Libreria Grafica SDL 2.0 (esempi)

http://www.libsdl.org/

Elaborazione Grafica

Visto l'ambiente Hardware/Software che permette di fare grafica, e i primi elementi sulla libreria SDL, facciamo alcuni ESEMPI:

- Disegno in coordinate intere/schermo (su "Viewport"): algoritmi di linea incrementale e di Bresenham (GCGraLib2/GCGraLib2.c funzione GC_DrawLine1)
- Disegno in coordinate floating point (su "Window") (SDL2prg0/polygonf2ren.c, SDL2prg0/draw_data.c)
- Proposta di due Esercizi

Disegno di Punti e Linee

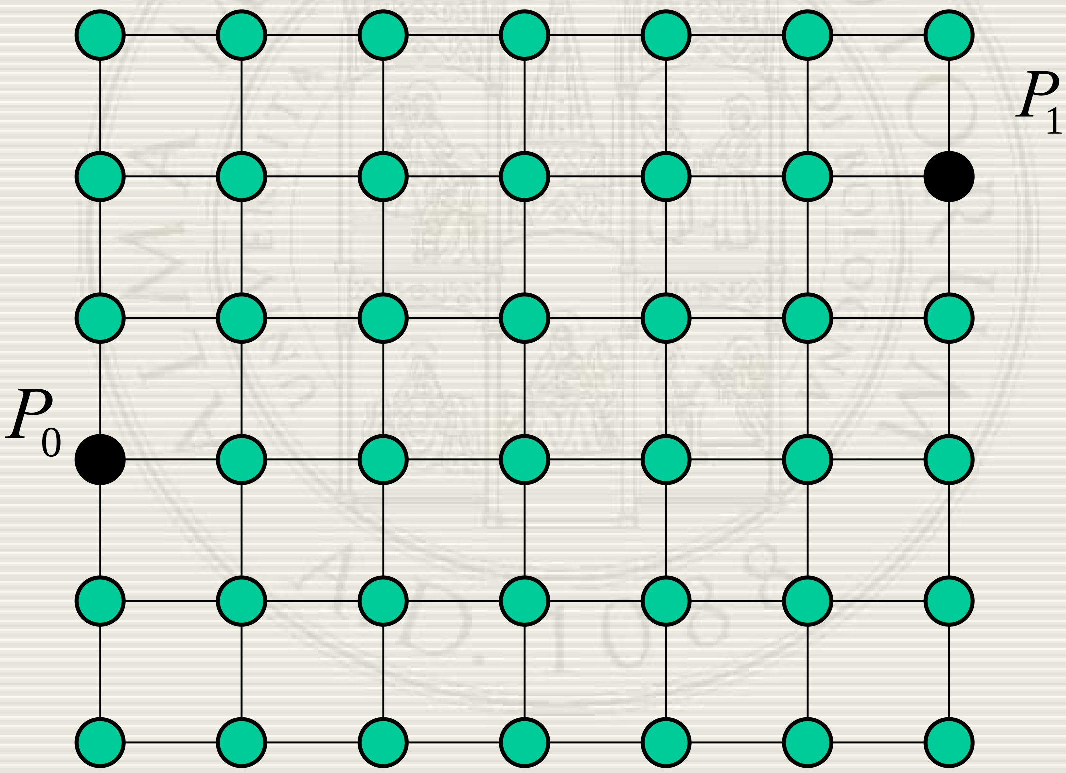
Abbiamo già visto in cosa consiste la primitiva punto cioè il disegno di un singolo punto

draw_point(x,y,color)

Vediamo ora la primitiva linea, cioè il disegno di una linea o segmeno di retta.

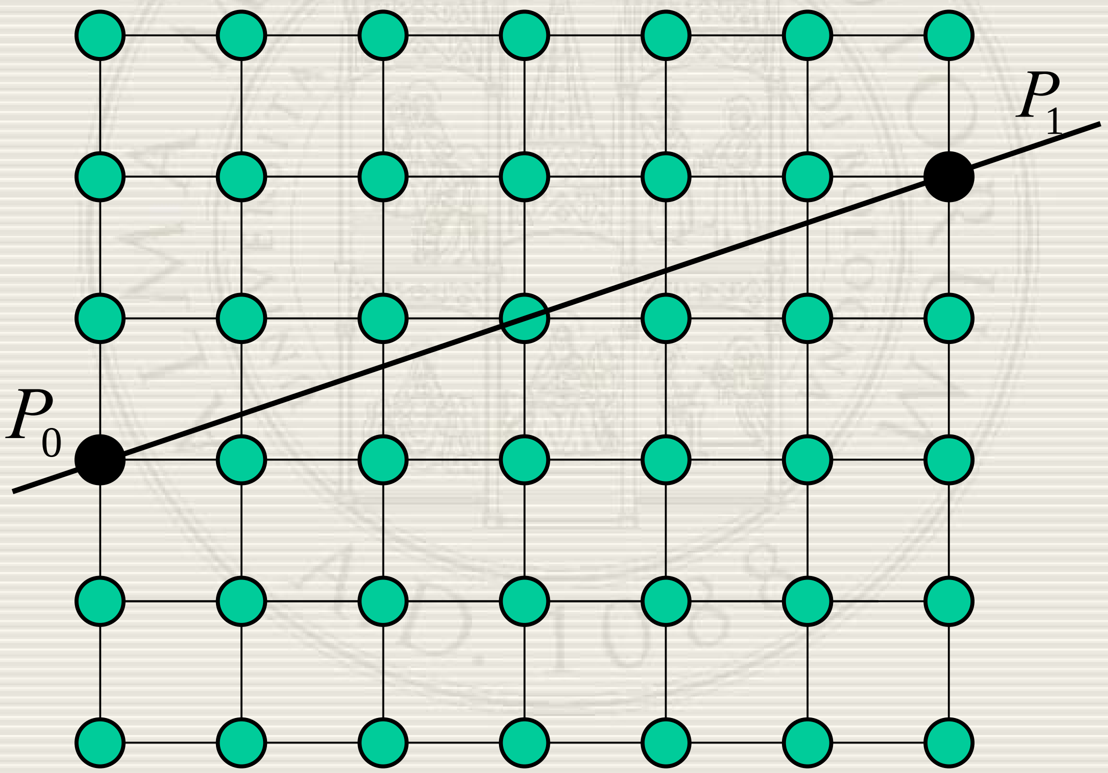
Disegno di Linee

- Dati due punti (P_0, P_1) sullo schermo (a coordinate intere) determinare quali pixel devono essere disegnati per visualizzare la linea (segmento retto) che definiscono.



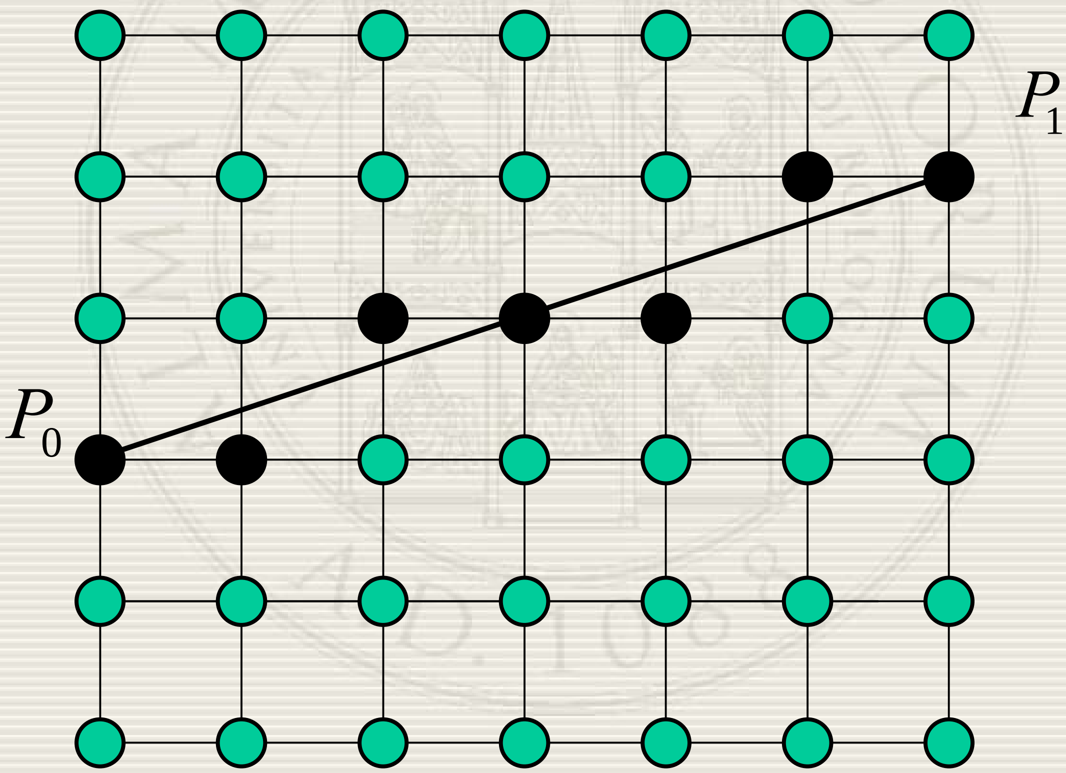
Disegno di Linee

- Dati due punti (P_0, P_1) sullo schermo (a coordinate intere) determinare quali pixel devono essere disegnati per visualizzare la linea (segmento retto) che definiscono.



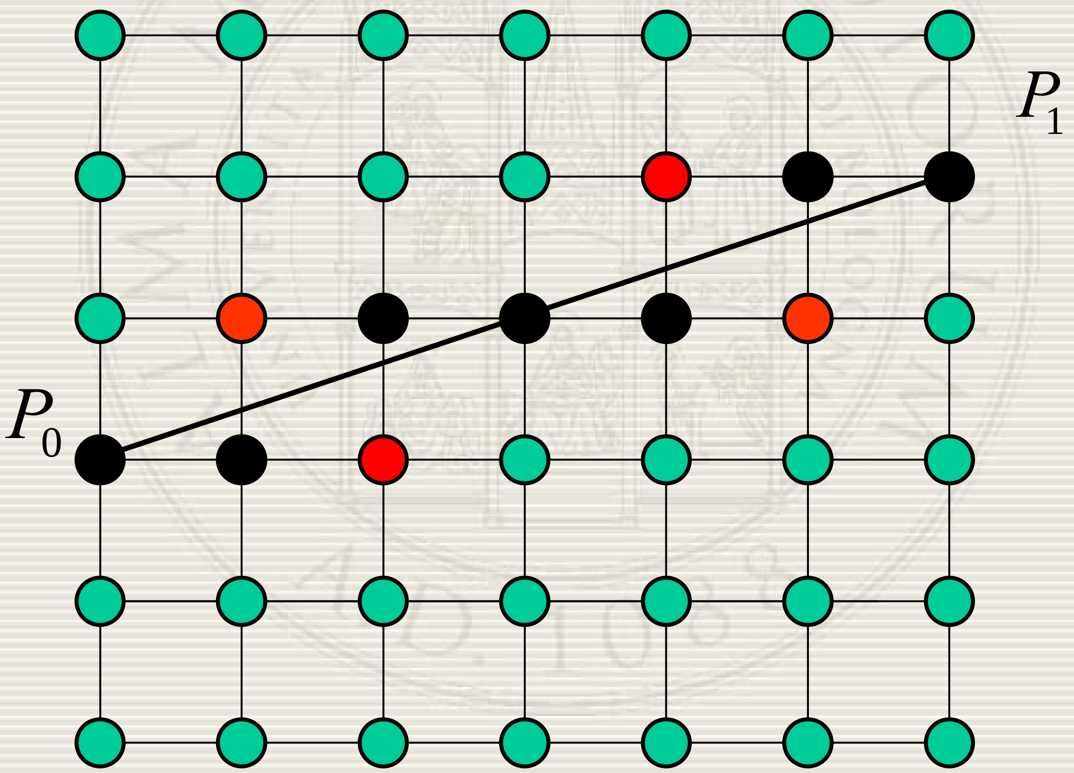
Disegno di Linee

- Dati due punti (P_0, P_1) sullo schermo (a coordinate intere) determinare quali pixel devono essere disegnati per visualizzare la linea (segmento retto) che definiscono.



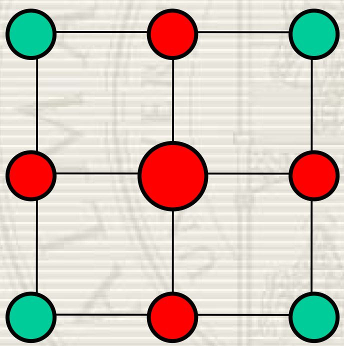
Disegno di Linee

- Perché vengono disegnati questi pixel e non anche altri? come per esempio i pixel evidenziati in rosso?

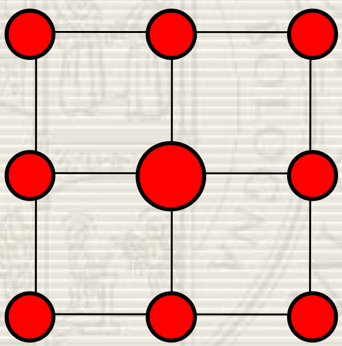


Disegno Continuo a Pixel

Si vuole procedere ad "accendere" pixel "adiacenti" per simulare un disegno continuo.
Questo porta ad una definizione di pixel adiacenti; ci sono due differenti modi:

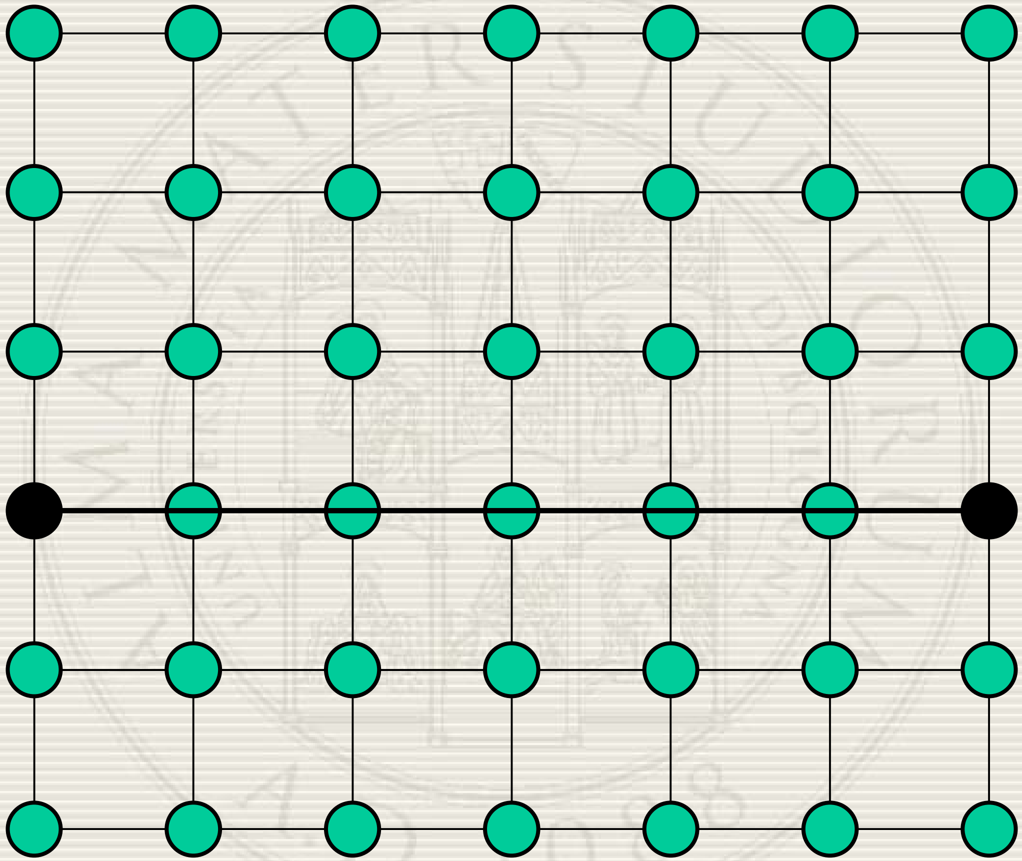


4-connected

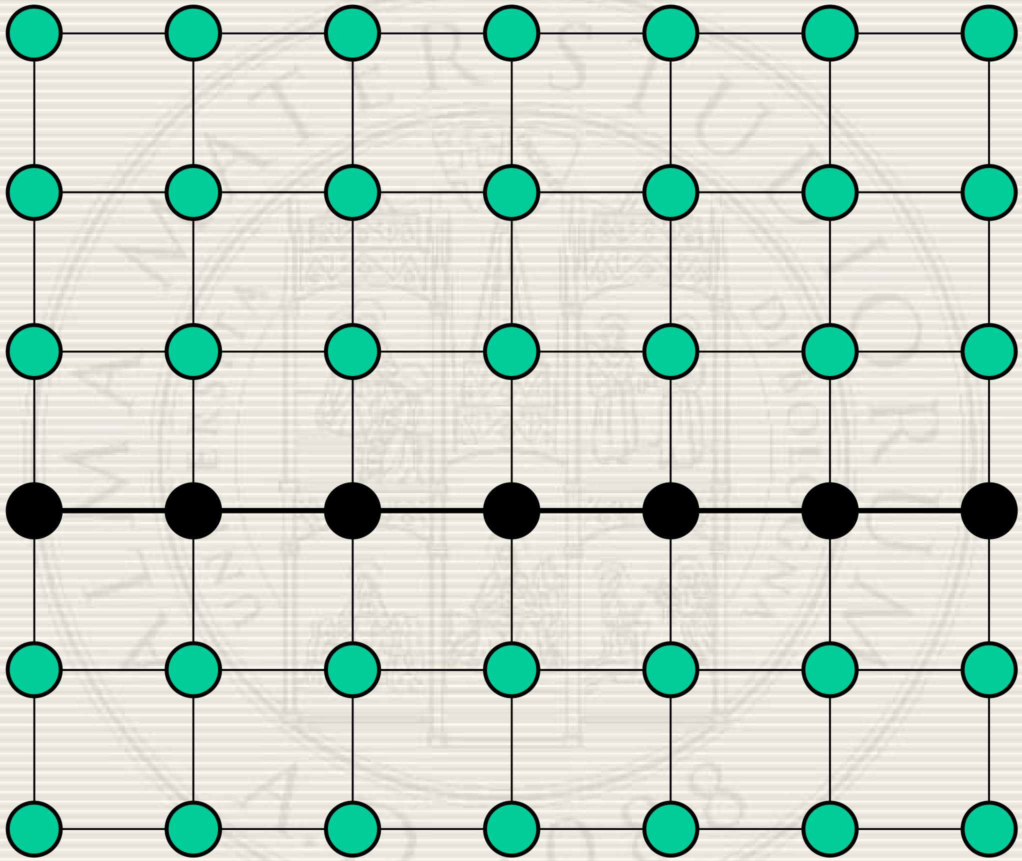


8-connected

Linee Speciali - Orizzontale

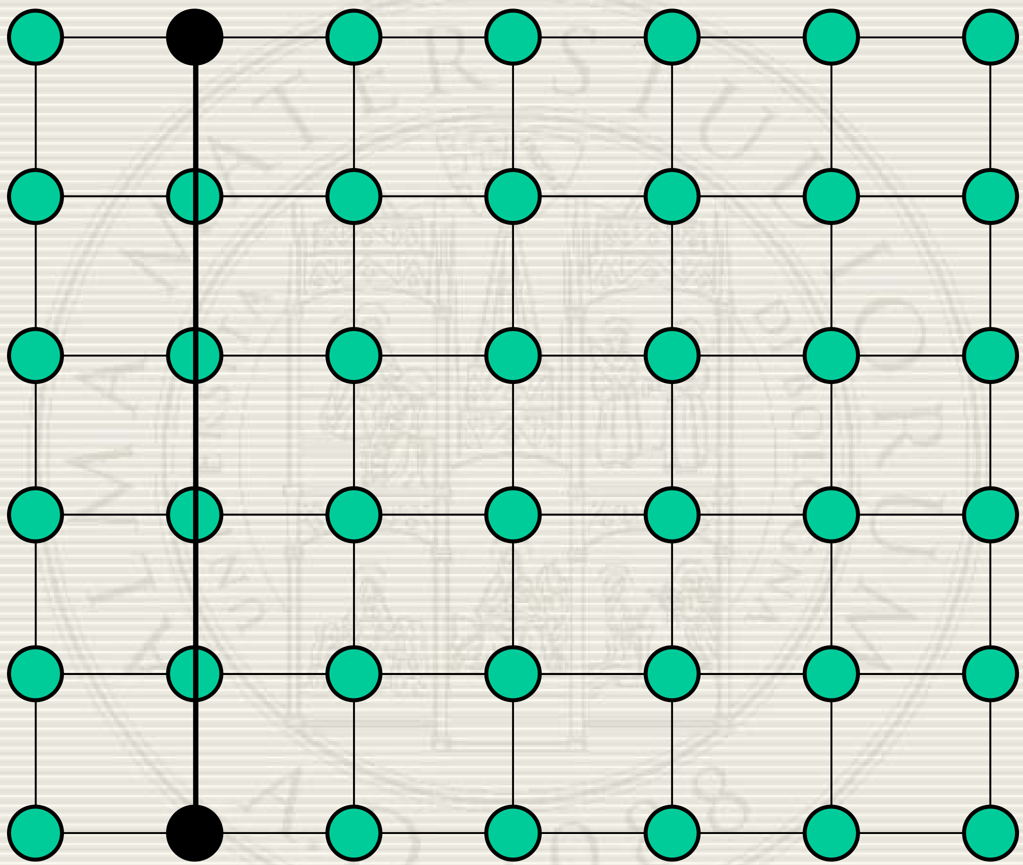


Linee Speciali - Orizzontale

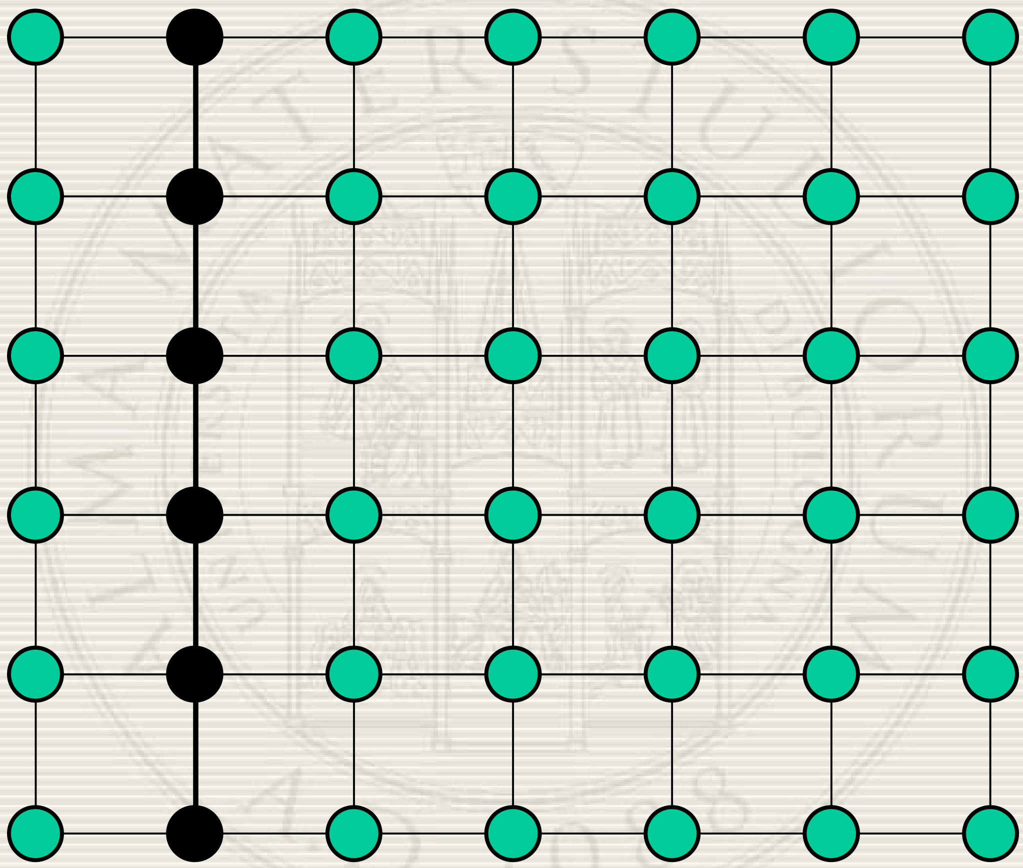


Incrementa la *x* di 1, tenendo la *y* constante

Linee Speciali - Verticale

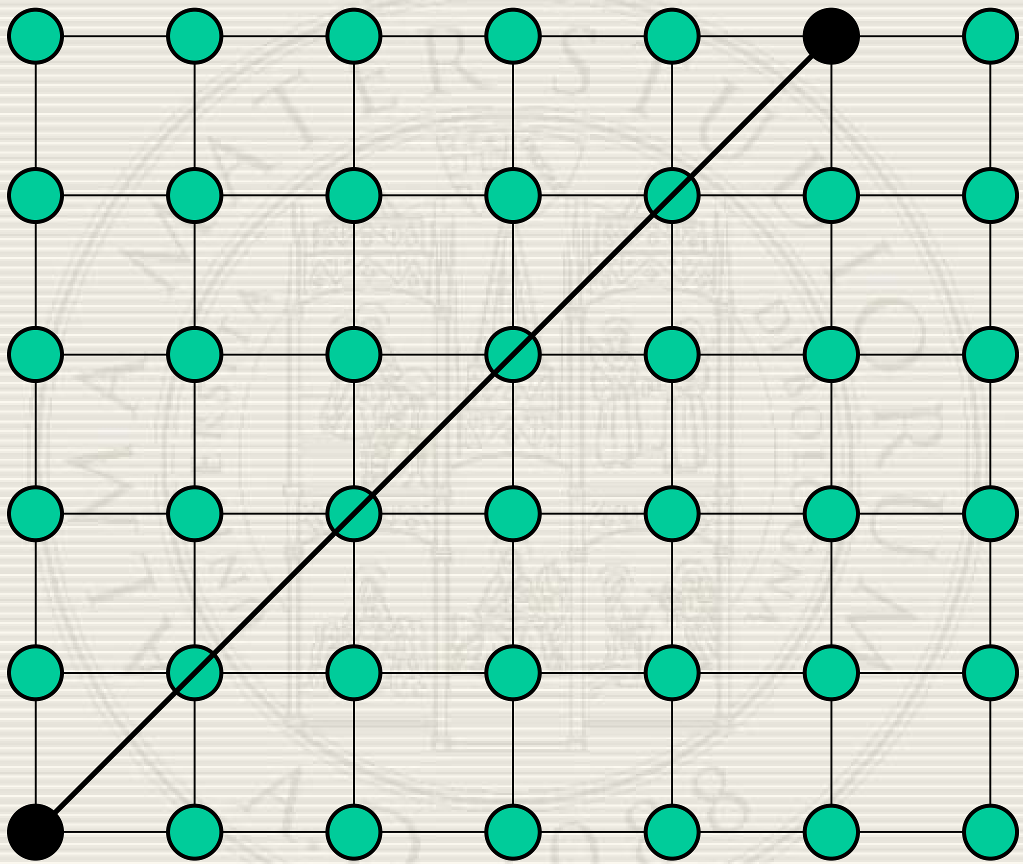


Linee Speciali - Verticale

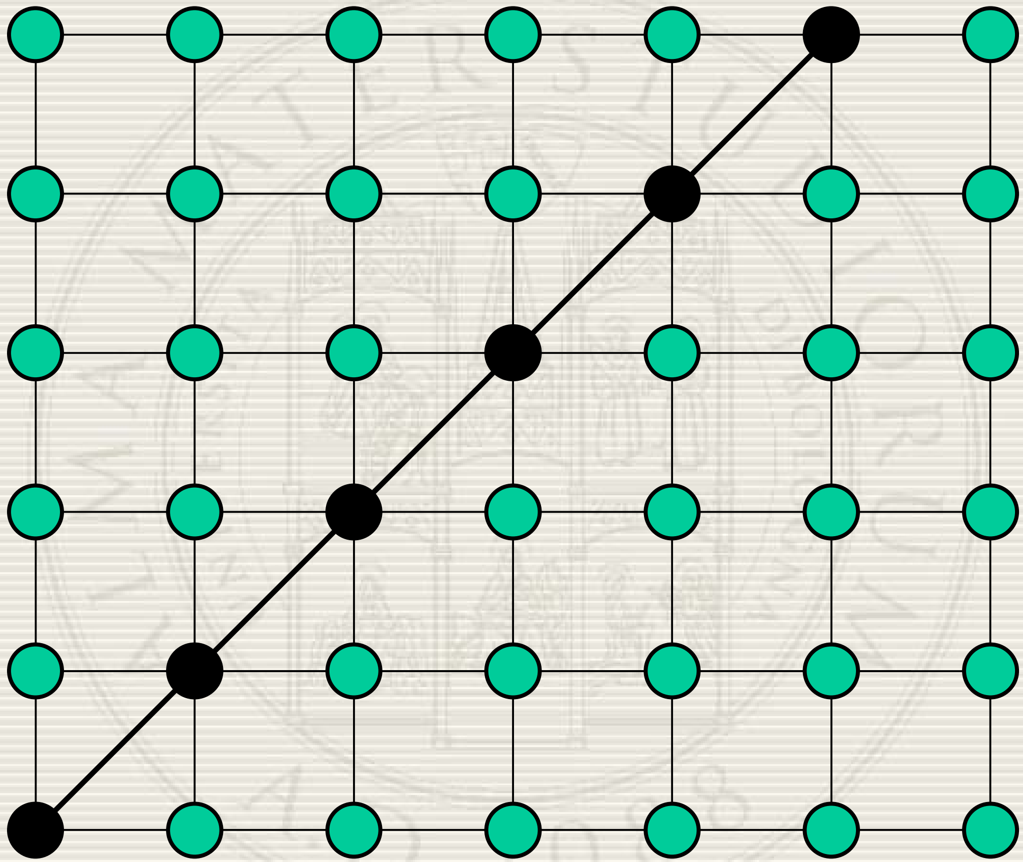


Tieni la *x* costante e incrementa la *y* di 1

Linee Speciali - Diagonale

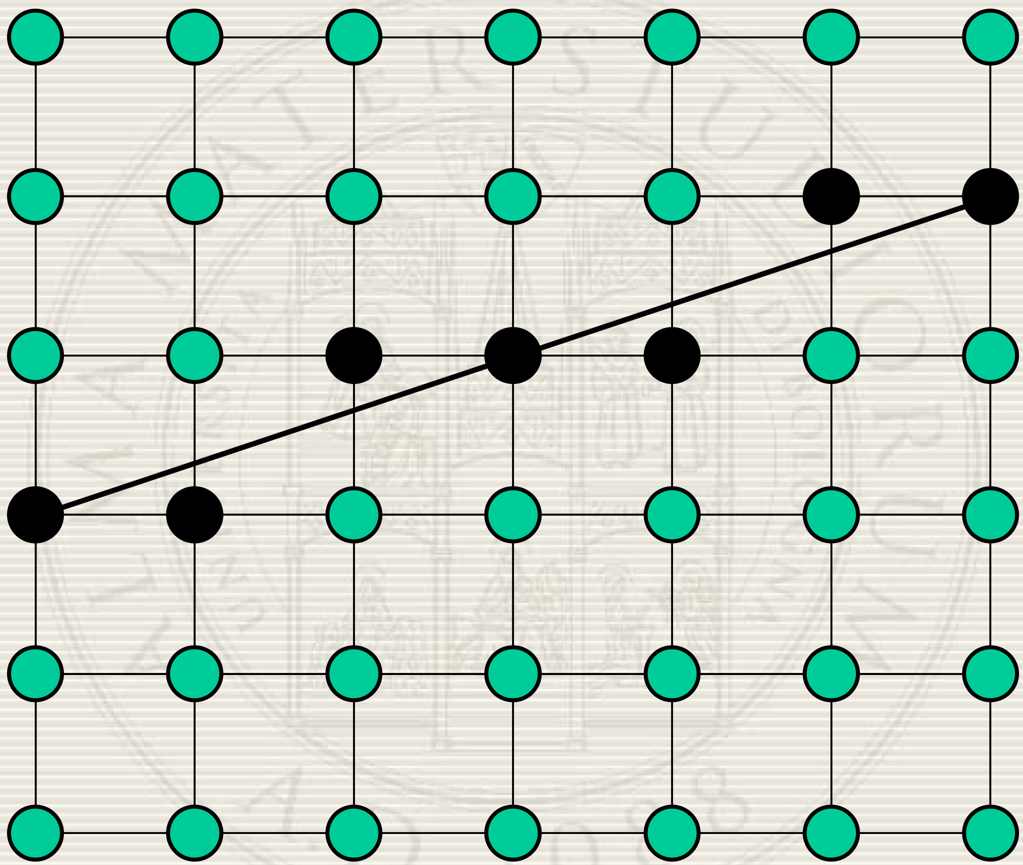


Linee Speciali - Diagonale



Incrementa la *x* di 1 e incrementa la *y* di 1

Come si procede per Linee Generiche?



Algoritmo di Linea Incrementale

Sia $L(t) = P_0 + (P_1 - P_0)\, t$ con $t \in [0, 1]$ l'espressione in forma parametrica del segmento di estremi $P_0 = (x_0, y_0)$ e $P_1 = (x_1, y_1)$. Dalla forma esplicita:

$$\begin{cases} x = x_0 + (x_1 - x_0)t \\ y = y_0 + (y_1 - y_0)t \qquad t \in [0, 1] \end{cases}$$

si possono determinare punti del segmento per opportuni valori del parametro; il numero di punti è dato dal numero di pixel necessari per rappresentare 8-connected il segmento.

Algoritmo:

```
n=max(abs(x1-x0),abs(y1-y0))
dx=(x1-x0)/n
dy=(y1-y0)/n
for ( i=0; i<=n; i++) {
   x=x0+i*dx
   y=y0+i*dy
   draw_point(round(x),round(y),col)
}
```

```
int round(float a) {
int k;
   k=(int)(a + 0.5);
   return k;
}
```

Algoritmo di Linea Incrementale

Il metodo incrementale consiste nel determinare le coordinate del nuovo punto da quelle del punto precedente, anziché dall'espressione parametrica:

$$x_{i+1} = x_0 + (i + 1)\, dx$$

$$x_i \quad = x_0 + i\, dx$$

sottraendo si ha: $x_{i+1} = x_i + dx$; (analogamente $y_{i+1} = y_i + dy$).

Algoritmo:

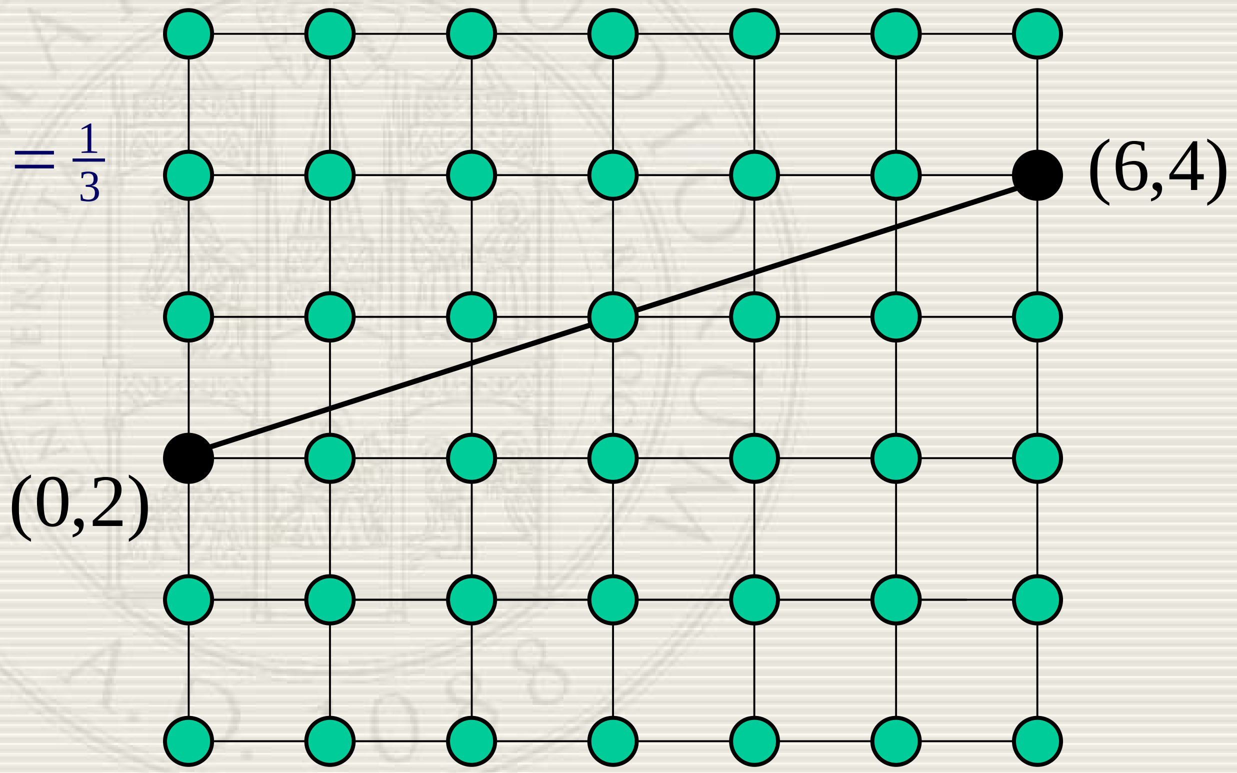
```
n=max(abs(x1-x0),abs(y1-y0))
dx=(x1-x0)/n
dy=(y1-y0)/n
x=x0
y=y0
draw_point(x,y,col)
for (i=1; i<=n, i++) {
  x=x+dx
  y=y+dy
  draw_point(round(x),round(y),col)
}
```

Algoritmo di Linea Incrementale: esempio

$P_0 = (0\ ,\ 2)$ $P_1 = (6\ ,\ 4)$

sarà: $n=6,\ dx=1$

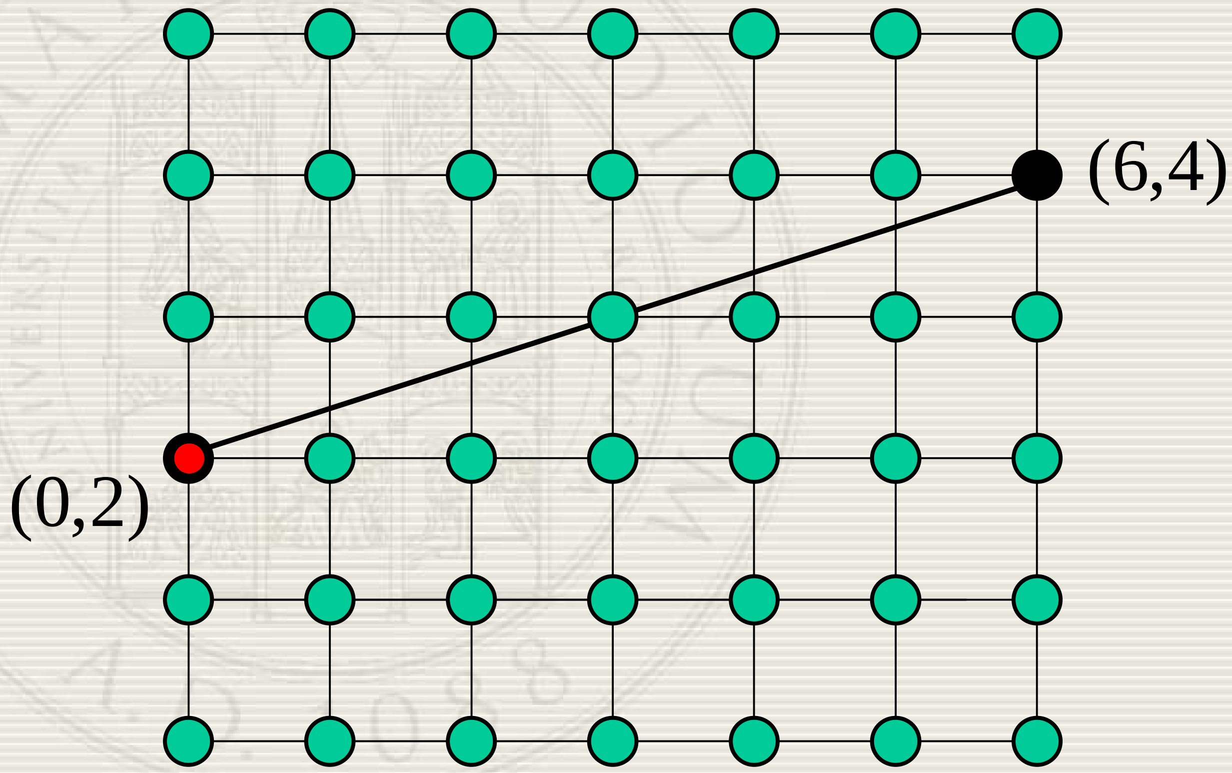
$$dy = \frac{y_1 - y_0}{x_1 - x_0} = \frac{4-2}{6-0} = \frac{1}{3}$$



Algoritmo di Linea Incrementale: esempio

$P_0 = (0\ ,\ 2)$ $P_1 = (6\ ,\ 4)$ $n=6,\ dx=1,\ dy=1/3 \cong 0.333$

--> (0,2)

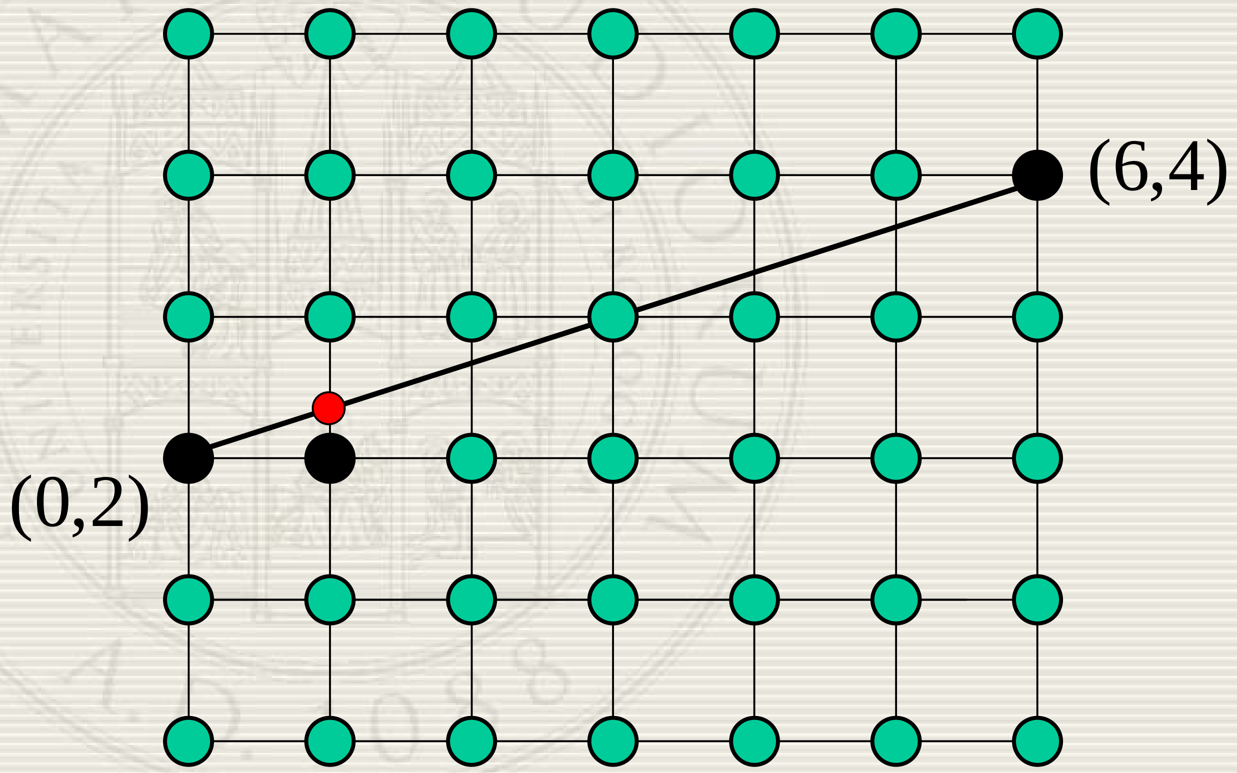


Algoritmo di Linea Incrementale: esempio

$P_0 = (0\ ,\ 2)$ $P_1 = (6\ ,\ 4)$ $n=6,\ dx=1,\ dy=1/3 \simeq 0.333$

--> (0,2)

(1,2.333) --> (1,2)



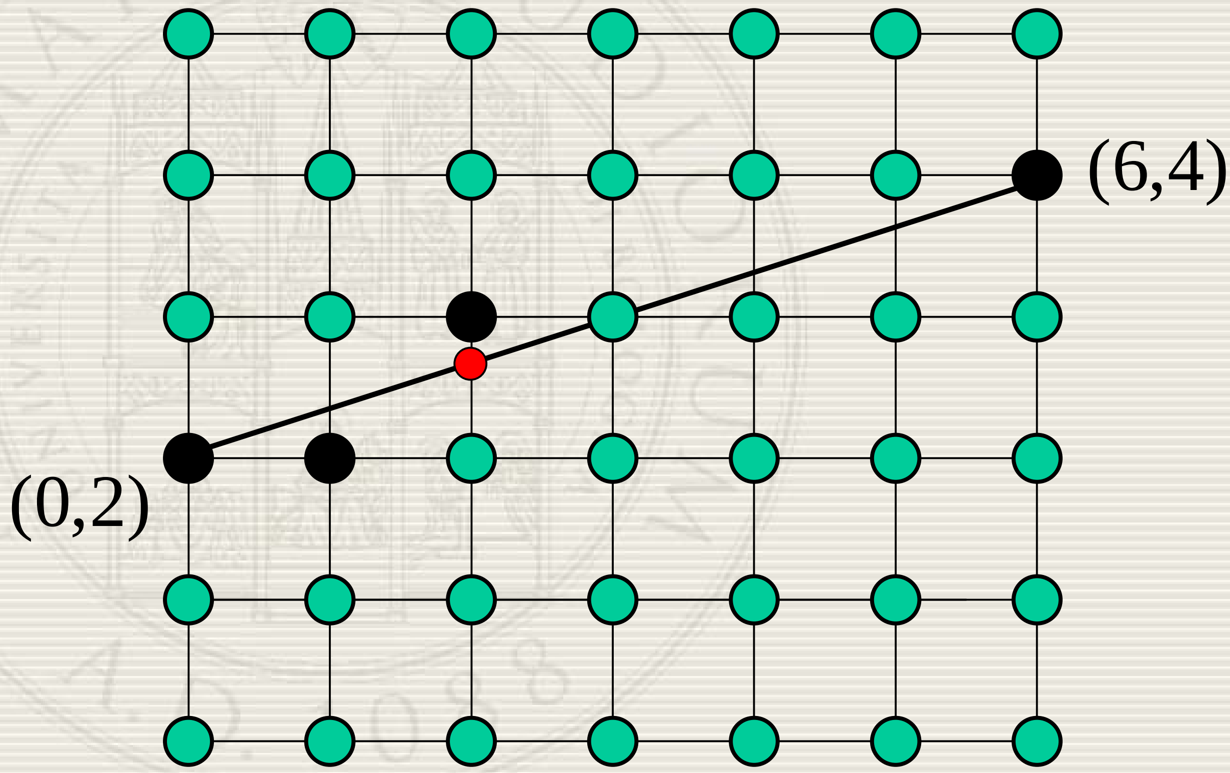
Algoritmo di Linea Incrementale: esempio

$P_0 = (0\ ,\ 2)$ $P_1 = (6\ ,\ 4)$ $n=6,\ dx=1,\ dy=1/3 \simeq 0.333$

--> (0,2)

(1,2.333) --> (1,2)

(2,2.666) --> (2,3)



Algoritmo di Linea Incrementale: esempio

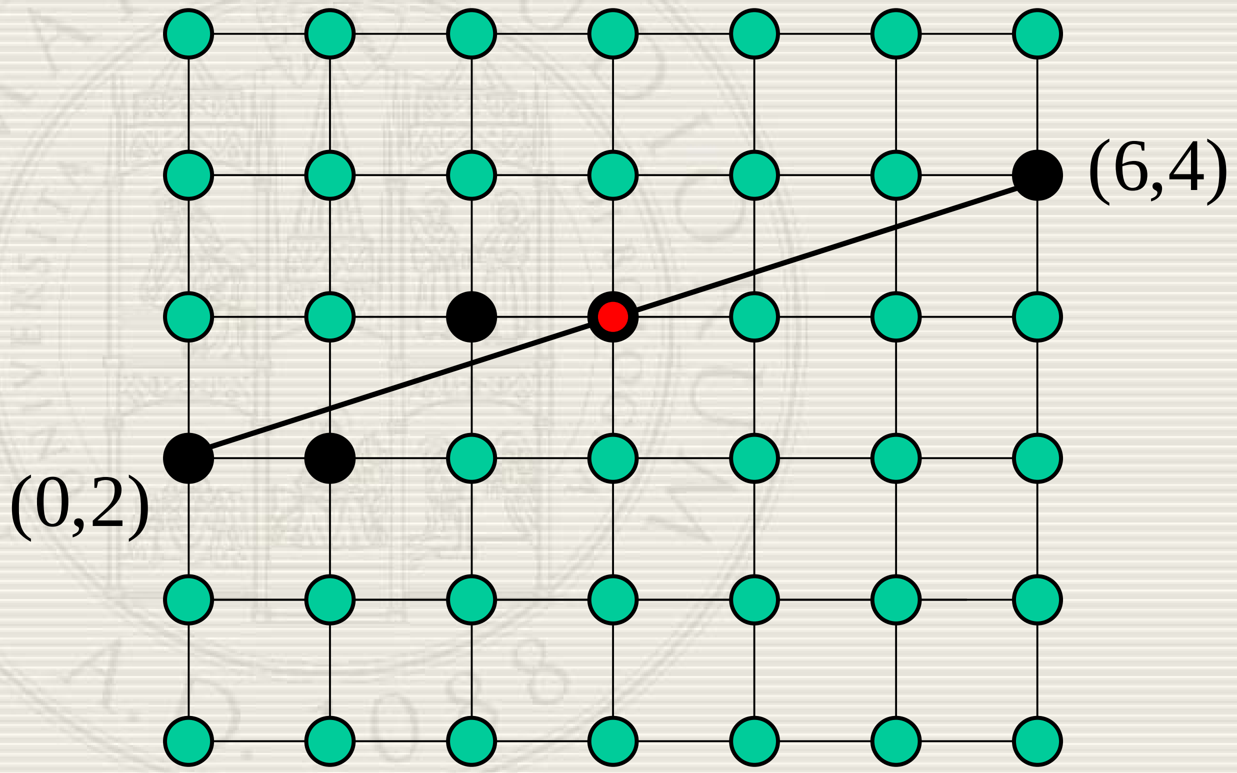
$P_0 = (0\ ,\ 2)$ $P_1 = (6\ ,\ 4)$ $n=6,\ dx=1,\ dy=1/3 \simeq 0.333$

--> (0,2)

(1,2.333) --> (1,2)

(2,2.666) --> (2,3)

(3,3.000) --> (3,3)



Algoritmo di Linea Incrementale: esempio

$P_0 = (0\ ,\ 2)$ $P_1 = (6\ ,\ 4)$ $n=6,\ dx=1,\ dy=1/3 \simeq 0.333$

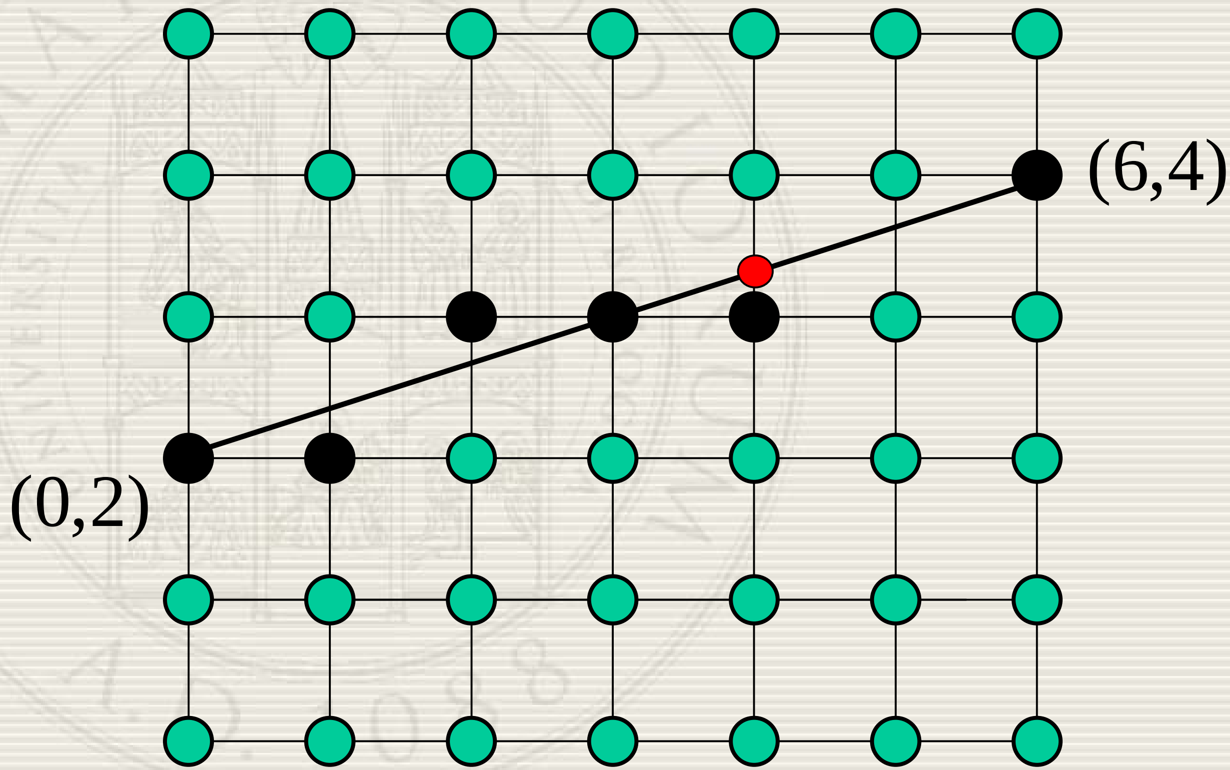
--> (0,2)

(1,2.333) --> (1,2)

(2,2.666) --> (2,3)

(3,3.000) --> (3,3)

(4,3.333) --> (4,3)



Algoritmo di Linea Incrementale: esempio

$P_0 = (0\ ,\ 2)$ $P_1 = (6\ ,\ 4)$ $n=6,\ dx=1,\ dy=1/3 \simeq 0.333$

--> (0,2)

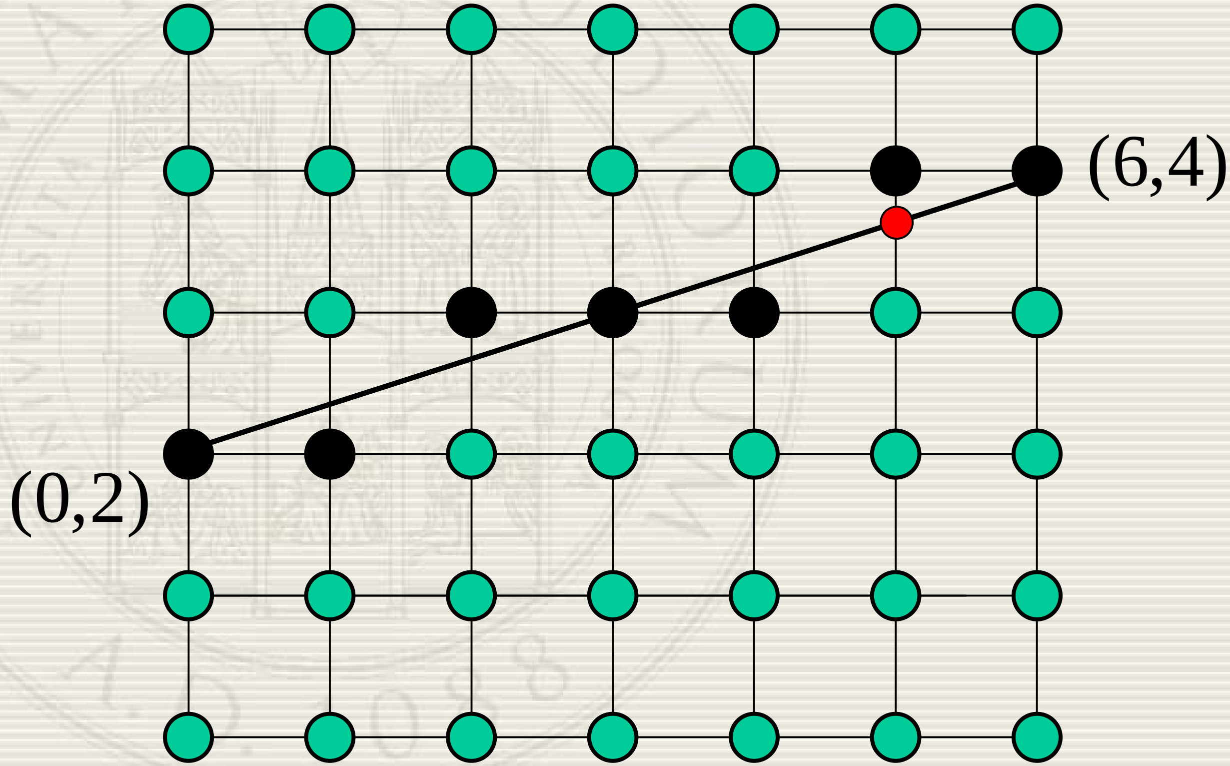
(1,2.333) --> (1,2)

(2,2.666) --> (2,3)

(3,3.000) --> (3,3)

(4,3.333) --> (4,3)

(5,3.666) --> (5,4)



Algoritmo di Linea Incrementale: esempio

$P_0 = (0\ ,\ 2) \quad P_1 = (6\ ,\ 4) \qquad n=6,\ dx=1,\ dy=1/3 \simeq 0.333$

--> (0,2)

(1,2.333) --> (1,2)

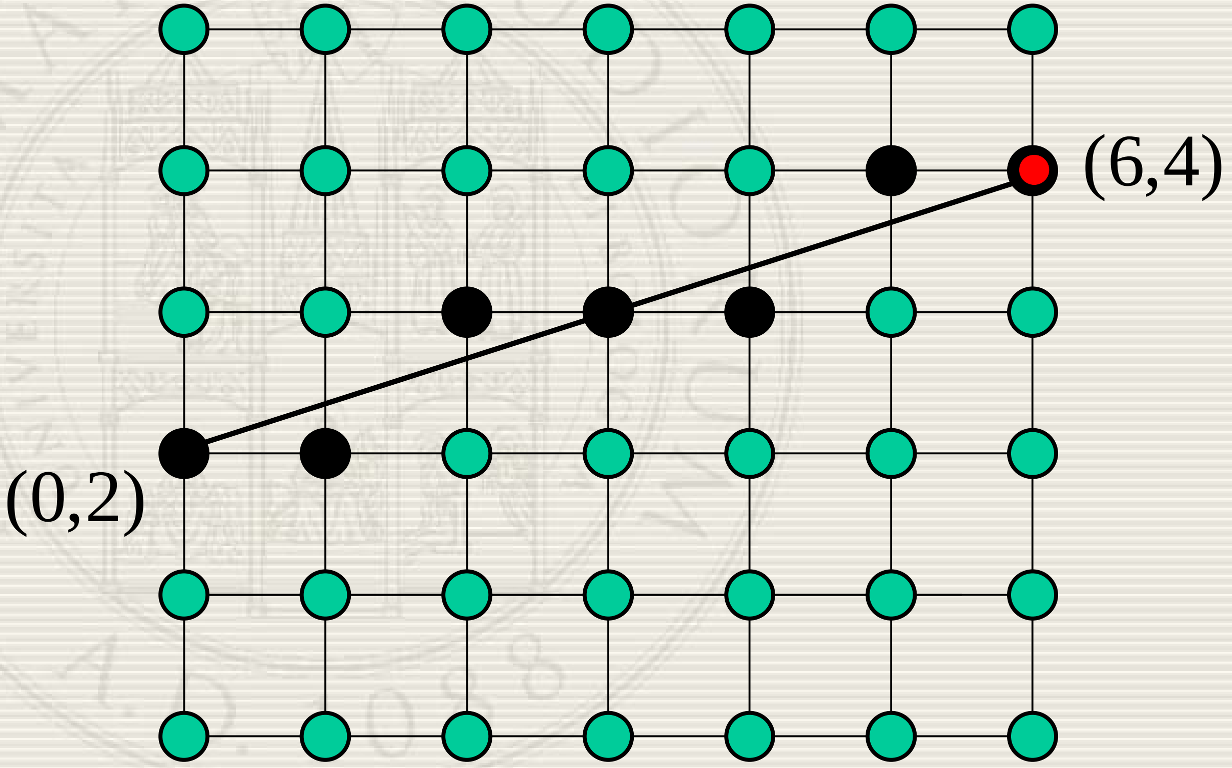
(2,2.666) --> (2,3)

(3,3.000) --> (3,3)

(4,3.333) --> (4,3)

(5,3.666) --> (5,4)

(6,4.000) --> (6,4)



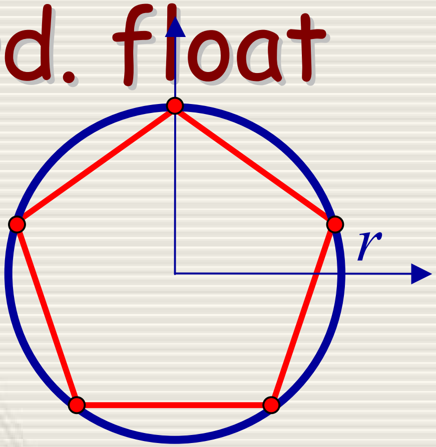
Algoritmo di Linea Incrementale: note

- Operazioni aritmetiche floating point
- Arrotondamento (funzione round( ))
- Si può fare meglio?

Algoritmo di Linea di Bresenham

- Solo operazioni aritmetiche fra interi
- Più precisamente addizioni, sottrazioni e shift di bit (moltiplicazioni per 2)
- Si estende ad altri tipi di forme (circonferenza, coniche)
- E' l'algoritmo implementato in hardware sulle schede grafiche
- Lo trovate implementato nella libreria GCGraLib2: file GCGraLib2.c, funzione GC_DrawLine1()

Problema: disegno in coord. float



Disegnare un poligono regolare di *n* lati.
Dati: *n* numero di lati (o vertici)
r raggio circonferenza circoscritta
Risultato: disegno a pixel del poligono
Metodo: si usa l'equazione della circonferenza di centro l'origine e raggio unitario e si determinano *n* punti equidistanti su di essa.

$$\begin{cases} x = \cos(t) \\ y = \sin(t) \end{cases} \quad t \in [0, 2\pi[$$

Algoritmo:

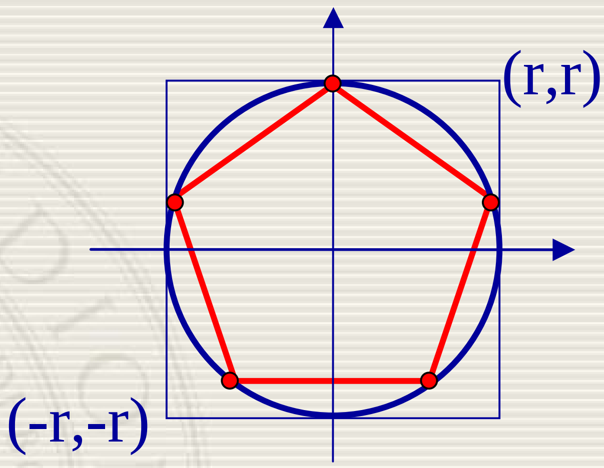
```
step = 6.28/n
for (i=0; i<=n; i++)
{
   alfa = i * step
   x[i] = r * cos(alfa)
   y[i] = r * sin(alfa)
}
```

Problema: disegno in coord. float

Per ottimizzare:

```
step = 6.28/n
c=cos(step)
s=sin(step)
x[0]=r
y[0]=0
for (i=1; i<=n; i++)
{
  x[i] = x[i-1]*c – y[i-1]*s
  y[i] = x[i-1]*s + y[i-1]*c
}
```

Vedi più avanti



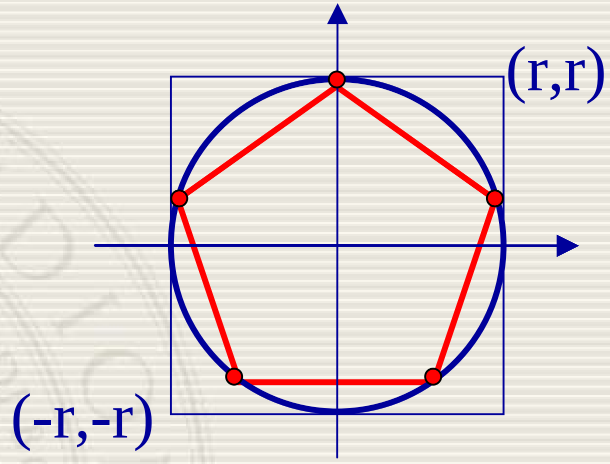
Ma i punti così determinati sono in coordinate floating point e in un quadrato $[-r,r]$x$[-r,r]$ che chiameremo Window;
Dobbiamo disegnare su una Viewport sullo schermo in coordinate intere.

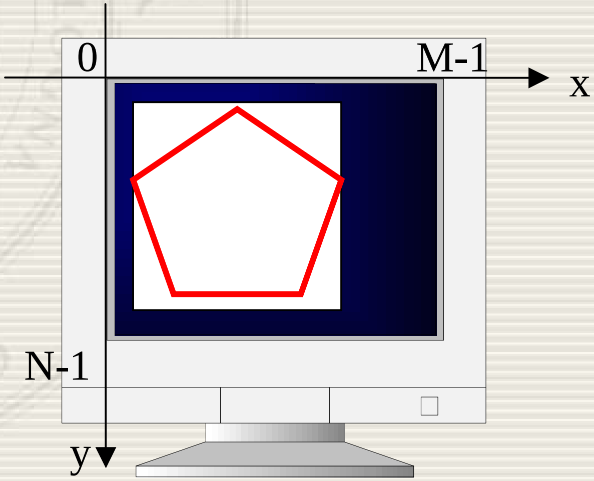
Definizioni di Window e Viewport

Chiameremo **Window** un'area rettangolare del piano di disegno che tipicamente contiene il nostro disegno.

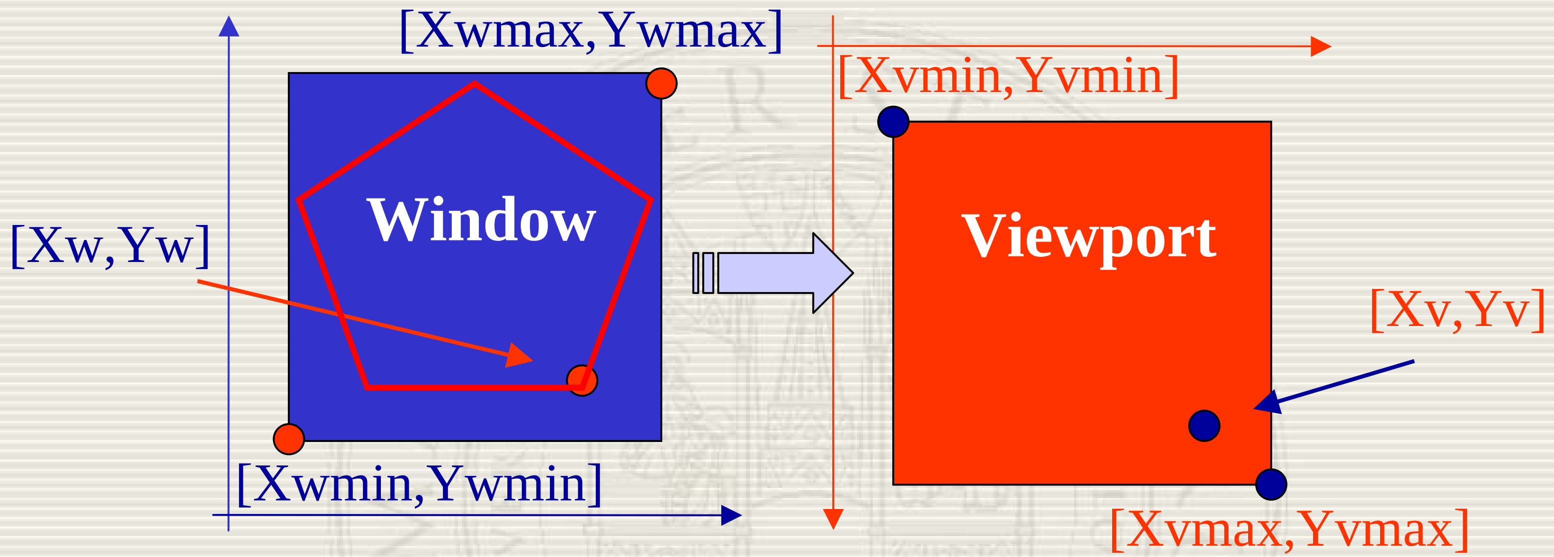
Chiameremo **Viewport** un'area rettangolare dello schermo su cui vogliamo rappresentare il disegno.

Problema: dovremo passare da coordinate floating point a coordinate intere!!





Trasformazione Window-Viewport



Gestiamo separatamente i due assi coordinati:

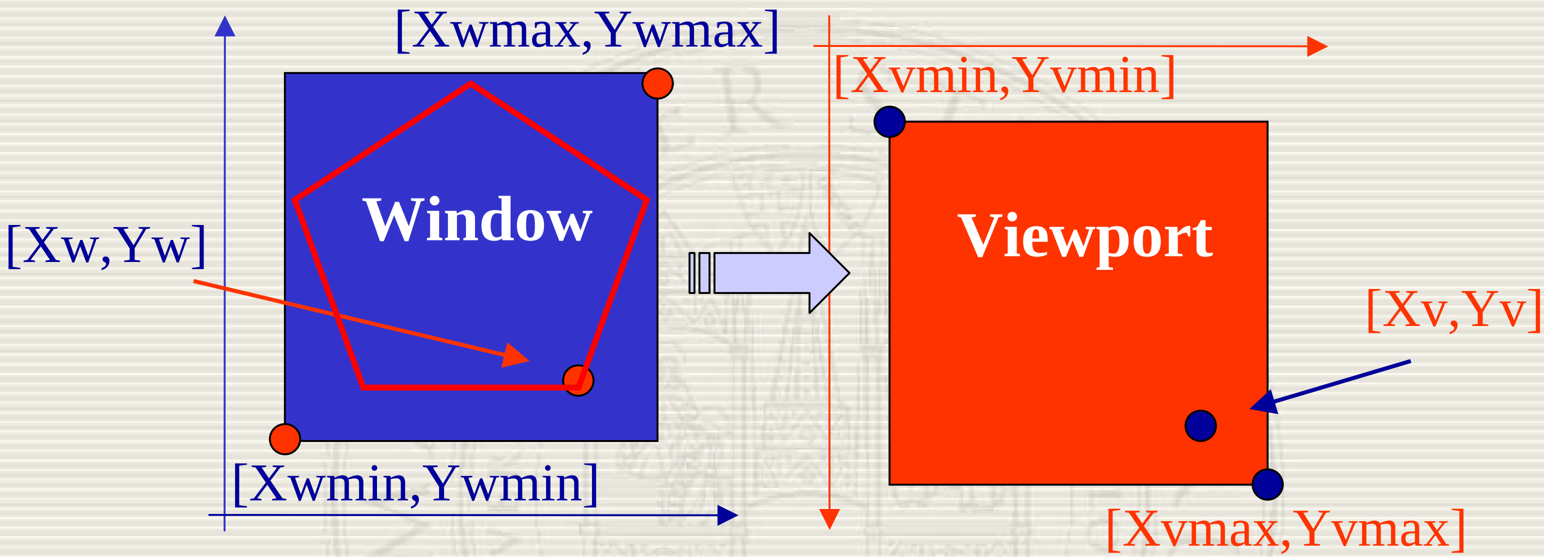
$$\boxed{(Xv-Xvmin) : (Xvmax-Xvmin) = (Xw-Xwmin) : (Xwmax-Xwmin)}$$

da cui: $Xv = Scx\,(Xw - Xwmin) + Xvmin$

$$\boxed{(Yvmax-Yv) : (Yvmax-Yvmin) = (Yw-Ywmin) : (Ywmax-Ywmin)}$$

da cui: $Yv = Scy\,(Ywmin - Yw) + Yvmax$

Trasformazione Window-Viewport



Scx = (Xvmax - Xvmin)/(Xwmax – Xwmin)

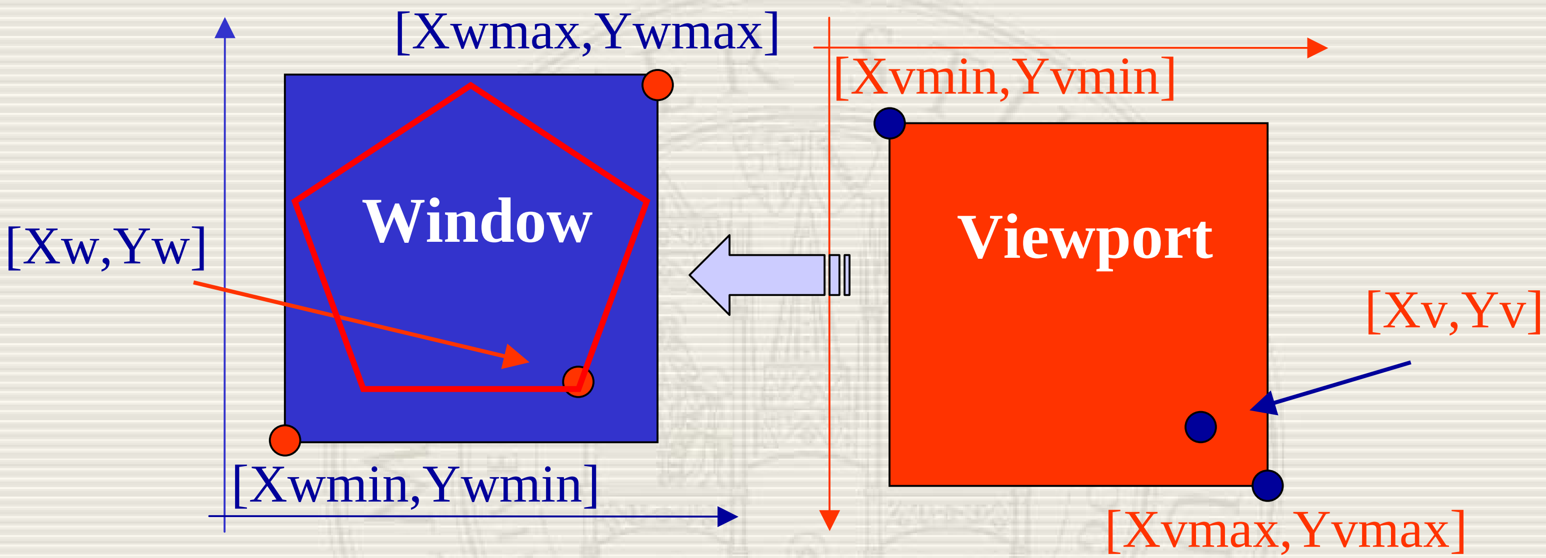
Scy = (Yvmax - Yvmin)/(Ywmax – Ywmin)

Che possiamo implementare così:

Xv = (int)(Scx * (Xw - Xwmin) + Xvmin + 0.5)

Yv = (int)(Scy * (Ywmin - Yw) + Yvmax + 0.5)

Trasformazione Viewport-Window

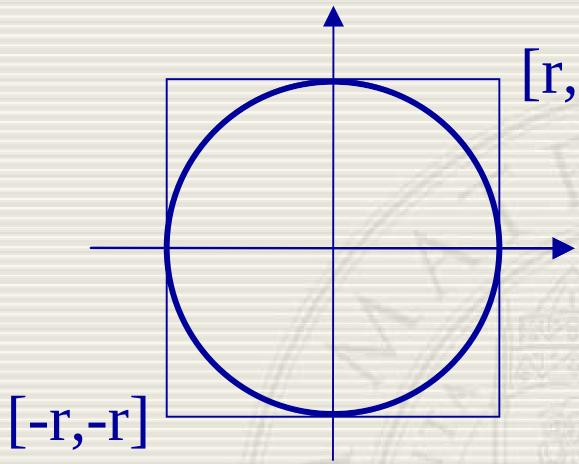


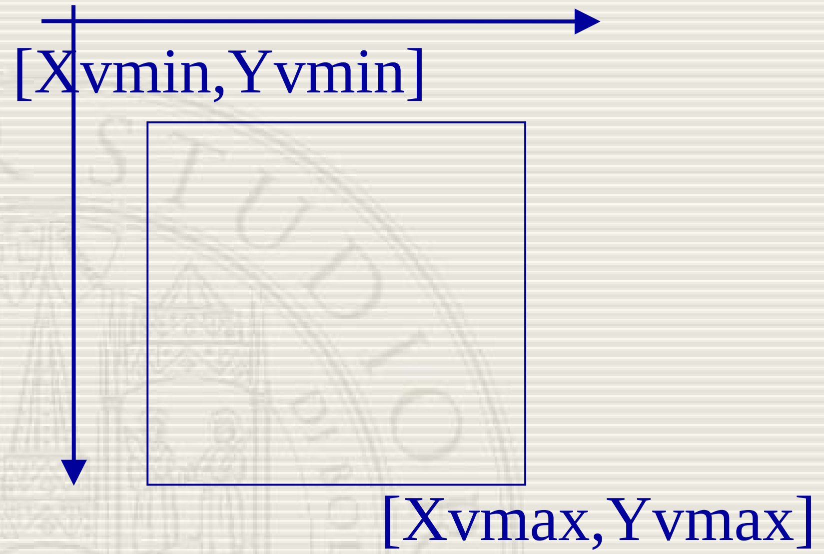
Trasformazione inversa:

$$Xw = (Xv - Xvmin) / Scx + Xwmin$$

$$Yw = (Yvmax - Yv) / Scy + Ywmin$$

Problema: disegno in coord. float



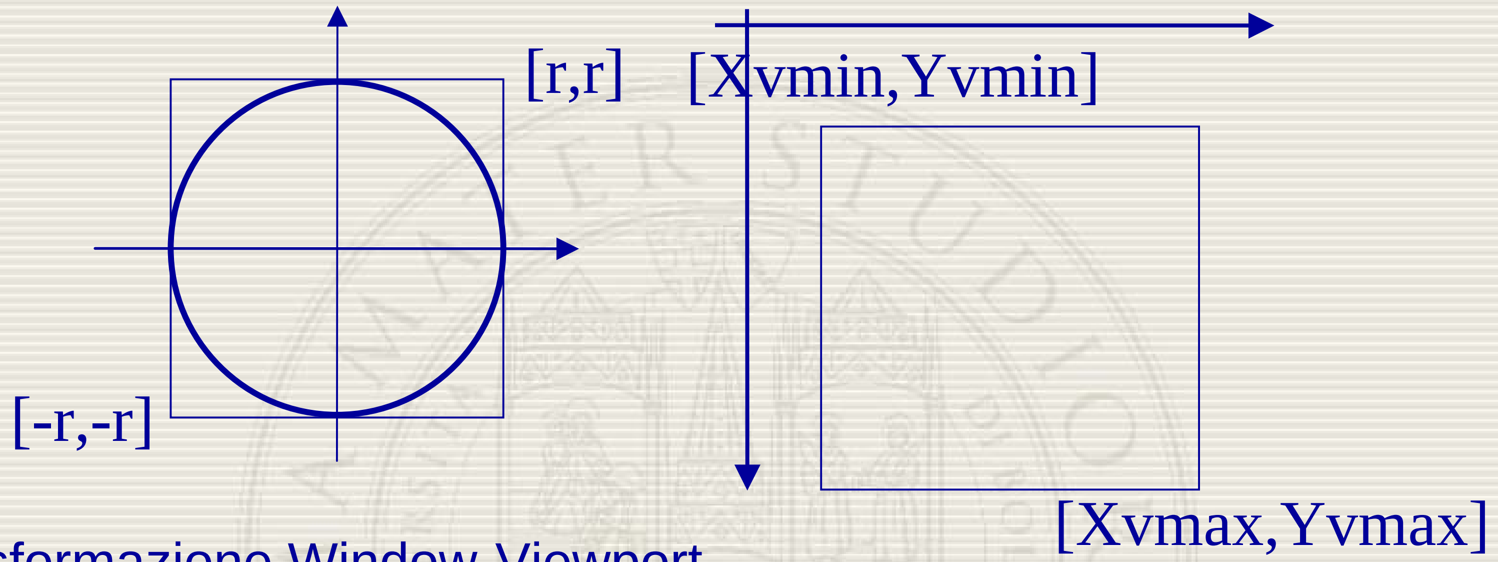


Trasformazione Window-Viewport
Definiamo due strutture:

```
typedef struct
{
        int vxmin;
        int vxmax;
        int vymin;
        int vymax;
}VIEW;
```

```
typedef struct
{
        float wxmin;
        float wxmax;
        float wymin;
        float wymax;
}WIN;
```

Problema: disegno in coord. float



Trasformazione Window-Viewport
Scriviamo una funzione

```
void wind_view(float px,float py,int *ix,int *iy,
               float scx,float scy,VIEW *view,WIN *win)
```

Si applichi la trasformazione suddetta ai punti (x[i],y[i]) i=0,..,n
e si disegni su schermo la spezzata di vertici così ottenuti.

Programma esempio: polygonf2ren.c

Problema: disegno in coord. float

Produrre il grafico di una funzione.

Dati: espressione *y=f(x)*

intervallo di definizione *[xa,xb]*

Risultato: grafico di *n+1* punti della funzione

Metodo: si campioni la funzione in *n+1* punti, per esempio equidistanti, nell'intervallo di definizione

Algoritmo: tabulazione

```
step = (xb-xa)/n
for (i=0; i<=n; i++)
{
  x[i] = xa + i * step
  y[i] = f(x[i])
}
```

Problema: disegno in coord. float

Bisogna determinare la Window, cioè il più piccolo rettangolo che contiene i punti (x[i],y[i]), i=0,..,n

```
Ywmin=y[0]
Ywmax=y[0]
for (i=1; i<=n; i++)
{
   if (y[i]>Ywmax)
      Ywmax=y[i]
   else
       if (y[i]<Ywmin)
          Ywmin=y[i]
}
```

La Window sarà: [Xwmin,Xwmax]x[Ywmin,Ywmax]

con Xwmin = xa

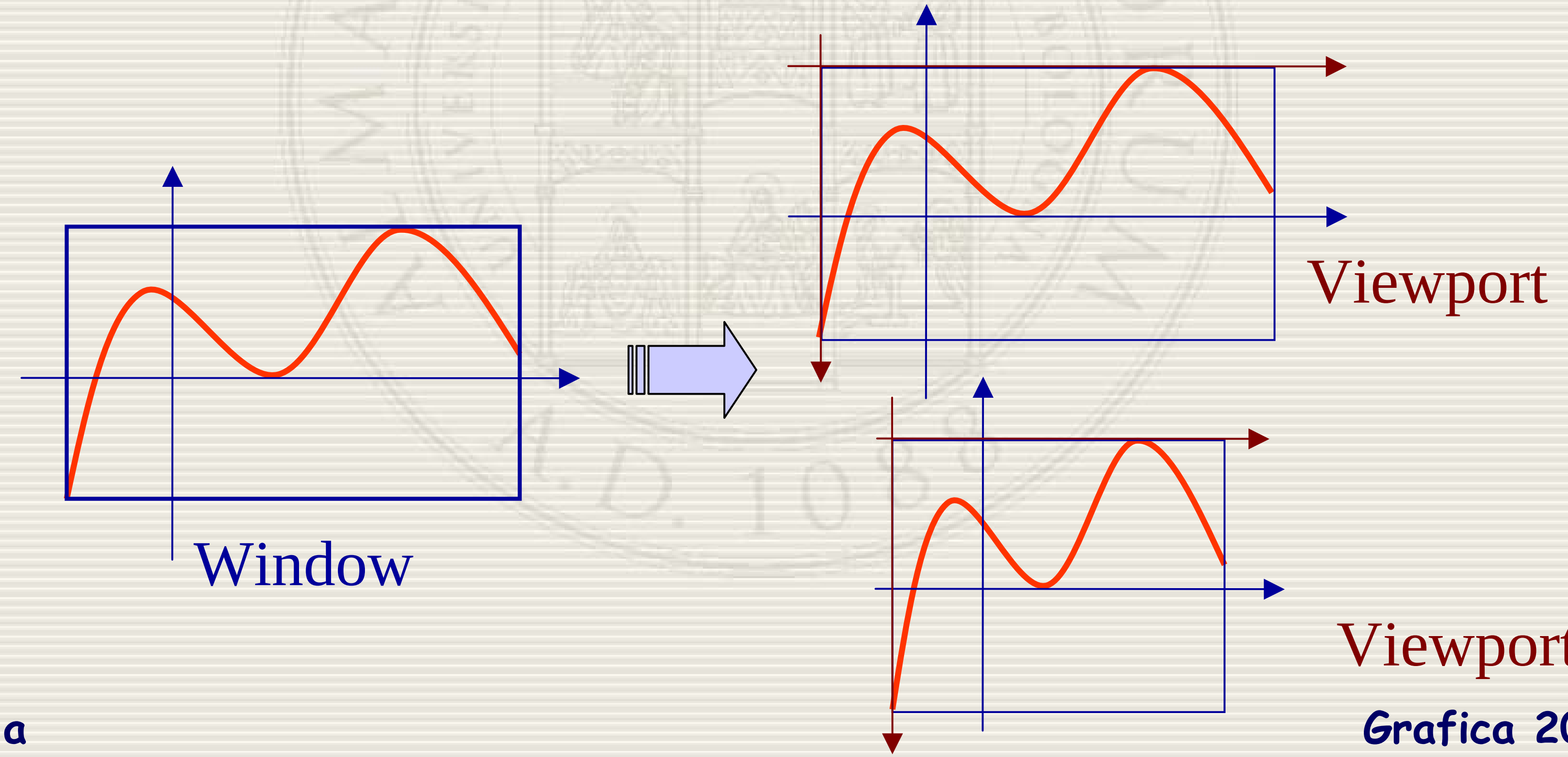
e Xwmax = xb

Problema: disegno in coord. float

Si definisca una Viewport con lo stesso aspect ratio (proporzioni fra i lati) della Window;

Definita quindi [Xvmin,Xvmax]x[Yvmin,Yvmax] si applichi la trasformazione Window-Viewport ai punti (x[i],y[i]) i=0,..,n

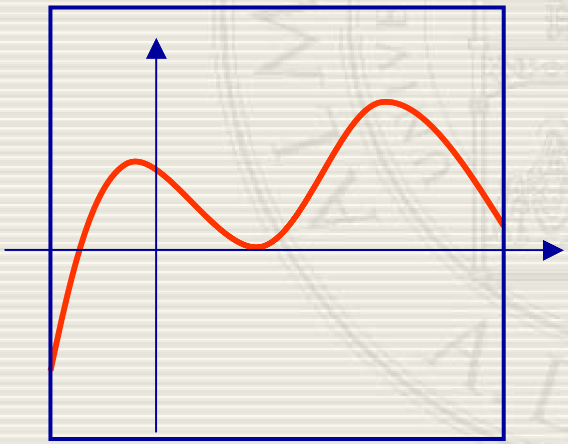
Si applichi la trasformazione suddetta ai punti (x[i],y[i]) i=0,..,n e si disegni su schermo la spezzata di vertici così ottenuti.



Problema: disegno in coord. float

In alternativa si definisca una Viewport quadrata e si determini la più piccola Window quadrata contenente i punti (x[i],y[i]), i=0,..,n, così da avere una rappresentazione corretta delle proporzioni.

Si applichi poi la trasformazione suddetta ai punti (x[i],y[i]) i=0,..,n e si disegni su schermo la spezzata di vertici così ottenuti.



Window

Window

Problema: disegno in coord. Float

```
dx=Xwmax-Xwmin
dy=Ywmax-Ywmin
if (dy>dx)
{
    diff=(dy-dx)/2
    Xwmin=Xwmin-diff
    Xwmax=Xwmax+diff
}
else
{
    diff=(dx-dy)/2
    Ywmin=Ywmin-diff
    Ywmax=Ywmax+diff
}
```

Programma esempio: draw_data.c

Esercizio 1

A partire dal codice SDL2prg0/draw_data.c che legge un file di punti in coordinate floating point e li disegna in una Viewport, realizzare un codice per il disegno di una curva piana definita in forma parametrica:

$$c(t) = \left[c_x(t), c_y(t) \right] \qquad t \in [0,1]$$

Lo si chiami draw_param_curve.c

Esercizio 2

A partire dal codice SDL2prg0/inter_polygon2ren.c che permette di definire una poligonale di n+1 vertici interattivamente, realizzare un codice che disegni la curva di Bézier di grado n di punti di controllo i vertici dati (si usi l'algoritmo di valutazione di de Casteljau).

Una volta disegnata la curva sia possibile modificarne la forma spostando col mouse i singoli punti di controllo.

Lo si chiami draw_bezier_curve.c

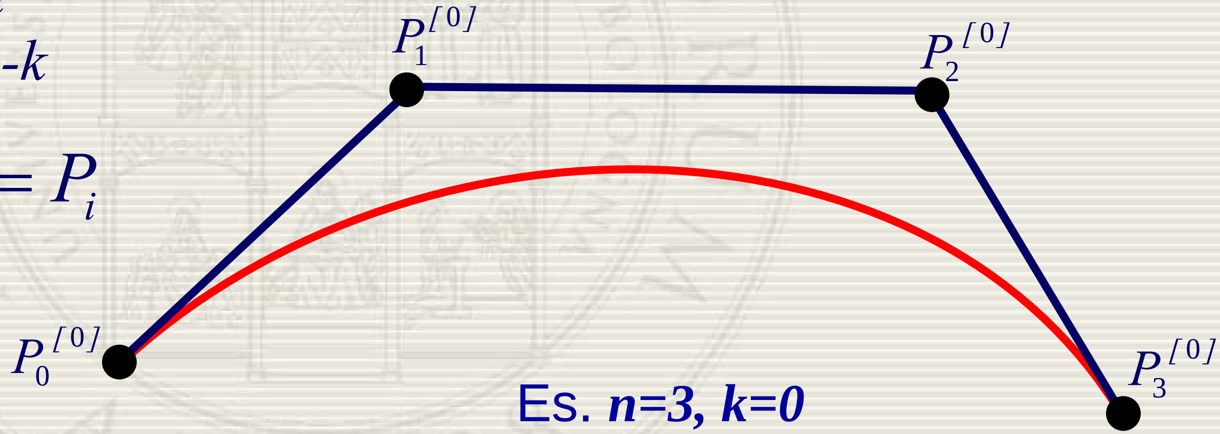
Le Curve di Bézier e de Casteljau

Il matematico francese de Casteljau, negli anni '60, diede una definizione di curva di Bézier basata su "corner cutting" successivi:

$$P_i^{[k]}(t) = (1-t)P_i^{[k-1]}(t) + tP_{i+1}^{[k-1]}(t) \qquad t \in [0,1]$$

dove $k = 1,..,n$
$i = 0,..,n\text{-}k$

con $P_i^{[0]}(t) = P_i$
$i = 0,..,n$



Nota: P_i sono i punti di controllo iniziali della curva di Bezier

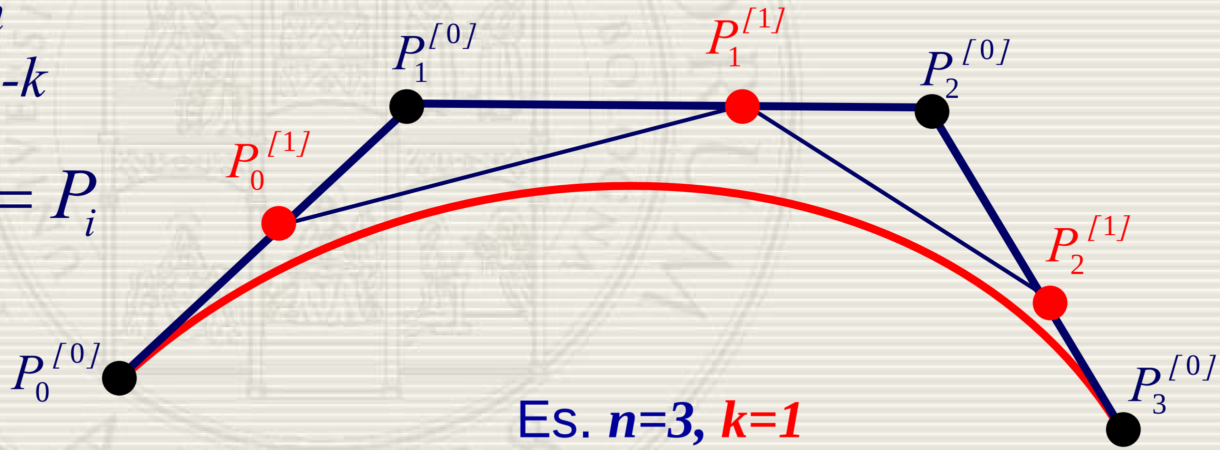
Le Curve di Bézier e de Casteljau

Il matematico francese de Casteljau, negli anni ’60, diede una definizione di curva di Bézier basata su “corner cutting” successivi:

$$P_i^{[k]}(t) = (1-t)P_i^{[k-1]}(t) + tP_{i+1}^{[k-1]}(t) \qquad t \in [0,1]$$

dove $k = 1,..,n$
$i = 0,..,n\text{-}k$

con $P_i^{[0]}(t) = P_i$
$i = 0,..,n$



Le Curve di Bézier e de Casteljau

Il matematico francese de Casteljau, negli anni '60, diede una definizione di curva di Bézier basata su "corner cutting" successivi:

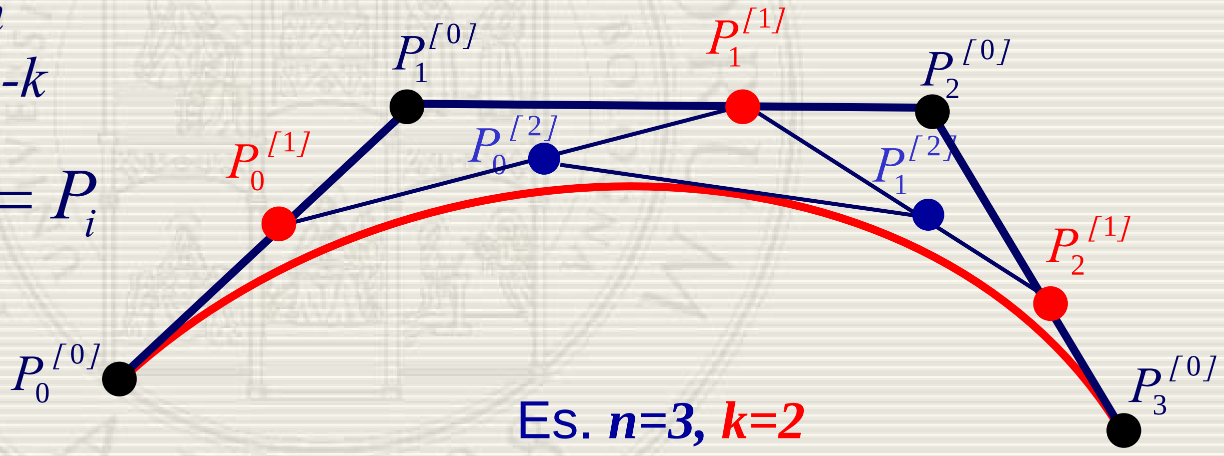
$$P_i^{[k]}(t) = (1-t)P_i^{[k-1]}(t) + tP_{i+1}^{[k-1]}(t) \qquad t \in [0,1]$$

dove $k = 1,..,n$

$i = 0,..,n\text{-}k$

con $P_i^{[0]}(t) = P_i$

$i = 0,..,n$



Le Curve di Bézier e de Casteljau

Il matematico francese de Casteljau, negli anni '60, diede una definizione di curva di Bézier basata su "corner cutting" successivi:

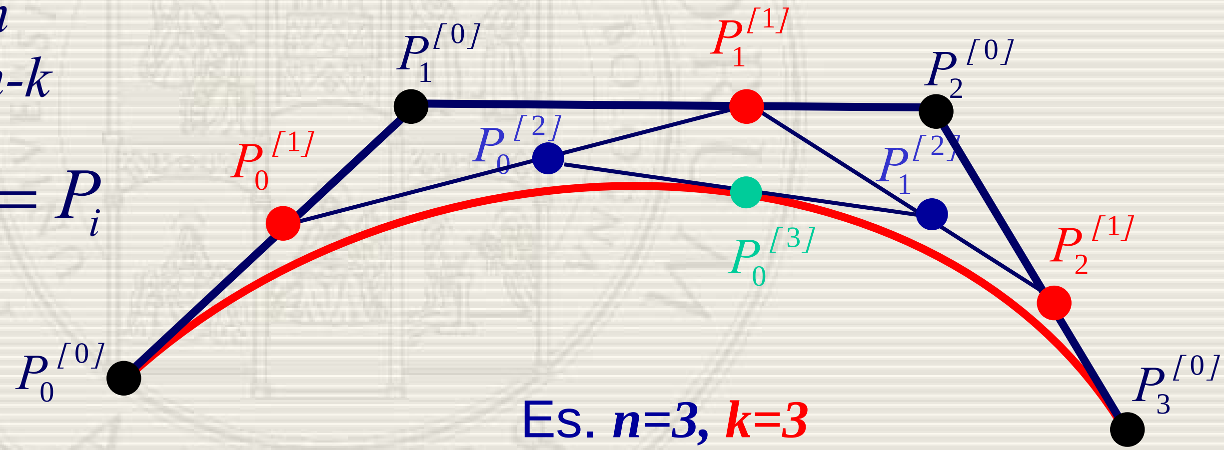
$$P_i^{[k]}(t) = (1-t)P_i^{[k-1]}(t) + tP_{i+1}^{[k-1]}(t) \qquad t \in [0,1]$$

dove $k = 1,..,n$

$i = 0,..,n\text{-}k$

con $P_i^{[0]}(t) = P_i$

$i = 0,..,n$



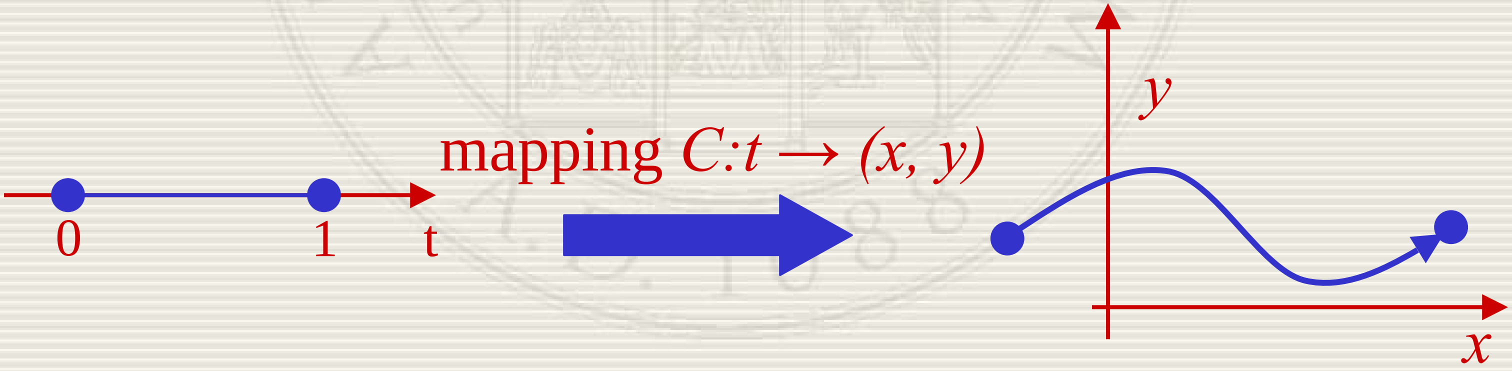
Questa definizione è anche un algoritmo numericamente stabile per il calcolo delle curve di Bézier.

Che cosa sono le curve di Bézier?

- Definiamo uno spazio parametrico
  - 1D per curve

- Definiamo un mapping fra lo spazio dei parametri e punti 2D
  - Una funzione che prende valori parametrici e restituisce punti 2D

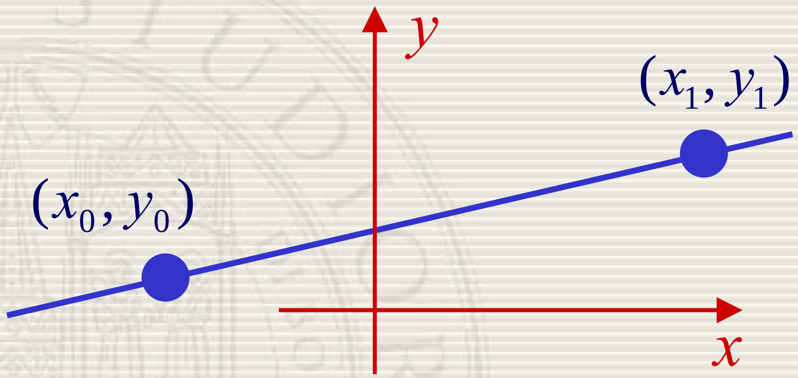
- Il risultato è una curva in forma parametrica



Curve Parametriche

- Abbiamo già visto una retta in forma parametrica

$$\begin{cases} x = x_0 t + (1-t)x_1 \\ y = y_0 t + (1-t)y_1 \end{cases}$$



- Si noti che x e y sono determinati ciascuno da una equazione che coinvolge:
  - il parametro t
  - certi punti $(x_{0,} y_{0,})$ e $(x_{1,} y_{1,})$ specificati dall'utente e detti punti di controllo
- Questo è un esempio di curva 2D in forma parametrica
- le singole componenti della curva sono polinomi espressi mediante le seguenti funzioni base { t , $(1-t)$ }

Curve di Bezier (1)

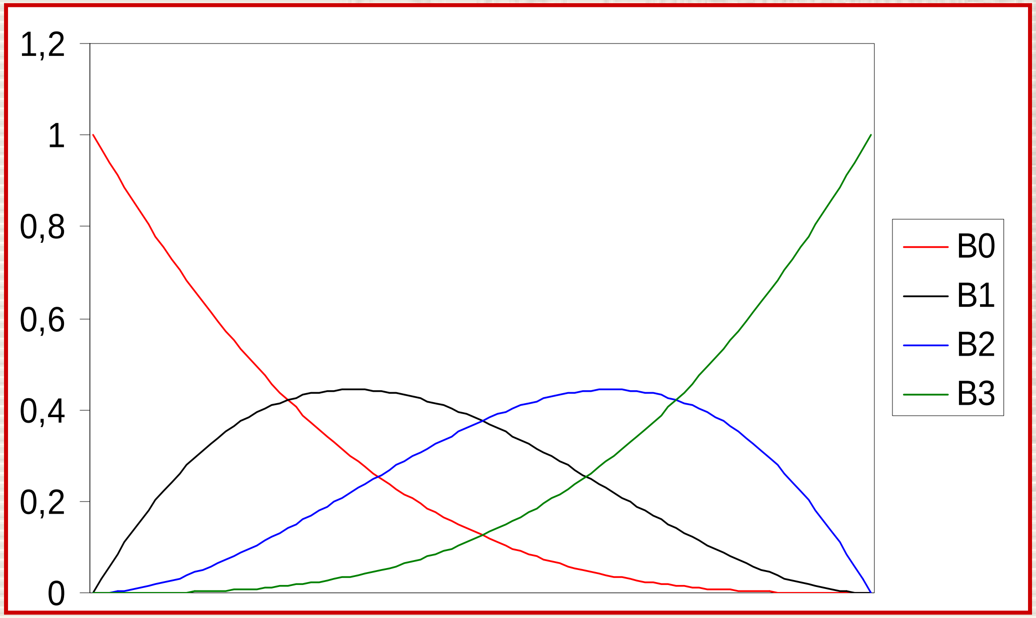
- Differenti scelte di funzioni base permettono di rappresentare una stessa curva in modi differenti
  - La scelta di una base di funzioni è importante sia per questioni numeriche e computazionali, ma soprattutto perché i punti di controllo (i coefficienti) siano informativi sulla forma della curva
- Per questi motivi la scelta cade sulla base polinomiale nota come base di Bernstein;
- L'utente dà *d+1* punti di controllo $\mathbf{p}_i$ (poligono di controllo)
- La curva si scrive come:

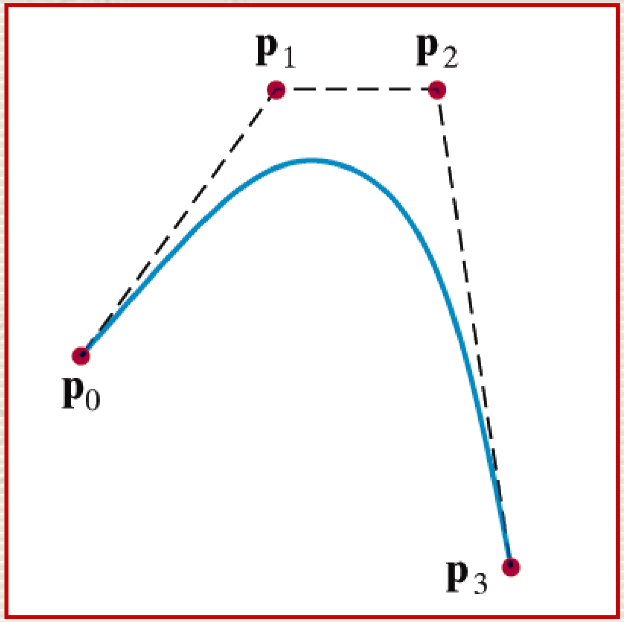
$$C(t) = \sum_{i=0}^{d} \mathbf{p}_i B_{i,d}(t) \qquad B_{i,d}(t) = \binom{d}{i} t^i (1-t)^{d-i}$$

- Le funzioni $B_{i,d}$ sono i *polinomi base di Bernstein* di grado d e sono dette *blending functions*; sono non negative e la loro somma vale 1.

Curve di Bezier (1)

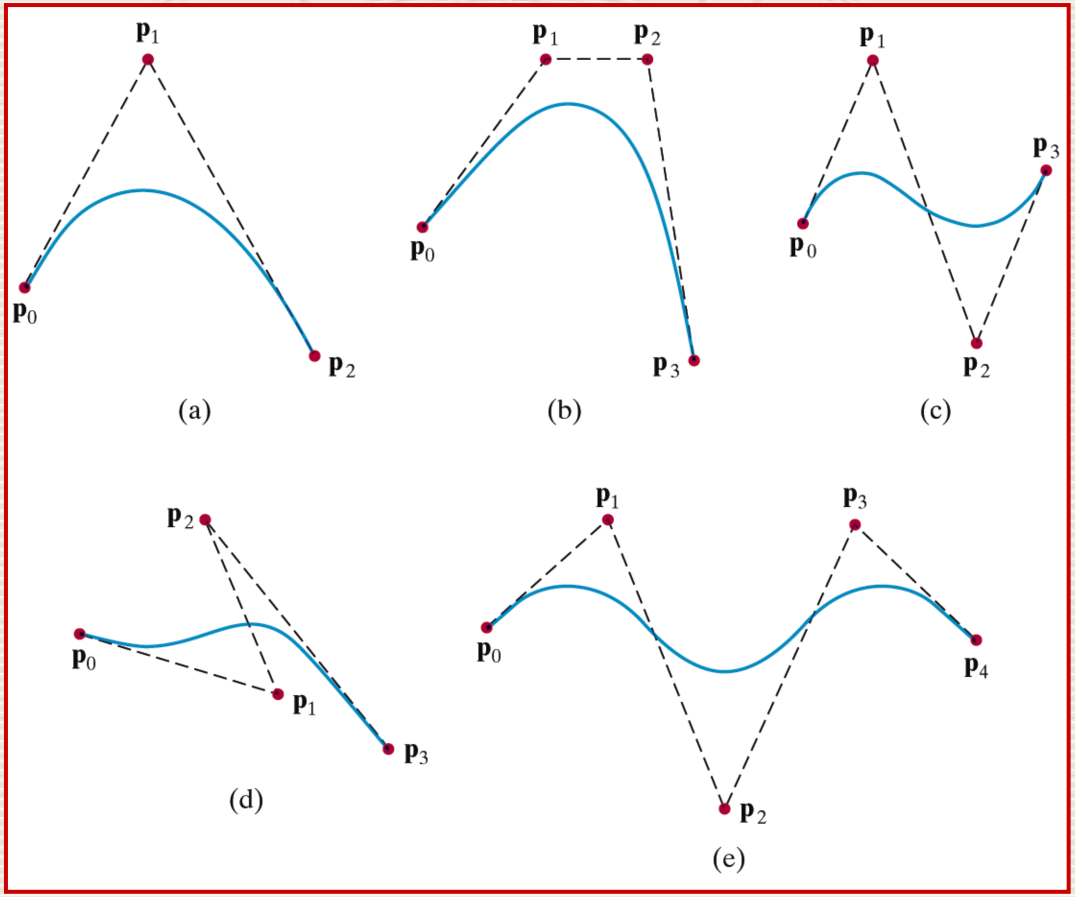
- Nel caso cubico ($d=3$) ci saranno 4 punti di controllo; i due punti ($\mathbf{p}_0$ e $\mathbf{p}_3$) definiscono gli estremi della curva e i due punti ($\mathbf{p}_1$ e $\mathbf{p}_2$) le tangenti (direzioni) alla curva nei punti estremi.
- La curva così definita è nota come curva di Bézier cubica.





Esempi di Curve di Bézier

Curve di Bézier di differente grado.



Le Curve di Bézier

Proprietà:

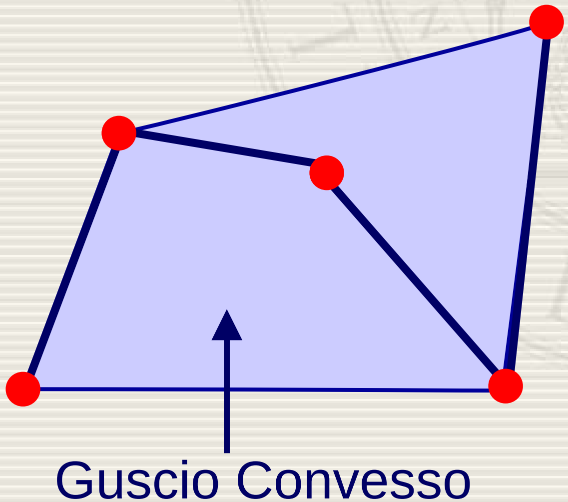
1. $P(t) \quad t \in [0,1]$ giace nel guscio convesso definito dai suoi punti di controllo;

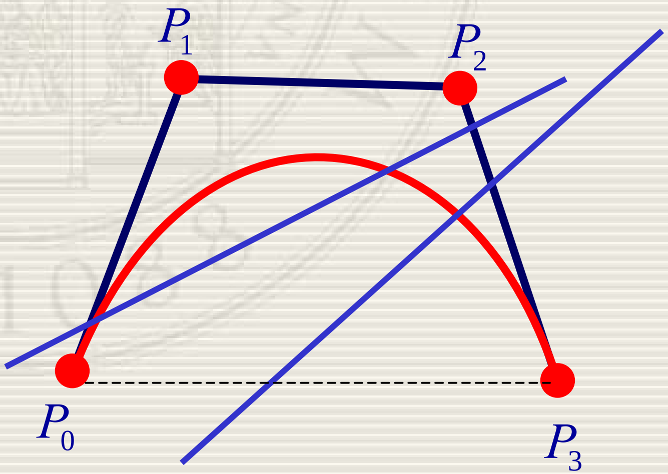
2. $P(0)=P_0$ e $P(1)=P_d$;

3. $P'(0)=d(P_1-P_0)$ e $P'(1)=d(P_d-P_{d-1})$;

4. $P(t)$ è invariante per trasformazioni affini; in particolare per traslazione, scala, rotazione e deformazione lineare (shear);

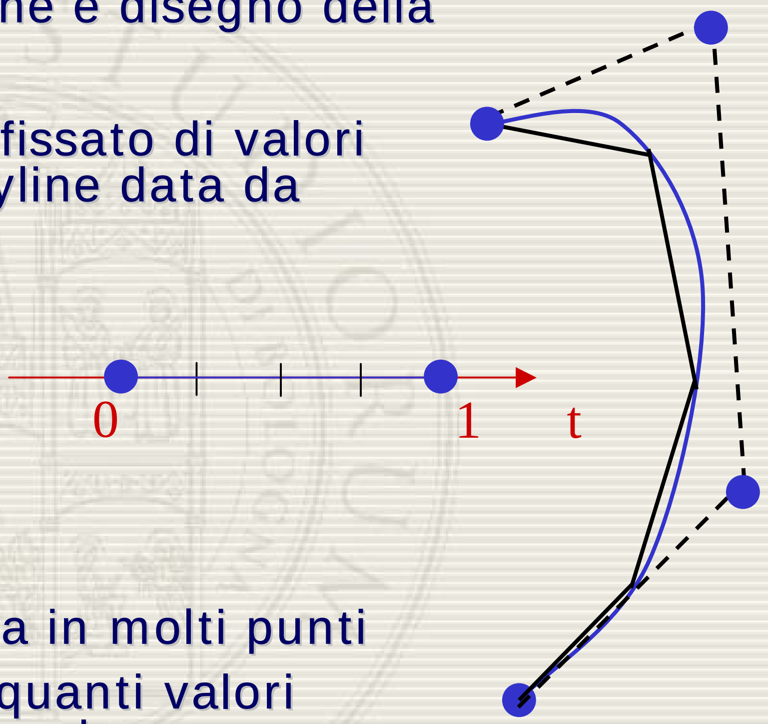
5. $P(t)$ è approssimante in forma della poligonale di controllo.





Rendering Curve di Bezier (1)

- Le curve possono essere disegnate in molti modi
  - Per esempio mediante tassellazione e disegno della polyline risultante
    - Si valuti la curva in un numero fissato di valori parametrici e si definisca la polyline data da questi punti della curva
    - Vantaggi:
      - molto semplice
    - Svantaggi:
      - Costoso per valutare la curva in molti punti
      - Non è facile determinare in quanti valori valutare e dove valutare lungo la curva
      - Non è facile renderlo adattivo; in particolare è difficile misurare la distanza della polyline dalla curva.

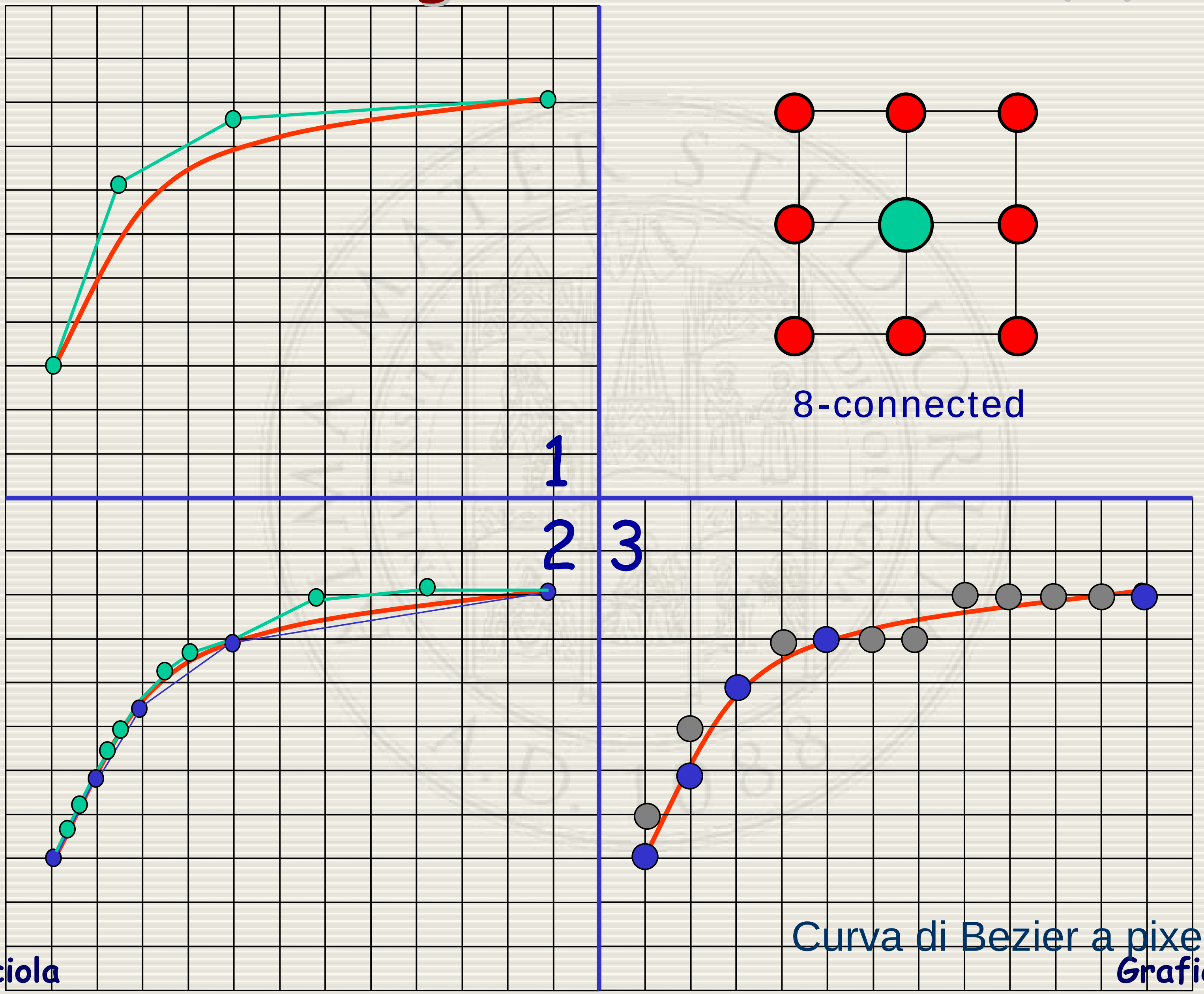


Rendering Curve di Bezier (2)

- Ricordiamo che una curva di Bézier giace interamente nel guscio convesso dei suoi punti di controllo
- Se i punti di controllo sono quasi allineati, allora il guscio convesso è quasi un segmento che approssima bene la curva
- Ancora, una curva di Bézier può essere divisa in due curve di Bézier più piccole che rappresentano esattamente la curva originale

- Questo suggerisce il seguente algoritmo di disegno:
  - dividi ricorsivamente la curva in due sotto-curve (algoritmo di suddivisione)
  - ferma il processo quando i punti di controllo di ogni sotto-curva sono quasi allineati
  - disegna il segmento di estremi il primo ed ultimo punto di controllo

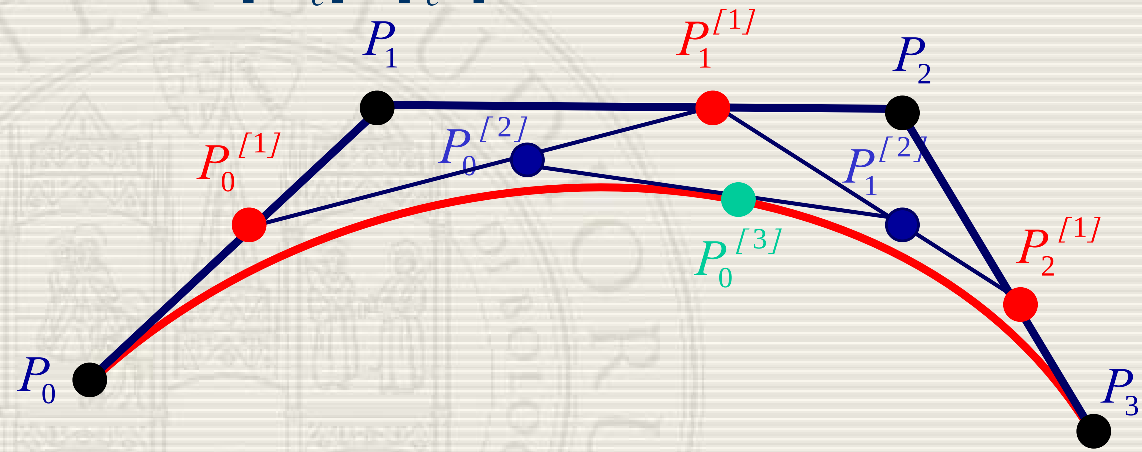
Rendering Curve di Bezier (3)



Le Curve di Bézier e la Suddivisione

La definizione o algoritmo di valutazione di de Casteljau di una curva di Bézier fornisce anche i punti di controllo delle curve di Bézier corrispondenti agli intervalli $[0,t_c]$ e $[t_c,1]$.

Vediamolo nel caso $d=3$:



$$
\begin{array}{cccccccc}
P_0^0 & & P_1^0 & & P_2^0 & & P_3^0 \\
\downarrow & \swarrow & \downarrow & \swarrow & \downarrow & \swarrow & \\
P_0^1 & & P_1^1 & & P_2^1 & & \\
\downarrow & \swarrow & \downarrow & \swarrow & & & \\
P_0^2 & & P_1^2 & & & & \\
\downarrow & \swarrow & & & & & \\
P_0^3 & & & & & &
\end{array}
$$

$$\boxed{(1-t)\downarrow \quad \swarrow\ t}$$

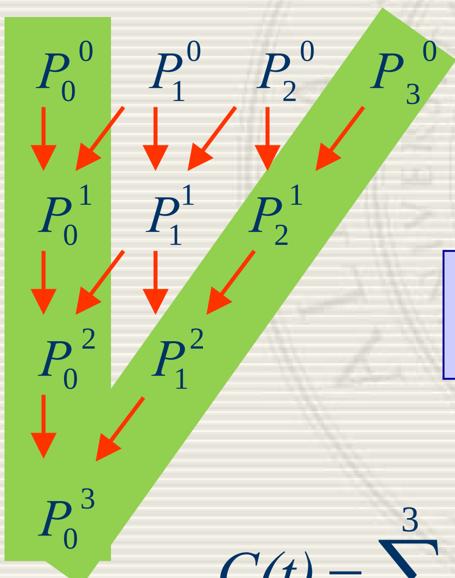
$$C(t)=\sum_{i=0}^{3} P_i^0 B_{i,3}(t)=\begin{cases}\displaystyle\sum_{i=0}^{3} P_0^i B_{i,3}(t) & t\in[0,t_c] \\ \displaystyle\sum_{i=0}^{3} P_i^{3-i} B_{i,3}(t) & t\in[t_c,1]\end{cases}$$

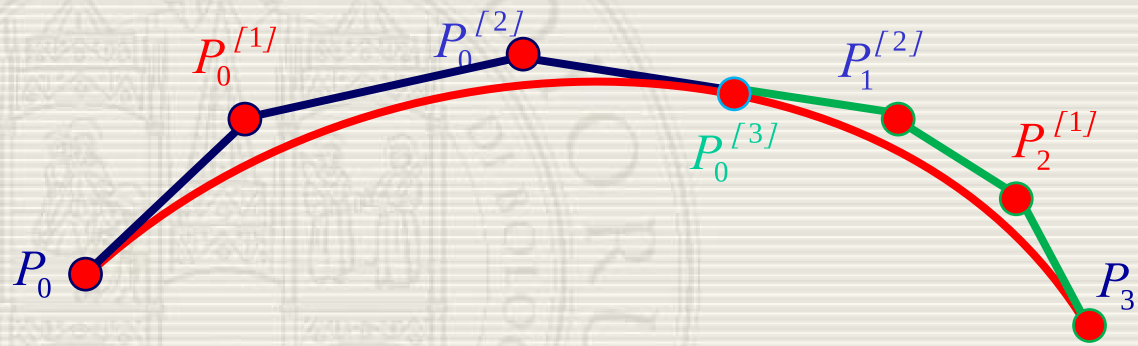
$t\in[0,1]$

Le Curve di Bézier e la Suddivisione

La definizione o algoritmo di valutazione di de Casteljau di una curva di Bézier fornisce anche i punti di controllo delle curve di Bézier corrispondenti agli intervalli $[0,t_c]$ e $[t_c,1]$.

Vediamolo nel caso *d=3*:





$(1\text{-}t)\downarrow \quad \swarrow t$

$$C(t) = \sum_{i=0}^{3} P_i^0 B_{i,3}(t) = \begin{cases} \displaystyle\sum_{i=0}^{3} P_0^i B_{i,3}(t) & t \in [0,t_c] \\ \displaystyle\sum_{i=0}^{3} P_i^{3-i} B_{i,3}(t) & t \in [t_c,1] \end{cases}$$

$t \in [0,1]$