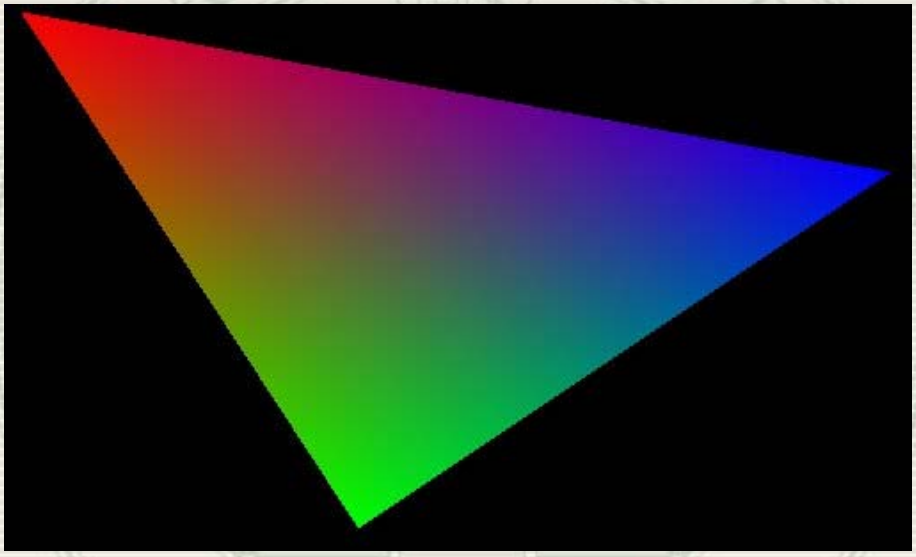
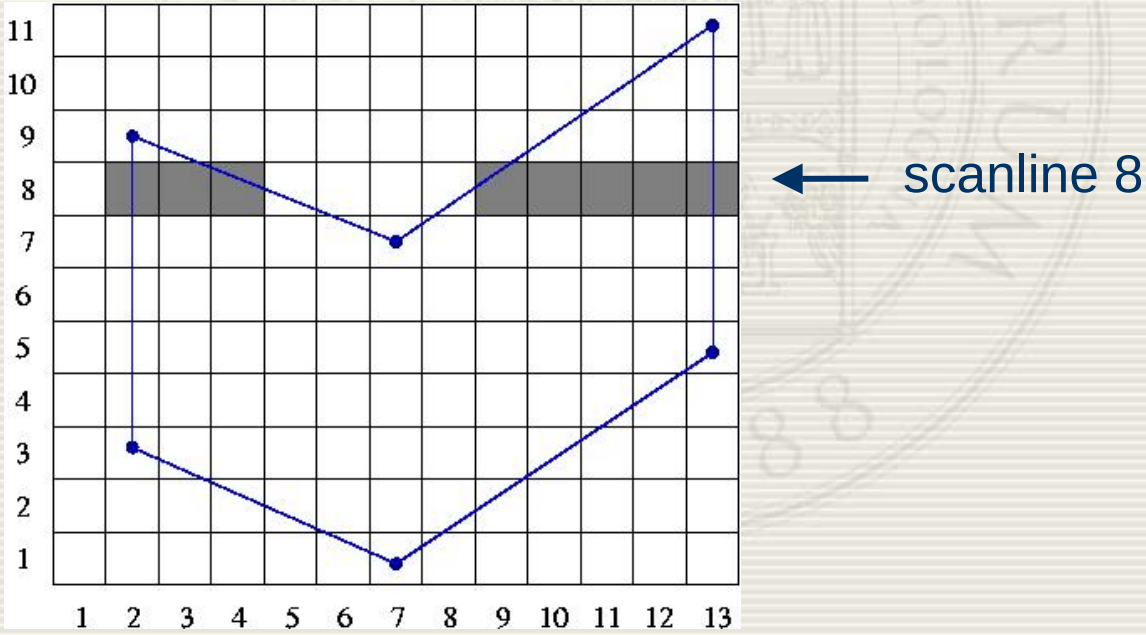
Algoritmi di Rasterizing



Un algoritmo di Rasterizing

Dato un poligono in coordinate schermo, la sua rasterizzazione consiste nel disegnare con un colore tutti i pixel del poligono (i pixel interni e/o sui lati del poligono).

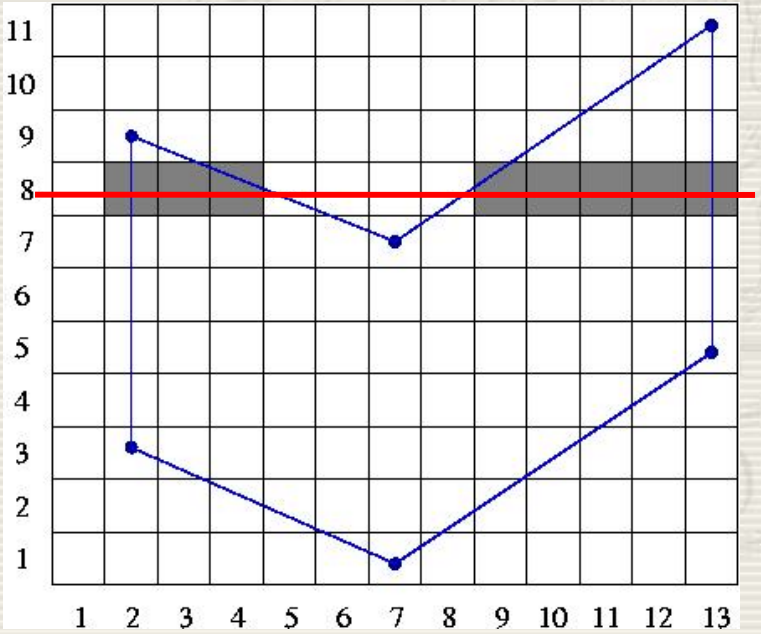
L'idea base dell'algoritmo che descriveremo è determinare una riga di pixel (scan-line) alla volta del poligono.



Un algoritmo di Rasterizing

L'algoritmo di rasterizzazione si può sintetizzare nei seguenti passi:

- Trovare l'intersezione della scan-line con tutti i lati del poligono
- Ordinare le intersezioni secondo le ascisse
- Accendere i pixel della scan-line fra coppie di intersezioni



scanline 8

Nell'esempio la lista ordinata delle ascisse è: (2, 4, 9, 13); si accenderanno quindi i pixel da 2 a 4 e da 9 a 13.

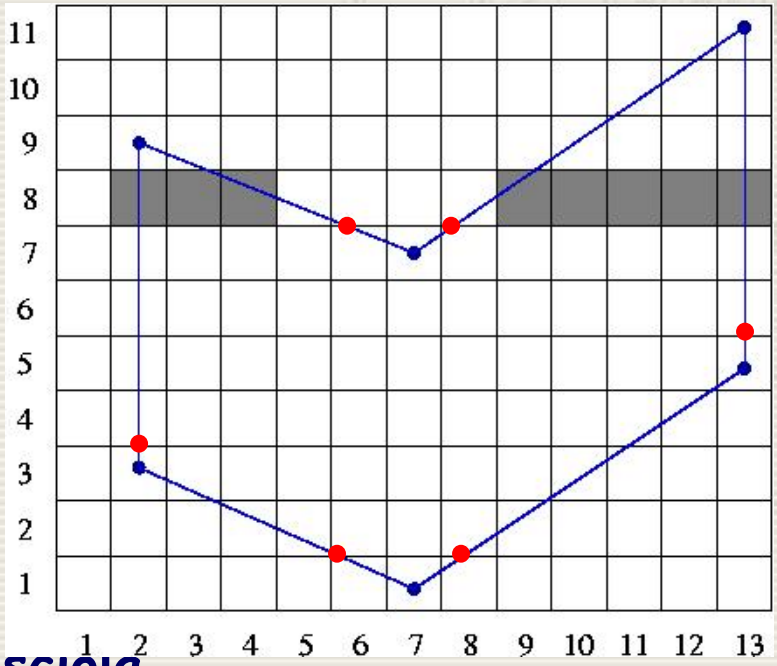
Un algoritmo di Rasterizing

Osservazioni sulla fase di intersezione:

•i lati orizzontali del poligono vengono scartati perché non producono intersezioni;

•i lati vengono accorciati per garantire che ogni scan-line intersecata con il poligono (con tutti i lati) fornisca un numero pari di intersezioni (teorema di Jordan);

•utilizzo dell'algoritmo di linea incrementale per determinare le intersezioni in modo semplice.



I lati abbiano estremi (x0,y0) e (x1,y1) e (x0,y0) sia sempre l'estremo con ordinata maggiore, allora:

n=abs(y1-y0)=y0-y1 (numero di scan-line – 1)

m=n/(x1-x0) (pendenza)

dx=1/m

dy= -1

$$x_{i+1} = x_i + dx$$

$$y_{i+1} = y_i + dy$$

Algoritmo di Linea Incrementale: richiamo

Sia n=abs(y1-y0) e applichiamo l'algoritmo visto:

P0 = (8 , 4) P1 = (1 , 1)

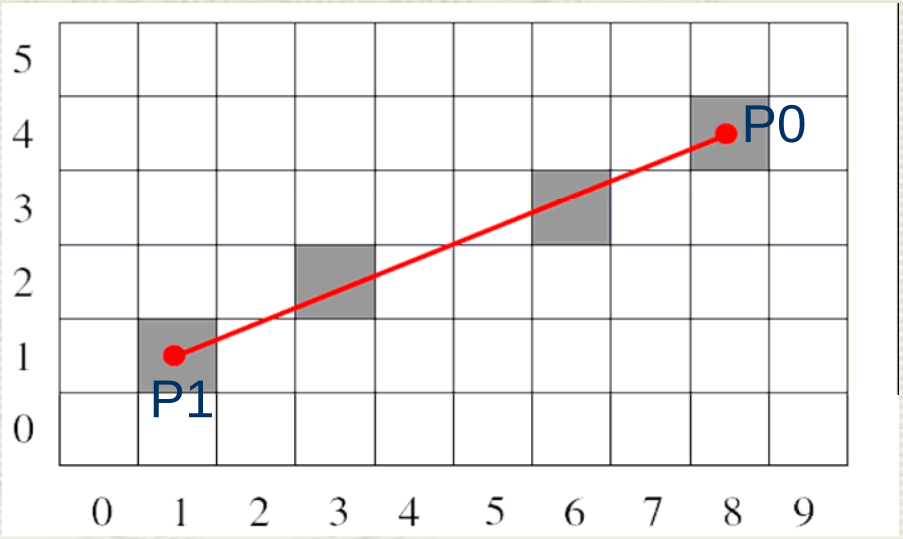
sarà: n=3, dx=-7/3 ≅ -2.333, dy=1

--> (8,4)

(5.666,3) --> (6,3)

(3.333,2) --> (3,2)

(1.000,1) --> (1,1)

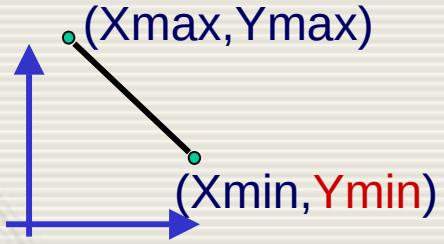


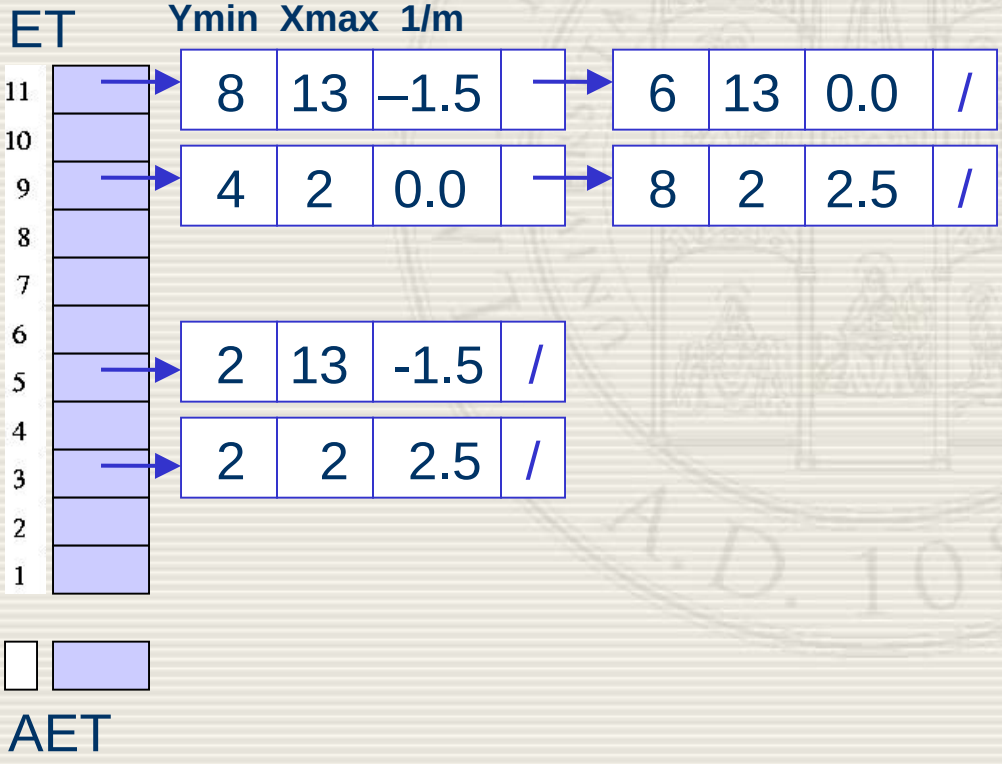
Un algoritmo di Rasterizing

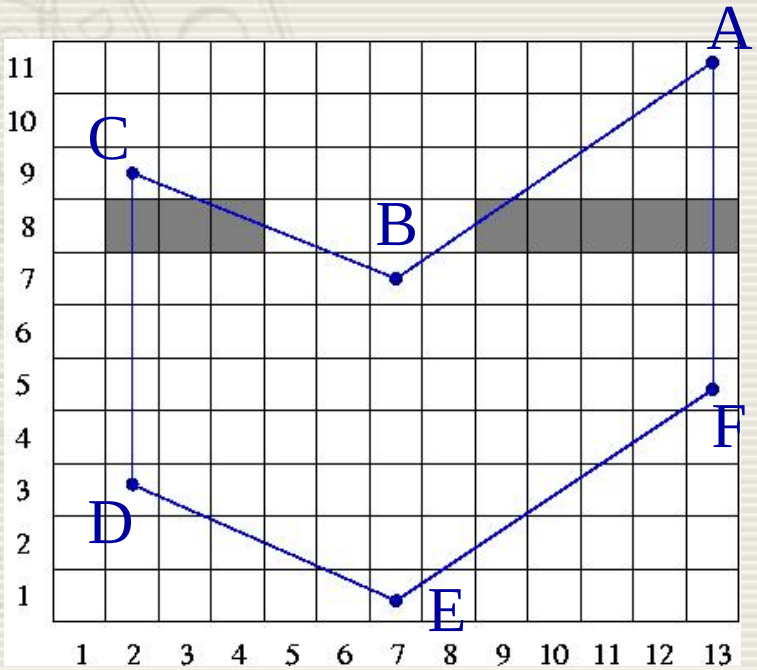
Per implementare l'algoritmo faremo uso di due strutture dati:

Edge Table (ET): contiene le informazioni sui lati secondo la loro ordinata maggiore;

Active Edge Table (AET): contiene la situazione dell' intersezione fra la scan-line corrente e i lati del poligono, così che sia facile determinare la sequenza dei pixel da disegnare.





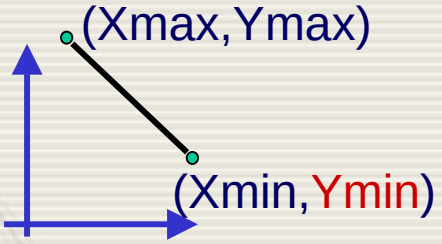


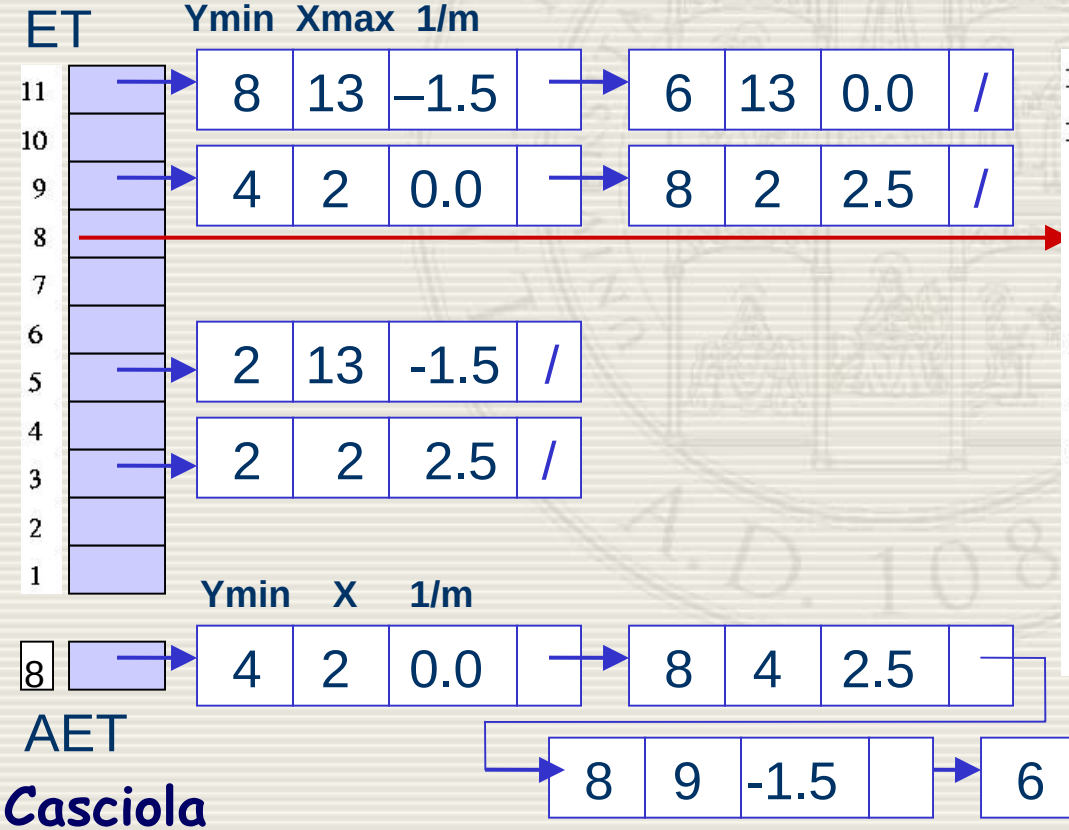
Un algoritmo di Rasterizing

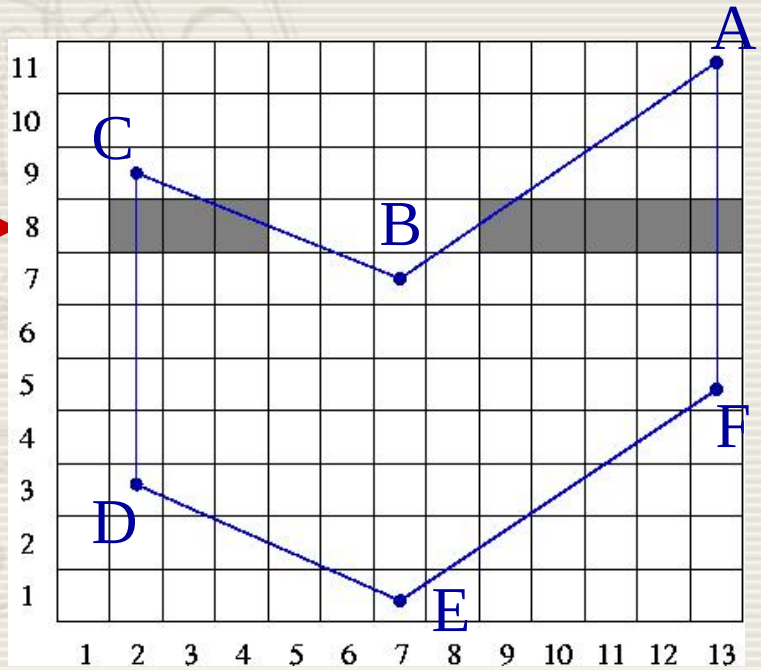
Per implementare l'algoritmo faremo uso di due strutture dati:

Edge Table (ET): contiene le informazioni sui lati secondo la loro ordinata maggiore;

Active Edge Table (AET): contiene la situazione dell' intersezione fra la scan-line corrente e i lati del poligono, così che sia facile determinare la sequenza dei pixel da disegnare.





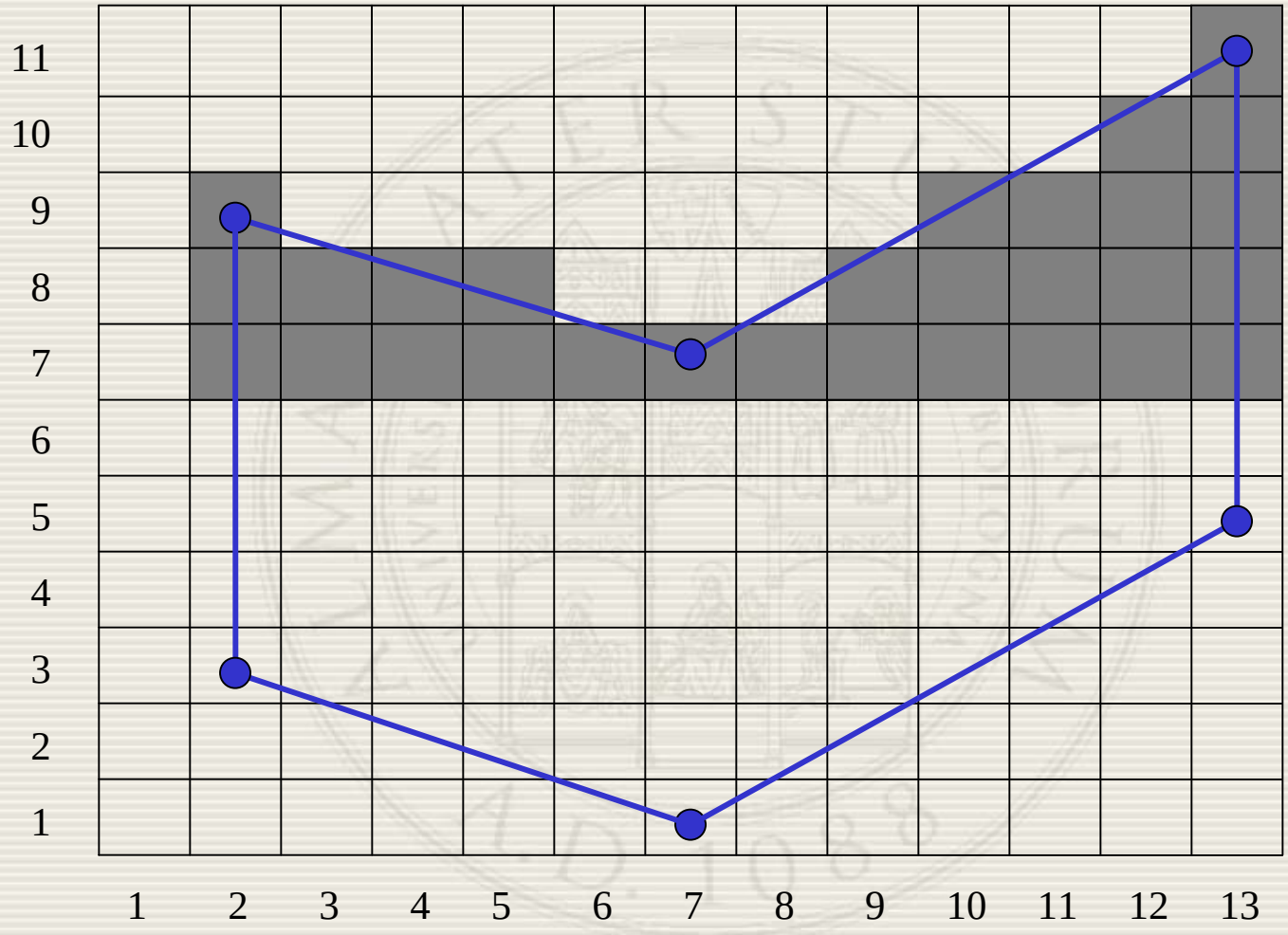


Un algoritmo di Rasterizing

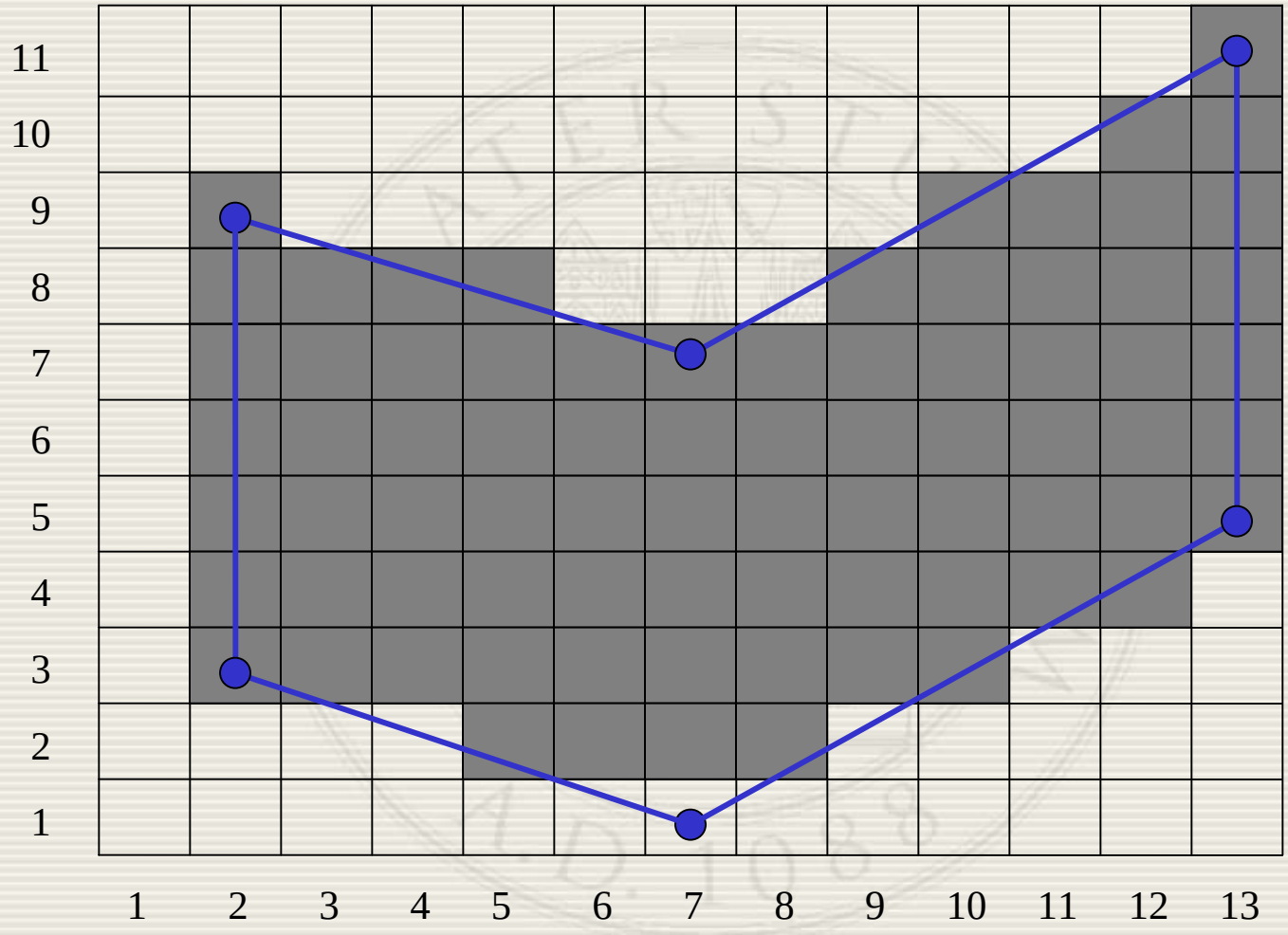
Una volta costruito l'ET, i passi dell'algoritmo sono:

1. Porre Y alla più grande ordinata dell'ET;
2. Inizializzare l'AET a vuoto;
3. Ripetere fino a che l'AET o l'ET sono vuoti:
   - 3.1 muovere l'informazione relativa a Y dall'ET all'AET, mantenendo l'AET ordinato sulle X;
   - 3.2 disegnare sulla scan-line Y i pixel utilizzando coppie di ascisse dall'AET;
   - 3.3 rimuovere dall'AET quelle informazioni per cui Y=Ymin;
   - 3.4 per ciascuna informazione rimasta nell'AET, aggiornare X con X+1/m;
   - 3.5 ordinare l'AET in base alle ascisse;
   - 3.6 Y=Y-1;

Simulazione dell'Algoritmo

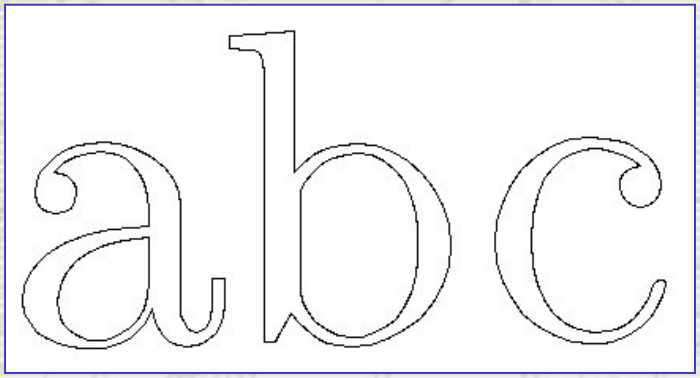


Simulazione dell'Algoritmo



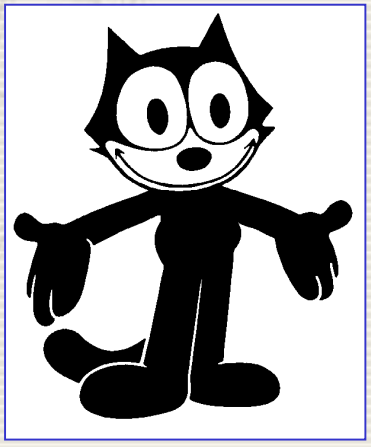
Esempi di Rasterizing

Rasterizzazione di poligonali ottenute discretizzando le curve outline della fonte cm10 (computer modern 10).



Esempi di Rasterizing

Rasterizzazione di poligonali chiuse ottenute mediante tracing di bitmap



Bitmap: 736x815 pixel

Bitmap: 640x801 pixel

Bitmap: 672x777 pixel

Esempio

Archvio SDL2prg_1516.tgz, cartella SDL2prg1, codice:

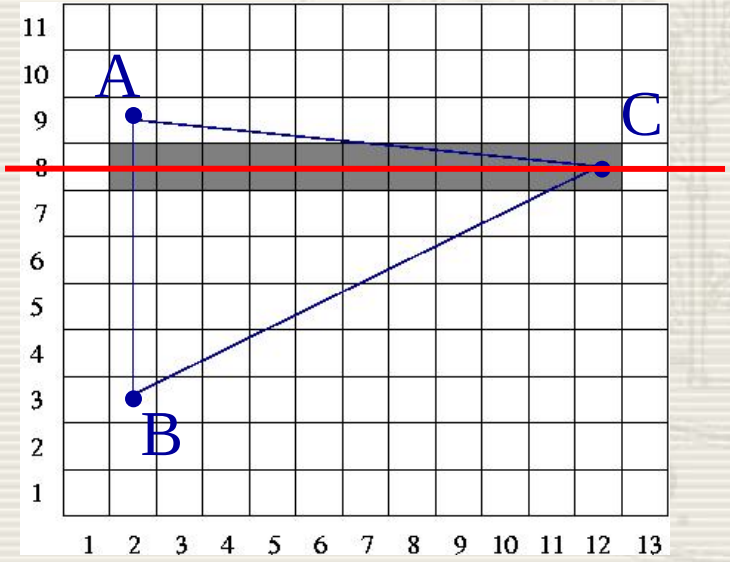
scan_conv.c

Rasterizing di Triangoli

L'algoritmo di scan conversion visto si può specializzare per triangoli.

Per ogni scan-line che ha a che fare con il triangolo:

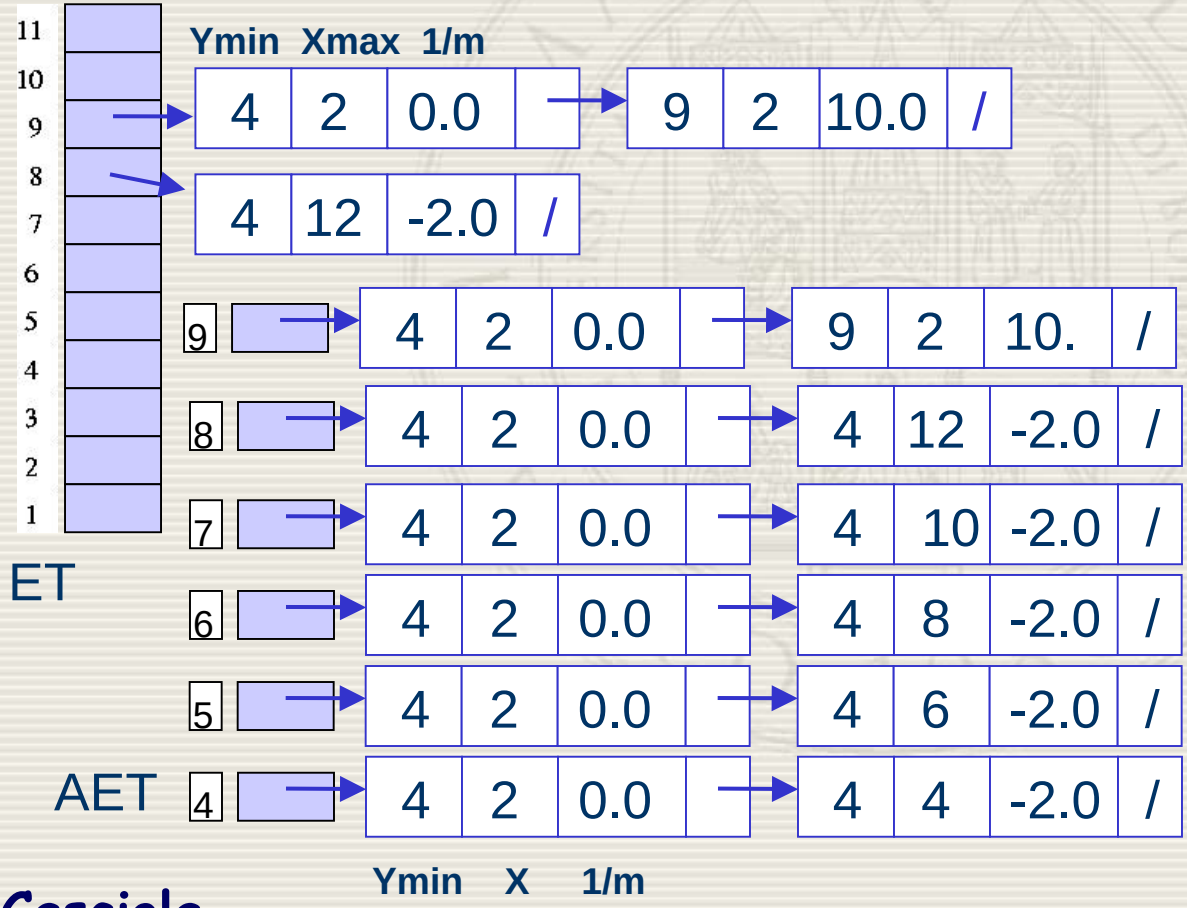
- Trovare l'intersezione della scan-line con i due lati del triangolo
- Accendere i pixel della scan-line fra le due intersezioni trovate

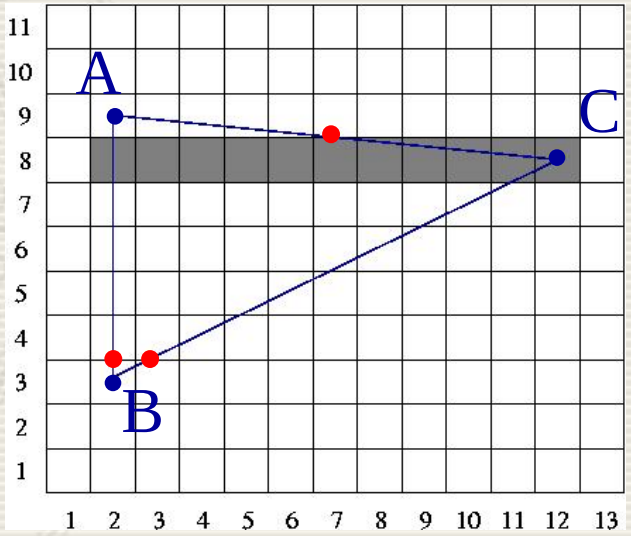


Per determinare le intersezioni di una scan-line con i lati del triangolo si utilizza ancora l'algoritmo incrementale di linea con incremento unitario in Y

Rasterizing di Triangoli

L'algoritmo può essere implementato specializzando o banalizzando le due strutture Edge Table (ET) e Active Edge Table (AET); la prima che contiene tanti record quanti sono i lati ora sarà limitata a tre, mentre la seconda avrà sempre solo due informazioni sui lati correnti.





Rasterizing di Triangoli per Z-buffer

Nel caso di triangoli ottenuti per proiezione con correzione prospettica per ogni vertice del triangolo 3D, avremo

$$\begin{cases} xs = xe/ze \\ ys = ye/ze \\ zs = alpha + beta/ze \end{cases}$$

con alpha e beta valori per correzione prospettica

quindi si applichi la rasterizzazione al “triangolo 2D”;

Dalle coordinate xs, ys ottenute per proiezione con correzione prospettica, l’algoritmo di rasterizzazione genererà le coordinate pixel del triangolo, le coordinate zs saranno utili per determinare la profondità del punto 3D nello Spazio del Piano di Proiezione a cui il pixel corrisponde.

Rasterizing di Triangoli per Z-buffer

Nel caso di triangoli in cui oltre alle informazioni xs ed ys dei vertici si abbiano anche le informazioni sulle coordinate zs dei vertici, l'algoritmo di rasterizzazione dovrà essere modificato e le strutture dati dovranno essere ampliate:

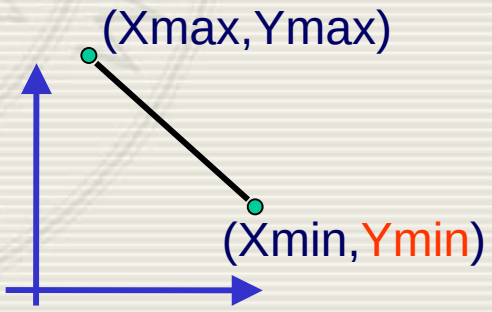
ET

$$\frac{1}{m} = \frac{Xmin-Xmax}{Ymax-Ymin} \qquad \frac{1}{mz} = \frac{Zmin-Zmax}{Ymax-Ymin}$$

AET


```
Y++;
X = X + 1/m;
Z = Z + 1/mz;
```

Nota: con Xmin si intende la coord. x associata al punto di minore ordinata



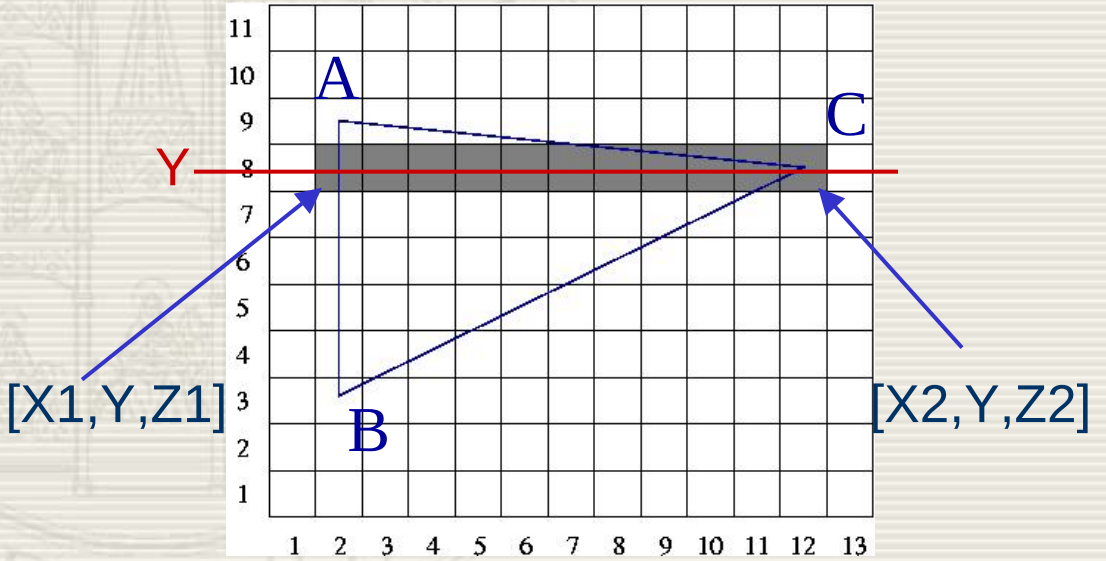
Rasterizing di Triangoli per Z-buffer

Sulla scanline Y, determinata la coppia di ascisse X1 e X2 fra cui accendere i pixel, avremo anche le coordinate Z1 e Z2 che rappresentano le profondità dei pixel (X1,Y) e (X2,Y).

Per ogni pixel (X,Y) con X=X1,…,X2, la relativa profondità Z potrà essere calcolata come:

```
n=X2-X1;
dZ=(Z2-Z1)/n;
X=X1;
Z=Z1;
for i=1,..,n
   X = X + 1;
   Z = Z + dZ;
```

Algoritmo di Linea incrementale



Rasterizing di Triangoli per Z-buffer

Nel caso di triangoli 3D in cui oltre alle informazioni x, y e z dei vertici si abbiano anche informazioni sulle componenti colore per vertice, si deve procedere alla proiezione con correzione prospettica per ogni vertice

$$\begin{cases} xs = xe/ze \\ ys = ye/ze \\ zs = alpha + beta/ze & \text{con alpha e beta} \\ Rs = Re/ze & \text{valori per correzione} \\ Gs = Ge/ze & \text{prospettica} \\ Bs = Be/ze \end{cases}$$

poi applicare la rasterizzazione al “triangolo 2D con le aggiunte e modifiche introdotte.

Ma, attenzione, anche le coordinate R, G e B dovranno essere elaborate in modo corretto e non semplicemente interpolando i valori proiettati; per essere corrette dovranno essere ricondotte ai valori del colore del triangolo 3D originale.

Rasterizing di Triangoli per Z-buffer

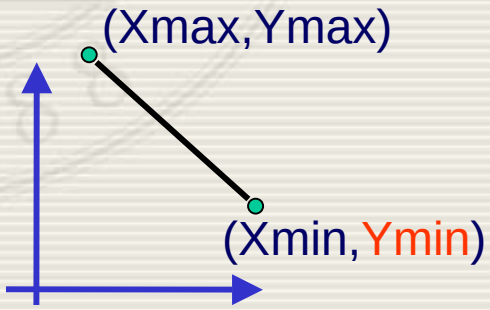
Nel caso di triangoli in cui oltre alle informazioni xs, ys e zs dei vertici si abbiano anche informazioni sulle componenti colore per vertice, l'algoritmo di rasterizzazione dovrà essere ulteriormente modificato e le strutture dati ampliate:

ET

$$\frac{1}{m} = \frac{Xmin-Xmax}{Ymax-Ymin} \qquad \frac{1}{mz} = \frac{Zmin-Zmax}{Ymax-Ymin} \qquad \frac{1}{mr} = \frac{Rmin-Rmax}{Ymax-Ymin} \qquad \text{ecc.}$$

AET

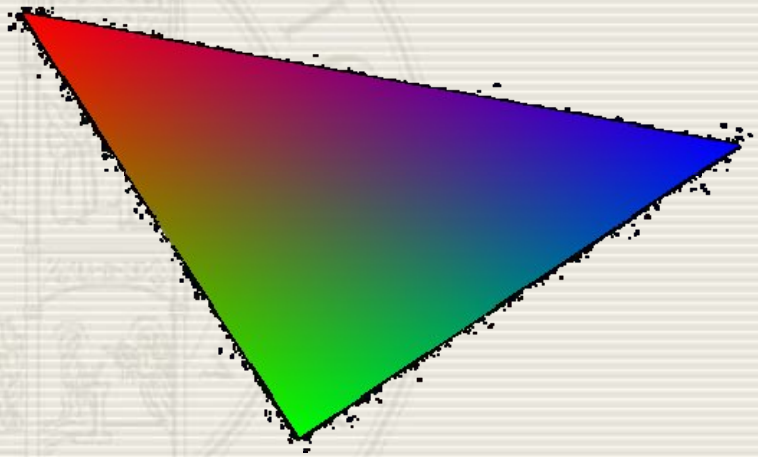

```
Y++;
X =  X + 1/m;
Z =  Z + 1/mz;
R = R + 1/mr;
G = G + 1/mg;
B =  B + 1/mb;
```



Rasterizing di Triangoli per Z-buffer

Per ottenere le componenti colore corrette, una volta determinate le componenti interpolate, dovranno essere moltiplicate per la componente ze (cooordinata zs nello Spazio dell'Osservatore) corrispondente al punto che si sta elaborando.

```
Ze = beta / (Z– alpha);
Re = R * Ze;
Ge = G * Ze;
Be =  B * Ze;
```



[Re, Ge, Be] sono le cordinate colore del punto sul triangolo 3D corrispondente al punto che si sta elaborando in 2D, mentre [R, G, B] sono le componenti colore interpolate di [Rs1, Gs1, Bs1], [Rs2, Gs2, Bs2] ed [Rs3, Gs3, Bs3] vertici del triangolo 2D.

Rasterizing di Triangoli per Z-buffer

Un modo alternativo, come già visto, ma più costoso, consiste nell'elaborare solo le informazioni xs, ys e zs (quest'ultima per i test di profondità), calcolare poi direttamente o incrementalmente le relative coordinate baricentriche o parametriche [u', v', s'] sul triangolo schermo e da queste elaborare le informazioni colore originali;

```
Re = u * Re1 + v * Re2 + s * Re3 ;
Ge = u * Ge1 + v * Ge2 + s * Ge3;
Be =  u * Be1 + v * Be2 + s * Be3;
```

Coord. baricentriche

dove

```
u = u' / ze1 / (1/z);
v = v' / ze2 / (1/z);
s = s' / ze3 / (1/z);
```

vedi formule (6), (7) e (8) slide coord. bar.

con `1/z = u'/ze1 + v'/ze2 + s'/ze3;` vedi formula (9) slide coord. bar.

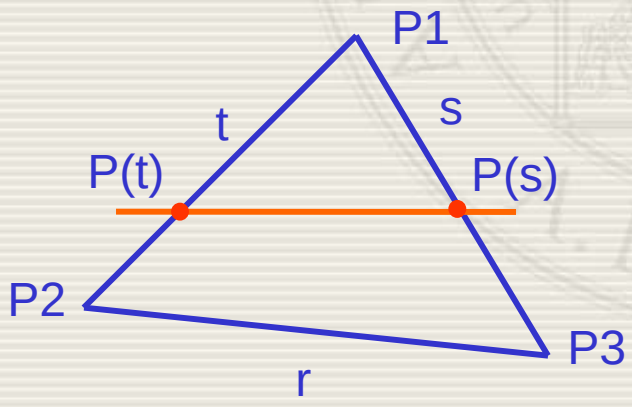
Rasterizing di Triangoli per Z-buffer

Questo è poi anche un modo nel caso si voglia texturare un triangolo a partire da una immagine idealmente applicata su un triangolo 3D.

Analogamente in forma parametrica:

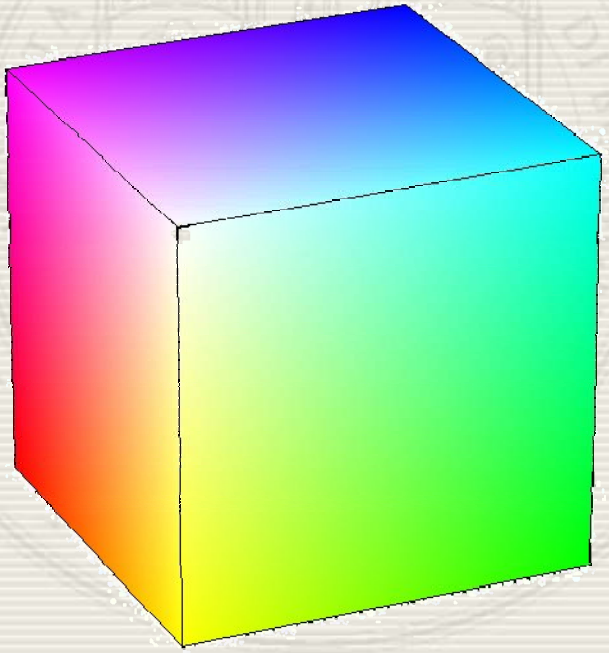
$$
\begin{aligned}
Re &= [1-(1-r)*t-r*s]*Re1 + [(1-r)*t]*Re2 + [r*s]*Re3;\\
Ge &= [1-(1-r)*t-r*s]*Ge1 + [(1-r)*t]*Ge2 + [r*s]*Ge3;\\
Be &= [1-(1-r)*t-r*s]*Be1 + [(1-r)*t]*Be2 + [r*s]*Be3;
\end{aligned}
$$

Coord. parametriche



$$
\begin{aligned}
P(t) &= (1-t)\,P1 + t\,P2\\
P(s) &= (1-s)\,P1 + s\,P3\\
P(r,s,t) &= (1-r)\,P(t) + r\,P(s) =\\
&= (1-r)\,[\,(1-t)\,P1 + t\,P2\,] + r\,[(1-s)\,P1 + s\,P3\,]
\end{aligned}
$$

Riprendere l'esempio:
persp_cube_color_sdl.c



Funzioni SDL utili per immagini-texture

Nella libreria GCGraLib2 è presente la funzione:

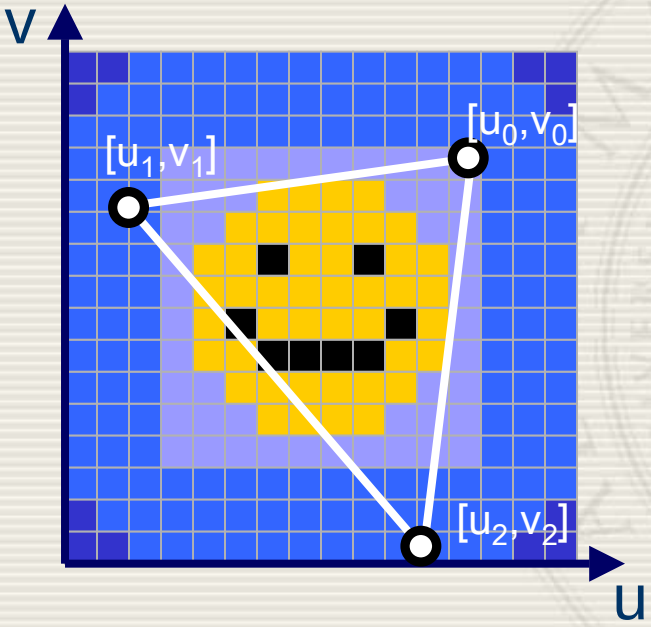
GC_GetPixelImage( )

Una volta letta un'immagine, se ne possiede un puntatore (per esempio image di tipo SDL_Surface*); per accedere in lettura e scrittura ad un pixel (x,y) dell'immagine si può fare:

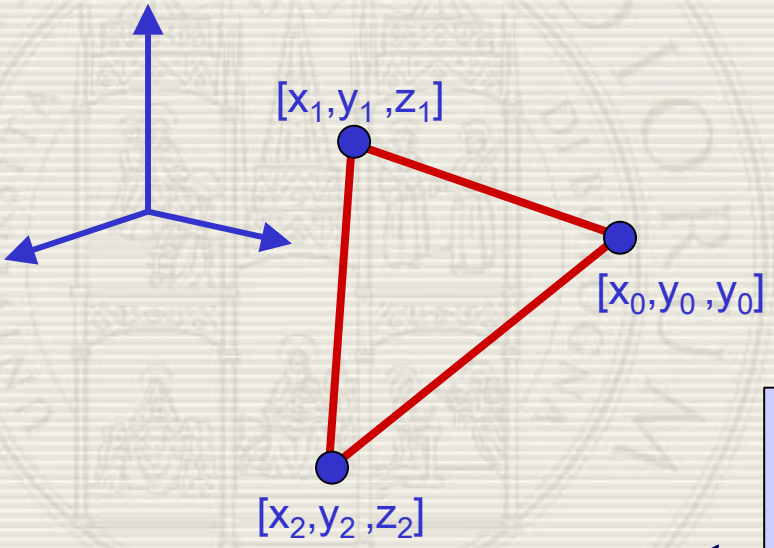
```
SDL_Surface *image;
Uint32 pix_val;
....
pix_val=GC_GetPixelImage(image,x,y,); //ritorna il colore del
                                      //pixel (x,y) nell'immagine;
....
GC_PutPixel1(image,x,y,pix_val); //assegna il colore pix_val al
                                 //pixel (x,y) dell'immagine.

SDL_SetRenderDrawColor(ren,pix_val[2],pix_val[1],pix_val[0],255);
SDL_RenderDrawPoint(ren,x,y); //assegna il colore pix_val al
                              //pixel (x,y) dello schermo.
```

Texture Mapping

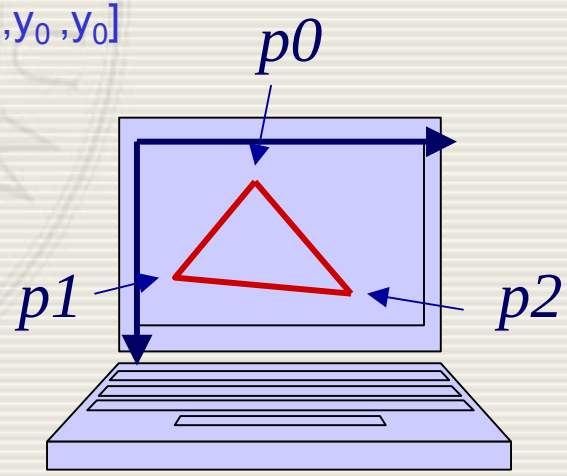
Ai vertici (di ogni triangolo 3D) si assegnano delle coordinate [*u,v*] nello spazio texture



Texture Space



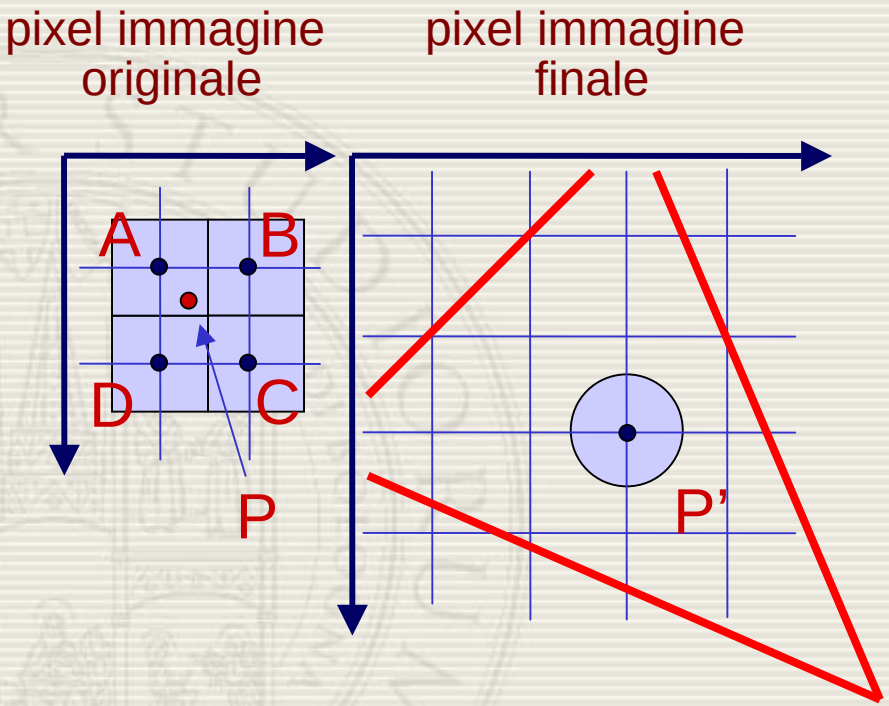
Object Space



Immagini Texture

1.Si considera ogni pixel P' di un triangolo schermo (proiezione di un triangolo 3D) e gli si applichi la trasformazione per portarlo nello spazio dell'immagine originale (P);

2.si procede alla determinazione dei pixel (A, B, C, D) dell'immagine originale più vicini a P.



Ci sono più tecniche per determinare un colore da attribuire al pixel dell'immagine finale (sul triangolo) a partire dal o dai pixel più prossimi dell'immagine originale; richiamiamone solo alcuni:

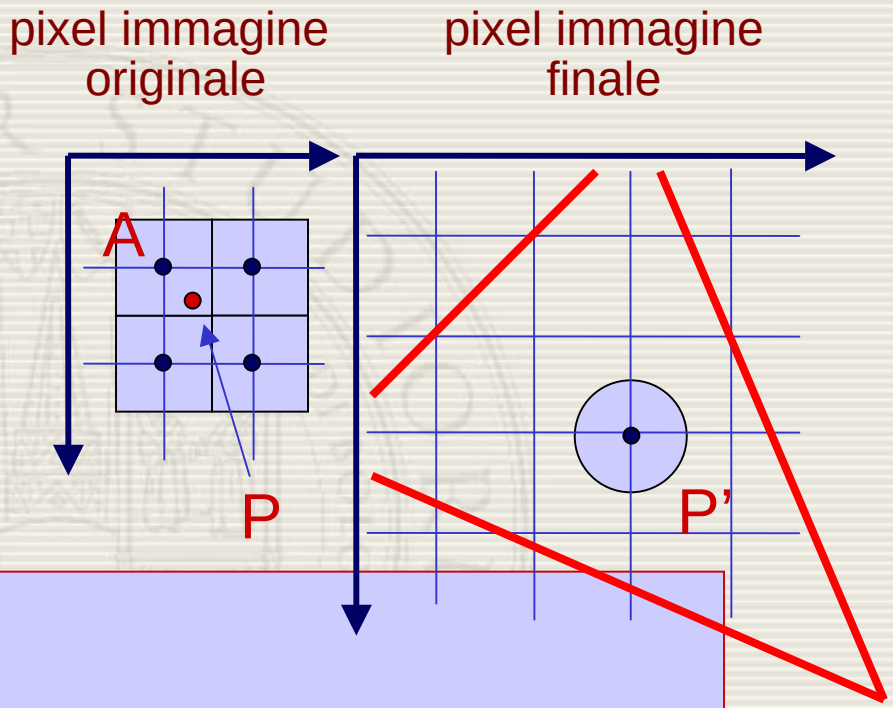
- Nearest Neighbor
- Bilinear Interpolation
- Bicubic Interpolation

Nearest Neighbor

Come dice il nome, consiste nel considerare il pixel dell'immagine originale più vicino a P.

Nel caso di figura, sarà il pixel A; si tratta dello schema più semplice possibile.

Dalle coordinate floating point (px,py) di P, vengono determinate le coordinate intere (ax,ay) di A come:

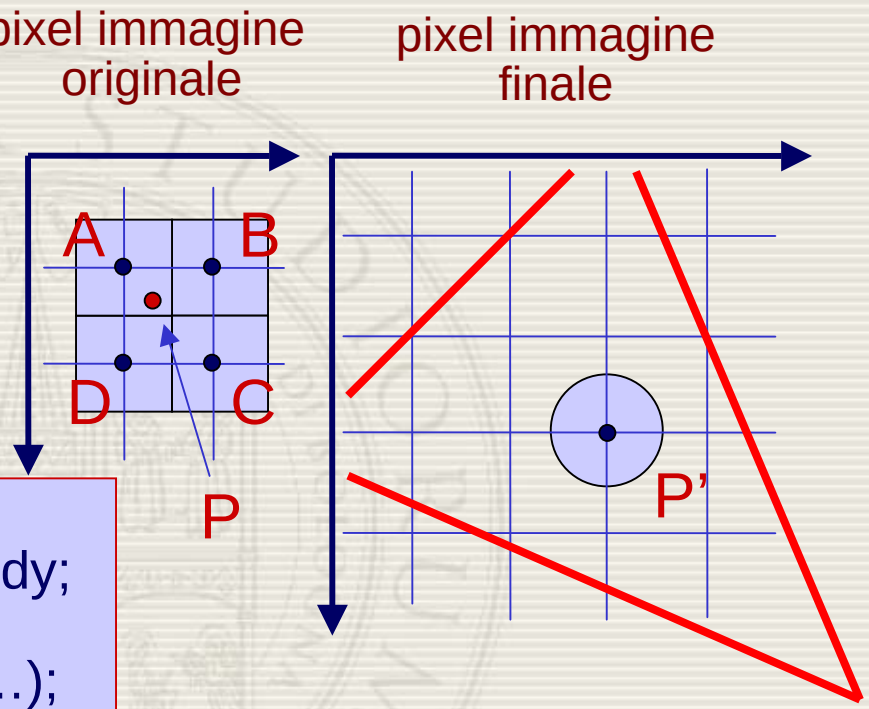



```
float px,py;
int ipx,ipy,ax,ay;
   inv_trasf(ipx, ipy, &px, &py, …);
   ax = (int) px;
   ay = (int) py;
   pixv=GC_GetPixelImage(image,ax,ay);
   SDL_SetRenderDrawColor(ren,pixv[2],pixv[1],pixv[0],255);
   SD_RenderDrawPoint(ren,ipx,ipy);
```

Bilinear Interpolation

Questa tecnica considera i 4 pixel, dell'immagine originale, più vicini a P;

le loro coordinate intere vengono determinate dalle coordinate floating point (px,py) di P, come:

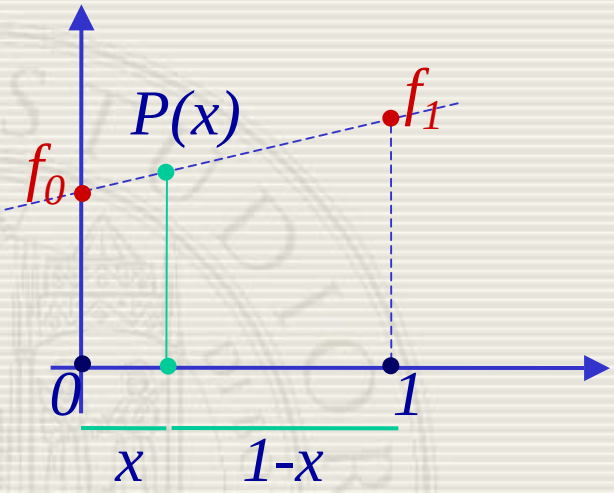



```
float px,py;
int ipx,ipy,ax,ay,bx,by,cx,cy,dx,dy;

   inv_trasf(ipx, ipy, &px, &py, …);
   ax = dx = (int) px;
   ay = by = (int) py;
   bx = cx = ax + 1;
   dy = cy = ay +1;
```

Le componenti colore dei 4 pixel più vicini vengono poi combinate in modo pesato (interpolazione) per arrivare alle componenti colore da attribuire a P'.

Linear Interpolation

Si considera l'interpolazione polinomiale di grado 1 di due valori scalari f_0 ed f_1 assegnati in corrispondenza delle ascisse *0* e *1*.

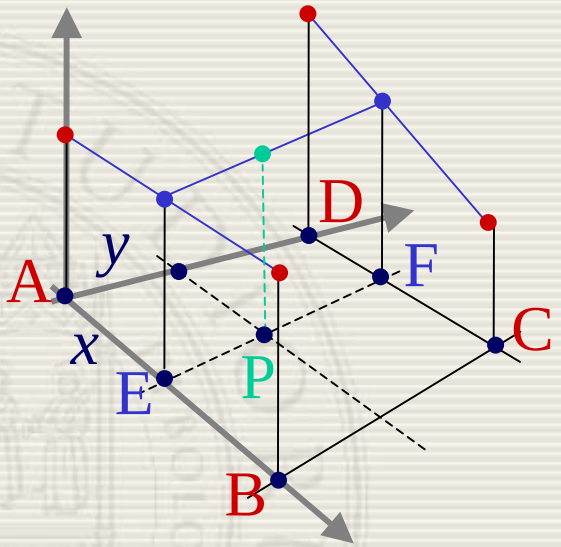


Si consideri la base polinomiale *(1-x)*, *x* per $\mathbf{P_1}$ (spazio dei polinomi lineari) che permette di scrivere l'interpolante lineare come:

$$P(x) = f_0 * (1-x) + f_1 * x$$

Bilinear Interpolation

Linterpolazione bilineare si costruisce mediante una sequenza di interpolazioni lineari. Sia $x=px-ax$ e $y=py-ay$, allora



$$fE = fA * (1-x) + fB * x$$

$$fF = fD * (1-x) + fC * x$$

$$fP = fE * (1-y) + fF * y$$

Od anche, sostituendo:

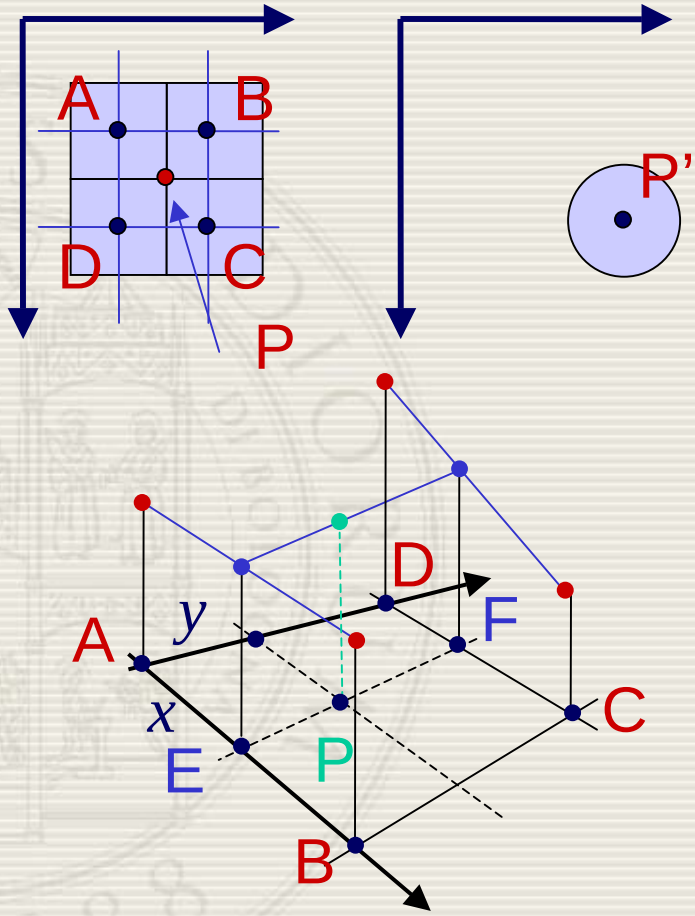
$$fP = fA*(1-x)*(1-y) + fB*x*(1-y) + fD*(1-x)*y + fC*x*y$$

Caso particolare di Bilinear Interpolation

Sia data un'immagine le cui dimensioni siano potenze di 2 e la si voglia scalare di ½.

In questa situazione ogni pixel dell'immagine finale, nella trasformazione inversa si troverà esattamente al centro di quattro pixel originali e il bilinear interpolation si ridurrà ad una media aritmetica dei quattro pixel:

Cioè per $x=y=1/2$:



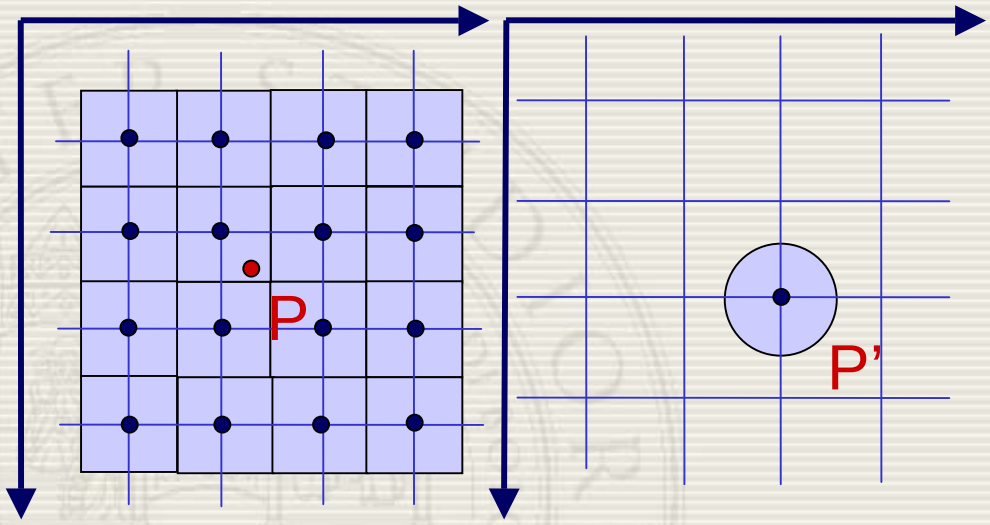
$$fP = fA*(1-x)*(1-y) + fB*x*(1-y) + fD*(1-x)*y + fC*x*y$$

$$= (fA + fB + fC + fD)/4$$ (base del MipMapping)

Bicubic Interpolation

Questa tecnica considera i 4x4 pixel, dell'immagine originale, più vicini a P;

le loro coordinate intere vengono determinate dalle coordinate floating point (px,py) di P:



Le componenti colore di questi 4x4 pixel vengono combinate in modo pesato (interpolazione) per arrivare alle componenti colore da attribuire a P'.

Questo metodo produce immagini migliori rispetto ai due metodi precedenti ed è forse la combinazione ideale fra tempo di calcolo e qualità. Per questo è il metodo standard in molti programmi di editing di immagini come Adobe Photoshop.

Vedi l'esempio:
persp_cube_image_sdl.c

Esercizi

Si propongono i seguenti esercizi da realizzare modificando il codice persp_cube_image_sdl.c nell'archivio messo a disposizione:

1. sostituire la rasterizzazione presente nel codice con l'algorimo presente in scan_conv.c, specializzandolo per triangoli;

2. implementare Z-buffer per rimozione facce nascoste;

3. texturare il cubo/mesh dando le coordinate immagine relative ad ogni vertice.

Lo si chiami persp_zbuffer.c