Algoritmi e strutture dati II

Lezione 12

3D convex hull

CompGeo

Cap. 11

Convex hull in 3 dimensioni

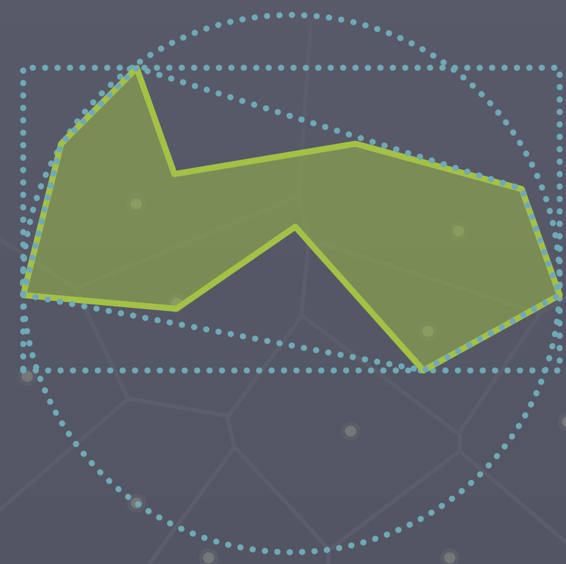
- Il problema di individuare il convex hull di un insieme di punti in uno spazio n-dimensionale ha una interpretazione che va ben oltre la pura interpretazione geometrica
- Se vogliamo operare una combinazione lineare di un insieme qualunque di elementi caratterizzati da n caratteristiche disgiunte possiamo pensare al convex hull come al sottospazio dello spazio di definizione che contiene tutte le combinazioni ottenibili

Convex hull in 3 dimensioni

- Se il numero di caratteristiche disgiunte è pari a 2 avremo un convex hull nel piano
- All'aumentare del numero di caratteristiche avremo convex hull in spazi n-dimensionali che, però, mantengono, tutte le proprietà del convex hull nel piano
- Ci interessa in particolar modo il convex hull nello spazio (3D) che ha un interesse pratico molto immediato in tutti i settori in cui si deve fare collision detection

Convex hull in 3 dimensioni

- Per velocizzare i tempi di calcolo delle possibili collisioni tra oggetti all'interno di una scena sintetica 3D si possono utilizzare delle approssimazioni degli oggetti
- Le principali sono
  - Il bounding box/sphere (semplice ma magari troppo approssimativo)
  - Il convex hull



Dimensione del convex hull

- Abbiamo visto che il convex hull di un insieme P di n punti nel piano è un poligono i cui vertici sono punti in P e quindi, al massimo, di n vertici
- In 3D vale la medesima proprietà: il convex hull di un insieme P di n punti è un poliedro con al più n vertici
- In 2D questo è un bound anche sul numero di edge (pari al numero di vertici), in 3D è leggermente diverso

Dimensione del convex hull

- Il numero di spigoli e facce è dato dalla formula di Eulero

$$n - n_e + n_f = 2$$

dalla quale possiamo dedurre, poiché ogni faccia ha almeno 3 spigoli e su ogni arco incidono esattamente due facce, che

$$2n_e \geq 3n_f$$

che, caso di una mesh chiusa (*watertight*) di triangoli diventa

$$2n_e = 3n_f$$

Dimensione del convex hull

- Sostituendo nella formula di Eulero si ottiene:

$$n - n_e + n_f = 2$$
$$2n - 2n_e + 2n_f = 4$$
$$2n - 3n_f + 2n_f = 4$$
$$2n - n_f = 4$$
$$n_f = 2n - 4$$

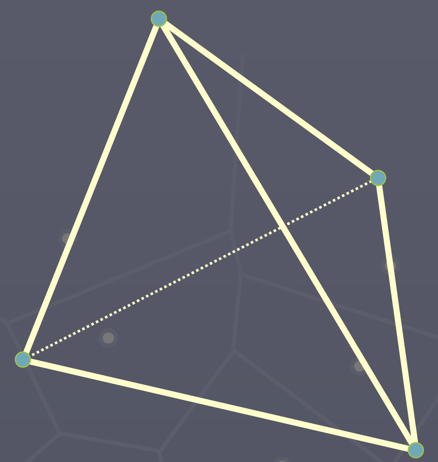
e:

$$2n_e = 6n - 12$$
$$n_e = 3n - 6$$

- Il numero di elementi del convex hull è quindi lineare con i punti dell'insieme P

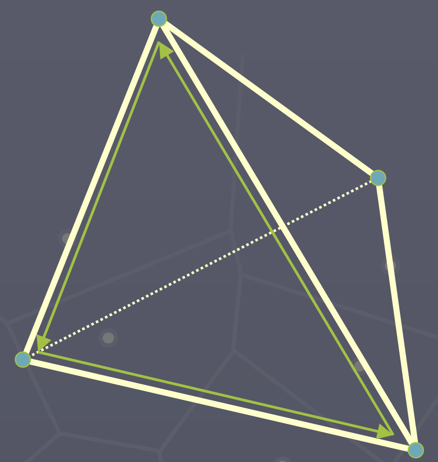
Calcolo del convex hull 3D

- L'algoritmo per il calcolo del convex hull di un insieme P di n punti nello spazio è incrementale randomizzato
- Si parte costruendo il convex hull di un insieme minimo di punti non coplanari (4) che formano un tetraedro estratti da una permutazione random dei punti



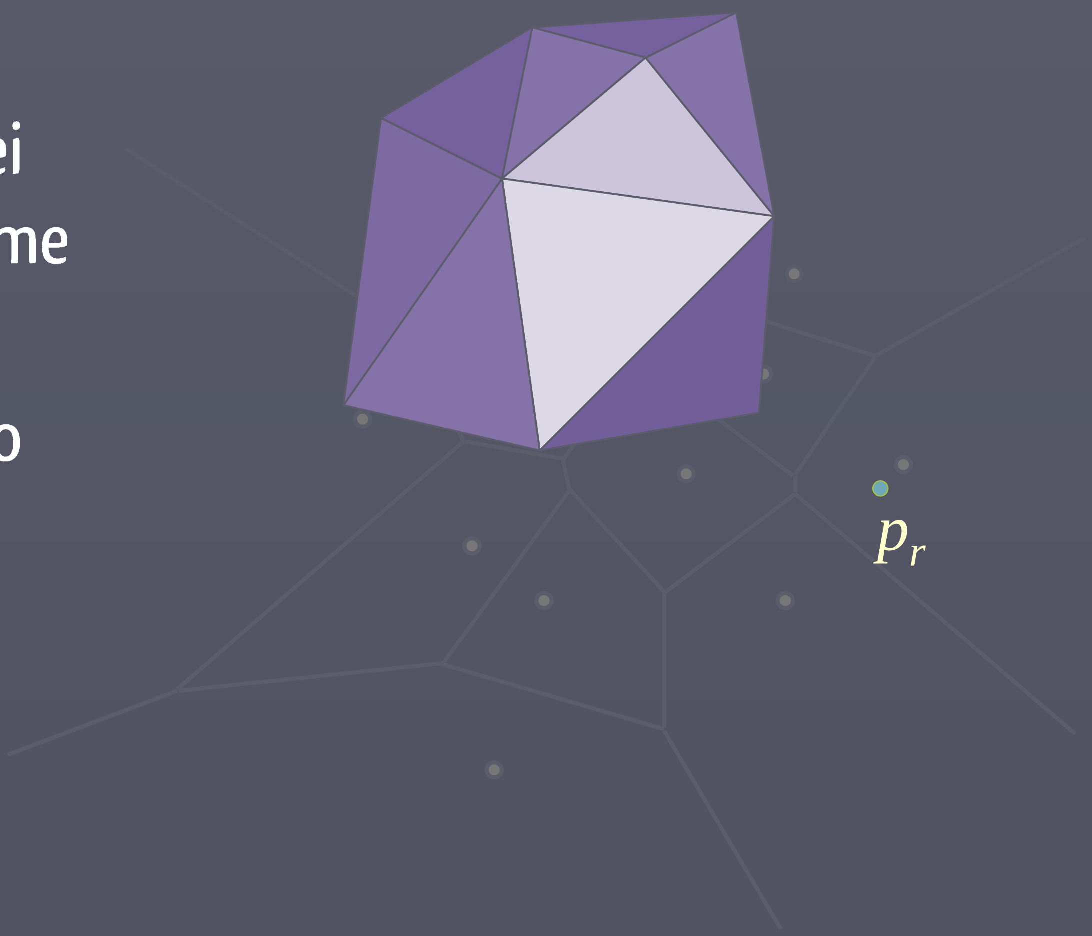
Calcolo del convex hull 3D

- Il poliedro sarà memorizzato in una DCEL adattata al 3D
  - I vertici contengono informazioni sulla terza coordinata
  - Il senso con cui vengono percorsi gli half-edge è tale per cui dall'esterno vedo la faccia percorsa in senso antiorario



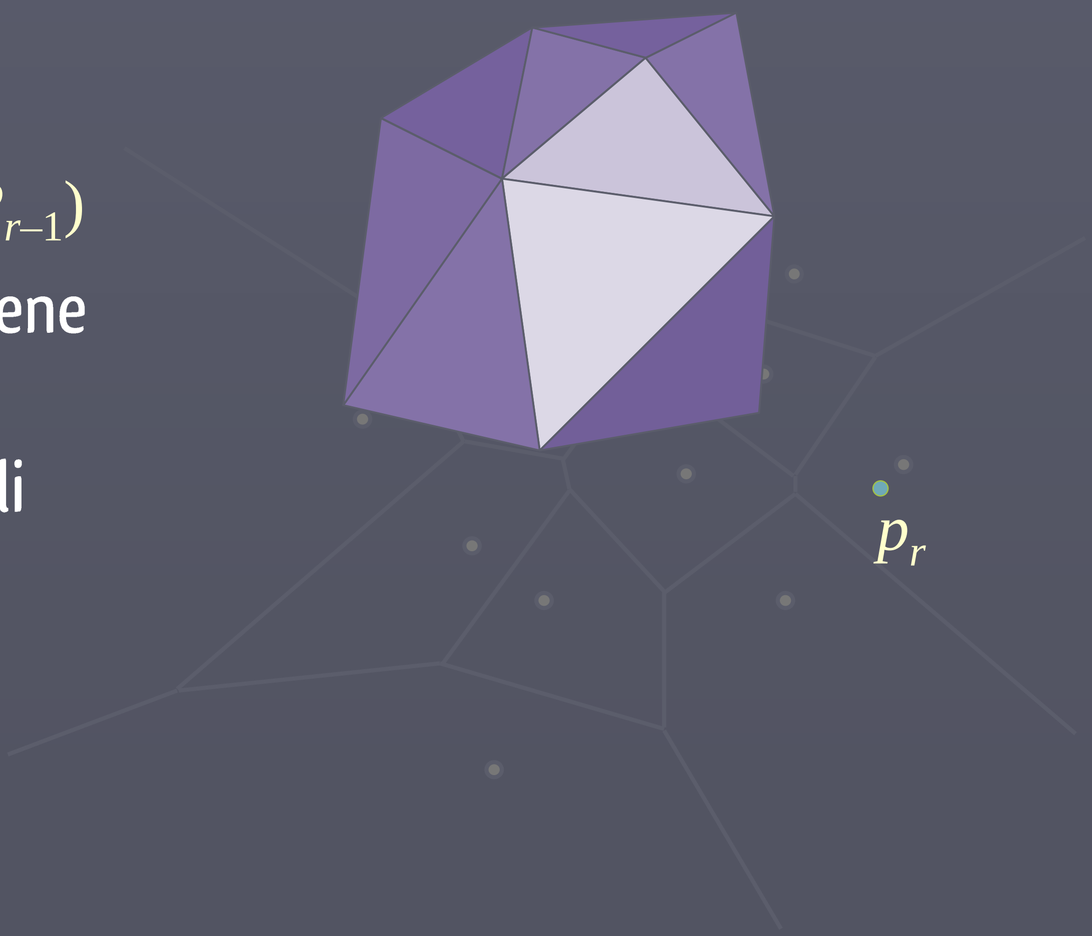
Calcolo del convex hull 3D

- Adesso restano da inserire gli altri $n-4$ punti
- Al passo generico r dell'algoritmo avremo costruito il convex hull dei primi $r-1$ punti dell'insieme randomizzato
- Dovremo inserire l'r-esimo punto



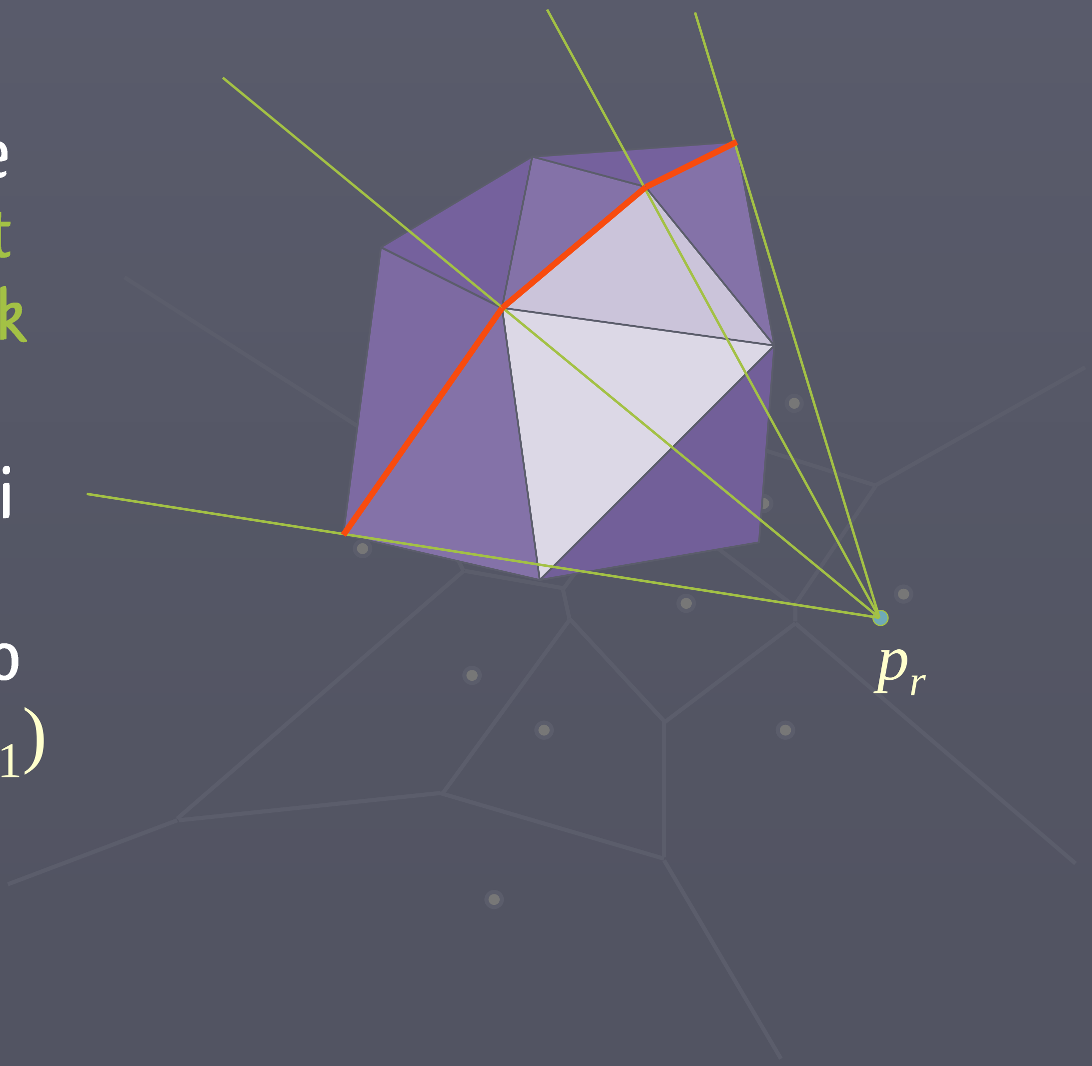
Calcolo del convex hull 3D

- Si possono presentare due casi:
  - Il punto è contenuto in $\mathcal{CH}(P_{r-1})$ e il convex hull rimane invariato
  - Il punto è esterno a $\mathcal{CH}(P_{r-1})$
- Nel secondo caso ci conviene esaminare il concetto di facce di un poliedro visibili da un punto



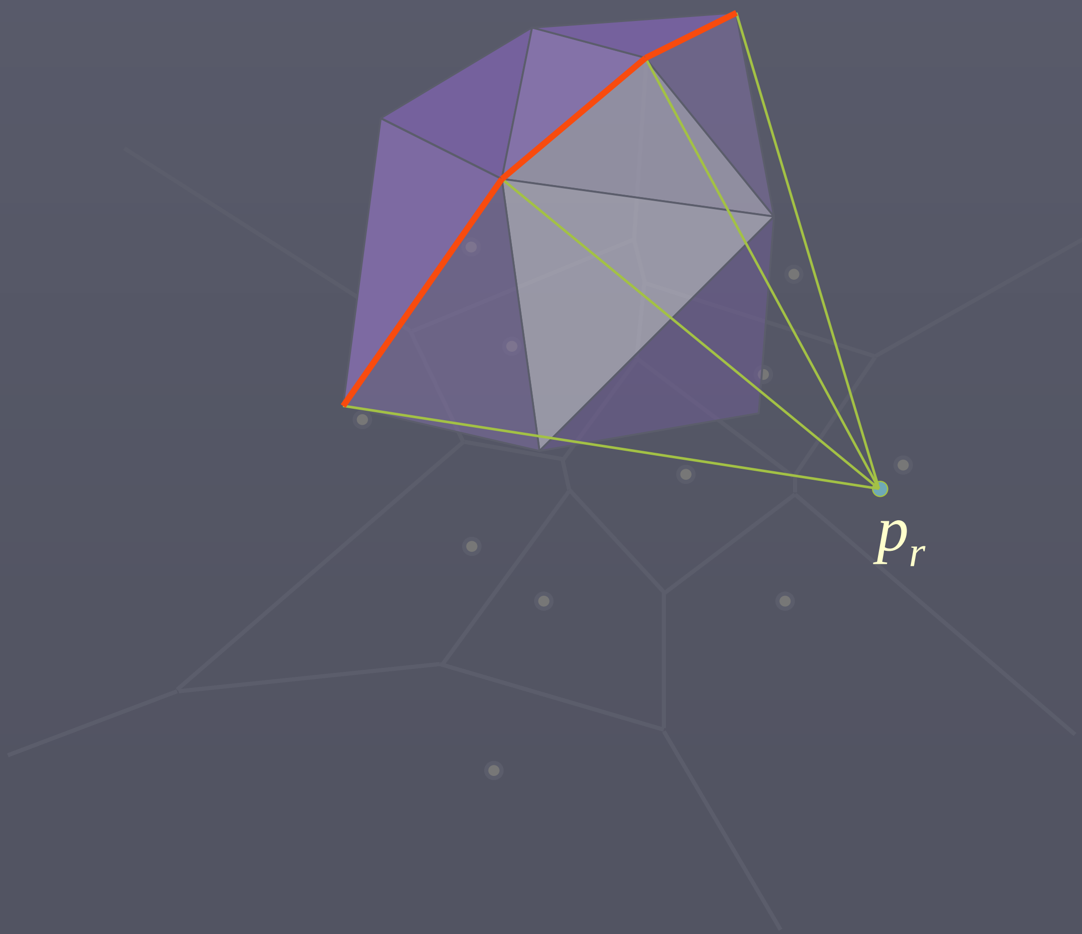
Calcolo del convex hull 3D

- In effetti ci interessano le facce di $\mathcal{CH}(P_{r-1})$ visibili da p_r
- Possiamo dividere le facce $\mathcal{CH}(P_{r-1})$ di in facce front (visibili da p_r) e facce back (invisibili da p_r)
- Entrambe formano regioni connesse con un bordo di separazione che definiamo orizzonte di p_r in $\mathcal{CH}(P_{r-1})$



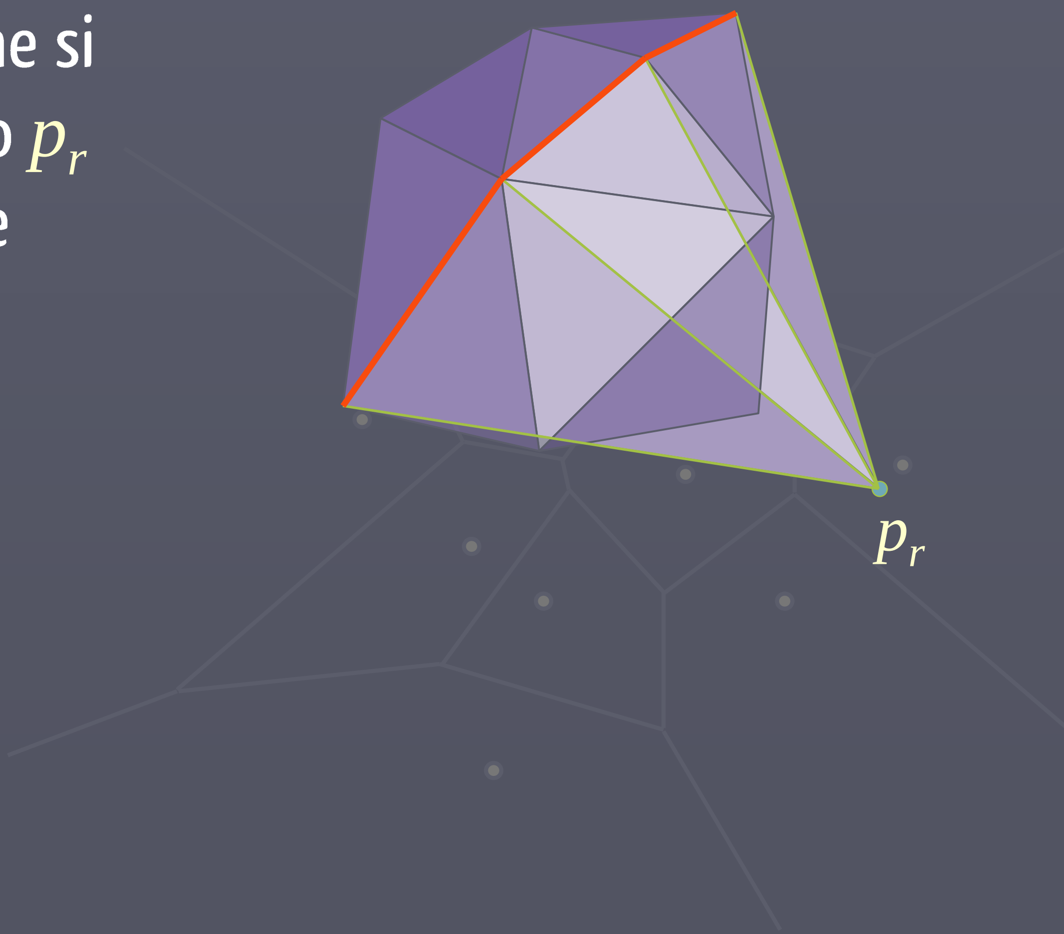
Calcolo del convex hull 3D

- L'orizzonte gioca un ruolo cruciale nella definizione di $\mathcal{CH}(P_r)$ a partire da $\mathcal{CH}(P_{r-1})$
- Separa infatti le facce di $\mathcal{CH}(P_{r-1})$ che saranno contenute in $\mathcal{CH}(P_r)$ da quelle che non lo saranno



Calcolo del convex hull 3D

- Quello che dobbiamo fare per trasformare $\mathcal{CH}(P_{r-1})$ in $\mathcal{CH}(P_r)$ è sostituire le facce visibili da p_r con quelle che si costruiscono connettendo p_r con gli edge dell'orizzonte



Calcolo del convex hull 3D

- Se desideriamo mantenere una mesh di triangoli possiamo anche evitare di verificare se un triangolo tra quelli inseriti risulta coplanare con l'adiacente sull'orizzonte
- Questo perché, comunque, vorremmo tenerli entrambi e non fonderli in un unico quadrilatero

Ricerca delle facce visibili

- Una volta enunciato il metodo rimane da risolvere il problema di come organizzare i dati per trovare efficientemente le facce visibili del $\mathcal{CH}(P_{r-1})$ da p_r
- Vogliamo evitare di testare la visibilità di ogni faccia di $\mathcal{CH}(P_{r-1})$ da p_r per non avere un algoritmo di costruzione $O(n^2)$
- Per questo ci serve una struttura dati aggiuntiva che tiene conto della visibilità delle facce dai punti e viceversa

Conflict list

- Si definisce conflict list la lista, per ogni faccia f in $\mathcal{CH}(P_r)$, di tutti i punti che possono vedere f:

$$P_{\text{conflict}}(f) \subseteq \{p_{r+1}, p_{r+2}, \ldots, p_n\}$$

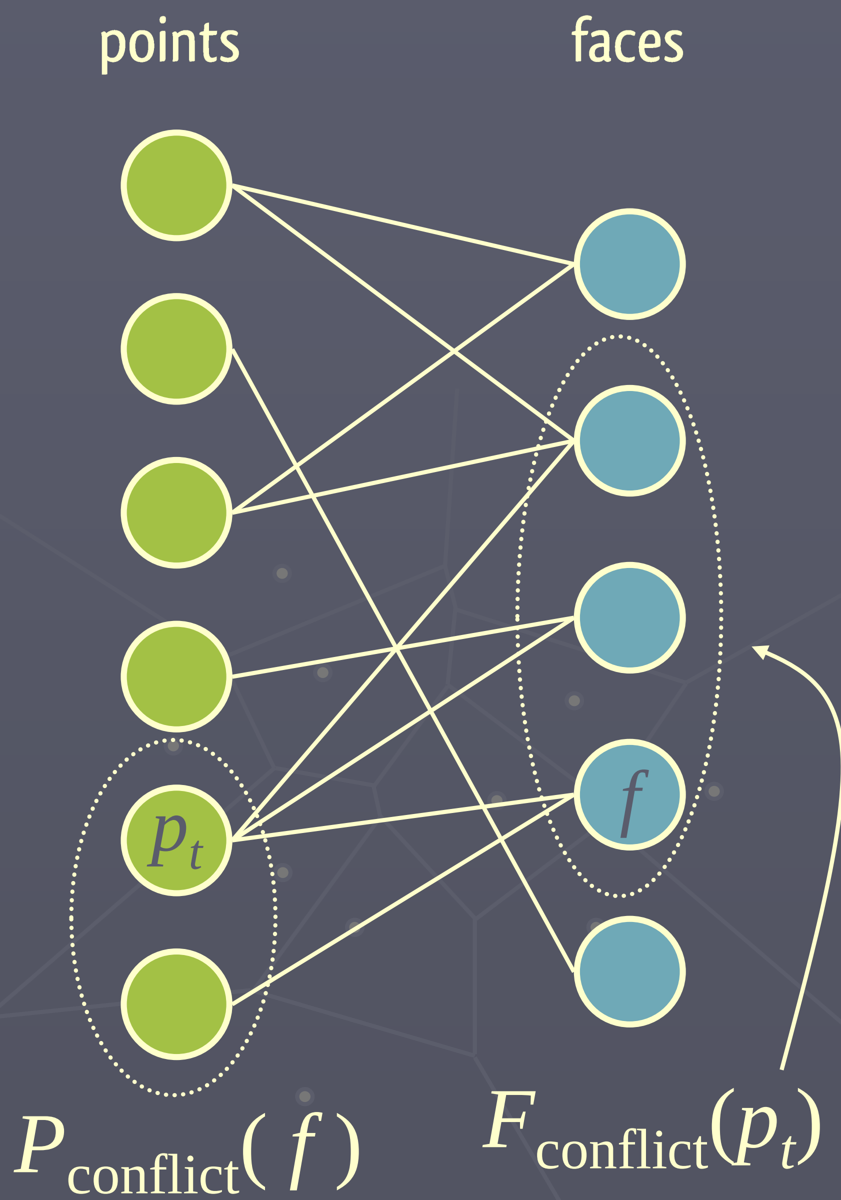
- Allo stesso tempo per ogni punto p_t in P per $t>r$ si mantiene una lista di tutte le facce visibili da p_t:

$$F_{\text{conflict}}(p_t) \subseteq \mathcal{CH}(P_r)$$

- Diremo che un punto p_i in $P_{\text{conflict}}(f)$ è in conflitto con f perché non posso averli entrambi nel convex hull

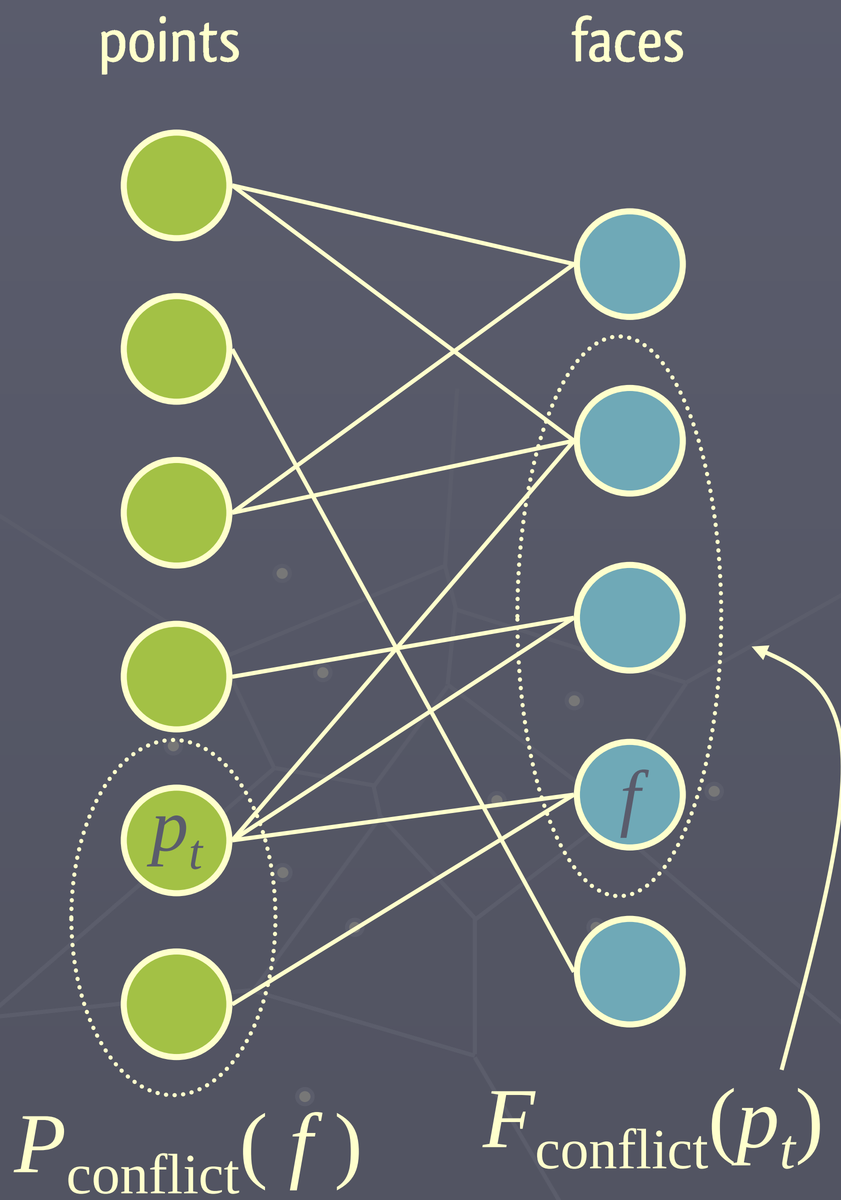
Conflict graph

- Manteniamo le due tipologie di conflict list in un conflict graph
- E' un grafo bipartito con due tipi di nodi
  - Punti
  - Facce
- Ci possono essere archi solamente tra nodi punto e nodi faccia e viceversa



Conflict graph

- Usando il conflict graph possiamo risolvere il problema della visibilità in maniera semplice
- Quando dobbiamo aggiornare p_r rimuoveremo tutte le facce in $F_{\text{conflict}}(p_r)$ e le sostituiremo con facce che hanno uno spigolo sull'orizzonte e il vertice opposto in p_r

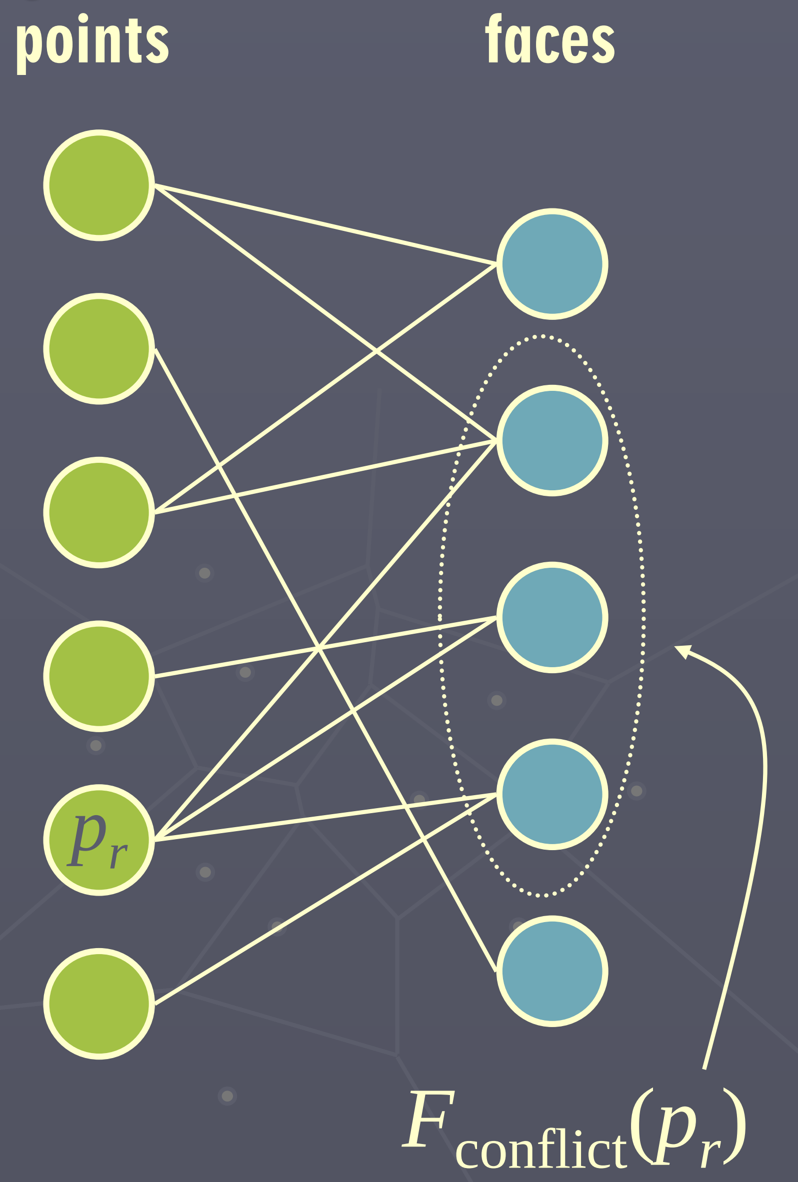


Gestione del conflict graph

- L'inizializzazione del conflict graph è fattibile in $O(n)$, dato che basta scorrere la lista dei punti e controllare, per ciascuno di essi, quale tra le 4 facce del tetraedro risulta visibile
- L'aggiornamento dopo un inserimento coinvolge invece la gestione delle due porzioni del grafo, sia la parte punti che la parte facce

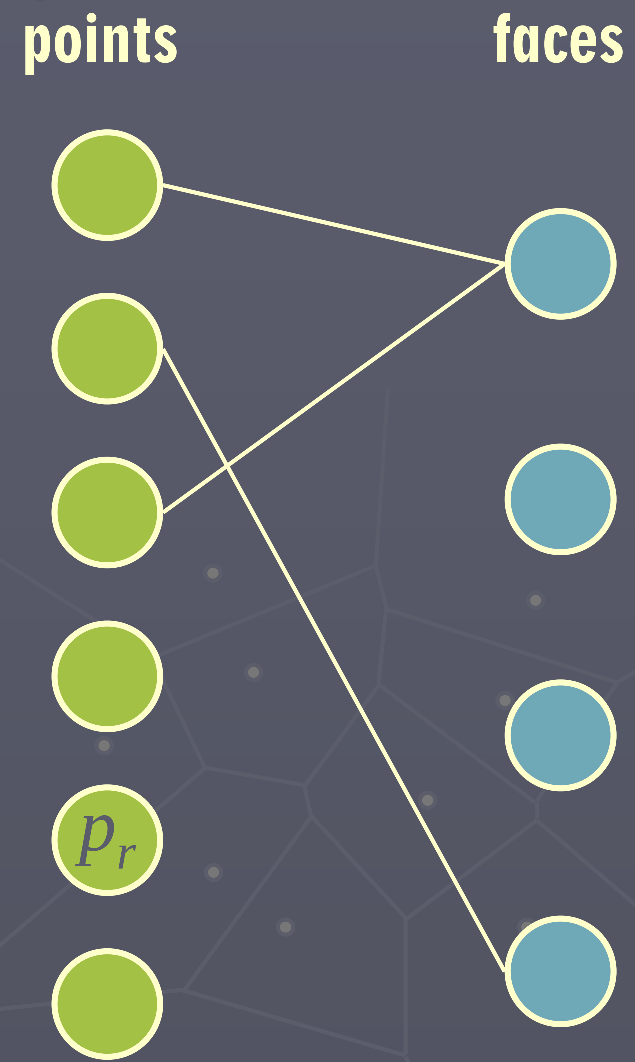
Gestione del conflict graph

- Per prima cosa si cancellano tutte le facce e gli archi su esse incidenti visibili da p_r ovvero $F_{\text{conflict}}(p_r)$



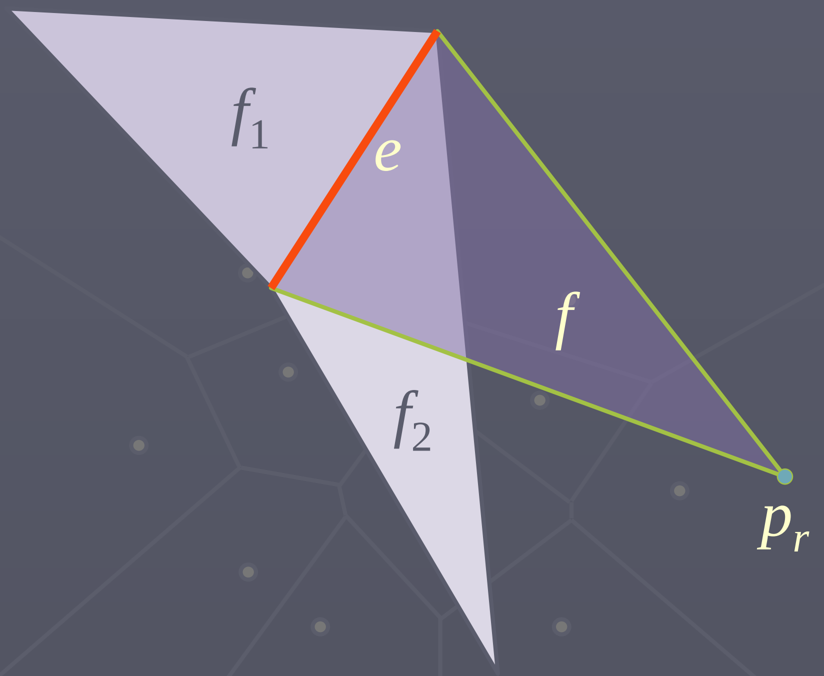
Gestione del conflict graph

- Per prima cosa si cancellano tutte le facce e gli archi su esse incidenti visibili da p_r ovvero $F_{\text{conflict}}(p_r)$
- Poi si aggiungono nuovi nodi per ogni faccia inserita in $\mathcal{CH}(P_r)$
- Per ciascuna di essere deve essere costruita la conflict list



Gestione del conflict graph

- La scelta dei punti per cui testare la visibilità è guidata dalla scelta delle nuove facce
- Se un punto p_t vede la nuova faccia f, ne vede anche il suo edge e , sull'orizzonte, opposto a p_t
- Ovvero: se vede f vedeva almeno una tra f_1 e f_2 e possiamo quindi costruire $P_{\text{conflict}}(f)$ a partire da $P_{\text{conflict}}(f_1)$ e $P_{\text{conflict}}(f_2)$



Algoritmo

ConvexHull(P)

Input. **Un insieme P di n punti nello spazio 3D**

Output. **Il convex hull $\mathcal{CH}(P)$ di P**

1. **Calcola un permutazione random degli n punti di P**
2. **Trova 4 punti p_1, p_2, p_3 e p_4 in P che formano un tetraedro**
3. $\mathcal{C} \leftarrow \mathcal{CH}(\{p_1, p_2, p_3, p_4\})$
4. **Inizializza il conflict graph $\mathcal{G}$ con tutte le coppie visibili (p_t,f) con f faccia in $\mathcal{C}$ e $t>4$**
5. **for** $r \leftarrow 5$ **to** n
6.   **do** **(* inserisci p_r in $\mathcal{C}$ *)**
7.   **if** $F_{\text{conflict}}(p_r)$ **non è vuoto (* p_r non è interno a $\mathcal{C}$ *)**
8.   **then** **cancella tutte le facce di $F_{\text{conflict}}(p_r)$ da $\mathcal{C}$**
9.   **scorrendo il boundary delle facce visibili da p_r ($F_{\text{conflict}}(p_r)$) crea la lista ordinata $\mathcal{L}$ di tutti gli edge sull'orizzonte**

Algoritmo

for ogni edge e in $\mathcal{L}$

  do connetti e con p_r per creare la nuova faccia f

    (* determina la conflict list per f *)

    crea un nodo per f in $\mathcal{G}$

    siano f_1 e f_2 le facce incidenti a e in $\mathcal{C}$

    $P(e) \leftarrow P_{\text{conflict}}(f_1) \cup P_{\text{conflict}}(f_2)$

    for ogni punto $p \in P(e)$

      do se f è visibile da p aggiungi (p,f) a $\mathcal{G}$

cancella il nodo p_r e i nodi corrispondenti alle facce in $F_{\text{conflict}}(p_r)$ da $\mathcal{G}$ assieme a tutti gli archi in esse incidenti

return $\mathcal{C}$

Complessità

- L'utilizzo del conflict graph, accoppiato alla strategia di inserimento randomizzata porta la complessità di costruzione del convex hull a $O(n \log n)$
- L'occupazione di memoria è invece lineare, si può inoltre dimostrare che l'upper bound del numero di facce create durante il processo è $6n–20$