**PROGETTO PROGRAMMAZIONE E CALCOLO SCIENTIFICO**

Definizione della struttura ***Polyhedron*** utilizzata dentro il namaspace *PolyhedronLibrary*:

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Definizione delle funzioni:  
**pair<vector<Vector3d>, vector<vector<*unsigned* *int*>>> SalvataggioDati(*unsigned* *int*& *q*)**

Serve per costruire i dati geometrici (vertici e facce) di un solido platonico, dati in input un intero q che rappresenta il tipo di solido (q= 3 tetraedro, q=4 ottaedro, q=5 icosaedro). Restituisce una coppia contenente: le coordinate normalizzate dei vertici del solido, le facce espresse come insiemi di id dei vertici.

Passaggi della funzione:

1. Dichiara due vettori vuoti (vertici, facce)
2. Controlla il valore di q per determinare quale solido costruire
3. Inserisce manualmente i vertici del solido scelto, dopo averli normalizzati
4. Definisce le facce, come un vettore di indici dei vertici
5. Restituisce una pair<vertici, facce> → da usare successivamente per costruire un oggetto Polyhedron

**Polyhedron CreazioneSolidoPlatonico(*unsigned* *int*& *q*)**

La funzione CreazioneSolidoPlatonico(unsigned int& q) costruisce e restituisce un oggetto Polyhedron, che rappresenta un solido platonico tridimensionale specificato da q. Organizza i dati in celle di dimensione 0 (vertici), 1 (lati), 2 (facce) e 3 (il solido stesso).

Passaggi della funzione:

1. Chiama SalvataggioDati(q) da cui riceve vertici e facce del solido platonico di partenza
2. Riempie le informazioni sui vertici (Cell0Ds)
3. Costruisce i lati (Cell1Ds), usando una mappa *MappaLati* per tenere traccia dei lati in modo univoco (ogni coppia di vertici definisce un lato)
4. Riempie le informazioni sulle facce (Cell2Ds)
5. Costruisce il poliedro (Cell3Ds)
6. Restituisce il poliedro P costruito

**Polyhedron TriangolazioneClasse1(const Polyhedron& *P*, const *unsigned* *int*& *t\_value*)**

La funzione TriangolazioneClasse1 prende un poliedro P con facce triangolari e restituisce un nuovo poliedro P1 in cui ogni faccia è ulteriormente suddivisa in triangoli più piccoli tramite una triangolazione detta “di classe 1”,regolata da un parametro intero t\_*value* (t\_value = b+c). Restituisce un elemento *Polyhedron* con i dati della triangolazione 1.

Passaggi della funzione:

1. Inizializza un nuovo poliedro vuoto P1
2. Crea strutture per ID e controllo duplicati:

* id\_punto\_successivo, id\_*lato*\_successivo, id\_*faccia*\_successiva: contatori ID
* *MappaLati*: mappa per evitare punti duplicati lati
* *MappaPunti*: mappa per evitare punti duplicati vertici

1. Per ogni faccia triangolare del poliedro originale:

* Estrae i 3 vertici della faccia (A, B, C)
* Calcola le coordinate dei punti interni alla faccia usando interpolazione baricentrica con parametro t\_*value*
* Inserisce i nuovi punti in P1, controllando che non siano già stati inseriti

1. Costruisce triangoli più piccoli (crea prima i triangoli su tutta la faccia tranne il triangolo in cima, a partire dalla linea con due punti)

* Organizza i punti interni in righe (*rows*)
* Crea triangoli tra questi punti
* Aggiorna la struct con i dati di vertici, lati e facce gestendo duplicati

1. Gestisce separatamente il triangolo finale in cima a ciascuna griglia
2. Aggiorna attributi finali del nuovo poliedro P1
3. Restituisce il nuovo poliedro P1 triangolato

**Polyhedron TriangolazioneClasse2(const Polyhedron& P, const unsigned int& b)**

La funzione TriangolazioneClasse2 effettua una raffinazione di classe 2 su un poliedro, basandosi sulla triangolazione di classe 1 e sul valore in input b: aggiunge baricentri delle facce e punti medi dei lati, collegandoli per formare nuovi triangoli. Restituisce un elemento *Polyhedron* con i dati della triangolazione 2.

Passaggi della funzione:

1. Effettua la triangolazione di partenza P1: ottiene la prima triangolazione a partire dal poliedro iniziale P
2. Crea nuovo poliedro P2: conterrà il risultato della triangolazione di classe 2
3. Copia tutti i vertici di P1 in P2: inserisce le coordinate e aggiorna *MappaPunti* per evitare duplicati.
4. Per ogni faccia calcola e salva i baricentri delle facce di P1
5. Per ogni faccia originale di P costruisce la mappa *FaccePerLati*, che indica ogni lato della triangolazione 1 a quante facce della triangolazione 1appartiene

* Il lato appartiene ad 1 faccia = lato esterno
* Il lato appartiene a 2 facce = lato interno

1. Collega baricentri ai vertici delle facce creando e salvando i lati
2. Caso 1: il lato appartiene a una sola faccia

* Calcola il punto medio e lo salva
* Collega il baricentro al punto medio e salva il lato
* Crea i due nuovi lati creati dalla divisione del lato di P1 dal punto medio e li salva
* Crea le due nuove facce e le salva
* Quando salva un nuovo elemento controlla sempre che non esista già

1. Caso 2: il lato appartiene a due facce

* Collega il baricentro della faccia al baricentro di quella vicino e salva il lato
* Crea le nuove facce interne e le salva
* Quando salva un nuovo elemento controlla sempre che non esista già

1. Aggiorna attributi finali del nuovo poliedro P2
2. Restituisce il nuovo poliedro P2 triangolato

**Polyhedron Dualizzazione(const Polyhedron& *P*)**

Costruisce il duale di un poliedro dato in input, cioè un nuovo poliedro dove:

* Ogni faccia del poliedro originale diventa un vertice del duale
* Ogni vertice del poliedro originale diventa una faccia del duale
* Ogni lato tra due facce adiacenti nel poliedro originale diventa un lato nel duale che collega i relativi nuovi vertici (baricentri delle facce)

Restituisce un elemento Polyhedron con i dati della dualizzazione.

Passaggi della funzione:

1. Crea i vertici del duale (baricentri delle facce originali): ogni faccia di P viene trasformata in un nuovo vertice nel duale
2. Costruisce la mappa: lato → facce adiacenti per identificare facce adiacenti
3. Crea i lati del duale: se due facce condividono un lato, significa che nel duale esisterà un lato tra i baricentri delle due facce (cioè tra due vertici del duale)
4. Crea le facce del duale: ogni vertice v del poliedro originale diventa una faccia nel duale
5. Aggiorna attributi finali del nuovo poliedro P\_duale

**Polyhedron ProiezioneSullaSfera(const Polyhedron& *P*)**

Proietta i vertici di un poliedro sulla superficie della sfera unitaria, cioè normalizzati in modo che abbiano tutti distanza 1 dall’origine. Restituisce un elemento Polyhedron con i punti normalizzati.

**void EsportazionePoliedro(const Polyhedron& P,**

**const vector<Gedim::UCDProperty<double>>& points\_properties = {},**

**const vector<Gedim::UCDProperty<double>>& segments\_properties = {})**

Esporta i punti e lati di Polyhedron in file esterni .inp, usando la libreria Gedim::UCDUtilities. È una funzione void (non restituisce nulla).

**void EsportazioneFile(const Polyhedron& P)**

Esporta su file .txt tutte le informazioni strutturali di un oggetto Polyhedron: i vertici (0D) → Cell0Ds.txt, i lati (1D) → Cell1Ds.txt, le facce (2D) → Cell2Ds.txt, le celle (3D) → Cell3Ds.txt. È una funzione void (non restituisce nulla).

**vector<unsigned int> CalcoloCamminoMinimo(const Polyhedron& P,**

**const unsigned int& v1,**

**const unsigned int& v2)**

Calcola il percorso più breve tra due vertici v1 e v2 all'interno del poliedro P, usando l'algoritmo di Dijkstra, dove il peso degli archi è la distanza euclidea tra vertici connessi. Restituisce un vector con gli ID dei vertici nel cammino minimo.

Passaggi della funzione:

* + 1. Estrae dati dal poliedro (Cell1DsExtrema, Cell0DsCoordinates)
    2. Costruisce il grafo (lista di adiacenza). Per ogni lato del poliedro:
* Calcola la distanza euclidea tra i vertici del lato
* Aggiunge un arco bidirezionale tra i due vertici con questo peso nella lista di adiacenza

1. Inizializza l'algoritmo di Dijkstra creando:

* Un array distanza[] con infiniti (eccetto v1, che ha distanza 0)
* Un array parent[] per tracciare i predecessori nel cammino
* Una min-heap (priority queue) con la coppia (distanza, nodo)

1. Esegue l'algoritmo di Dijkstra

* Estrae il nodo con distanza minima dalla heap
* Se è v2, interrompe: cammino trovato
* Altrimenti, esplora i suoi vicini: se trova un cammino più corto, aggiorna distanza[] e parent[] e aggiunge il vicino nella heap

1. Ricostruisce il cammino minimo

* Se la distanza verso v2 è ancora infinita → nessun cammino trovato
* Altrimenti: ripercorre parent[] da v2 a v1, inserisce ogni nodo nel vettore cammino, lo inverte per ottenere il cammino ordinato da v1 a v2

1. Stampa e restituisce il risultato (numero di lati del cammino minimo, lunghezza totale del cammino)
2. Restituisce il vettore cammino, che contiene gli ID dei vertici attraversati