elemento **<feMerge>** realizza il merging grafico di più sorgenti:  
**<feMerge>**  
    **<feMergeNode in="">**  
    **<feMergeNode in="">**  
**</feMerge>**

Ciao SVG!

# Blending

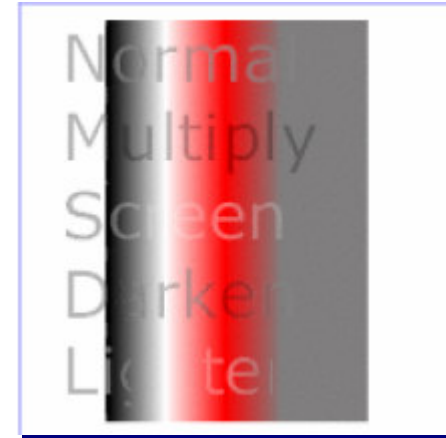
- La primitiva **<feBlend>** sovrappone due oggetti grafici, pixel per pixel, secondo diverse modalità:

*normal / multiply / screen / darken / lighten*

Sintassi:

**<feBlend in2="", in="", mode="">**

# Blending (2)



L'opacità dell'immagine risultante  $q_r$ ,  
date le opacità  $q_a$  e  $q_b$  delle immagini di input A e B, si ottiene:

$$q_r = 1 - (1 - q_a) * (1 - q_b)$$

Il colore finale pixel per pixel  $c_r$ , si ottiene a partire dai colori  $c_a$  e  $c_b$  nelle diverse modalità:

<b>normal</b>	$c_r = (1 - q_a) * c_b + c_a$
<b>multiply</b>	$c_r = (1 - q_a) * c_b + (1 - q_b) * c_a + c_a * c_b$
<b>screen</b>	$c_r = c_b + c_a - c_a * c_b$
<b>darken</b>	$c_r = \text{Min} ((1 - q_a) * c_b + c_a, (1 - q_b) * c_a + c_b)$
<b>lighten</b>	$c_r = \text{Max} ((1 - q_a) * c_b + c_a, (1 - q_b) * c_a + c_b)$

# Utilizzare il background..

- E' possibile accedere in maniera esplicita al *background* e al canale *alpha* di un oggetto grafico, per essere utilizzati come input ai vari filtri.
- La proprietà che abilita l'accesso all'immagine *background* è:  
**enable-background="new"**

# Turbulence

- Una delle primitive filtro che nonostante la sua semplicità permette di realizzare interessanti effetti visivi è **<feTurbulence>**, che utilizza la funzione di turbolenza di Perlin:

**<feTurbulence**

**in=**“”

**type=**“fractalnoise|turbulence”

**baseFrequency=**“”

**numOctaves=**“”

**Seed=**“”

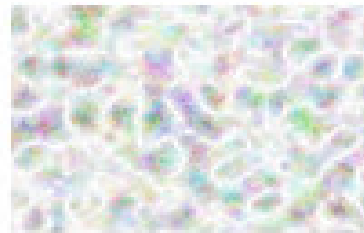
**</feTurbulence>**



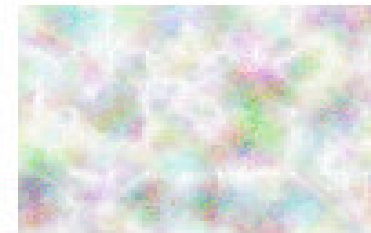
# Esempi di <feTurbulence>



type=turbulence  
baseFrequency=0.05  
numOctaves=2



type=turbulence  
baseFrequency=0.1  
numOctaves=2



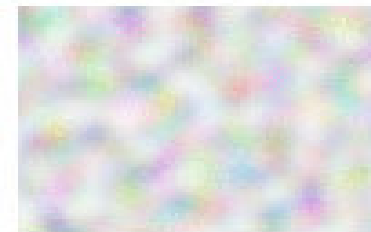
type=turbulence  
baseFrequency=0.05  
numOctaves=8



type=fractalNoise  
baseFrequency=0.1  
numOctaves=4



type=fractalNoise  
baseFrequency=0.4  
numOctaves=4



type=fractalNoise  
baseFrequency=0.1  
numOctaves=1

# Ancora <feTurbulence>

- Aumentando la frequenza di base invece di un effetto a onde si può ottenere un effetto a macchie.
- E' possibile utilizzare sfondi semplici a tinta unita o con gradienti lineari/radiali come input per la primitiva filtro <feTurbulence>.

# <feConvolveMatrix>

Il classico operatore di convoluzione può essere generato attraverso l'utilizzo della primitiva **<feConvolveMatrix>**. E' quindi possibile generare effetti di edge detection, sharpening, blurring, ecc.

La sintassi:

```
<feConvolveMatrix in=""  
    order="" kernelMatrix=""  
    edgeMode="duplicate|wrap|none"  
    preserveAlpha = "false|true"  
    result=""/>
```

# Breve panoramica

- <**feColorMatrix**> permette di applicare delle matrici di trasformazione ai colori;
- <**feComponentTransfer**> come sopra ma agisce solo su una componente.
- <**feImage**> permette di utilizzare una sorgente esterna come input per un filtro.

# L'intera gamma

- feBlend
- feColorMatrix
- feComponentTransfer
- feComposite
- feConvolveMatrix
- feDiffuseLighting
- feDisplacementMap
- feFlood
- feGaussianBlur
- feImage
- feMerge
- feMorphology
- feOffset
- feSpecularLighting
- feTile
- feTurbulence