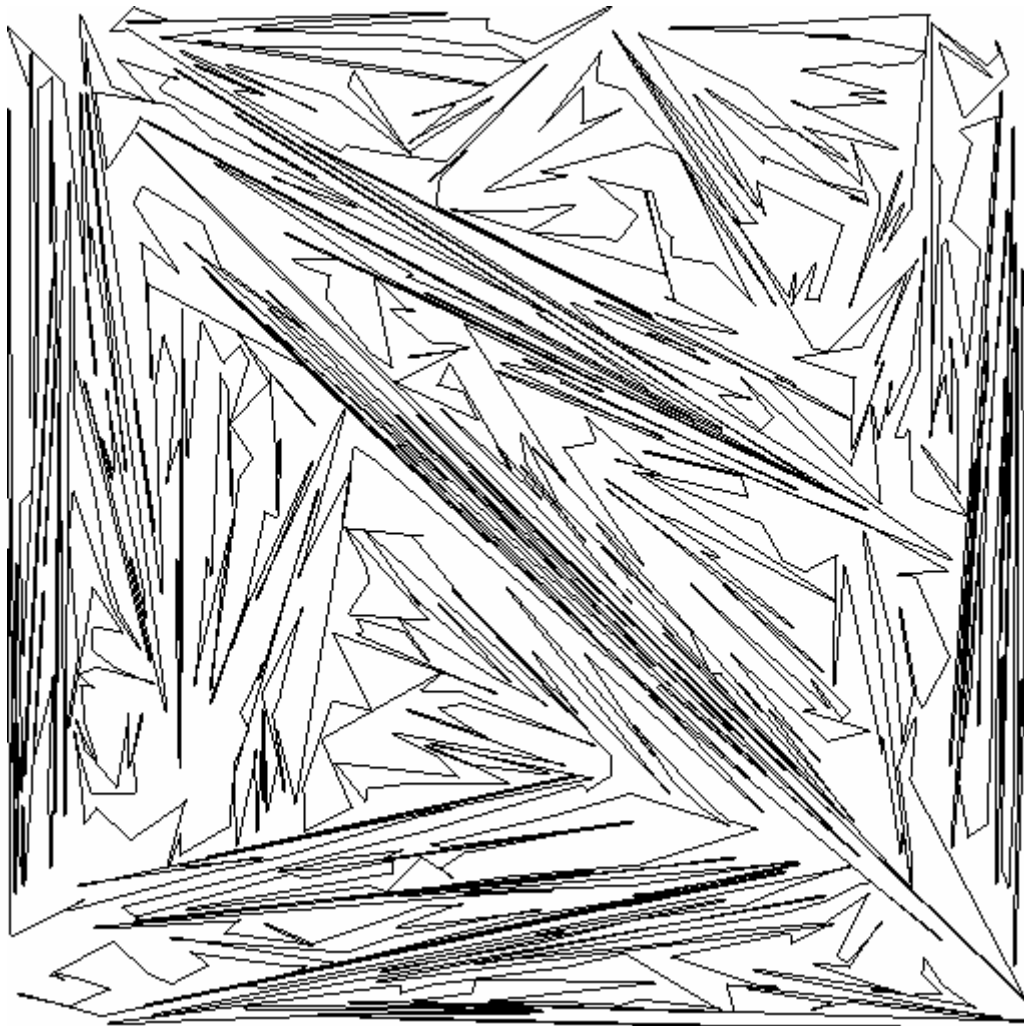


GENERAZIONE DI POLIGONI SEMPLICI ALEATORI (*Simple Closed Polygon*)

Stefano Caschi

Corso di Laurea in Scienze dell'Informazione



Università degli Studi di Udine

Dipartimento di Matematica e Informatica

a.a. 1996-1997

Sommario

1. Introduzione	3
2. Le tecniche possibili	4
3. L'algoritmo GENPOLY	7
3.1. <i>Funzionamento dell'algoritmo</i>	8
3.2. <i>Situazione di non convergenza</i>	10
3.3. <i>Implementazione</i>	11
3.3.1. <i>Scelta del linguaggio di programmazione</i>	11
3.3.2. <i>Visualizzazione grafica</i>	11
3.3.3. <i>Descrizione dell'algoritmo</i>	12
3.4. <i>Costo computazionale</i>	29
4. Sperimentazione	30
5. Conclusioni	34
6. Bibliografia	35
7. Appendice: libreria Hips	36

In copertina: un poligono aleatorio con 1000 lati
generato con l'algoritmo genpoly

1. Introduzione

Tra gli argomenti più discussi nell'ambito della "Geometria Computazionale" vi sono tecniche che trattano intersezioni di poligoni, nuclei di poligoni, triangolazioni, problemi di visibilità.

La ricerca di algoritmi sempre più efficienti, porta inevitabilmente al successivo testing degli stessi tramite l'inserimento delle coordinate dei vertici dei poligoni, per verificarne il comportamento nei casi più generali e particolari, con enorme perdita di tempo da parte del programmatore.

Sarebbe molto utile in questi casi un sistema automatico in grado di generare poligoni semplici (poligoni in cui spigoli non adiacenti hanno intersezione nulla) in modo casuale, e con un numero di lati fissato a piacimento.

Sebbene a prima vista si potrebbe pensare che un algoritmo di questo tipo sia di banale costruzione, in realtà i problemi che si pongono sono di difficile soluzione.

Lo scopo della presente ricerca è di individuare le diverse tecniche per la costruzione di poligoni semplici, illustrandone le caratteristiche, quindi implementare un algoritmo e verificarne l'efficienza in termini di costo computazionale e di convergenza.

Nel Capitolo 2 è fornita una descrizione delle diverse tipologie di costruzione dei poligoni, dando risalto ai pregi e difetti delle stesse, nonché ai problemi connessi all'implementazione.

Nel Capitolo 3 verrà proposto ed implementato un algoritmo, e calcolato il relativo costo computazionale.

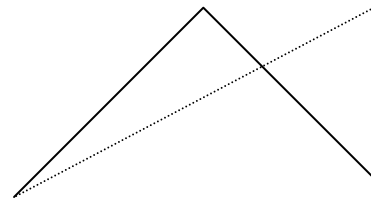
Il Capitolo 4 è invece dedicato al lavoro sperimentale, per la verifica del buon comportamento dell'algoritmo.

2. Le tecniche possibili

Supponiamo di voler costruire un poligono per mezzo di un algoritmo che simuli il comportamento umano secondo lo schema seguente:

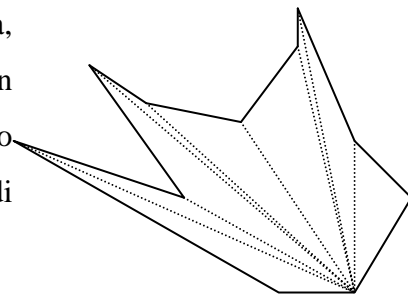
1. parte da una posizione a caso nel piano e costruisce un segmento di lunghezza aleatoria;
2. disegna un altro segmento a partire dal precedente verificando che non vi siano intersezioni;
3. ripete 2. fino al raggiungimento di $n-1$ lati;
4. chiude il poligono se il punto di partenza è raggiungibile senza intersezioni, altrimenti riprende l'esecuzione da 1.

Come si vede nella figura a lato, un tale algoritmo crea problemi di convergenza, già con poligoni di $n > 3$ lati, e la probabilità di non convergenza cresce molto rapidamente all'aumentare del numero dei lati del poligono.



Per evitare problemi di convergenza si potrebbe pensare allora ad un metodo che funzioni nel modo seguente:

dati n punti, si ordinano in base alla direzione della congiungente ciascun punto con quello di ordinata minima, quindi si collegano seguendo l'ordine stabilito; questo è un metodo che garantisce la convergenza, che ha un costo computazionale non elevato, ma che ha il grave difetto di generare soltanto una piccola sottoclasse di poligoni.



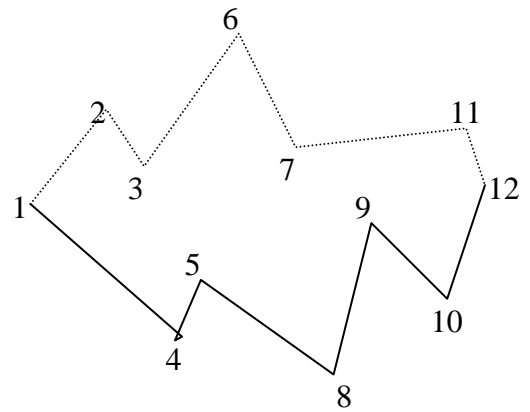
Un algoritmo per la generazione di poligoni a forma “stellata” (si noti che di questa classe fanno parte anche i poligoni convessi) è stato proposto da *L. Deneen e G. Shute [1]* nel 1988.

L'algoritmo ricerca tutti i poligoni “stellati” costruibili sugli n punti dati, aventi nucleo non vuoto; essi sono chiamati “*nondegenerate star-sharped polygons*”.

La complessità è $O(N^5)$.

Il metodo proposto nel 1996 da C. Zhu, G. Sundaram, J. Snoeyink e J.S.B. Mitchell [10], analizza il caso specifico di generazione di poligoni “*x-monotoni*” su un insieme dato di n vertici. I poligoni *x-monotoni* sono costituiti da due catene: “*top chain*” e “*bottom chain*”.

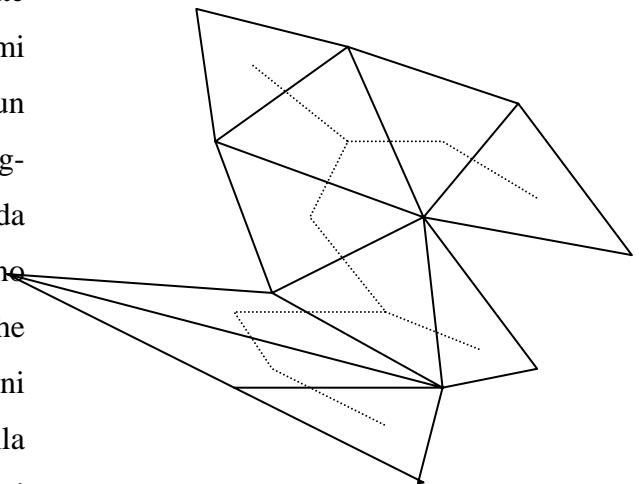
La costruzione è molto semplice, e si basa su un ordinamento delle ascisse dei punti in senso crescente per una catena e decrescente per l'altra.



Un altro metodo potrebbe essere basato sul fatto che ogni poligono è triangolabile, ed il collegamento tra i vari triangoli può essere rappresentato da un albero binario.

Si può allora supporre di costruire un albero binario in cui ogni nodo rappresenta il baricentro di un triangolo, e ad ogni ramo si associa una lunghezza a caso ed un angolo a caso, fino a che si sono raggiunti gli n lati.

In questo caso si pongono dei problemi relativamente alla quantificazione della lunghezza dei rami dell'albero binario, della probabilità di uscita da un nodo di uno o due rami, nonché problemi di floating-point dovuti al calcolo di angoli e lunghezze da trasformare poi in coordinate cartesiane che sono invece numeri Naturali; questo causerebbe anche problemi non indifferenti sulla verifica di intersezioni con gli altri lati, verifica che potrebbe cadere nella conversione delle coordinate dei punti da numeri floating-point a interi.



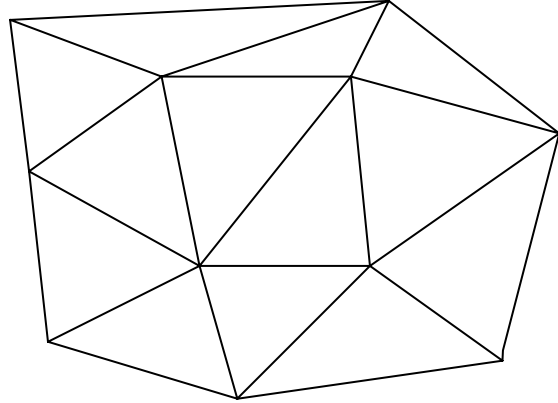
Per evitare i problemi di calcolo relativi a numeri di tipo floating-point, ed anche qualsiasi tipo di vincolo sulla posizione dei vertici del poligono, è conveniente generare i vertici del poligono ancor prima di collegarli fra loro.

Il problema del collegamento tra i punti può essere risolto calcolando il *duale del diagramma di Voronoi* [7] in un tempo $O(N \lg N)$, successivamente è possibile creare il poligono eliminando i lati in eccesso secondo opportuni criteri; ad esempio, partendo dal perimetro esterno si può “tagliare” un

lato (se non è un’*“orecchia”*), e ripetere il procedimento fino a che il numero dei vertici è uguale al numero dei lati.

Questo è un buon metodo, che potrebbe garantire la convergenza, ma che impone implicitamente un vincolo sulla lunghezza dei lati.

Infatti, per come vengono collegati i punti, la lunghezza dei lati dipende dalla minima distanza tra punti vicini.



3. L'algoritmo GENPOLY

L'algoritmo di seguito esposto è stato pensato per evitare ogni tipo di vincolo sulla lunghezza dei lati e posizione dei vertici del poligono, nonché per evitare ogni tipo di calcolo su numeri Razionali, rappresentati nel calcolatore in modo floating-point con una approssimazione dipendente dal numero dei bit gestito dall'Hardware.

Per questo motivo i punti corrispondenti ai vertici del poligono verranno generati in modo casuale, indipendentemente e uniformemente su tutto l'insieme dei possibili valori; solo successivamente verranno collegati dagli spigoli con i criteri che verranno di seguito illustrati.

La costruzione di poligoni semplici ottenuta tramite la connessione di un insieme di n punti nel piano viene detta *poligonizzazione* di punti (*polygonization*).

Si noti che questo problema è differente da quello conosciuto sotto il nome di *Traveling Salesman Problem*, in cui si deve trovare una linea di costo minimo che unisce i punti dati in precedenza, evitando che le linee si intersechino (i punti possono rappresentare le tappe di un percorso che il navigatore deve effettuare, senza passare due volte nello stesso luogo).

Si vuole qui invece trovare un algoritmo in grado di generare qualunque tipo di poligono, di modo che ogni poligono semplice, che sia costruibile sugli n vertici dati, abbia una probabilità $P > 0$ di essere generato dall'algoritmo.

Il numero minimo di poligonizzazioni possibili su un range di n punti è stato ricavato da *R. B. Hayward [3]* ed è esponenziale rispetto a n .

Le condizioni che dovranno essere rispettate sono le seguenti:

- 1) la posizione dei vertici del poligono dovrà essere generata in modo casuale su tutto il range dei possibili valori dei pixel dello schermo; questo sarà garantito dal fatto che i vertici verranno generati prima dei lati;
- 2) a vertici diversi dovranno corrispondere coordinate diverse;
- 3) i lati del poligono dovranno avere una lunghezza l che rispetti la condizione: $1 \leq l \leq \sqrt{x_{\max}^2 + y_{\max}^2}$;
- 4) dovrà essere in ogni modo evitato il calcolo con numeri di tipo floating-point.

3.1. Funzionamento dell'algoritmo

Il meccanismo di funzionamento dell'algoritmo segue questi passi:

1. Si generano n punti a caso sullo schermo;
2. si calcola il poligono convesso sugli n punti;
3. si seleziona un lato a caso;
4. si seleziona un punto a caso;
5. si verifica che il punto e il lato selezionati siano collegabili tra loro senza che vengano coperti o inclusi altri punti dalle due nuove linee, e che queste non intersechino gli altri lati; se così non è, allora riprendi dal passo 4. finchè ci sono punti non ancora esaminati, altrimenti riprendi dal passo 3.
6. Cancella il lato selezionato e sostituiscilo con i nuovi due;
7. se ci sono ancora vertici da collegare torna al passo 4., altrimenti esci dal ciclo.

Questo algoritmo può essere spiegato in modo figurativo come segue:

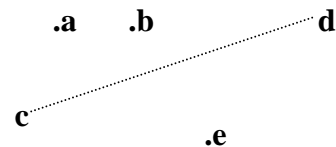
si supponga che i punti siano in realtà dei chiodi piantati in posizioni a caso su una superficie rettangolare; ebbene, se prendiamo un elastico in modo da racchiudere tutti i chiodi, questo formerà una figura che rappresenta il poligono convesso (*Convex Hull*).

Ora, se prendiamo l'elastico in un punto a caso e lo allunghiamo fino a fargli raggiungere un chiodo all'interno, senza che vada a toccare altri chiodi, e ripetiamo il procedimento, evitando delle sovrapposizioni, fino a che tutti i chiodi sono raggiunti, ecco che la figura che si viene a formare è un poligono semplice.

Si noti che il fatto di non poter collegare fra loro due punti che appartengono al Convex Hull nel caso non siano consecutivi, non è un vincolo; infatti se ciò si verificasse, ossia se un lato del poligono semplice fosse una diagonale del Convex Hull, allora i punti verrebbero così suddivisi in due insiemi non collegabili fra loro se non con una intersezione.

L'algoritmo può essere ottimizzato modificando il passo 5. nel modo seguente:

5. si verifica che il punto e il lato selezionati siano collegabili tra loro senza che vengano coperti o inclusi altri punti dalle due nuove linee, e che queste non intersechino gli altri lati; se così non è,



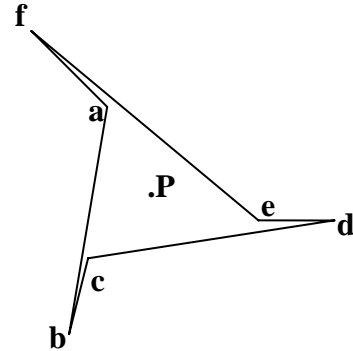
allora riprendi dal passo 4. finchè ci sono punti non ancora esaminati, altrimenti marcalo perchè sarà un lato definitivo del poligono, poi riprendi dal passo 3.

Infatti, se un lato non è collegabile ad alcun punto all' i -esimo passo, non lo sarà neppure nei passi $(i+k)$ -esimi ($0 < k < n-i$) poichè i punti sono già tutti generati.

3.2. Situazione di non convergenza

Con un numero di lati $n \geq 7$, può accadere che l'algoritmo non converga a causa di una situazione di *deadlock*.

Come si vede nell'esempio a fianco, il lato (a,b) non può essere collegato al punto **P** perchè non visibile a causa del lato (c,d), a sua volta non visibile a causa del lato (e,f), coperto a sua volta dal primo lato.



Per risolvere questo problema si potrebbe pensare di riprendere l'esecuzione del programma dal passo k -esimo, decrementando k ogni volta che non si riesce a collegare tutti i punti.

Essendo però la procedura incentrata su algoritmi *random*, questo non garantirebbe comunque la convergenza, ed equivarrebbe a far ripartire l'algoritmo dall'inizio.

Una soluzione parrebbe essere quella di spostare il punto **P** lungo una direzione, fino ad essere visibile da un lato, ma in questo modo si violerebbe la condizione 1) che era stata imposta proprio per evitare ogni tipo di vincolo sulla realizzazione del poligono.

Per questo motivo si è optato per una terminazione controllata del programma.

Il programma prevede in questo caso la stampa sullo schermo di un messaggio di errore.

Il numero dei casi di non convergenza, come si può intuire, cresce all'aumentare del numero dei lati del poligono.

La probabilità che si verifichi un simile evento è stata testata in modo sperimentale, ed i risultati sono visibili nel successivo Capitolo 4.

3.3. Implementazione

3.3.1. Scelta del linguaggio di programmazione

Per l'implementazione si è scelto il Linguaggio C, nella versione Standard ANSI.

Molte caratteristiche del C derivano dal linguaggio BCPL, sviluppato da *Martin Richards*, e dal linguaggio B ideato da *Thompson* nel 1970 per il primo sistema UNIX.

Anche il C, progettato da *Dennis Ritchie* [4], è stato in origine implementato su un sistema operativo UNIX.

Tuttavia esso non è stato progettato per alcun hardware o sistema operativo particolare.

Nel 1983, l'Istituto Nazionale Americano per gli Standard (ANSI) ha costituito un comitato per la definizione aggiornata e completa del C non ambigua e non dipendente dalla macchina.

Il risultato di questo lavoro è lo standard ANSI o "ANSI C" approvato nel 1989.

Un grosso contributo derivante dallo standard è la specifica di una libreria standard, con un insieme esteso di funzioni di Input/Output, gestione della memoria, manipolazione di stringhe ecc. Tutti i programmi che utilizzano questa libreria hanno la garanzia della compatibilità di comportamento su ogni sistema.

Il C è un linguaggio di programmazione di uso generale, relativamente "a basso livello"; questo significa che tratta gli oggetti in modo molto simile a quello utilizzato dalla maggior parte dei calcolatori.

Anche se vengono sfruttate le potenzialità di molti calcolatori, il C è indipendente dall'architettura della macchina. E' quindi possibile costruire programmi portabili, ossia programmi che possano essere eseguiti, senza bisogno di modifiche, su hardware differenti.

3.3.2. Visualizzazione grafica

Per quanto concerne la visualizzazione grafica, si è scelto lo standard *Hips* (Human Information Processing Laboratory Image Processing System) la cui documentazione è disponibile *sul World Wide Web* all'indirizzo "<http://www.cns.nyu.edu/home/msl/hipsdescr.cgi>".

Hips è un sistema software per l'elaborazione di immagini che opera in ambiente UNIX, sviluppato inizialmente da *Michael Landy*, *Yoav Cohen* e *George Sperling* presso l'Università di New York, Dipartimento di Psicologia - Human Information Processing Laboratory.

Nonostante l'obiettivo iniziale che fece nascere *Hips* fosse la ricerca nell'ambito dell'*ASL* (American Sign Language), un linguaggio composto di gesti per la comunicazione tra/con persone udiolese,

quando il sistema venne presentato e reso disponibile ad altri ricercatori, apparve evidente che molti degli strumenti software di Hips potevano essere utilizzati anche per numerose altre applicazioni nell'ambito del trattamento di segnali ed immagini, e più in generale nelle scienze della visione.

Per questo motivo gli autori cercarono di creare un sistema molto flessibile e facile da usare.

I vantaggi di Hips sono i seguenti:

- Stesura in "C"
- Uso di comandi UNIX-like o di procedure interfacciabili ad altri programmi "C".
- Totale indipendenza dall'Hardware per il 98% del codice.
- Disponibilità di driver per numerosi sistemi di visualizzazione.
- Facilità d'apprendimento, uso, estensione.
- Disponibilità di più di 200 procedure di trasformazione di immagini singole o sequenze.
- Disponibilità del codice sorgente, insieme alle pagine di manuale in linea per ogni comando o procedura

Il fatto di poter essere facilmente adattato ad altri Sistemi Operativi, ne ha favorito una larga diffusione, e si trova oggi in uso su una moltitudine di piattaforme tra cui Vax e MicroVax, Sun, Apollo, Masscomp, NCR Tower, Iris, IBM AT, Macintosh II.

Avere quindi un programma che preveda l'output in formato Hips, significa poter visualizzare l'immagine sullo schermo con la semplice linea di comando "genpoly | xhips", nonché poter salvare l'immagine in altri formati grafici, o elaborarla tramite i filtri e trasformazioni rese disponibili dal pacchetto (tutte le funzioni sono visibili in Appendice).

Nel caso la visualizzazione grafica non fosse possibile, viene prevista la stampa delle coordinate dei punti in formato ascii, seguendo l'ordine dei vertici antiorario.

3.3.3. Descrizione dell'algoritmo

Gli strumenti di base della programmazione in C nell'ambito della Geometria Computazionale, sono forniti da *J. O'Rourke* [5], e sono disponibili anche nella rete *Internet* presso l'indirizzo "ftp://grendel.csc.smith.edu/pub/compgeom".

A questi ci si è ispirati nella stesura del codice.

```
/*  
*****  
* GENPOLY: GENERAZIONE DI POLIGONI ALEATORI *  
*****  
*/
```

```

*****/

#include <time.h> /* Consente di leggere l'ora di sistema */
#include <stdio.h> /* Libreria standard i/o */
#include <stdlib.h> /* Libreria di conversione caratteri */
enum { X,Y,DIM }; /* def. spazio bidimensionale */
typedef enum { FALSE,TRUE } bool;
typedef int tPointi [DIM];
typedef tPointi *tPolygoni;

```

Differentemente da come indicato da *J. O'Rourke* [5], il tipo `tPolygoni` è definito come un puntatore a interi; lo spazio di memoria necessario potrà così essere allocato in funzione degli n lati del poligono.

Solo in questo modo si può garantire un costo spaziale di tipo lineare.

```

#if !defined (_SYS_TYPES_H)
    typedef unsigned int    uint;
    typedef unsigned long   ulong;
#endif

```

I tipi *unsigned int* e *unsigned long* vengono ridefiniti in modo più compatto come *uint* e *ulong* (semprechè non siano già stati ridefiniti in precedenza).

```

#define NPOINTS      100 /*Numero max vertici poligono di default*/
#define XMAX        256 /* Dimensione X della finestra di default */
#define YMAX        256 /* Dimensione Y */
#define max(A,B)    ((A) > (B) ? (A) : (B))
#define min(A,B)    ((A) < (B) ? (A) : (B))
#define abs(A)      ((A) > 0 ? (A) : (-(A)))
#define absdiff(A,B) ((A) > (B) ? (A) - (B) : (B) - (A))
#define C_POINT 255 /* Colore dei punti sul video */
#define C_LINE 127 /* Colore delle linee sul video */
#define C_NULL 0 /* Colore dello sfondo */

```

Nel *preprocessing*, fase che precede la compilazione vera e propria, viene attuata una sostituzione delle macro all'interno del testo del programma, quindi ad esempio $\max(x,y)$ viene espanso così come indicato nella definizione in $(x>y ? x : y)$.

```
/**
 * Area doppia del triangolo
 */
int Area2 (tPointi a, tPointi b, tPointi c) {
return
    a[X]*b[Y]-a[Y]*b[X]+
    b[X]*c[Y]-b[Y]*c[X]+
    c[X]*a[Y]-c[Y]*a[X];
}
```

Questa funzione calcola l'area doppia del triangolo definito dai punti a,b,c.

Il calcolo viene fatto in forma vettoriale, per evitare errori dovuti a rappresentazione in forma floating-point.

Il calcolo dall'area viene ricondotto alla soluzione del seguente determinante:

$$\begin{vmatrix} Bx- & Cx-Ax \\ Ax \\ By- & Cy-Ay \\ Ay \end{vmatrix} = \begin{vmatrix} Ax & Bx & Cx \\ Ay & By & Cy \\ 1 & 1 & 1 \end{vmatrix} = \begin{vmatrix} Bx & Cx \\ By & Cy \end{vmatrix} - \begin{vmatrix} Ax & Cx \\ Ay & Cy \end{vmatrix} + \begin{vmatrix} Ax & Bx \\ Ay & By \end{vmatrix}$$

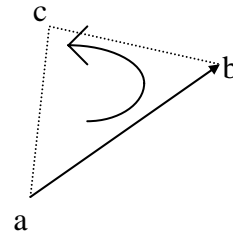
L'area è positiva se sovrapponendo il vettore $\mathbf{b-a}$ al vettore $\mathbf{c-a}$ in senso antiorario l'angolo è minore di π , altrimenti è negativa.

```
/**
 * Left=TRUE se il punto c e' a sx rispetto al vettore a -> b
 * Collinear=TRUE se a,b,c sono punti collineari
 */

#define Left(a,b,c)      (Area2((a),(b),(c))>0)
#define LeftOn(a,b,c)   (Area2((a),(b),(c))>=0)
```

```
#define Collinear(a,b,c) (Area2((a),(b),(c))==0)
```

il punto c è a sinistra, se percorrendo la retta orientata a-b nella sua direzione, il punto è a sinistra, oppure se percorrendo il triangolo a,b,c la percorrenza è antioraria.



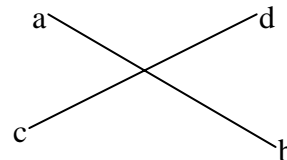
```
/**
```

```
 * Intersezione propria
```

```
 */
```

```
bool IntersectProp (tPointi a, tPointi b, tPointi c, tPointi d) {
return
    ((Left(a,b,c)&&!LeftOn(a,b,d)) ||
(!LeftOn(a,b,c)&&Left(a,b,d)))
    &&
    ((Left(c,d,a)&&!LeftOn(c,d,b)) ||
(!LeftOn(c,d,a)&&Left(c,d,b)));
}
```

L'intersezione è propria se esiste un punto comune interno ad entrambi i segmenti.



```
/**
```

```
 * Between=TRUE se il punto c si trova sul segmento (a,b)
```

```
 */
```

```
bool Between (tPointi a, tPointi b, tPointi c) {
if (!Collinear(a,b,c))
    return FALSE;
if (a[X]!=b[X])
    return
    (((a[X]<=c[X])&&(c[X]<=b[X])) ||
    ((b[X]<=c[X])&&(c[X]<=a[X])));
else
    return
```

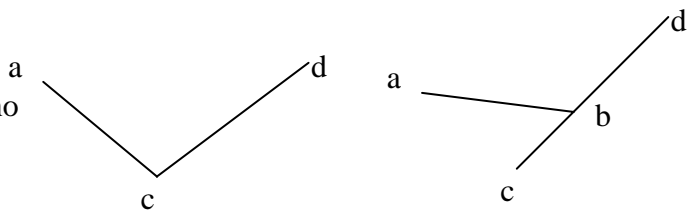
```

        ((a[Y]<=c[Y])&&(c[Y]<=b[Y])) ||
        ((b[Y]<=c[Y])&&(c[Y]<=a[Y]));
    }

/**
 * Intersect=TRUE se la linea a-b ha un punto di intersezione con
 c-d
 */
bool Intersect (tPointi a, tPointi b, tPointi c, tPointi d) {
return    IntersectProp(a,b,c,d) ||
        Between(a,b,c) ||
        Between(a,b,d) ||
        Between(c,d,a) ||
        Between(c,d,b);
}

```

C'è intersezione tra ab e cd, se c'è una intersezione propria, oppure se si verificano i casi riportati a lato.



```

/**
 * Crea una copia del punto a nel punto b
 */
CopyPoint (tPointi a, tPointi b) {
register uint dim;
for (dim=0; dim<DIM; dim++)
    b[dim]=a[dim];
}

```

Viene fatta una copia delle coordinate del punto **a**, nel punto **b**.

La variabile *dim* è forzatamente posta in un registro interno al microprocessore in modo che le somme sugli spiazziamenti vengano eseguite nel modo più rapido possibile.

```

/**

```



```

* Crea una copia del poligono P nel poligono Pcopy
*/
CopyPoly (int n, tPolygoni P, tPolygoni Pcopy) {
register uint k;
for (k=0;k<n;k++)
    CopyPoint(P[k],Pcopy[k]);
}

/**
* Aggiunge al poligono il punto v nella posizione i
*/
void CreateEar(uint i, uint v, uint *pnpoints, tPolygoni Points,
uint *pnlati, tPolygoni Poly) {
register uint k;
for (k=++*pnlati; k>i+1; k--)
    CopyPoint(Poly[k-1],Poly[k]);
CopyPoint(Points[v],Poly[i+1]);
for (--*pnpoints,k=v; k<=*pnpoints; k++)
    CopyPoint(Points[k+1],Points[k]);
}

```

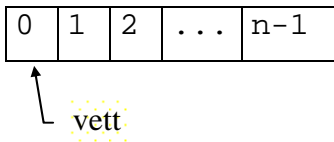
Per aggiungere un nuovo punto al poligono, viene prima creato lo spazio nella posizione corretta all'interno del vettore Poly, quindi inserito il punto e compattato il vettore che contiene la lista dei punti restanti.

```

/**
* Crea un vettore di n indici
*/
void IndPos(uint *vett, uint n) {
register uint k;
for(k=0;k<n;k++)
    vett[k]=k;
}

```

Viene creato un vettore che contiene gli indici da 0 a $n-1$.

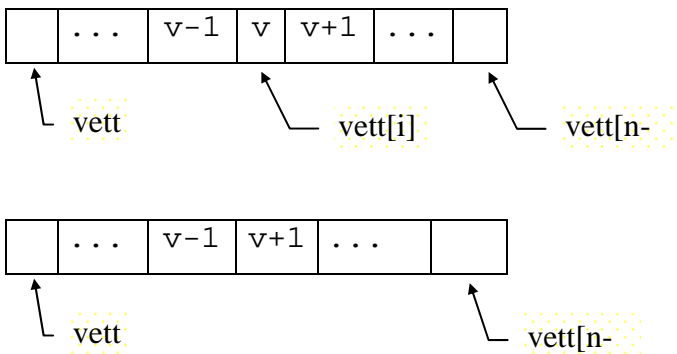


```

/**
 * Restituisce una posizione casuale all'interno del vettore e lo
 comprime
 */
int RandPos (uint *vett, uint *n) {
register uint i=rand()*rand()/*n;
register uint v=vett[i];
for(--*n;i<=*n;i++)
    vett[i]=vett[i+1];
return v;
}

```

Questa procedura permette di selezionare un elemento a caso dal vettore che contiene gli indici, ed eliminarlo dal vettore stesso; in questo modo ogni elemento del vettore di indici viene selezionato una sola volta.



```

/**
 * Genera punti pseudocasuali nel range (0,xmax),(0,ymax)
 */
void Randomize (tPolygoni P, int n, int xmax, int ymax) {
uint k,posmax=xmax*ymax;
uint *pos=calloc(posmax,sizeof(int));
IndPos(pos,posmax);
while (n--) {

```

```

        k=RandPos(pos,&posmax);
        P[n][X]=k%xmax;
        P[n][Y]=k/xmax;
    }
free(pos);
}

/**
 * Invia l'immagine allo standard output in formato Hips
 */
void OutHips (char *Screen, int xmax, int ymax) {
register uint k;
printf("HIPS\n\n\n1\n\n%d\n%d\n%d\n%d\n0\n0\n0\n1\n0\n1\n\n0\n
0\n",ymax,xmax,ymax,xmax);
for (k=0;k<xmax*ymax;k++)
    putchar(Screen[k]);
}

```

L'immagine consiste in uno Header che contiene le informazioni sull'immagine (altezza, larghezza, numero di colori, ecc.), in questo caso stampato in una sola riga formattata secondo lo standard Hips, e in un corpo che contiene colori e posizioni dei pixel, in questo caso non compressi.

```

/**
 * Disegna una linea tra due punti
 */
void DrawLine (tPointi a, tPointi b, char *Screen, int xmax, int
ymax) {
int i,
    x1=min(a[X],b[X]),
    x2=max(a[X],b[X]),
    y1=min(a[Y],b[Y]),
    y2=max(a[Y],b[Y]),
    dx=b[X]-a[X],
    dy=b[Y]-a[Y];
if ((x2-x1)>(y2-y1))
    for(i=x1;i<=x2;i++)

```

```

        Screen[i+((i-a[X])*dy/dx+a[Y])*xmax]=C_LINE;
else
    for(i=y1;i<=y2;i++)
        Screen[(i-a[Y])*dx/dy+a[X]+i*xmax]=C_LINE;
}

```

La linea tra due punti viene disegnata sullo schermo con una approssimazione di $\frac{1}{2}$ pixel.

Essendo una rappresentazione “imprecisa”, viene data una luminosità differente a quella che avranno i vertici del poligono.

```

/**
 * Pulisce lo schermo prima di disegnare
 */
void ClearScreen (char *Screen, int xmax, int ymax) {
register ulong k;
for(k=0; k<xmax*ymax; k++)
    Screen[k]=C_NULL;
}

```

Questa procedura non sarebbe necessaria se lo sfondo avesse colore C_NULL=0, in quanto il vettore puntato da Screen è già stato azzerato nella fase di allocazione dinamica (malloc).

```

/**
 * Disegna i punti sullo schermo
 */
void DrawPoints (int n, tPolygoni P, char *Screen, int xmax, int
ymax) {
while (n--)
    Screen[P[n][X]+P[n][Y]*xmax]=C_POINT;
}

```

```

/**
 * Disegna un poligono
 */

```

```

void DrawPoly (int n, tPolygoni P, char *Screen, int xmax, int
ymax) {
register uint i;
for(i=1;i<n;i++)
    DrawLine(P[i-1],P[i],Screen,xmax,ymax);
DrawLine(P[n-1],P[0],Screen,xmax,ymax); /* chiusura poligono */
}

/**
 * Compatta un vettore di punti
 */
void Compatta (int i, int *pn, tPolygoni P) {
register uint k;
for (--*pn,k=i; k<*pn; k++)
    CopyPoint(P[k+1],P[k]);
}

/**
 * SwapPoint: scambia 2 punti
 */
void SwapPoint (tPointi a, tPointi b) {
tPointi t;
CopyPoint(a,t);
CopyPoint(b,a);
CopyPoint(t,b);
}

/**
 * DownHeap: procedura chiamata da HeapSort
 */
void DownHeap(int k, tPolygoni P, int n) {
register int i;
tPointi v;
CopyPoint(P[k],v);
while(k<=n>>1) {
    i=k<<1;

```

```

    if ((i<n)&&!Left(P[0],P[i],P[i+1]))
        i++;
    if (Left(P[0],v,P[i])) goto rtn;
    CopyPoint(P[i],P[k]);
    k=i;
}
rtn:
CopyPoint(v,P[k]);
}

/**
 * HeapSort: procedura di ordinamento vettore in base all'angolo
polare
 */
void HeapSort (tPolygoni P, int npoints) {
register int n,k,min;
for(min=0,n=1;n<npoints;n++)
    if(P[n][Y]<P[min][Y])
        min=n;
SwapPoint(P[0],P[min]);
for(n=npoints-1,k=npoints>>1; k>=1; k--)
    DownHeap(k,P,n);
do {
    SwapPoint(P[1],P[n]);
    DownHeap(1,P,--n);
}
while (n>1);
}

```

La procedura HeapSort è la versione presentata nel 1983 da *Robert Sedgewick* [8] in “Algorithms”, convertita in C, ottimizzata tramite le potenzialità del linguaggio, e adattata al caso specifico.

```

/**
 * Costruisce il poligono convesso sui *pnpoints punti di P
(Graham)
 */

```

```

int ConvexHull (tPolygona P, uint *pnpoints, tPolygona Poly) {
register uint n,i;
HeapSort(P,*pnpoints);
for(n=2,i=3;i<*pnpoints;i++,n++) {
    while (!LeftOn(P[n],P[n-1],P[i]))
        n--;
    SwapPoint(P[i],P[n-1]);
}
CopyPoly(++n,P,Poly);
for(i=n; i<*pnpoints; i++)
    CopyPoint(P[i],P[i-n]);
*pnpoints-=n;
return n;
}

```

Questo metodo, che è stato inventato da *R.L. Graham* [2] nel 1972, garantisce delle prestazioni temporali dipendenti solo dal tempo necessario all'ordinamento dei punti.

```

/**
 * GoodEar=TRUE se il punto p puo' essere aggiunto al poligono tra
v e v+1
 */
bool GoodEar (int v, int p, int npoints, tPolygona Points, int
nlati, tPolygona Poly) {
register int k,k1;
register int v1=(v+1)%nlati;
if (Collinear(Points[p],Poly[v],Poly[v1]))
    return FALSE;
for(k=0;k<nlati;k++) {
    k1=(k+1)%nlati;
    if((Intersect(Poly[v],Points[p],Poly[k],Poly[k1])&&(k!=v)&&(k1
=v))
        ||(Intersect(Points[p],Poly[v1],Poly[k],Poly[k1])&&(k!=v1)&&(k1
!=v1)))
        return FALSE;
}
}

```

```

for(k=0;k<p;k++)
    if(LeftOn(Points[p],Poly[v1],Points[k])
        &&LeftOn(Poly[v],Points[p],Points[k])
        &&LeftOn(Poly[v1],Poly[v],Points[k]))
        return FALSE;
for(k=p+1;k<npoints;k++)
    if(LeftOn(Points[p],Poly[v1],Points[k])
        &&LeftOn(Poly[v],Points[p],Points[k])
        &&LeftOn(Poly[v1],Poly[v],Points[k]))
        return FALSE;
return TRUE;
}

```

L' "orecchia" va bene se non ci sono intersezioni con gli altri lati del poligono, e se non ci sono punti che vengono inclusi.

```

/**
 * Aggiunge un lato al poligono
 */
int CreatePoly(uint *lato, uint *pn, uint *pnpoints, tPolygoni
Points, uint *pnlati, tPolygoni Poly) {
uint i,rp,rl,npoints=0;
uint *point=(uint *)calloc(*pnpoints,sizeof(int));
do {
    if (npoints==0) {
        if ((*pn)==0) return (-1); /* non converge */
        npoints=*pnpoints;
        IndPos(point,npoints); /* inizializza il vett. di indici
*/
        rl=RandPos(lato,pn);
    }
    rp=RandPos(point,&npoints);
}
while (!GoodEar(rl,rp,*pnpoints,Points,*pnlati,Poly));
CreateEar(rl,rp,pnpoints,Points,pnlati,Poly);
for(i=0;i<(*pn);i++) /* aggiorna gli indici > rl */

```



```

        if(lato[i]>r1)
            lato[i]++;
lato[(*pn)++]=r1; /* aggiunge gli indici dei 2 nuovi lati */
lato[(*pn)++]=r1+1;
free (point);
}

/**
 * ParseArgs: tratta i parametri della riga di comando
 */
int ParseArgs (int argc, char *argv[], bool *paflag, uint *npoints,
uint *xmax, uint *ymax) {
char c,*prgname=argv[0];
newoption:
while(--argc>0 && (*++argv)[0]!='-')
    while (c=*++argv[0])
        switch (c) {
        case 'n':
            *npoints=atoi(++*argv);
            if (*npoints==FALSE) {
                *npoints=atoi(*++argv);
                --argc;
            }
            goto newoption;
        case 'a':
            *paflag=TRUE;
            break;
        case 's':
            *xmax=*ymax=atoi(++*argv);
            if (*xmax==FALSE) {
                *xmax=*ymax=atoi(*++argv);
                --argc;
            }
            if (argc>1)
                *ymax=atoi(*++argv);
            if (*ymax==FALSE) {

```

```

        *--argv;
        *ymax=*xmax;
    }
    else
        --argc;
        goto newoption;
    default:
        fprintf(stderr,"%s: illegal option --
%c\n",prgname,c);
        fprintf(stderr,"Usage: %s [-n nedges] [-a | -s xmax
[ymax]]\n",prgname);
        return -1;
    }
}

```

Questa procedura tratta i parametri che vengono forniti al comando, secondo lo standard Unix.

Ad esempio, i seguenti comandi sono equivalenti:

```

genpoly -n40 -s256 256
genpoly -n 40 -s 256 256
genpoly -s 256 256 -n40
oppure
genpoly -an40
genpoly -n 40 -a

```

```

/**

```

```

 * Output in formato testo (coordinata X, coordinata Y) per ogni
 punto

```

```

 */

```

```

void Outascii (register int k, tPolygoni Poly) {
while (k--)
    printf("%d,%d\n",Poly[k][X],Poly[k][Y]);
}

```

```

/*****

```

```

 * M A I N

```

```

 */

```

```

main (int argc, char *argv[]) {
bool aflag=FALSE; /* output ascii */
uint n,npoints,nlati,*lato,
      xmax=XMAX,ymax=YMAX;
tPolygوني Poly,Points;
srand(time(NULL));

```

Il seme iniziale della funzione rand() viene settato uguale all'ora di sistema

```

npoints=rand()%(NPOINTS-3)+3; /* numero casuale 3-NPOINTS lati */
if (ParseArgs(argc,argv,&aflag,&npoints,&xmax,&ymax)==-1)
    return FALSE;
lato=(uint *)calloc(npoints,sizeof(int));
Points=(tPolygوني)calloc(npoints,sizeof(tPointi));
Poly=(tPolygوني)calloc(npoints,sizeof(tPointi));
Randomize(Points,npoints,xmax,ymax);
nlati=ConvexHull(Points,&npoints,Poly);
IndPos(lato,n=nlati);
while (npoints)
    if (CreatePoly(lato,&n,&npoints,Points,&nlati,Poly)==-1) {
        fprintf(stderr,"L'algoritmo non converge!\n");
        break;
    }
free (lato);
if (aflag)
    Outascii(nlati,Poly); /* Output in formato coordinate ascii
*/
else {
    char *Screen=malloc(xmax*ymax);
    ClearScreen(Screen,xmax,ymax);
    DrawPoly(nlati,Poly,Screen,xmax,ymax); /* Disegna il
poligono */
    DrawPoints(nlati,Poly,Screen,xmax,ymax); /* Disegna i vertici
*/
    if (npoints) /* Se l'algoritmo non converge, mancano da
disegnare */

```

```

        DrawPoints(npoints,Points,Screen,xmax,ymax);/*questi
punti*/
        OutHips(Screen,xmax,ymax);      /* Output in formato standard
Hips */
        free(Screen);
    }
free (Points);
free (Poly);
return TRUE;
}

```

Lo spazio di memoria occorrente alla rappresentazione grafica viene allocata solo se necessario, e rilasciata successivamente da free(Screen).

3.4. Costo computazionale

Spazio:

O(N) per la memorizzazione degli N punti.

A questo si deve aggiungere O(XMAX*YMAX) nel caso di rappresentazione grafica, per la memorizzazione dei pixel.

Tempo massimo:

Se H è il numero dei punti che costituiscono il Convex Hull, numero che può variare da 3 a N, la procedura GoodEar viene invocata al più per ogni lato (N) e per ogni punto $K \leq (N-H)$ che rimane all'interno del poligono; la procedura stessa effettua $(1+nlati+npoints-1)=N$ confronti.

In totale O(N³)

Tempo minimo:

Nel 1994 P. Valtr [5] ha dimostrato che:

“n punti aleatori selezionati in modo uniforme e indipendente da un parallelogramma sono in posizione convessa con probabilità

$$\frac{\binom{2n-2}{n-1}^2}{n!^2}$$

Un insieme finito di punti nel piano è detto convesso se essi sono vertici di un poligono convesso.”

Nel caso specifico il parallelogramma è lo schermo di rappresentazione di dimensioni XMAX*YMAX, e i punti sono i vertici del poligono.

Quindi con probabilità $P=\frac{(2N-2)!}{((N-1)!^2 * N!)^2}$ il poligono è convesso (N=H).

In questo caso il tempo di calcolo si riduce a quello necessario per il calcolo del Convex Hull, che a sua volta dipende dal tempo necessario ad eseguire l'ordinamento, eseguito con *HeapSort* che, come è stato dimostrato in dettaglio da R. Schaffer e R. Sedgewick [10], ha complessità O(N lgN)

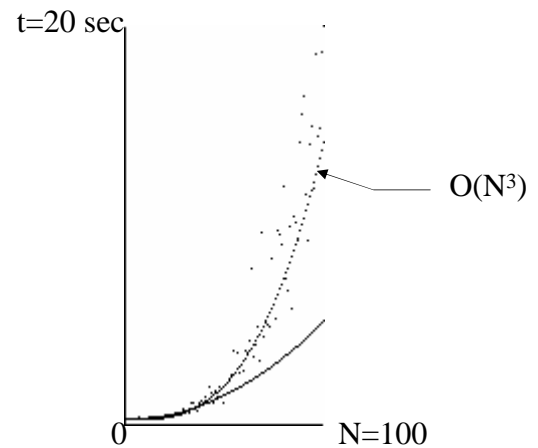
4. Sperimentazione

Il test è stato eseguito per un numero di lati del poligono variabile fra 3 e 100.

Poligoni superiori a 100 lati, oltre a richiedere tempi di elaborazione superiori ai 10 sec., cominciano ad avere problemi di rappresentazione grafica (i lati sono talmente vicini tra loro che non si riesce a capire la forma del poligono).

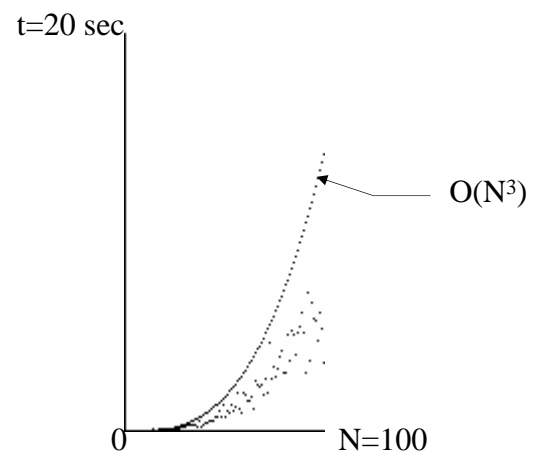
Si riportano qui in un grafico i tempi occorsi all'algoritmo per l'elaborazione dei dati, dopo una prima stesura.

A fianco il grafico relativo ai tempi di CPU necessari ad un sistema SUN SparcStation con Sistema Operativo UNIX, per il calcolo del poligono con numero di lati variabile da 3 a 100. Come si può vedere, la complessità massima supera $O(N^3)$.



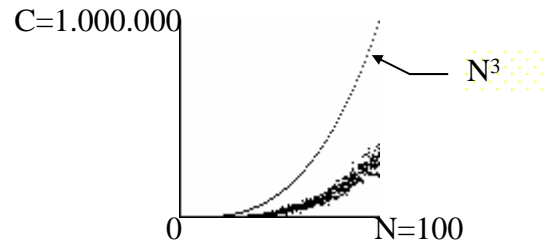
Ecco qui sotto, i tempi di CPU dopo aver ottimizzato l'algoritmo (modificando il passo 5. come indicato nel III capitolo), ed aver raggiunto un tempo massimo di elaborazione di complessità cubica.

Il limite superiore è rappresentato dalla curva $O(N^3)$, e il limite minimo da $O(N \lg N)$.



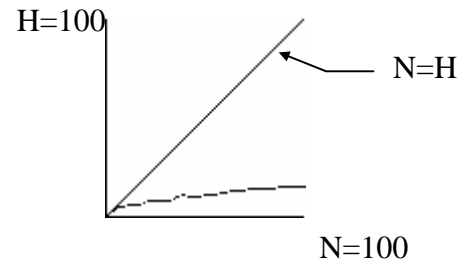
Questa è una verifica sperimentale più precisa e indipendente dall'Hardware.

Si è fatto ciclare il programma per 10 volte, con numero di lati da 3 a 100, contando le chiamate alla funzione GoodEar, moltiplicate per la sua complessità computazionale che è $(1+nlati+npoints-1)=N$



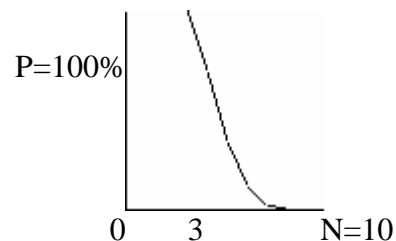
Si è eseguito un test anche per vedere quanto incida il calcolo del Convex Hull rispetto al tempo totale di elaborazione, in funzione del numero dei punti del poligono.

Il grafico a fianco rappresenta il numero dei lati calcolati inizialmente dal Convex Hull in $O(N \lg N)$ rispetto al numero totale di lati del poligono.

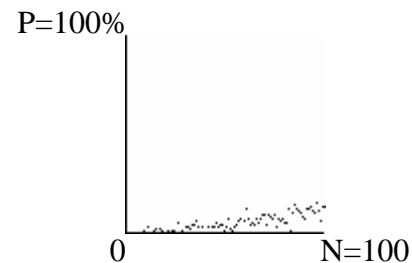


A fianco il grafico che rappresenta la probabilità P che un poligono di N lati sia convesso, calcolata come indicato da $P. Valtr [5]$.

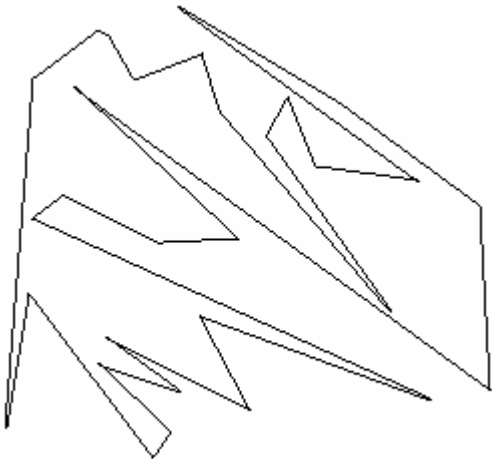
Come si può notare, con un poligono di $N > 10$ lati la convessità è molto improbabile.



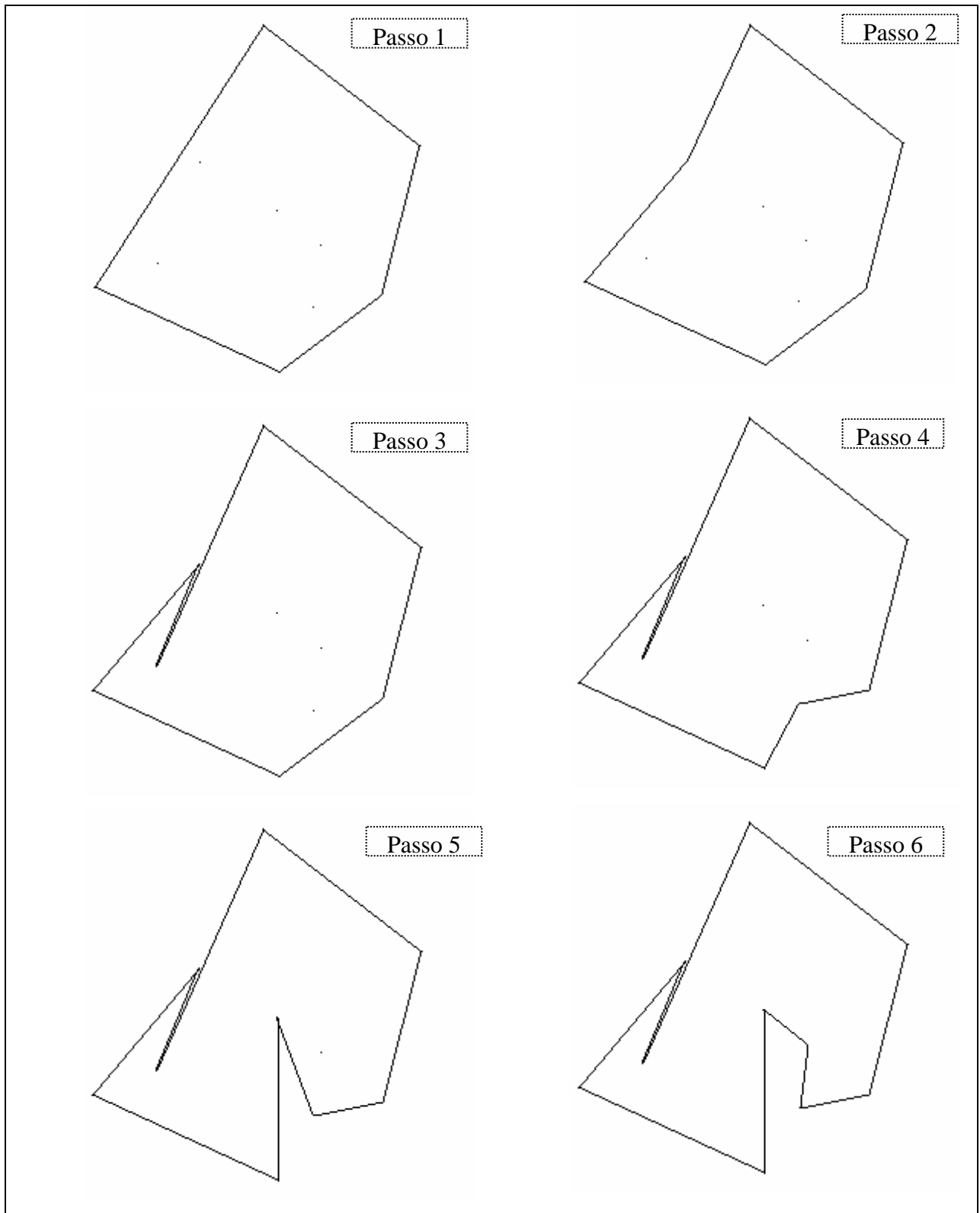
Per testare quale sia la probabilità che l'algoritmo non converga, è stato eseguito un conteggio automatico dei casi di non convergenza su un totale di 100 poligoni, e per ogni n -agone con $7 \leq n \leq 100$



Si riporta qui sotto un esempio concreto di funzionamento del programma:
viene realizzato un poligono aleatorio con 30 lati.

Output in formato ascii: genpoly -an30	Output in formato grafico: genpoly -n30 xhips
<pre> 92,33 216,158 56,67 165,165 97,53 151,200 60,80 140,180 132,44 202,93 107,91 117,193 222,49 108,222 23,139 74,209 39,151 62,230 88,127 55,233 126,129 24,210 44,205 10,35 252,54 21,102 247,145 83,20 176,197 96,245 </pre>	

Si può vedere ora come avviene la costruzione passo dopo passo di un poligono aleatorio di $n=10$ lati:



5. Conclusioni

Nonostante non si sia riusciti a trovare un metodo esente da problemi di convergenza, e che allo stesso tempo non imponga alcun tipo di vincolo, questo algoritmo rappresenta comunque un utile strumento per la verifica di correttezza di alcuni programmi.

Essendo possibile, anche se poco probabile, la non convergenza, questa viene comunque individuata dall'algoritmo, che termina con un messaggio stampato a video.

Si noti che il verificarsi di due simili eventi consecutivamente è altamente improbabile per poligoni con numero di lati $n < 100$, per cui in tale caso è sufficiente ripetere l'esecuzione, e verrà così generato un poligono con lo stesso numero di lati, ma con diversa configurazione.

Infatti, se supponiamo che $p=3\%$ sia la probabilità che l'algoritmo per il calcolo di un poligono con n lati non converga, allora $p \cdot p = 0.09\%$ sarà la probabilità che l'algoritmo non converga per due volte consecutive.

Il vantaggio principale dell'algoritmo è di riuscire a generare ogni tipo di poligono; tramite piccoli accorgimenti nell'implementazione è possibile far sì che la probabilità di generazione di ogni poligono sia distribuita in modo uniforme.

Altro vantaggio è nella possibilità di estendere in modo agevole l'implementazione al caso tridimensionale (poliedri): il Convex Hull può essere calcolato in $O(NH)$ con "Gift Wrapping", i lati diventano facce triangolari, l'"orecchia" è rappresentata da un tetraedro.

Il fatto di non avere imposto alcun tipo di vincolo, porta alla costruzione di poligoni difficilmente immaginabili a priori.

Risulta quindi un valido strumento per il "testing" di algoritmi che trattano poligoni semplici, in quanto possono verificarsi delle situazioni non previste dal programmatore, e quindi non testate manualmente. Riassumendo:

Vantaggi:

- ogni poligono semplice può essere generato con Probabilità $P > 0$;
- l'algoritmo può essere esteso al caso tridimensionale;
- i risultati sono altamente affidabili, in quanto i numeri vengono tutti rappresentati nel calcolatore come "unsigned int", quindi non si pongono i problemi connessi alla rappresentazione di numeri floating-point.

Svantaggi:

- la convergenza non è garantita
- il costo computazionale temporale massimo è abbastanza elevato

6. Bibliografia

- [1] **L. Deneen, G. Shute** “Polygonizations of Point Sets in the Plane” - *Discrete & Computational Geometry*, 1988, 77-87
———
- [2] **R. L. Graham** “An efficient algorithm for determining the convex hull of a finite planar set” - *Inform. Processing Letters*, 1972, 132-133
———
- [3] **R. B. Hayward**, “A lower bound for the optimal crossing-free Hamiltonian cycle problem” - *Discrete & Computational Geometry*, 1987, 327-343
———
- [4] **B. W. Kernighan, D. M. Ritchie** “The C programming language, second edition” - *Prentice Hall*, 1989
———
- [5] **M. S. Landy, Y. Cohen, and G. Sperling** “HIPS: A Unix-Based Image Processing System” - *Computer Vision, Graphics, and Image Processing* 25, 1984, 331-347
———
- [6] **J. O’Rourke** “Computational Geometry in C” - *Cambridge University Press*, 1994
———
- [7] **A. Pascoletti** Appunti dal Corso di “Controlli Automatici” - *a.a. 1996-97*
———
- [8] **F.P. Preparata, M.I. Shamos** “Computational Geometry” - *Springer Verlag*, 1985
———
- [9] **R. Sedgewick** “Algorithms” - *Addison-Wesley*, 1983
———
- [10] **R. Schaffer & R. Sedgewick** “The Analysis of Heapsort” - *Journal of Algorithms*, 1993, 76
———
- [11] **P. Valtr** “Probability that n Random Points Are in Convex Position” - *Discrete & Computational Geometry*, 1995, 637-643
———
- [12] **C. Zhu, G. Sundaram, J. Snoeyink, J. S. B. Mitchell** “Generating random polygons with given vertices” - *Computational Geometry Theory and Applications*, 1996, 277-290

7. Appendice: libreria Hips

Viene di seguito elencata l'intera libreria standard del pacchetto Hips aggiornata a maggio 1997.

A questi comandi si aggiungono poi quelli forniti da studenti e ricercatori che incrementano ulteriormente le potenzialità del pacchetto.

Ad esempio, se volessimo visualizzare l'output del programma su una finestra x-windows, il comando da usare è:

```
genpoly | xhips
```

oppure:

```
genpoly | xvhips
```

per memorizzare l'output in formato sunraster sul file out.ras:

```
genpoly | hipstosun > out.ras
```

per memorizzare l'output in formato sunraster sul file out.ras, ma con i colori invertiti:

```
genpoly | neg | hipstosun > out.ras
```

Per la visualizzazione dell'immagine su altre piattaforme, vengono forniti assieme all'intero pacchetto i comandi creati da alcuni utilizzatori esterni all'Università di New York.

Ad esempio, per visualizzare l'output su un Macintosh II si può usare il comando:

```
genpoly | macview
```

Se si dispone di piattaforma NeXT, l'equivalente al comando precedente è:

```
genpoly | nhips
```