Programmazione Grafica 2d

Sommario

Abbiamo visto l’ambiente Hardware/Software che permette di fare grafica, i primi elementi su HTML5, canvas e contesto ‘2d’.
Adesso facciamo alcuni esempi di programmazione grafica 2d.

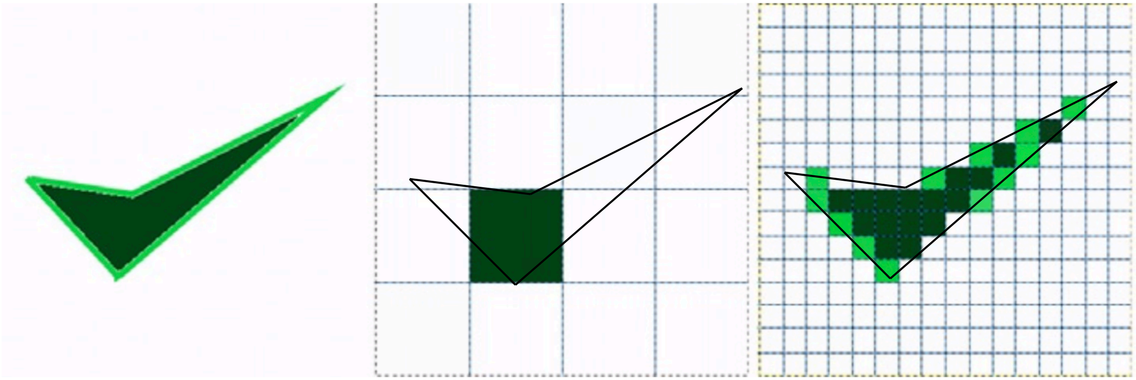
Vedremo:

1. algoritmo di linea incrementale (per disegno in coordinate intere/schermo di un segmento su viewport)
2. algoritmo di linea di Bresenham (per disegno di un segmento su viewport usando aritmetica intera)
3. disegno in coordinate floating point (su “window”)
4. facciamo un Esercizio insieme
5. Esercizio lasciato da fare da soli

Immagini e Pixel

Quando rappresentiamo un’immagine digitale sullo schermo di un computer, essa è composta da una griglia rettangolare di punti colorati, noti come pixel. Questo tipo di rappresentazione è nota come *raster graphic* o *bitmap*. Quanto l’immagine visualizzata appaia fedele all'immagine originale dipende dal numero di pixel sullo schermo: la **risoluzione**.

In figura si vede, a sinistra un’immagine, al centro la sua rappresentazione su una griglia 4x4 di pixel e a destra su una griglia 16x16. All’aumentare della risoluzione la differenza fra l’immagine originale e il rendering dell’immagine diminuisce.



Disegno di Punti e Linee

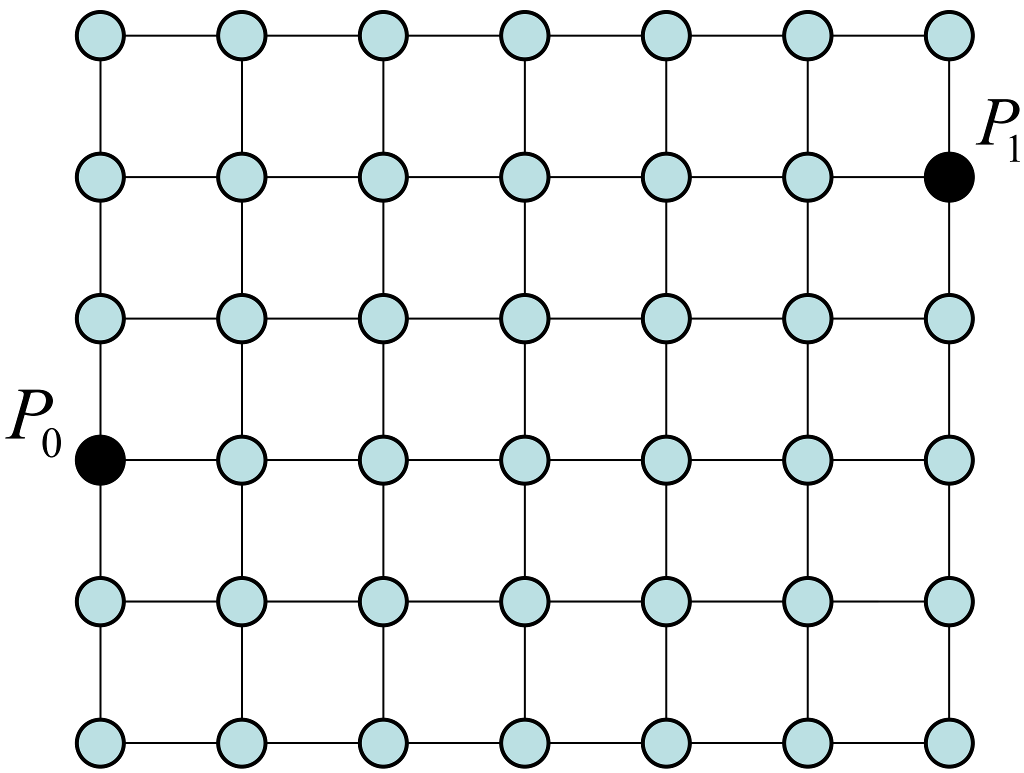
Abbiamo già visto che non viene esposta una primitiva punto cioè il disegno di un singolo pixel, mentre esiste la primitiva linea;

```
var canvas = document.getElementById("myCanvas");
var ctx = canvas.getContext("2d");
ctx.moveTo(0, 0);
ctx.lineTo(200, 100);
ctx.stroke();
```

Vediamo cosa c’è dietro alla primitiva linea, cioè al disegno di una linea o segmento di retta.

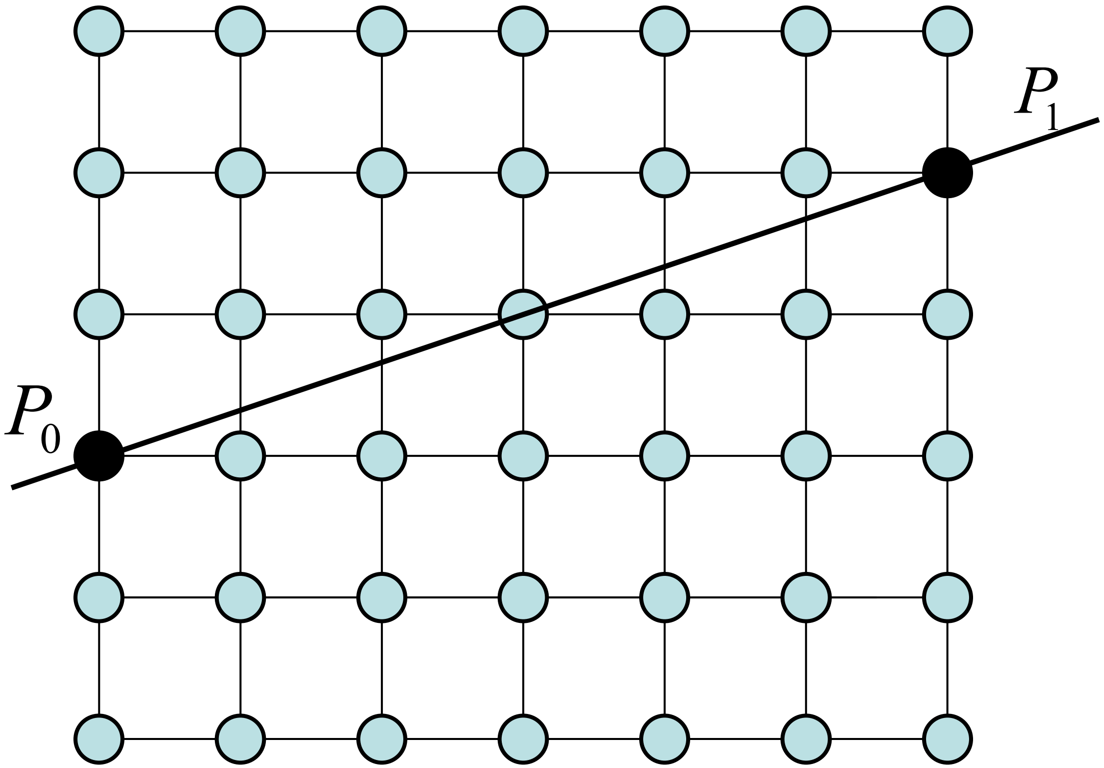
Disegno di Linee

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



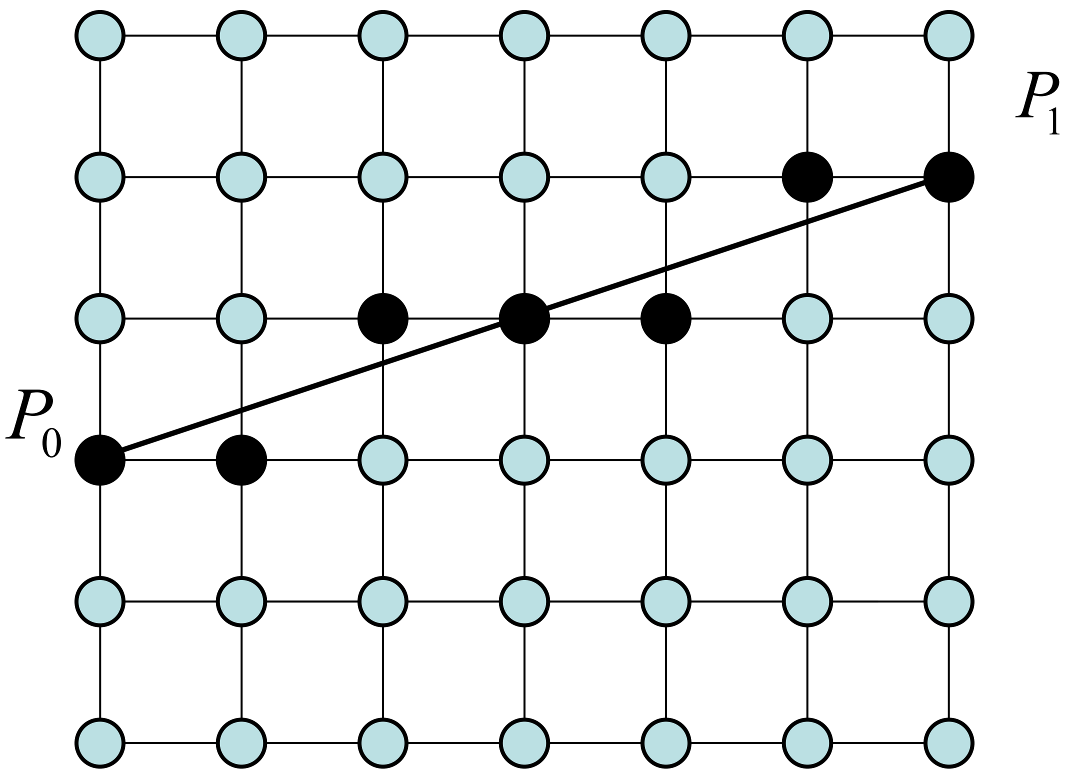
Disegno di Linee

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



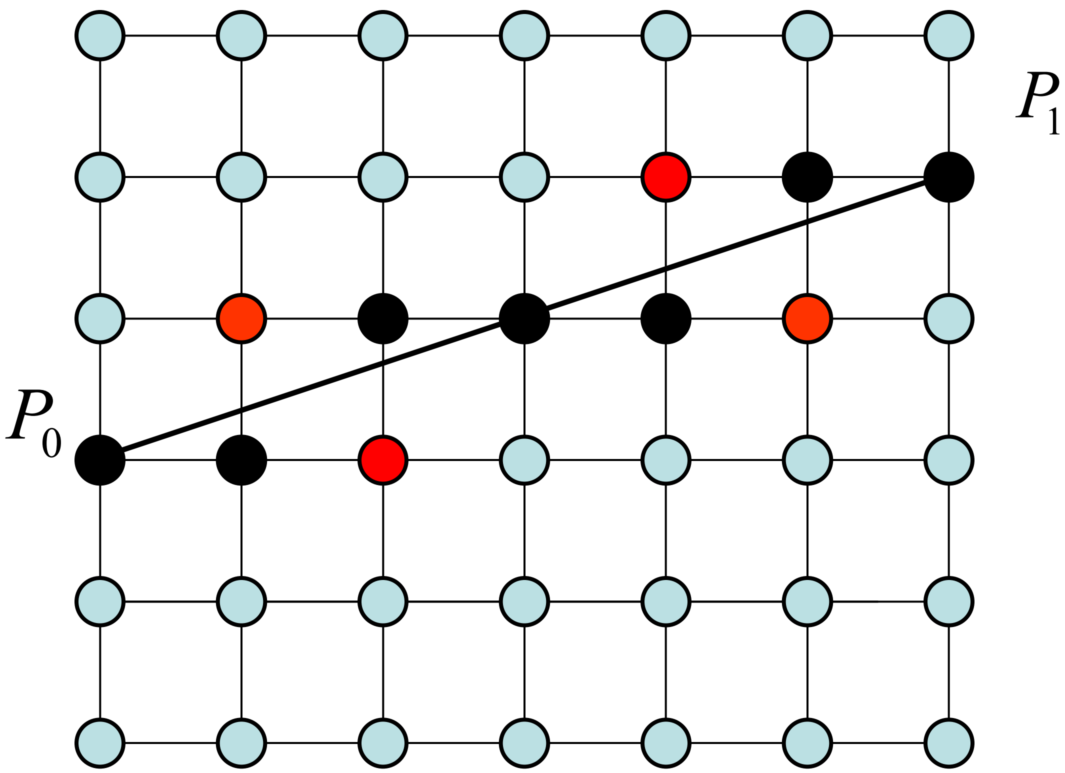
Disegno di Linee

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



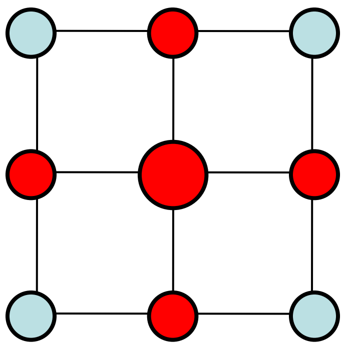
Disegno di Linee

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

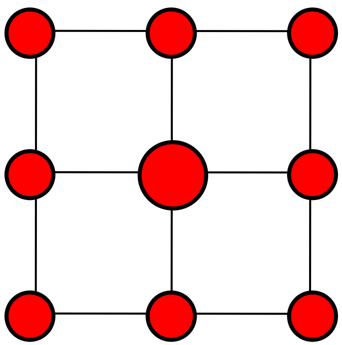


Disegno Continuo a Pixel

La logica è di procedere ad “accendere” pixel “adiacenti” per simulare un disegno continuo.
Questo porta alla definizione di pixel adiacenti; ci sono due differenti modi o definizioni:

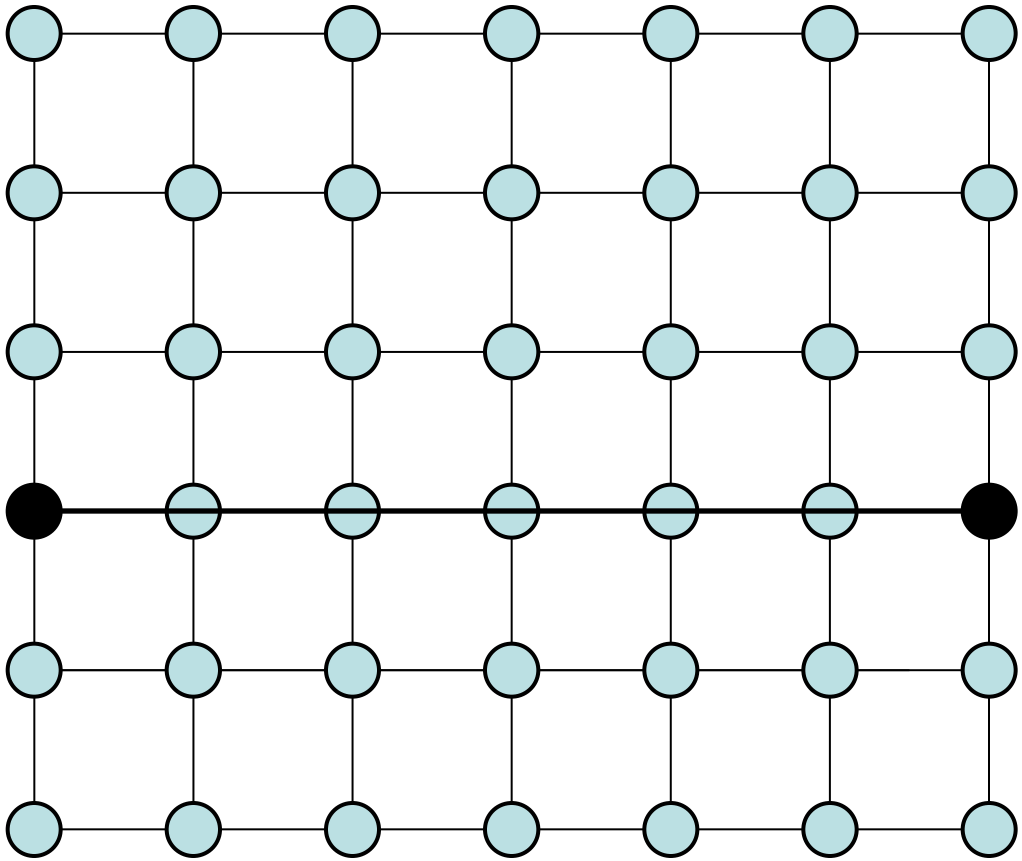


4-connected

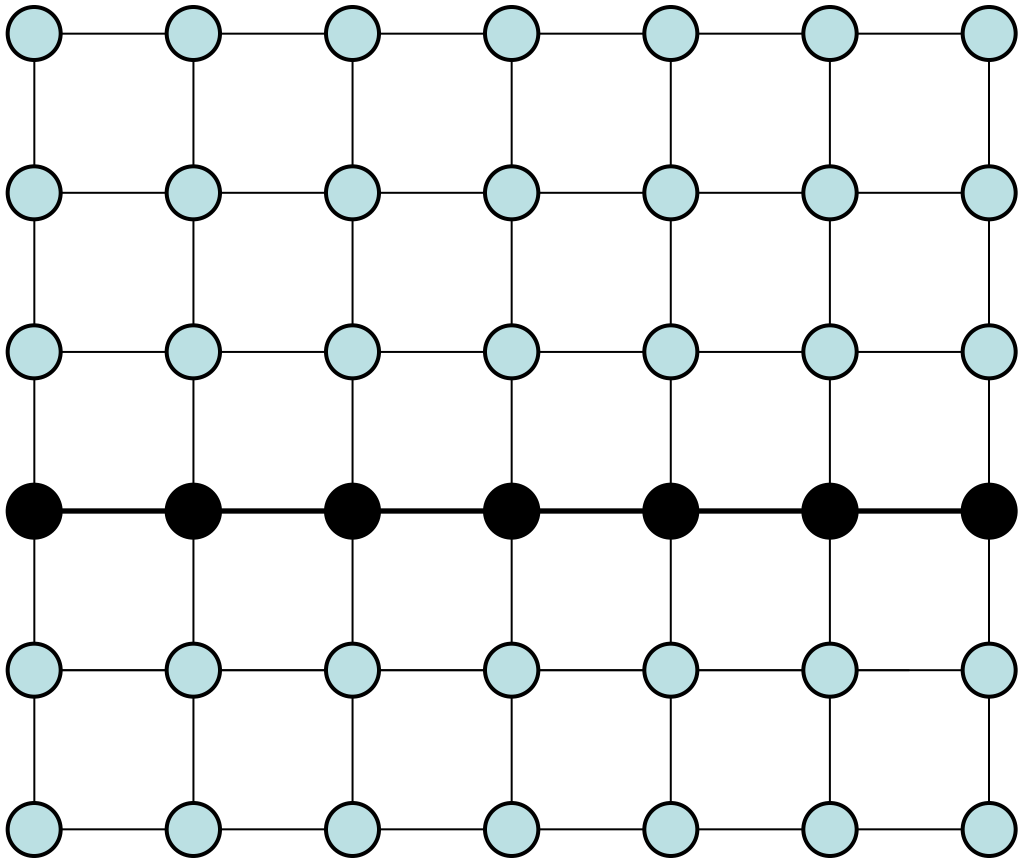


8-connected

Linee Speciali - Orizzontale

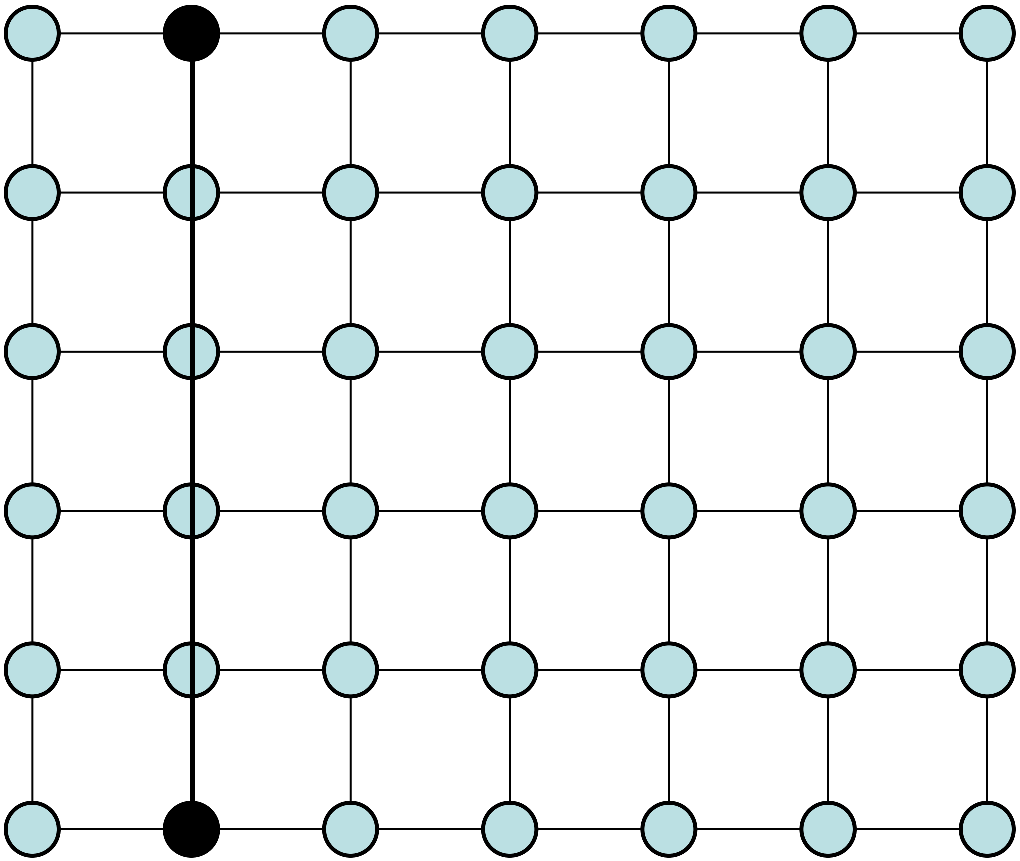


Linee Speciali - Orizzontale

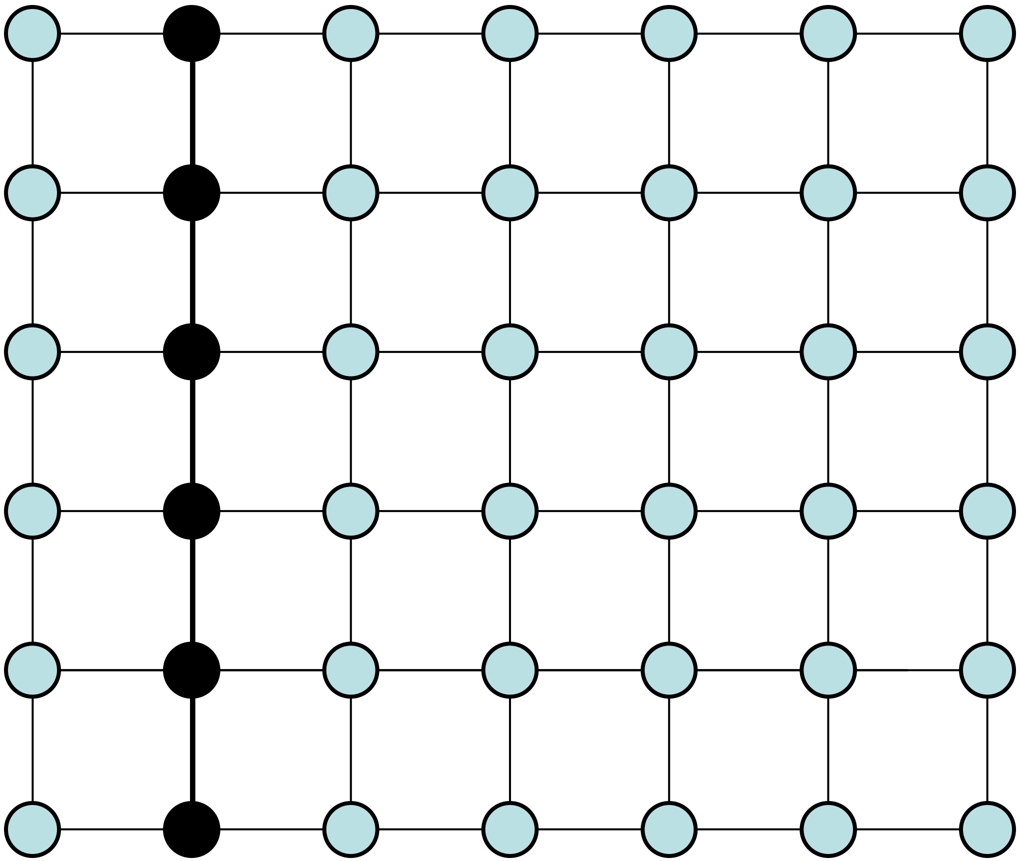


Incrementa la *x* di 1, tenendo la *y* costante

Linee Speciali - Verticale

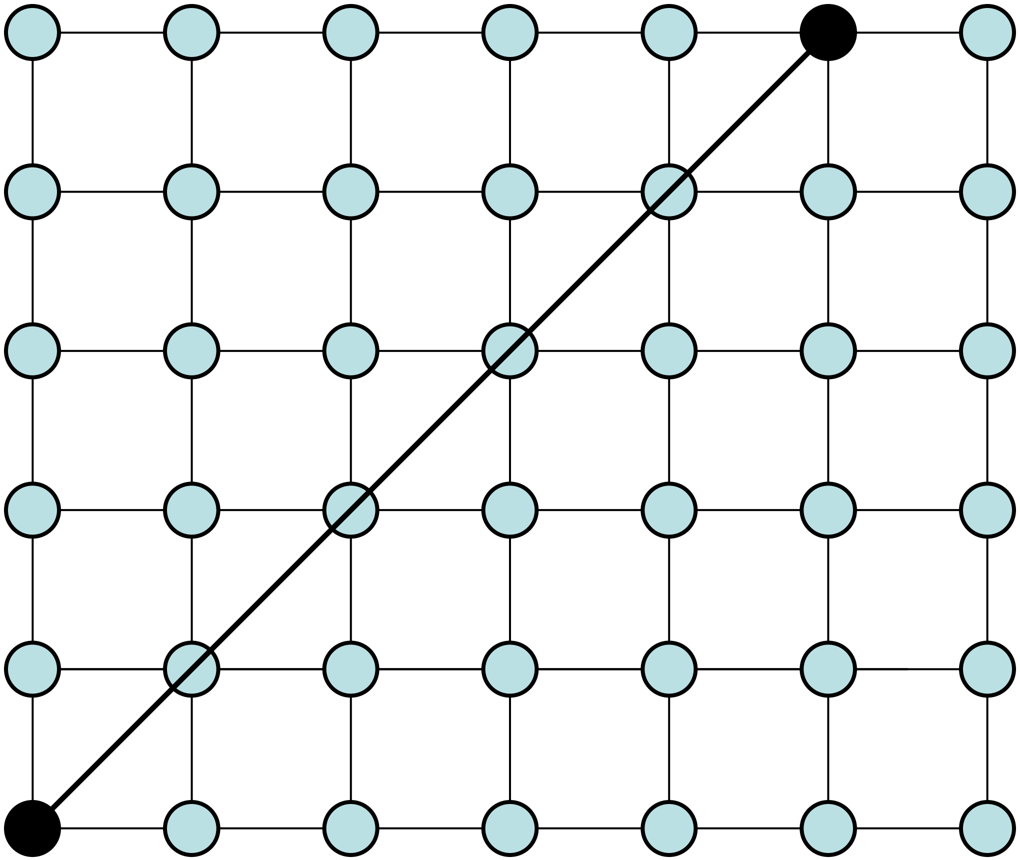


Linee Speciali - Verticale

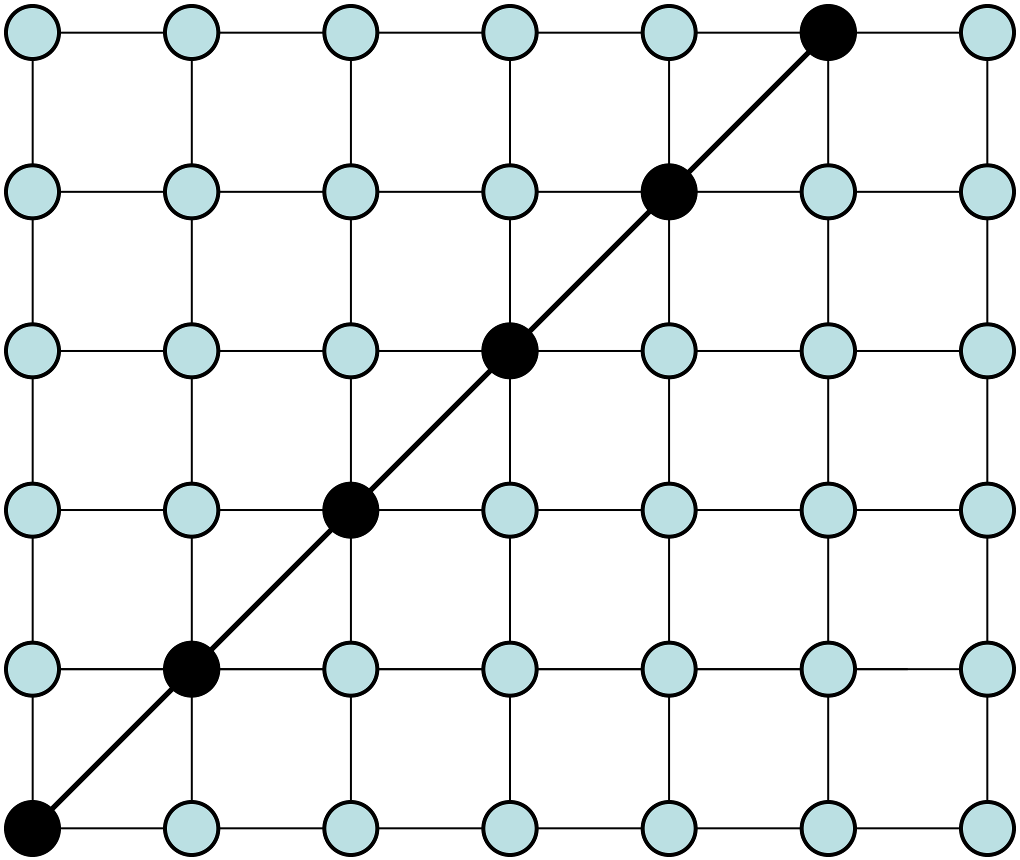


Tieni la *x* constante e incrementa la *y* di 1

Linee Speciali - Diagonale

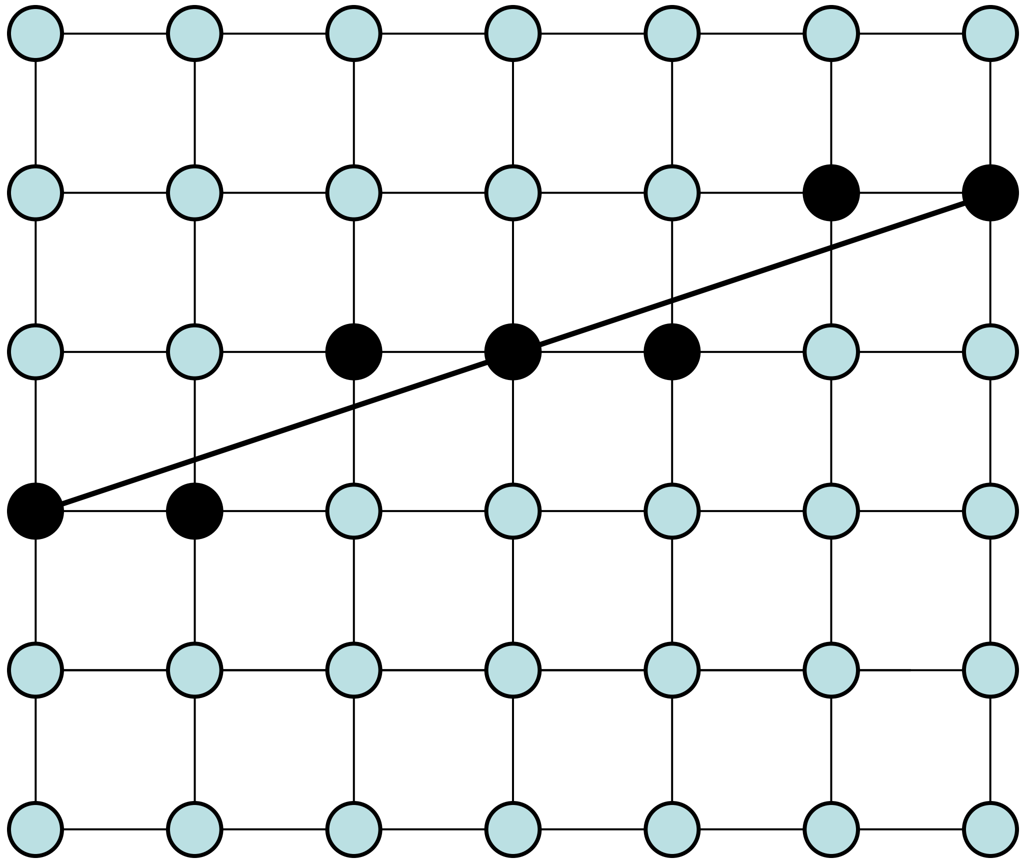


Linee Speciali - Diagonale



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

Ma 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
   setPixel(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 vista, infatti per le ascisse si ha:

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

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

Algoritmo:

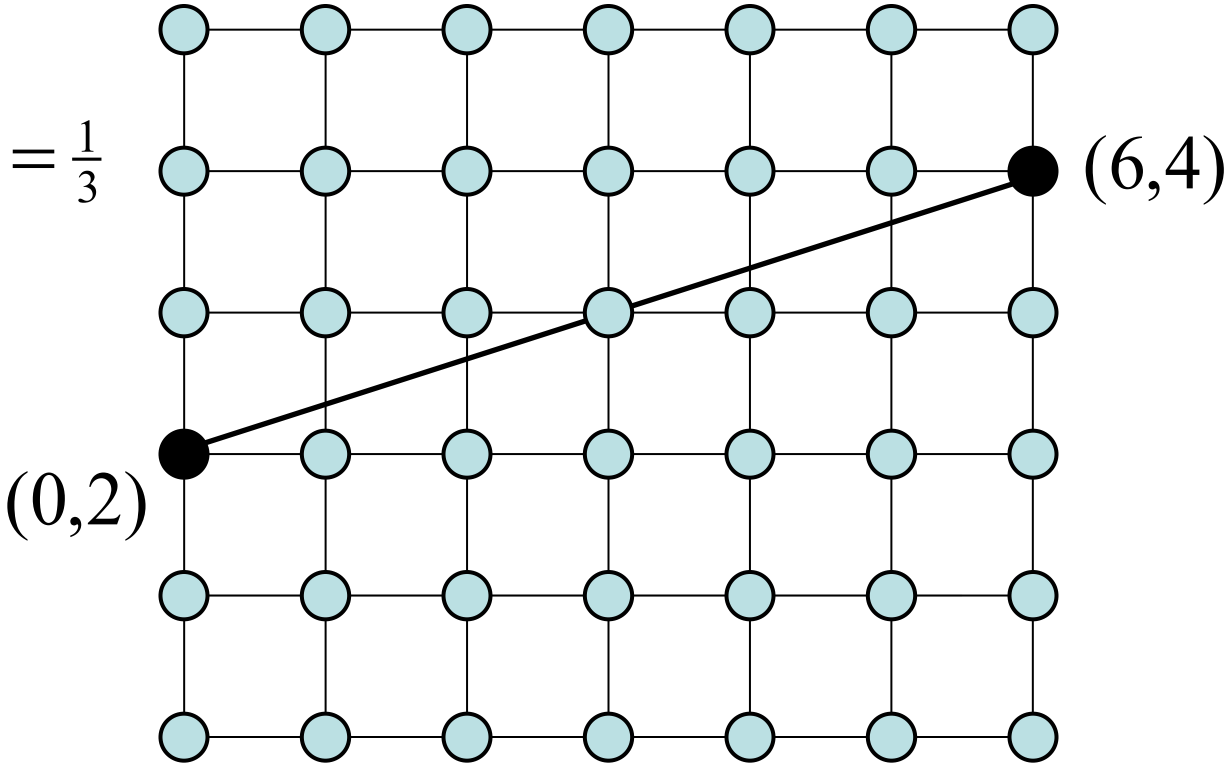
```
n=max(abs(x1-x0),abs(y1-y0))
dx=(x1-x0)/n
dy=(y1-y0)/n
x=x0
y=y0
setPixel(x,y,col)
for (i=1; i<=n; i++) {
   x=x+dx
   y=y+dy
   setPixel(round(x), round(y), col) }
```

Algoritmo di Linea Incrementale: esempio

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

sarà: $n=6,\ dx=1$

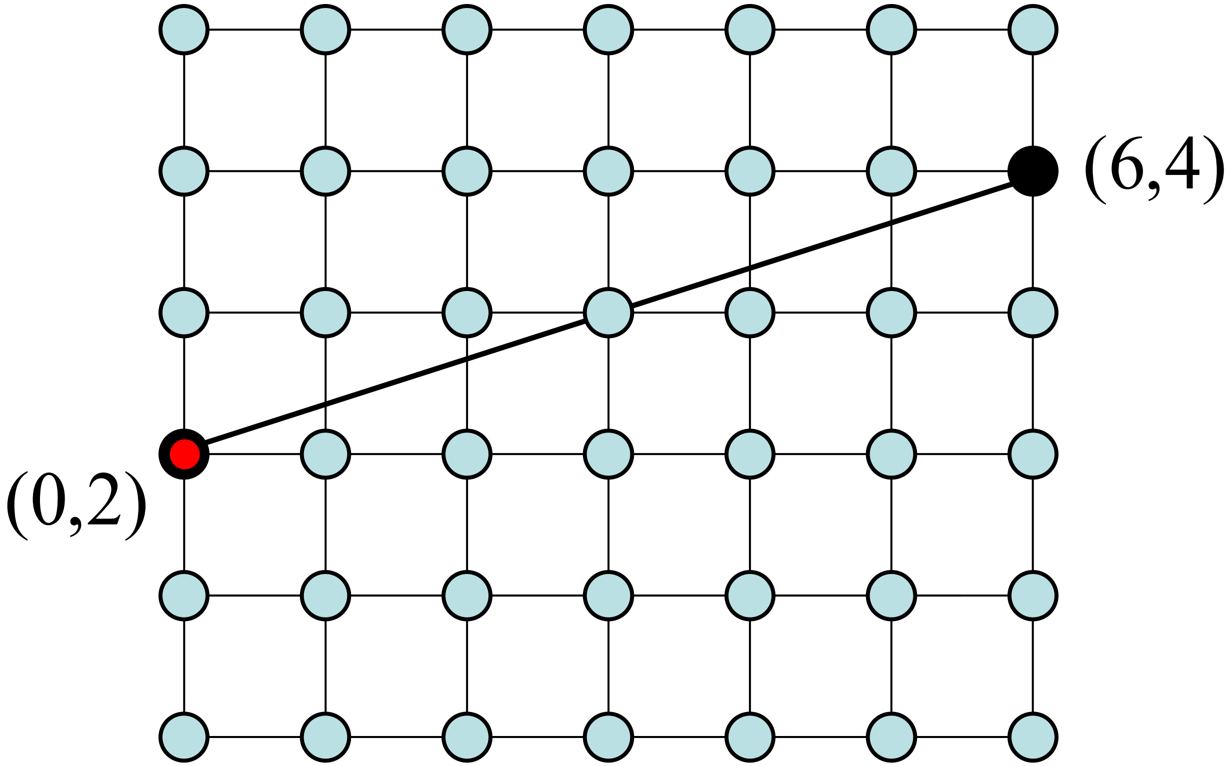
$$dy = \frac{y_1 - y_0}{x_1 - x_0} = \frac{4-2}{6-0} = \frac{1}{3}$$



Algoritmo di Linea Incrementale: esempio

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

--> (0,2)

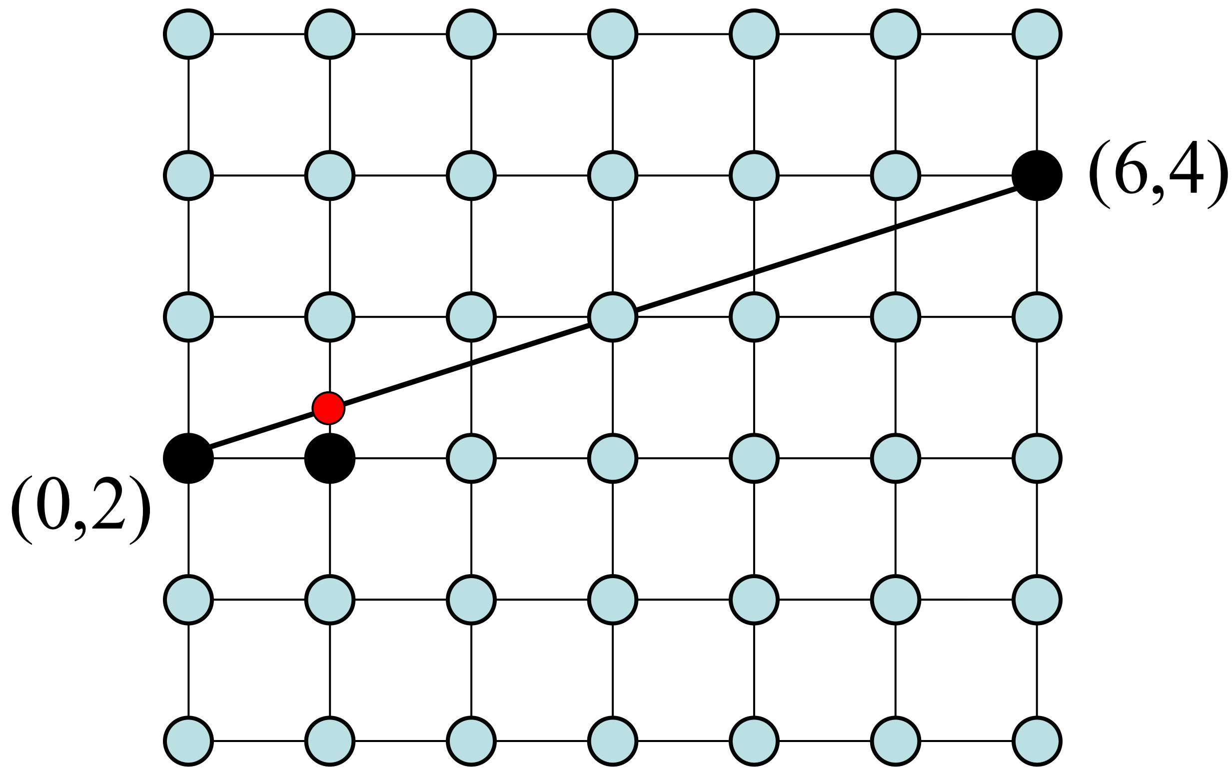


Algoritmo di Linea Incrementale: esempio

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

--> (0,2)

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



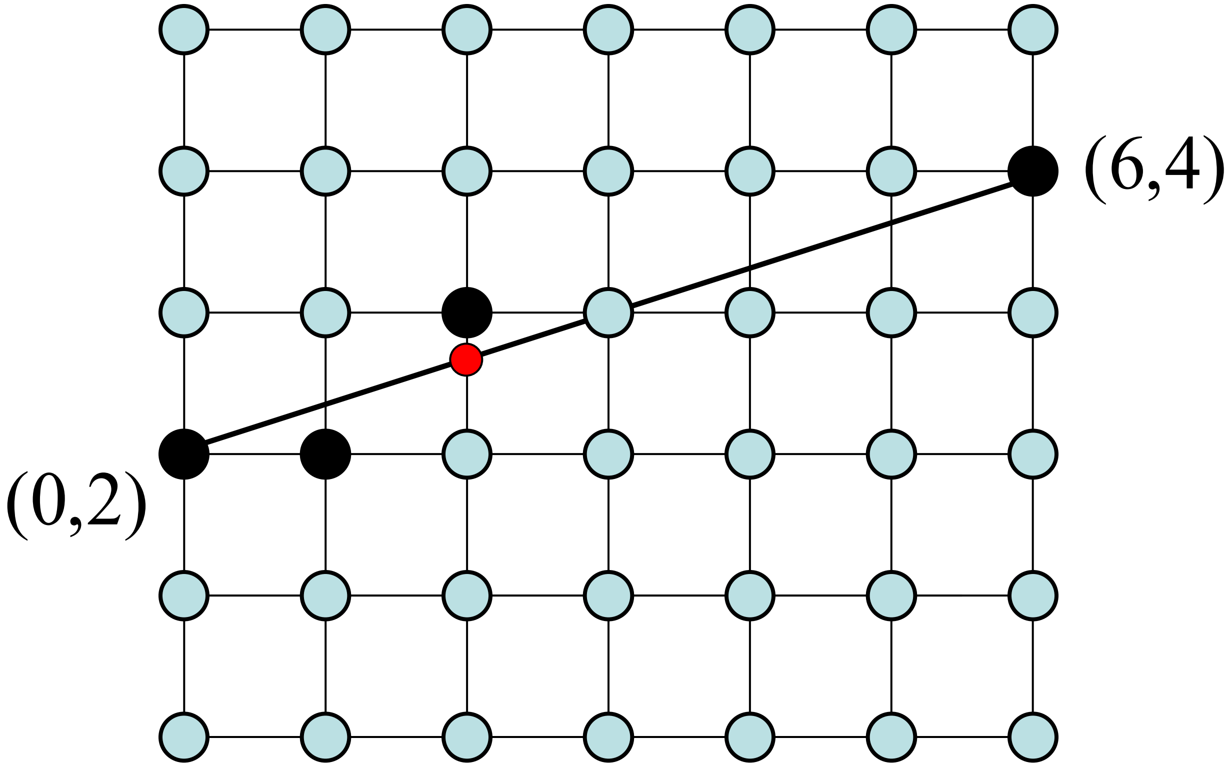
Algoritmo di Linea Incrementale: esempio

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

--> (0,2)

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

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



Algoritmo di Linea Incrementale: esempio

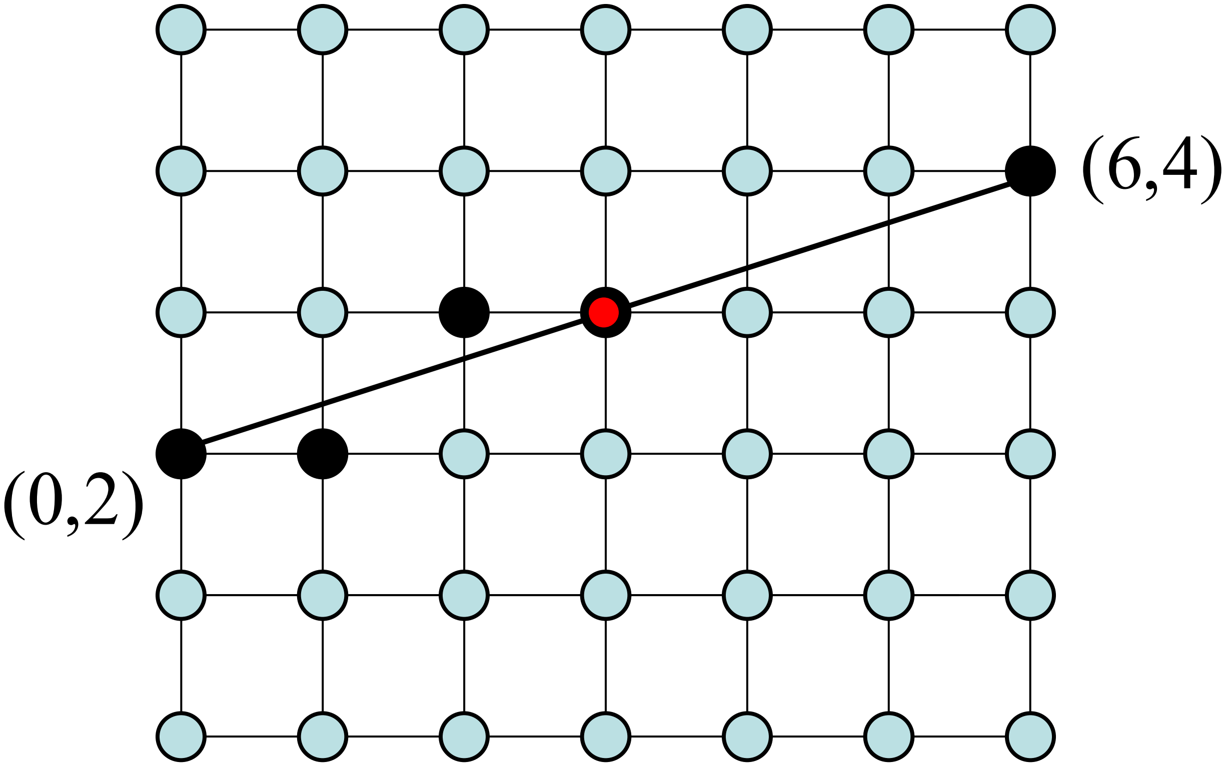
$P_0 = (0\ ,\ 2)\quad P_1 = (6\ ,\ 4)\qquad n=6,\ dx=1,\ dy=1/3 \cong 0.333$

--> (0,2)

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

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

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



Algoritmo di Linea Incrementale: esempio

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

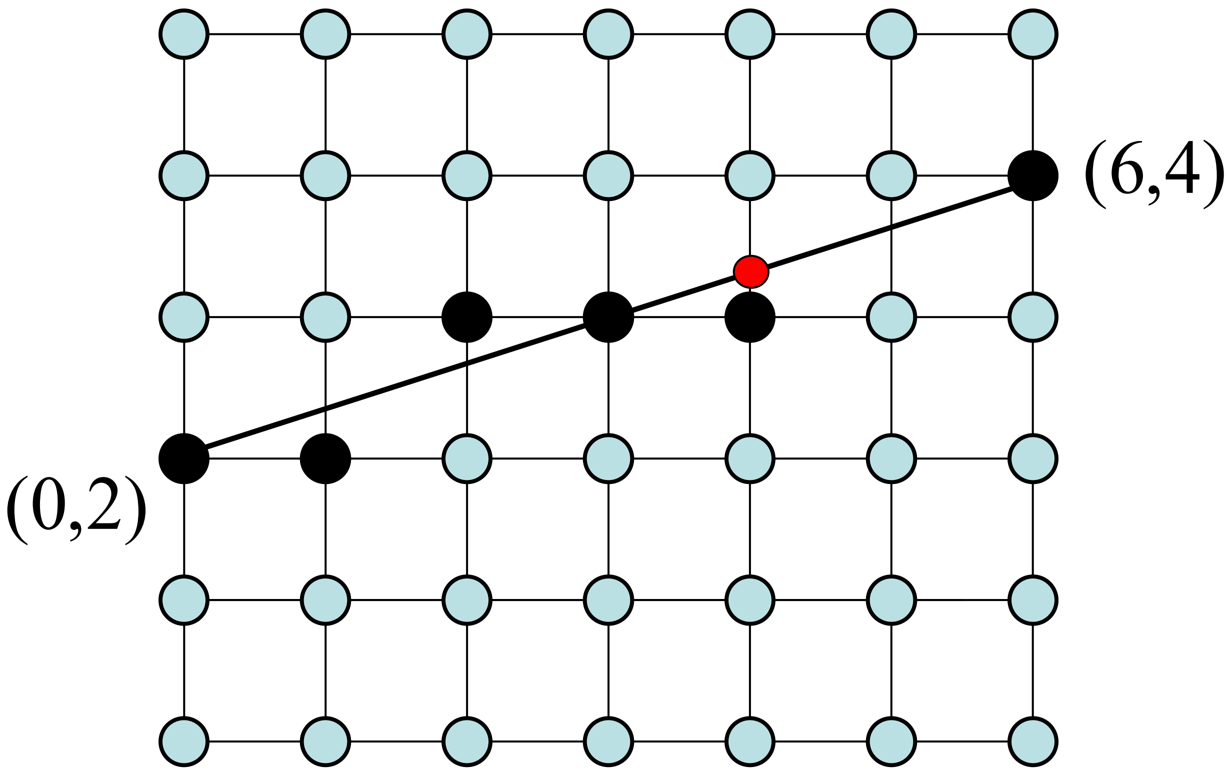
--> (0,2)

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

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

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

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



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)

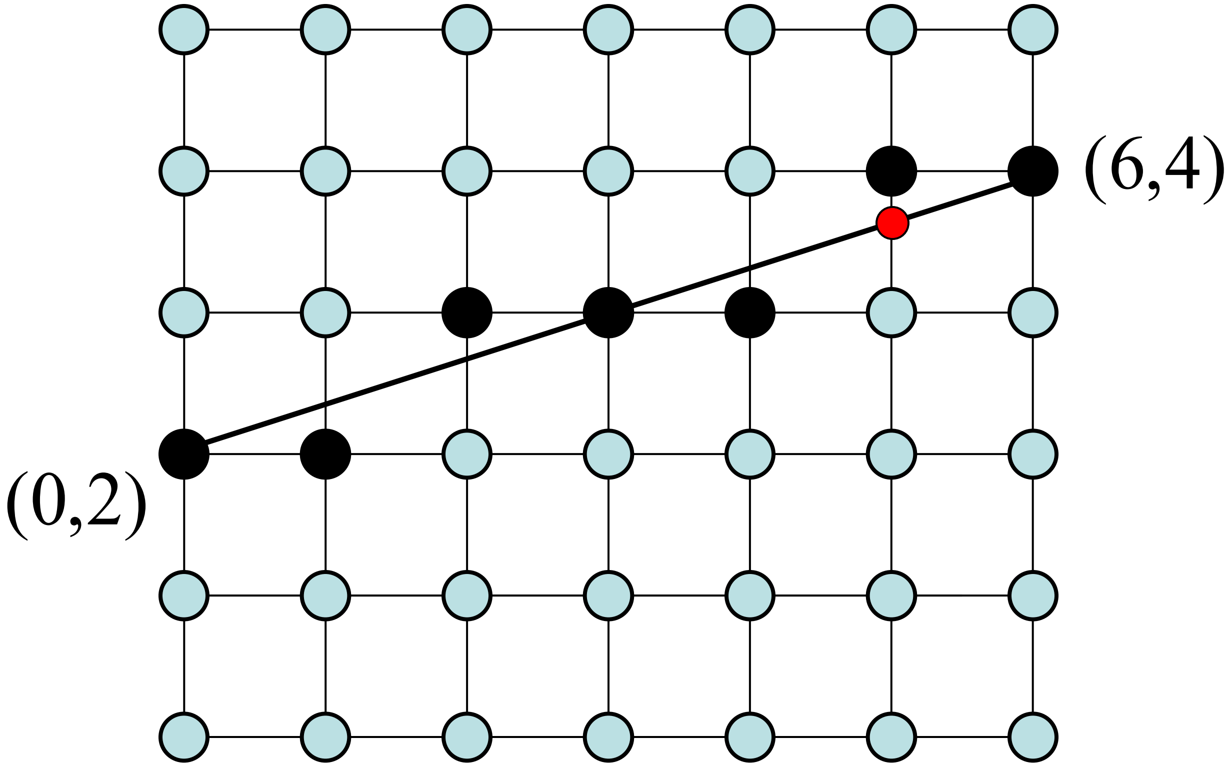
(1,2.333) --> (1,2)

(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

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

--> (0,2)

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

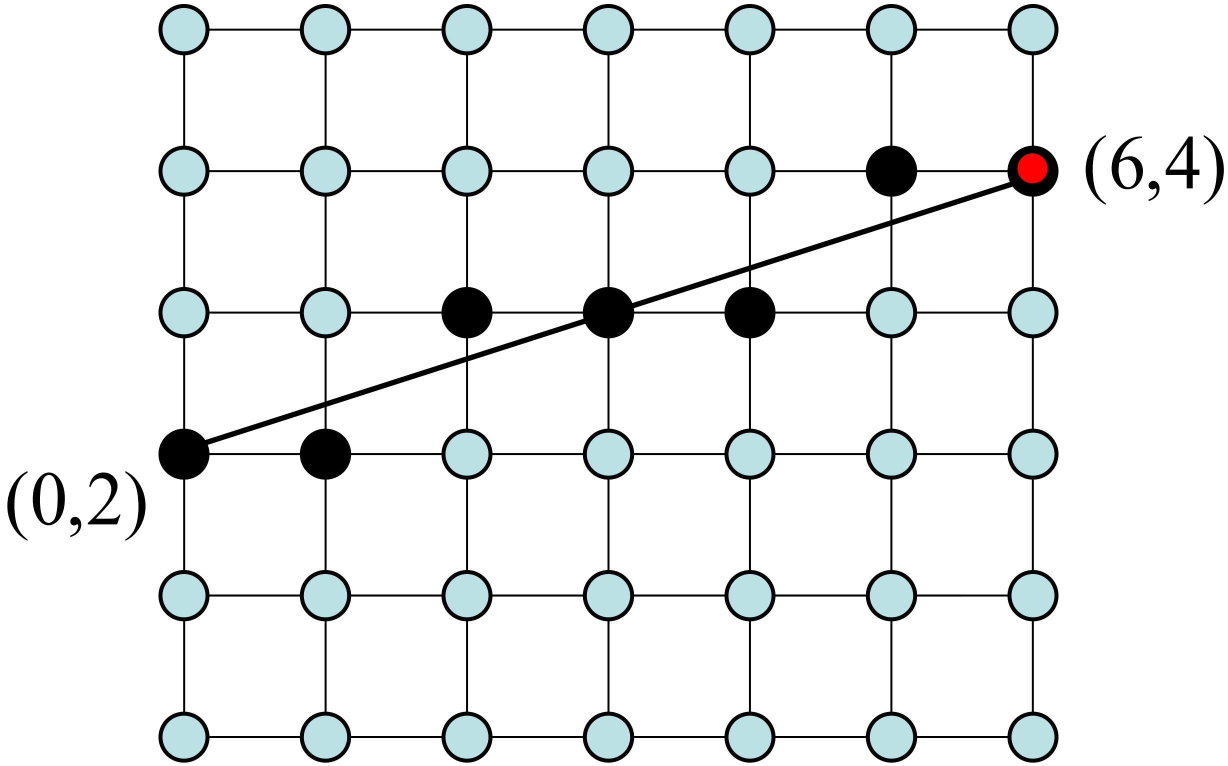
(2,2.666) --> (2,3)

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

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

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

(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
- Ne trovate una implementazione software nella cartella HTML5_2d_1 file draw_line.js utilizzato dallo script polygon_pixel.html

Problema: disegno in coord. float

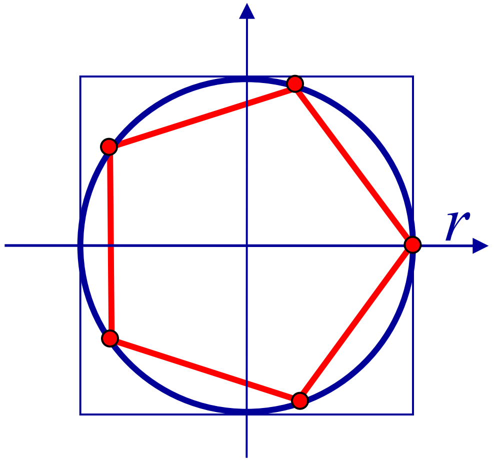
Disegnare un poligono regolare di *n* lati.

Dati: *n* numero di lati (o vertici)

r raggio circonferenza circoscritta

Risultato: disegno a pixel del poligono

Metodo: si usa l'equazione parametrica della circonferenza di centro l'origine e raggio unitario e si determinano *n* punti equidistanti su di essa.



Algoritmo:

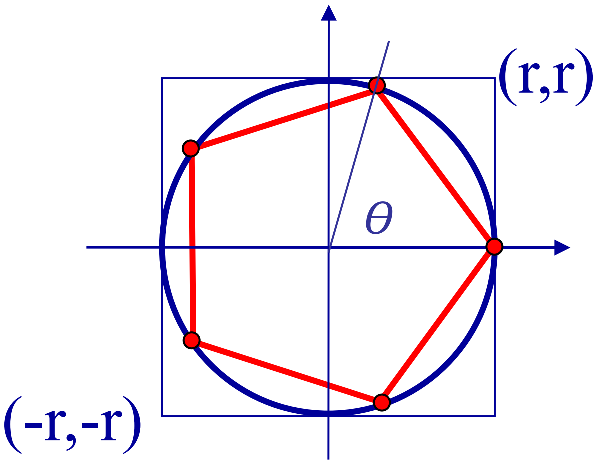
```
theta = 6.28/n
for (i = 0; i<=n; i++)
{
   t = i * theta
   x[ i ] = r * cos( t )
   y[ i ] = r * sin( t )
}
```

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

Problema: disegno in coord. float

Per ottimizzare:

```
theta = 6.28/n
c=cos(theta)
s=sin(theta)
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
}
```



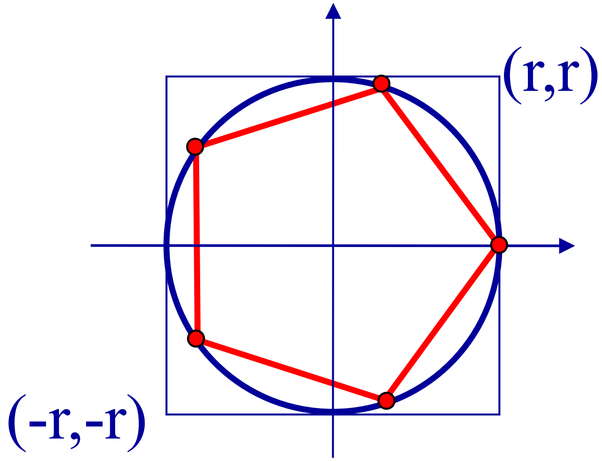
$$\begin{bmatrix} p'_x \\ p'_y \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \bullet \begin{bmatrix} p_x \\ p_y \end{bmatrix}$$

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.

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

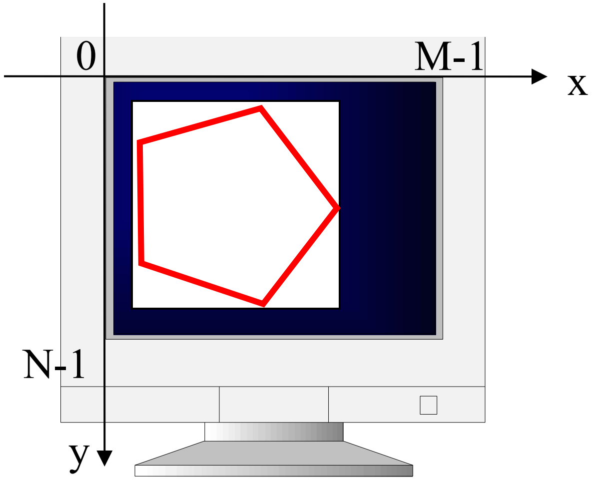
Definizioni di Window e Viewport

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

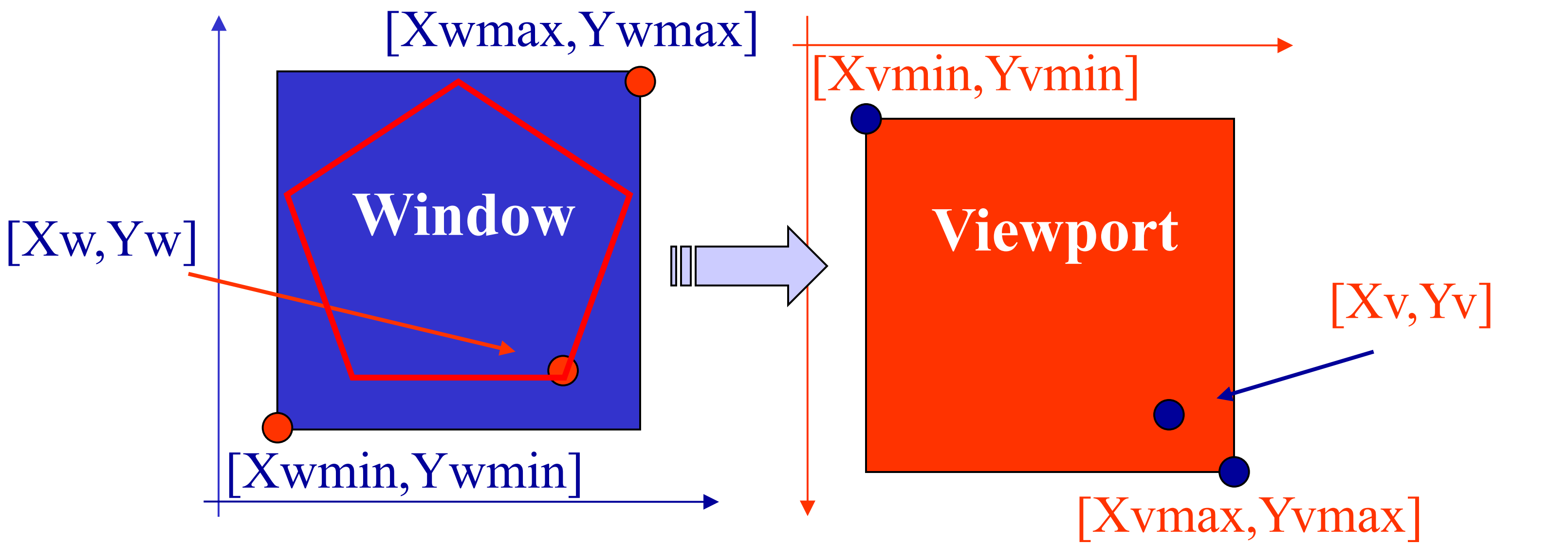


Chiameremo **Viewport** un'area rettangolare dello schermo su cui vogliamo rappresentare il disegno.

Problema: dovremo trasformare le coordinate floating point (Window) in coordinate intere (Viewport)



Trasformazione Window-Viewport



Gestiamo separatamente i due assi coordinati:

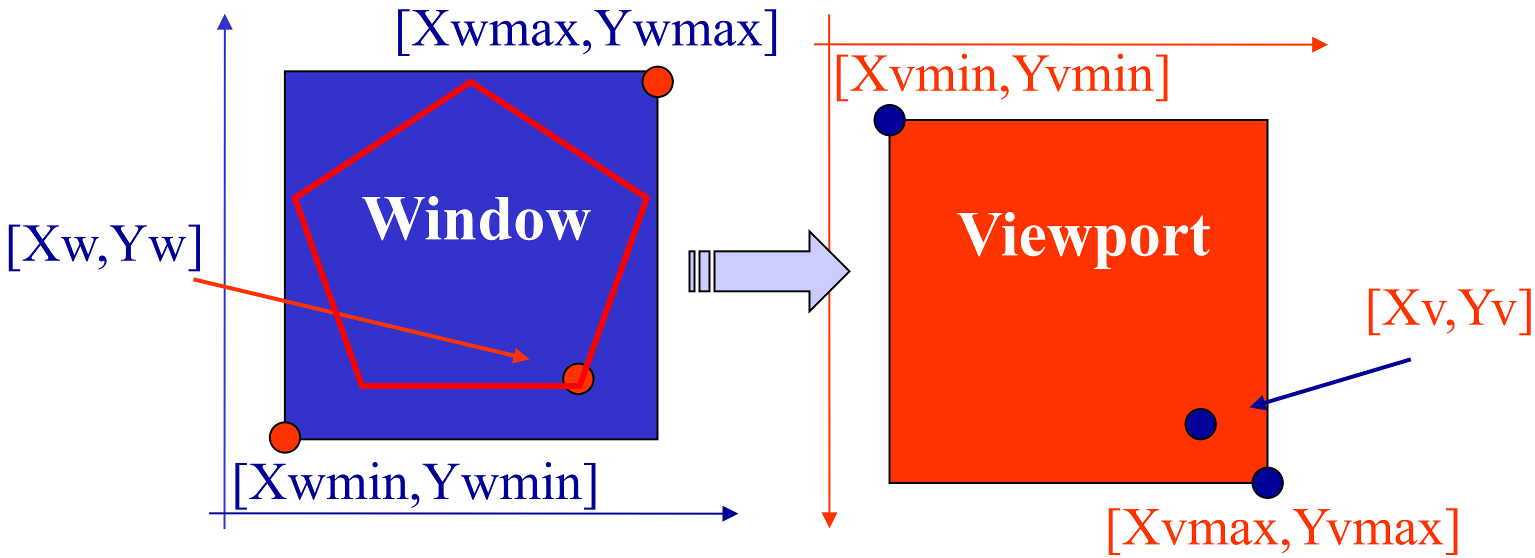
(Xv-Xvmin) : (Xvmax-Xvmin) = (Xw-Xwmin) : (Xwmax-Xwmin)

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

(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)

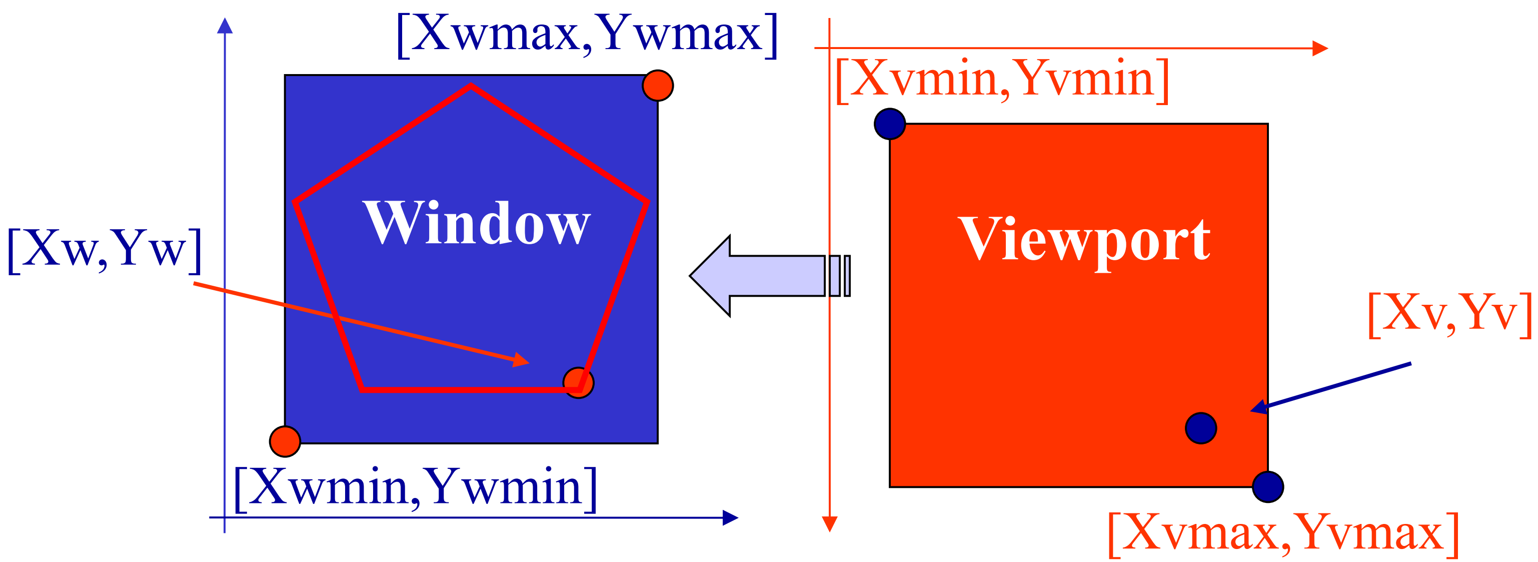
Yv = (int)(Scy * (Ywmin - Yw) + Yvmax + 0.5)

Trasformazione Window-Viewport

Disegnare in coordinate floating point ha notevoli vantaggi:

- la viewport può essere posizionata in diversi modi, avere differenti dimensioni e aspect ratio, ma non sarà necessario cambiare il disegno, bensì basterà riapplicare il mapping window-viewport;
- ridefinire la window rispetto a quanto disegnato e applicare il mapping window-viewport permette di vedere a pieno schermo dettagli di quanto disegnato (zoom);
- la trasformazione inversa viewport-window permette di definire interattivamente le parti del disegno da zoomare;
- ecc.

Trasformazione Viewport-Window

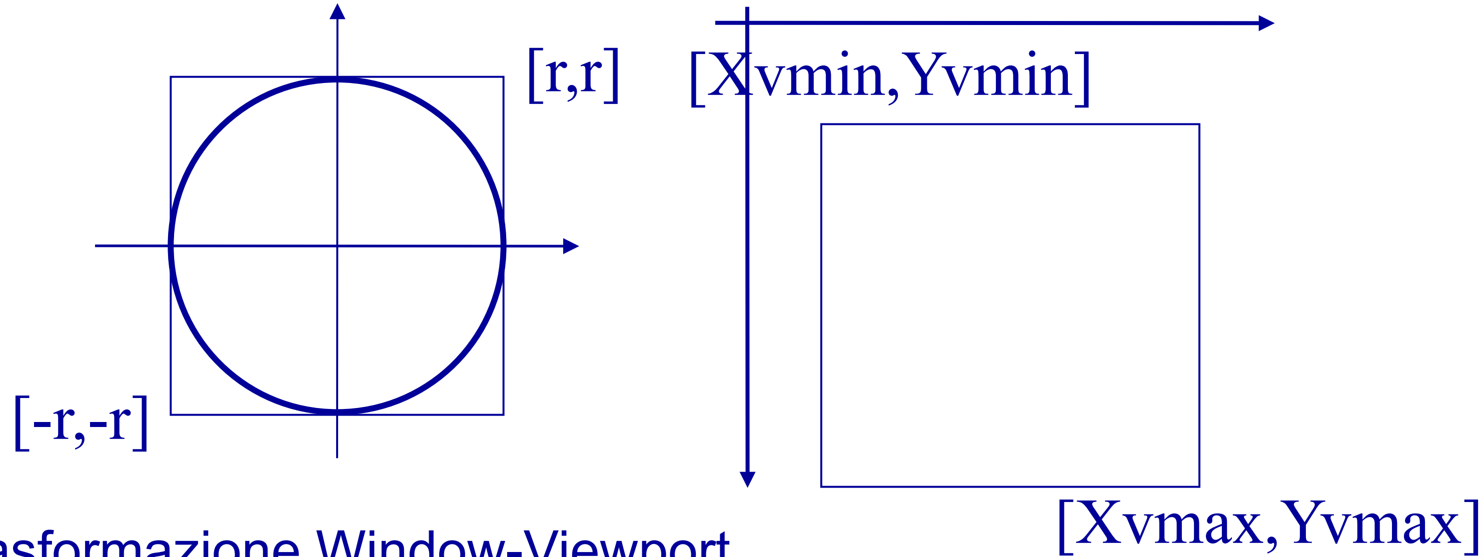


Trasformazione inversa:

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

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

Problema: disegno in coord. float

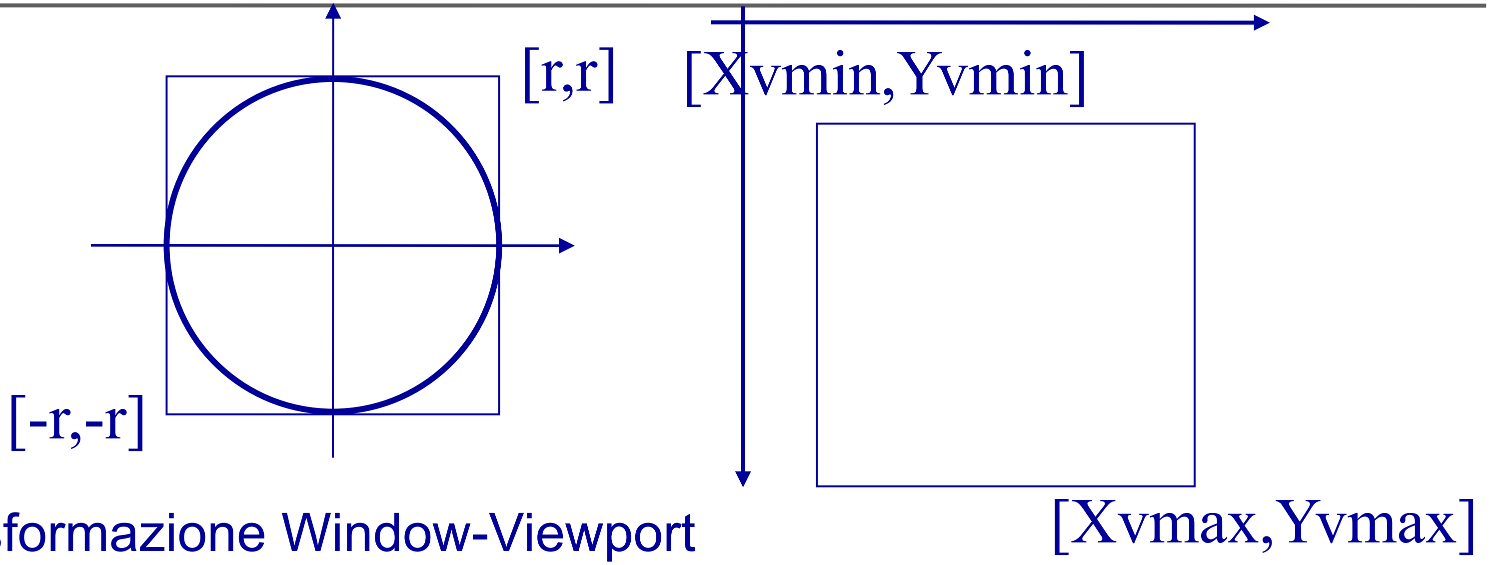


Trasformazione Window-Viewport
Definiamo due strutture/oggetti:

```
var view={
   vxmin:  0;
   vxmax:  300;
   vymin:  0;
   vymax: 150;
}
```

```
var win={
    wxmin: -1.0;
    wxmax:  1.0;
    wymin: -1.0;
    wymax:  1.0;
}
```

Problema: disegno in coord. float



Trasformazione Window-Viewport
Scriviamo una funzione

function win_view(px,py,scx,scy,view,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.

codice: polygon_float.html

Esercizio

Avendo chiaro il codice che disegna i punti floating point di un poligono discretizzando una circonferenza su un Viewport (HTML5_2d_1/polygon_float.js), realizzare un codice per il disegno di una curva piana definita in forma parametrica:

$$C(t) = \left[C_x(t), C_y(t) \right] \quad t \in [0,1]$$

lo si chiami draw_param_curve.html (.js)

Per esempi di curve in forma parametrica si consulti:

http://mathshistory.st-andrews.ac.uk/Curves/Curves.html

Soluzione Esercizio

Problema: produrre il grafico di una curva in forma parametrica.

Dati: espressione *c(t) = [f(t), g(t)]*
con t $\in$ *[a,b]* intervallo di definizione
Risultato: grafico di *n+1* punti della curva
Metodo: si campioni la curva in *n+1* punti, per esempio equidistanti, nell'intervallo di definizione
Algoritmo: tabulazione della curva

```
h=(b-a)/n;
for (var i=0; i<=n; i++){
  t=a + i*h;
  x[ i ] = f ( t )
  y[ i ] = g ( t )
}
```

Soluzione

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

```
win.xmin =x[0]
win.xmax=x[0]
for (var i=0; i<n; i++)
{
   if (x[i]>win.xmax)
      win.xmax=x[i]
   else
       if (x[i]<win.xmin)
          win.xmin=x[i]
}
```

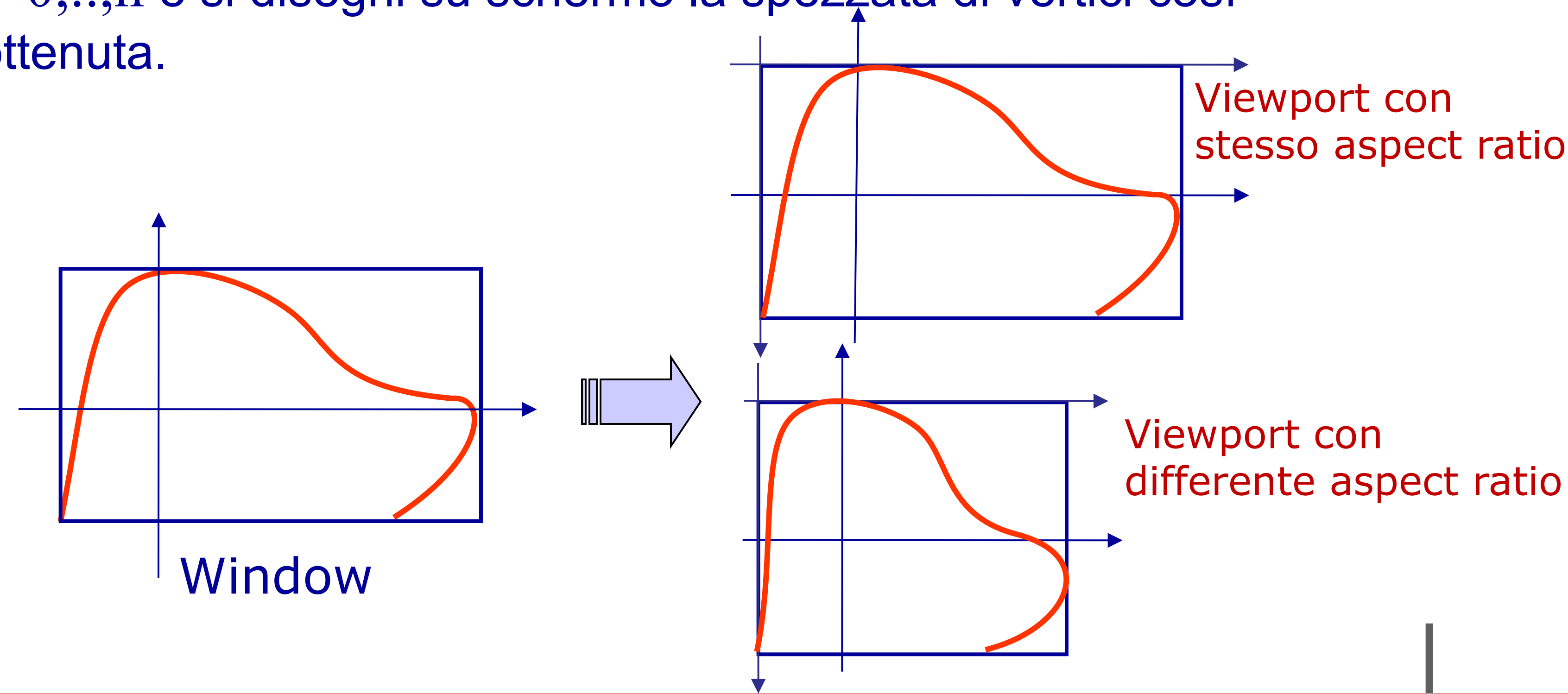
```
win.ymin =y[0]
in.ymax=y[0]
for (var i=0; i<n; i++)
{
   if (y[i]>win.ymax)
      win.ymax=y[i]
   else
       if (y[i]<win.ymin)
          win.ymin=y[i]
}
```

La Window sarà: [win.xmin,win.xmax]x[win.ymin,win.ymax]

Soluzione

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

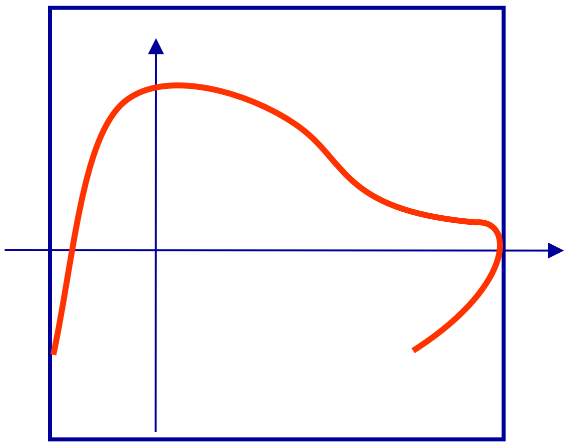
Definito quindi [view.xmin,view.xmax]x[view.ymin,view.ymax] si applichi la trasformazione Window-Viewport ai punti (x[i],y[i]) i=0,..,n e si disegni su schermo la spezzata di vertici così ottenuta.



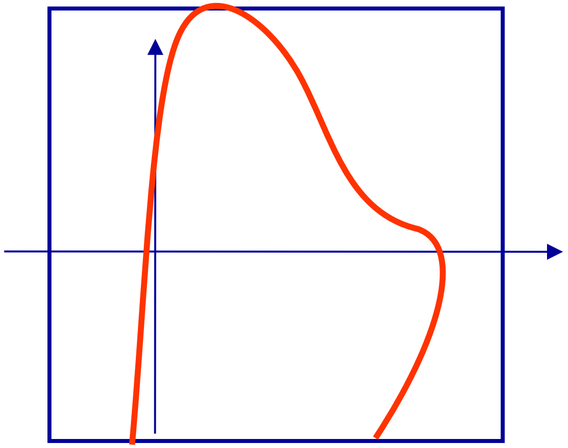
Soluzione

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 dei vertici così ottenuti.



Window



Window

Soluzione

```
dx=win.wxmax-win.wxmin;
dy=win.wymax-win.wymin;
if (dy > dx){
    diff=(dy-dx)/2;
    win.wxmin=win.wxmin-diff;
    win.wxmax=win.wxmax+diff;
}
else {
    diff=(dx-dy)/2;
    win.wymin=win.wymin-diff;
    win.wymax=win.wymax+diff;
}
```

Esercizio 1

Realizzare un codice che permetta di definire e disegnare interattivamente una poligonale di *n+1* vertici floating point (coordinate window), quindi disegni la curva di Bézier di grado *n,* di punti di controllo i vertici dati (si usi l'algoritmo di valutazione di de Casteljau).

$$C(t) = \sum_{i=0}^{n} P_i \, B_{i,n}(t) \qquad t \in [0,1]$$

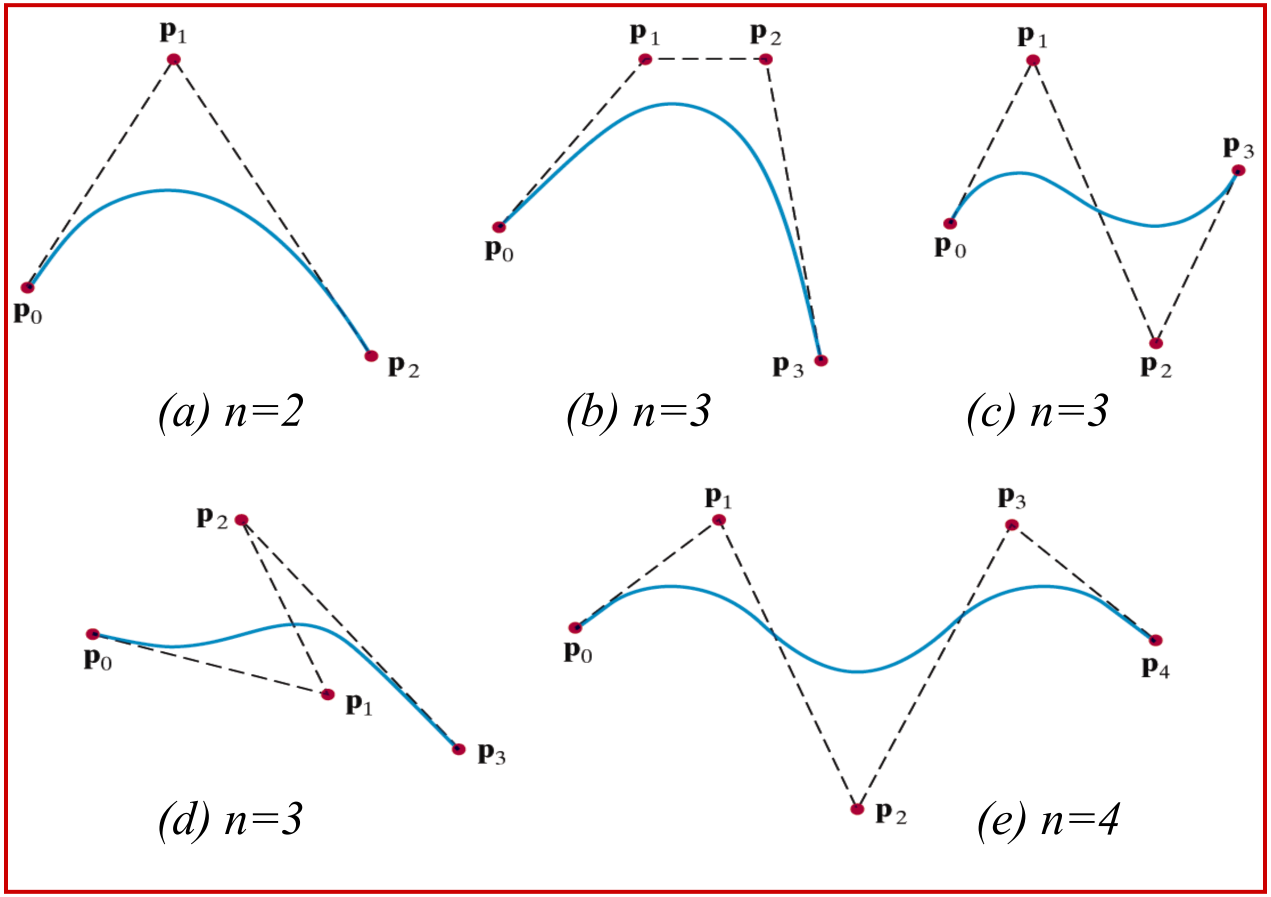
Una volta disegnata la curva sia possibile modificarne la forma spostando col mouse singoli punti di controllo;

Io si chiami draw_bezier_curve.html (.js)

Sugg. Prima si analizzi il codice HTML5_2d_1/bezier.html e .js che permette di disegnare una curva di Bézier cubica (viene utilizzata la function bezierCurveTo) e di interagire con i suoi punti di controllo.

Esempi di Curve di Bézier

Curve di Bézier di differenti gradi n



Curve di Bézier e algoritmo di valutazione de de Casteljau

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

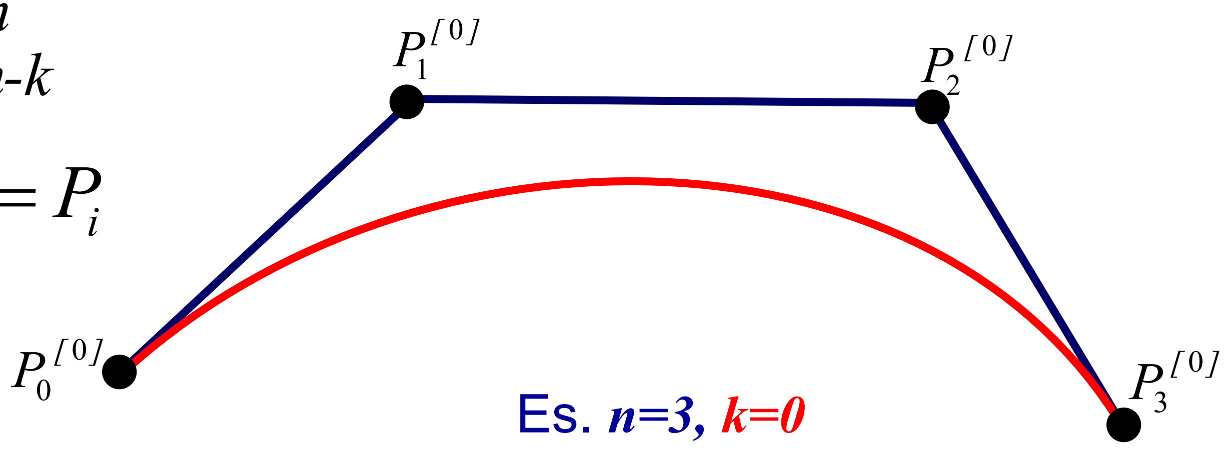
$$P_i^{[k]}(t) = (1-t)P_i^{[k-1]}(t) + tP_{i+1}^{[k-1]}(t) \qquad t \in [0,1]$$

dove $\quad k = 1,\dots,n$

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

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

$\quad\quad\quad i = 0,\dots,n$



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

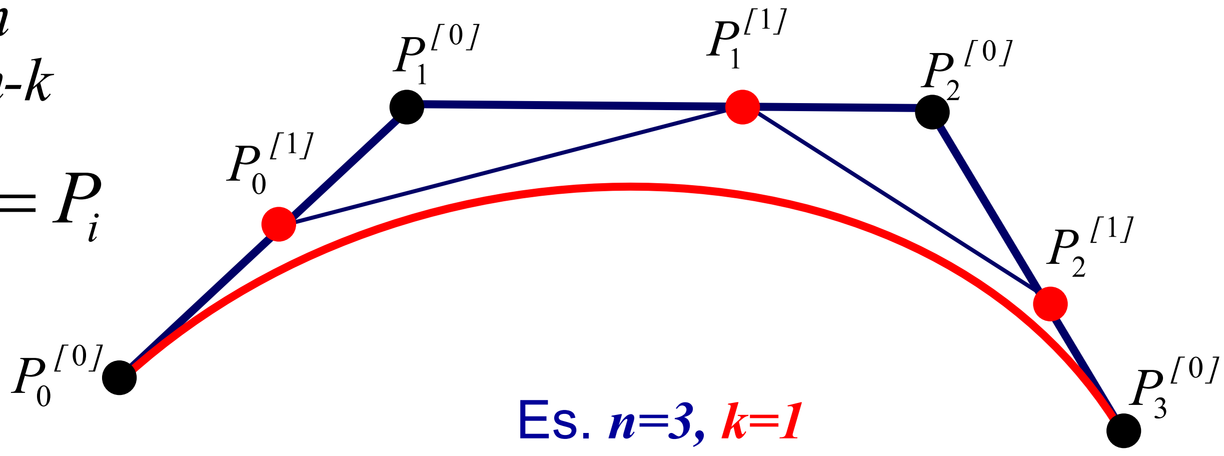
Curve di Bézier e algoritmo di valutazione di de Casteljau

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 $\begin{array}{l} k = 1,\dots,n \\ i = 0,\dots,n\text{-}k \end{array}$

con $\begin{array}{l} P_i^{[0]}(t) = P_i \\ i = 0,\dots,n \end{array}$



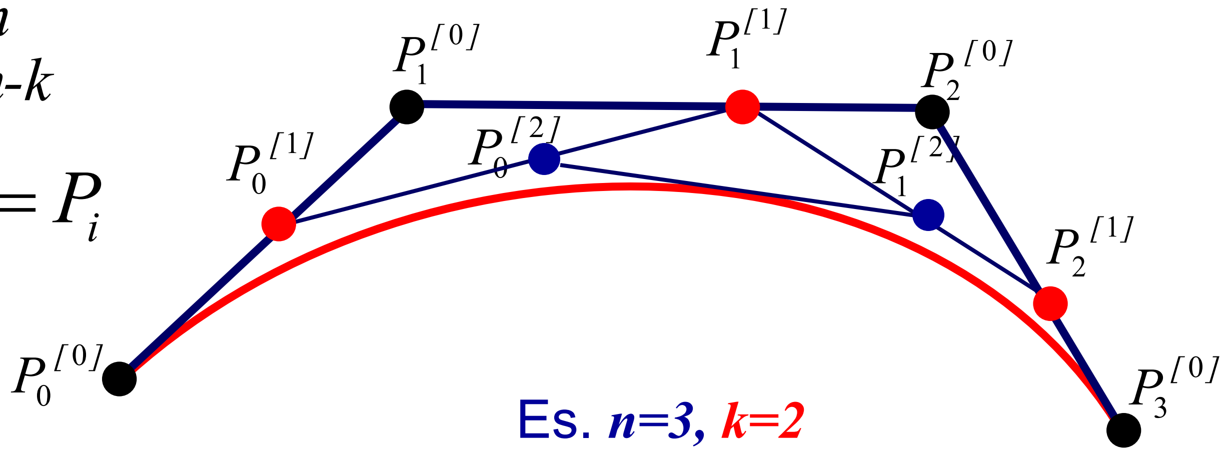
Curve di Bézier e algoritmo di valutazione di de Casteljau

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

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



Es. n=3, k=2

Curve di Bézier e algoritmo di valutazione di de Casteljau

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

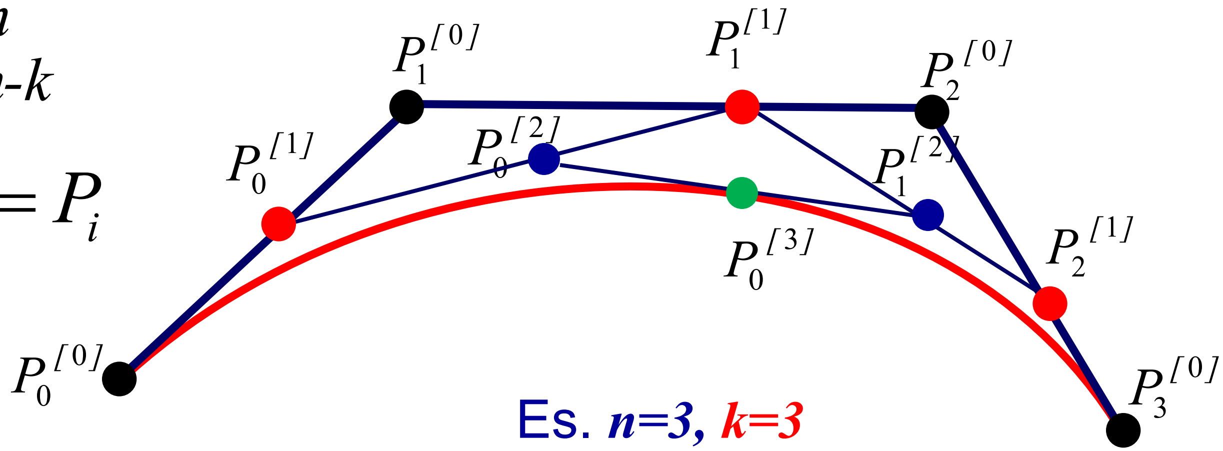
$$P_i^{[k]}(t) = (1-t)P_i^{[k-1]}(t) + tP_{i+1}^{[k-1]}(t) \qquad t \in [0,1]$$

dove $k = 1,\ldots,n$

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

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

$i = 0,\ldots,n$



Questa definizione è anche un algoritmo numericamente stabile per il calcolo delle curve di Bézier (implementarlo senza ricorsione).

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Giulio Casciola

Dip. di Matematica

giulio.casciola at unibo.it