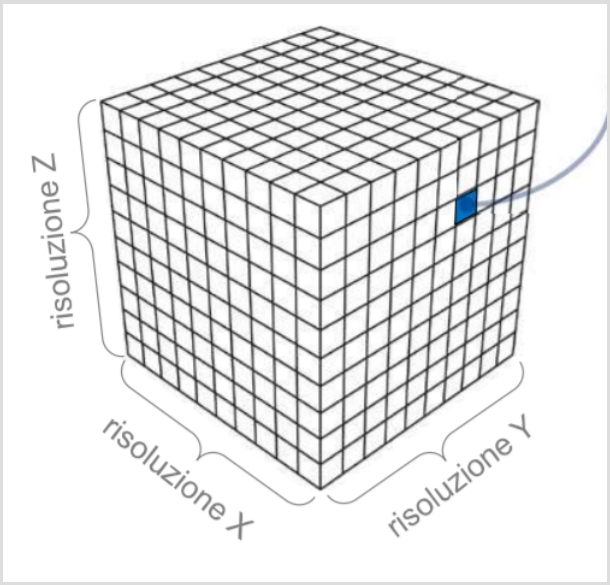
H1 Voxelized Models

Sono strutture dati che rappresentano non solo la superficie, ma anche il volume dell'oggetto 👉 modelli volumetrici .

H2 Introduzione

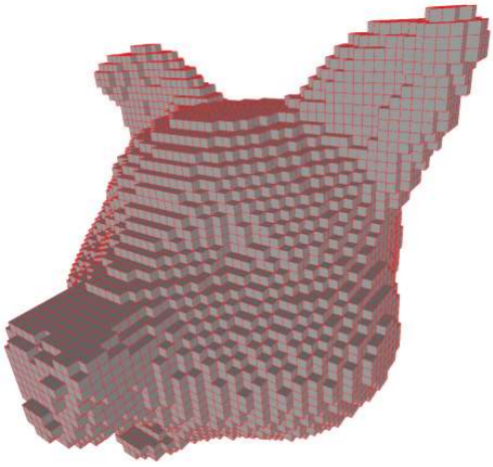
Si tratta una **matrice 3D regolare**. Ogni casella della matrice è un ***voxel***.

```
Voxel[][][] data = new Voxel[resX][resY][resZ]
```



*giglia regolare 3D, le cui celle sono **voxel** (evidenziato in blu)*

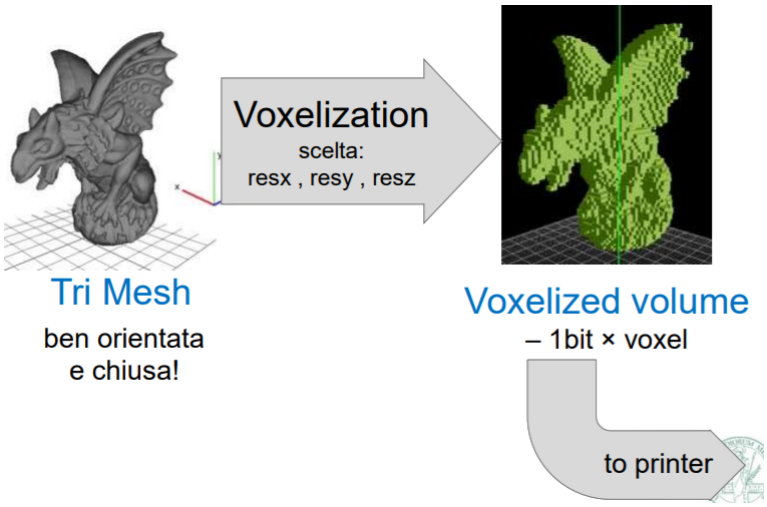
I voxel possono essere molti tipi di dato diverso, ad esempio dei `boolean` che codificano se lo spazio in quel determinato punto è pieno o vuoto, come nell'esempio sottostante:



Come è facile osservare, la **risoluzione** in questo tipo di modelli è data dalla dimensione della griglia 3D, quindi dal prodotto delle 3 tridimensioni:

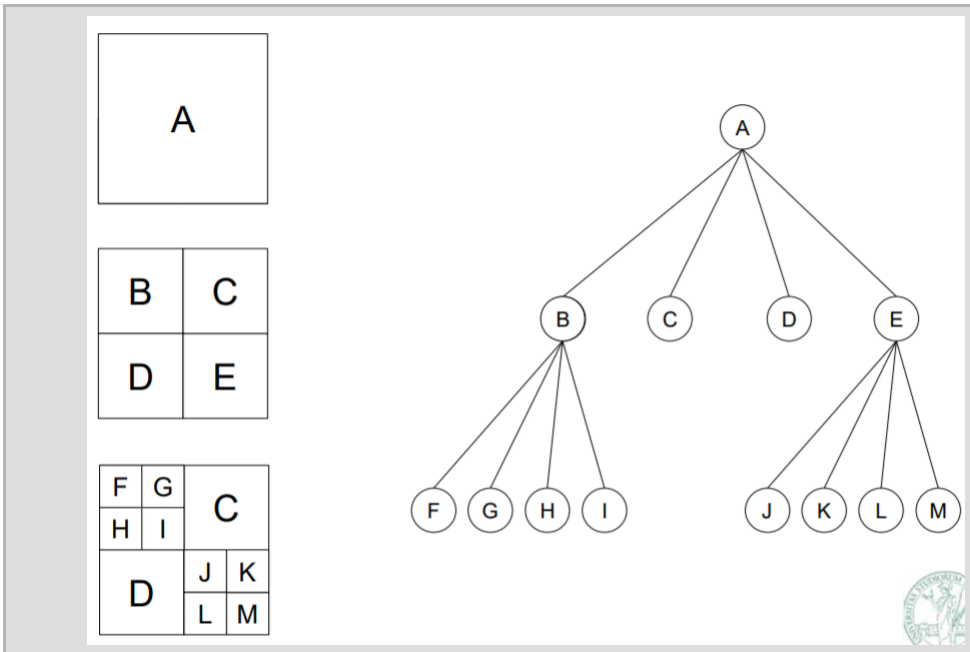
La risoluzione è fissa e quindi la **risoluzione adattiva** non è possibile, mentre la **multirisoluzione** è chiaramente ottenibile.

Questo tipo di dati voxelized sono input naturale delle stampanti 3D. Infatti è spesso eseguito un processo di *"voxelization"*, passando ad esempio da una mesh (che deve essere 2-manifold, chiusa e ben orientata) ad un modello voxelizzato.



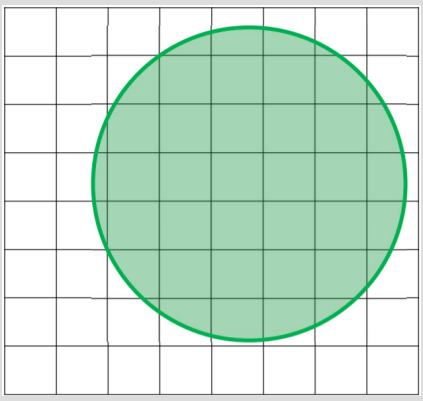
Memorizzare questa struttura direttamente in un array 3D come abbiamo visto non è però conveniente per l'uso della memoria. Anche se un voxel fosse un valore binario, chiaramente lo spazio usato sarebbe cubico.
Per memorizzare il dataset in modo più efficiente una delle strutture dati più comuni è l'**oct tree**. Vediamo prima la sua versione 2D, il **quad tree**.

H2 Quad Tree

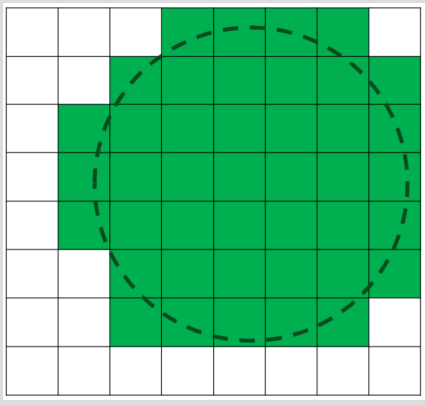


l'idea è quella di memorizzare in un albero i dati, e più un punto della figura ha bisogno di dettaglio (ad esempio una curva) più il relativo sottoalbero sarà profondo

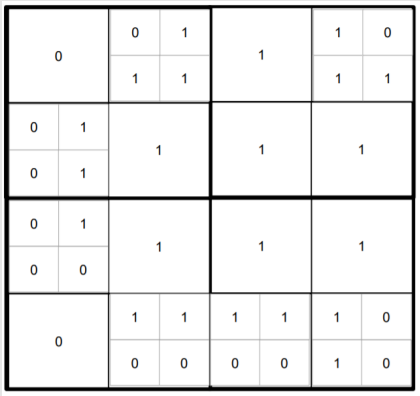
Un esempio di applicazione del quad tree è il seguente:



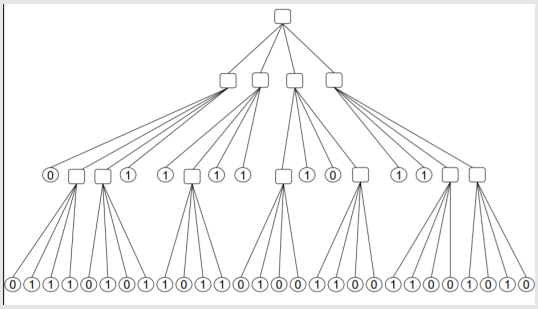
la figura da memorizzare



sua "voxelizzazione 2D"



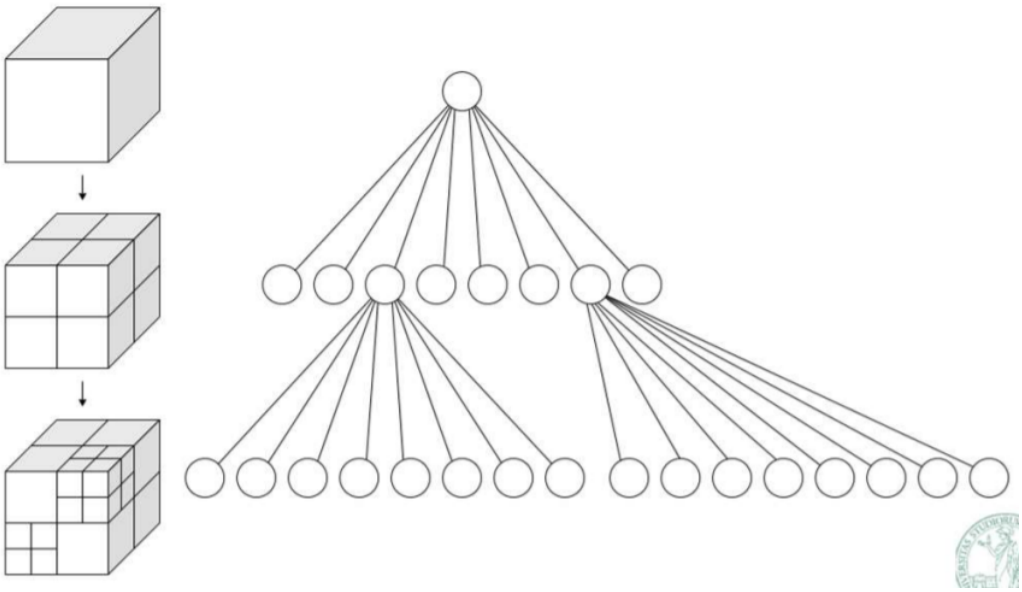
rappresentazione con voxel binari



albero risultante

H2 Oct Tree

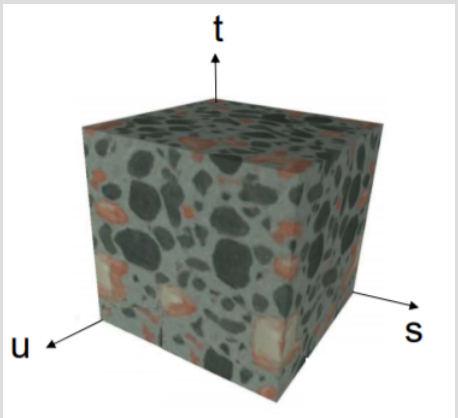
Il concetto è lo stesso dei quad tree, solo che applicato nelle 3 dimensioni. Ciò comporta che ogni nodo dell'albero abbia invece che $2^2 = 4$ figli, $2^3 = \mathbf{8}$ figli.



Rimane comunque un'alternativa molto più valida dell'array 3D, in quanto il numero totali di nodi è proporzionale al bordo dell'oggetto rappresentato. Quindi invece di essere cubico con la risoluzione, è solo **quadratico**!

H1 Texture Volumetriche

Il concetto che sta dietro ai voxel e ai modelli 3D voxelizzati può essere sfruttato per creare le cosiddette **texture volumetriche** (o **texture solide**).



1 voxel = 1 valore RGB

Quest'idea ha diversi vantaggi:

✔ Anche l'interno del modello viene texturizzato

✔ Non c'è bisogno di parametrizzare la superficie come con texture 2D, in quanto si può direttamente mappare un vertice della mesh in un punto a coordinate 3D della texture solida!

✔ È utile per rappresentare oggetti che si rompono (ad esempio in un videogame)

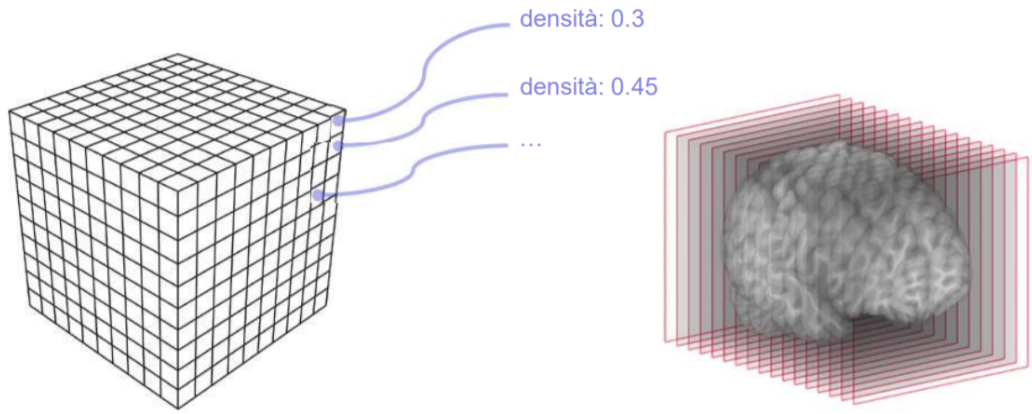
✔ È supportata dall'HW, come ogni altra texture

Ma ha un enorme svantaggio:

✘ Occupa spesso troppa memoria, a causa della dimensione in più.

H1 Voxelized Models ~ Reprise

Come abbiamo già detto, un voxel può essere un qualsiasi tipo di dato. Può risultare utile associare a un voxel un numero reale reale tra 0.0 e 1.0, ad esempio se si vuole associare a un voxel un *valore di densità*, come nel caso del formato file *DICOM* usato in ambito medico:




```
float[][][] volume = new float[resX][resY][resZ]
```

Oltre a questo, usare dei valori continui come voxel permette l'applicazione di algoritmi per trasformare un modello voxelizzato in una mesh, che è sempre utile data la loro popolarità e importanza. L'algoritmo che vedremo è detto **marching cubes**.

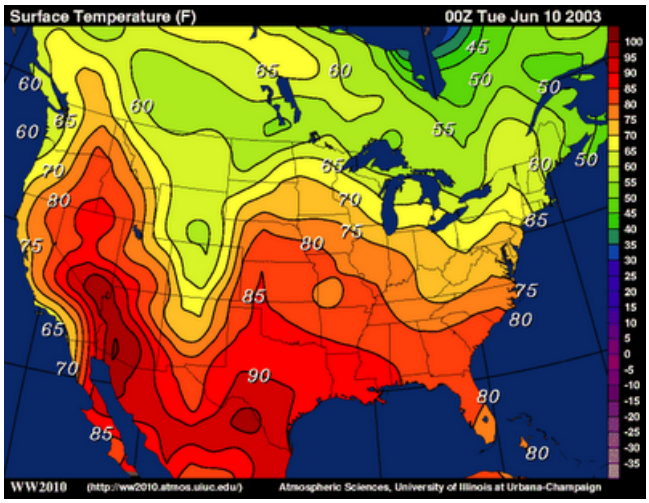
H2 Marching Cubes

L'algoritmo sfrutta il concetto di ***isosuperficie***, cioè una *isolinea* in 3D.

H4 Isolinee

Ogni punto del piano 2D ha assegnato un valore scalare (pressione, temperatura, etc.). Definisco un valore soglia σ:

- considero tutta la regione 2D i cui valori sono $> \sigma$
- considero la linea che racchiude tale regione, quindi che avrà tutti valori $= \sigma$.
  Tale linea è detta isolinea di valore σ.



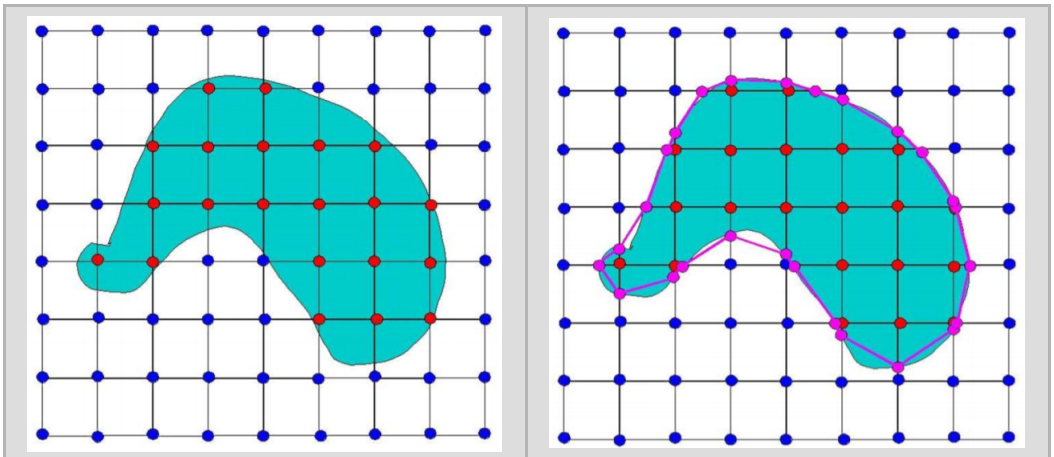
H4 Isosuperfici

A ogni punto di un volume 3D ha assegnato un valore scalare (densità, pressione, etc.). Definisco un valore soglia σ:

- considero tutta la regione 3D i cui valori sono $> \sigma$
- considero la *superficie* che racchiude tale regione, quindi che avrà tutti i valori $= \sigma$.
  Tale superficie è detta *isosuperficie* di valore σ.

Tornando all'algoritmo **marching cubes**, per capire il suo funzionamento vediamo prima la sua versione 2D che sfrutta le isolinee.

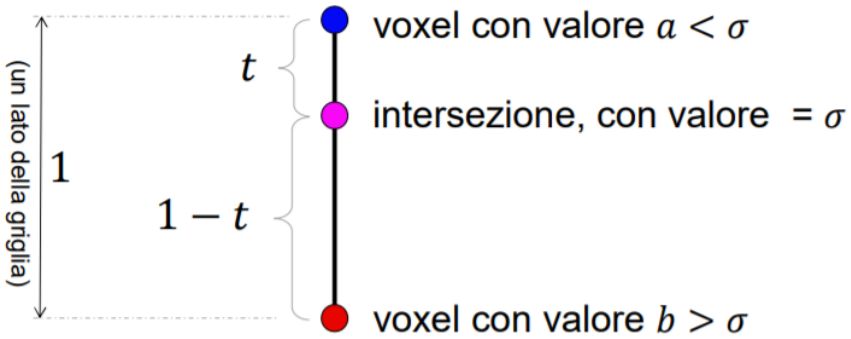
H3 Marching Squares



Come si nota dalle figure, l'aver posto una soglia ha creato una distinzione che permette di identificare una approssimazione del perimetro della figura come quello composto dalle intersezioni fra il perimetro "reale" e i lati della griglia.

- Come facciamo a trovare le intersezioni ( ) in figura?

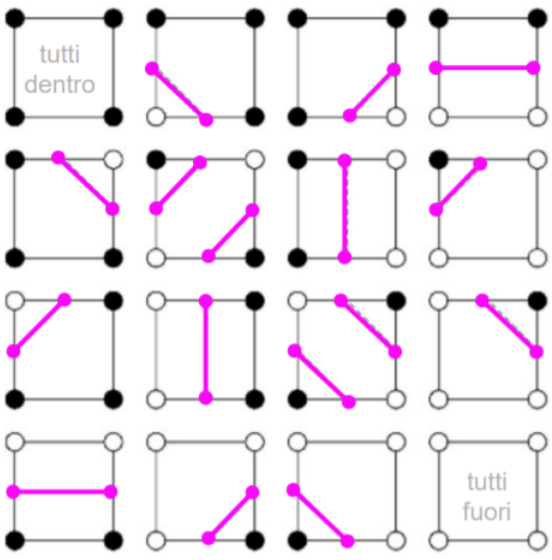
  Tali intersezioni sono sempre comprese tra un punto e un punto . L'idea è quella di interpolare i valori relativi a quei punti per trovare il punto di intersezione il cui valore sarà $= \sigma$.



$$a \cdot (1-t) + b \cdot t \Leftrightarrow t = \frac{\sigma - a}{b - a} \tag{1}$$

- Come facciamo a trovare i segmenti che connettono le intersezioni?

  Non approcceremo questo problema proceduralmente, bensì dal punto di vista combinatorio. Infatti possiamo notare come i segmenti vengano definiti tutti all'interno di un quadrato della griglia, i cui vertici sono 4 punti di valori o $< \sigma$ (cioè fuori dalla figura) o $\geq \sigma$ (cioè dentro). Possiamo esplicitare tutte le 16 combinazioni:

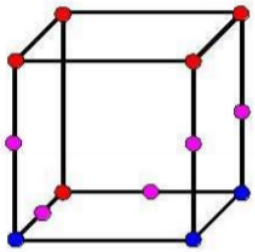


H3 Marching Cubes - Descrizione

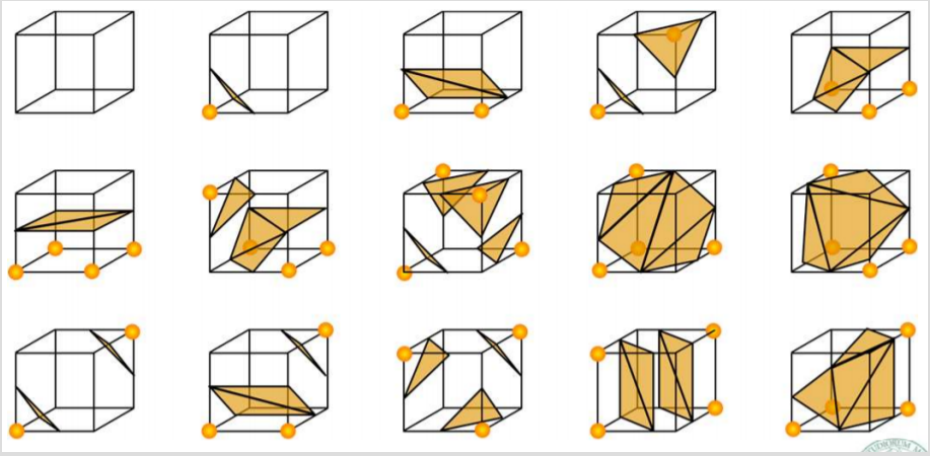
Possiamo quindi generalizzare l'algoritmo anche per la terza dimensione, ottenendo quindi l'**algoritmo Marching Cubes**:

- Ogni voxel del modello sarà o dentro o fuori in base a una soglia σ .
- Ogni edge delle griglia che connette un voxel dentro ad un voxel fuori, ha al suo interno una intersezione con l'isosuperficie cercata.

  Tale intersezione la si trova come visto in (1).

  Infine, si trasformano i punti di intersezione trovati in vertici della mesh. Ciò significa che abbiamo definito la **geometria** della mesh che stiamo costruendo!
- Per ottenere la **connettività**, applichiamo il ragionamento combinatorio visto prima, questa volta però avremo cubi da 8 vertici, ciascuno dei quali può essere o dentro o fuori $\rightarrow 2^8 = 256$ possibili combinazioni:

  qui sono visualizzati solo 15 casi, ma tutti gli altri sono ottenibili per simmetria

H5 Efficienza

Per come descritto, l'algoritmo non è assai efficiente, in quanto vorrebbe dire analizzare tutti i voxel presenti nello spazio contenente il modello, quindi un numero cubico di cubi. In realtà spesso sono pochi i voxel che contengono intersezioni con l'isosuperficie al loro interno, quindi analizzare tutti i cubi della griglia 3D è uno spreco.
L'ottimizzazione che da il nome all'algoritmo è la seguente:

1. quando trovo un cubo non vuoto, lo processo
2. passo ad analizzare i cubi vicini, *marciando* lungo i cubi che probabilmente contengono l'isosuperficie.

H5 Output

L'output ottenuto è una mesh triangolare 2-manifold, chiusa e ben orientata, ma spesso non è una mesh di buona qualità. Infatti i triangoli risultanti sono spesso allungati o addirittura degeneri. È comunque un processo spesso usato per la comodità di trattare mesh: