UNIVERSITÀ DEGLI STUDI DI CATANIA

Dipartimento di Matematica e Informatica

Corso di Laurea Triennale in Informatica

Michele Ferro

Sviluppo di add-on per Blender: applicazioni nell'archeologia e nell'ingegneria edile

Relazione Progetto Finale

Relatore: prof. Giovanni Gallo

Anno Accademico 2019 - 2020

Introduzione

Questo progetto di Tesi è stato realizzato al fine di estrarre informazioni da modelli 3D mediante il software di modellazione grafica ***Blender***. Tali informazioni, saranno poi utili affinché si possa effettuare uno studio (anche mediante applicazioni di machine learning, a seconda delle necessità che lo specifico caso applicativo richiede) sui modelli presi in esame.

Si esaminano due diversi casi applicativi.
Il primo è quello **archeologico**, e prevede che vengano estratte informazioni sulle **caratteristiche geometriche** dai modelli delle pietre che compongono una parete, ottenuti per fotogrammetria. Grazie a queste informazioni, si potrebbe poi essere in grado di posizionare temporalmente – in modo approssimativo, facendo uso del machine learning – la data di costruzione di quella specifica parete antica.
Il secondo è quello **ingegneristico-edilizio**: la grande richiesta in campo edile, ha infatti portato ingegneri e architetti dall'utilizzo di "carta e penna" come strumenti di studio delle planimetrie degli edifici, a quello di avanzati software di disegno tecnico (è impossibile non citare, a tal proposito *Autocad* e lo stesso *Blender*). In questo specifico caso, si prevede di calcolare la totale superficie di modelli 3D in un ambiente Blender, e soprattutto di poter assegnare ad ogni modello, in base alle esigenze dell'utilizzatore, una proprietà che lo possa descrivere.

Prendendo come riferimento i casi applicativi analizzati, sono quindi stati sviluppati, in questo progetto di Tesi, due componenti aggiuntivi scritti in *Python*: si tratta di due **add-on** per il software di modellazione grafica 3D *Blender*, in grado di estrarre informazioni di carattere geometrico dai modelli presenti in una scena.

I componenti aggiuntivi sono stati realizzati per la produzione dell'articolo[1] ***Abstracting stone walls for visualization and analysis***.

Indice

Capitolo 1

Blender

1.1 Cos'è Blender?

Blender è un software di modellazione, animazione, montaggio video, composizione e rendering di immagini tridimensionali e bidimensionali; open source e multipiattaforma, esso compete nel campo artistico e cinematografico con altri noti software di modellazione 3D, quali ad esempio *Maya* e *3D Studio Max*.

L'interfaccia di Blender si può riassumere in una **struttura ad albero**, i cui elementi sono visivamente rappresentati in una scena.
Tale struttura è composta anzitutto da una radice, il nodo principale dell' albero che rappresenta l'intera scena; a cascata, seguiranno poi i vari oggetti, che siano mesh o meno, e le collezioni. Una scena può quindi contenere uno o più o oggetti e/o una o più collezioni, che a loro volta possono contenere uno o più oggetti.

Vista la presenza di un **interprete Python** specializzato per l'ambiente di Blender, è impossibile non citare la caratteristica che fa da base portante per questa Tesi, ossia lo scripting. Mediante lo scripting, è infatti possibile sfruttare la versatilità e potenza del suddetto linguaggio di programmazione per realizzare procedure automatizzate di ogni tipo: dalla modellazione alla scansione di modelli già esistenti all'interno di una scena Blender.

Inoltre, se la sua polifunzionalità non dovesse ancora essere adatta ad uno specifico utilizzo, Blender consente di installare dei moduli aggiuntivi, degli **add-on**, che ne ampliano ancora di più le funzionalità in modo nativo. Questa parentesi è molto importante per un passaggio di questo progetto, che pre-

vede la conversione da semplice script Python a componente aggiuntivo per Blender.

Le parole chiave di Blender necessarie per avere una più profonda comprensione di questa Tesi sono le seguenti:

- **Scena**: l'ambiente in cui è possibile posizionare oggetti di diverso tipo, come luci, telecamere oggetti; è l'equivalente digitale di un set cinematografico.
- **Oggetto**: oggetti di diverso tipo che è possibile posizionare all'interno di una scena, quali ad esempio mesh, luci e telecamere.
- **Collezione**: raggruppamento di oggetti. Quando si vogliono raggruppare oggetti sotto lo stesso nome a seconda del loro utilizzo, è preferibile utilizzare una collezione. Più collezioni possono coesistere nella medesima scena. Ogni collezione può contenere uno o più oggetti e/o una o più ulteriori collezioni.
- **Mesh**: tra tutti gli oggetti, i più comuni sono le mesh poligonali. Tradotto in italiano "=maglia", una mesh è un reticolo che definisce una forma in uno spazio, composta da vertici e spigoli. In modo più formale: una mesh M è una doppia (V, K) dove $V = \{v_i \in \mathbb{R}^3 \ \forall \ i = 1, \ldots, N_v\}$ è l'insieme dei vertici del modello, i quali quindi sono punti in $\mathbb{R}^3$, mentre K è l'informazione sull'adiacenza dei punti per formare uno spigolo o una faccia della mesh.

1.2 Blender API: gli Application Modules

Le API di base di Blender, conosciute sotto l'identificativo **`bpy`** (da "**B**lender" e "**Py**thon", quasi a voler evidenziare quanto tale linguaggio di programmazione sia importante come base del software) sono organizzate secondo nove diversi **moduli applicativi**, ognuno dei quali fa da "colonna portante" all'intero ambiente di Blender. Ogni singolo aspetto di Blender è gestito da un Application Module, il quale naturalmente comunica con gli altri restanti.
Nello specifico, essi sono:

- **Context Access** (`bpy.context`): i membri del modulo relativo al Context di Blender dipendono dalla specifica area di accesso. I valori di questi ultimi, in sola lettura (sebbene possano essere modificati mediante Data Access o Operators), contengono informazioni relative ad oggetti o collezioni selezionate e/o visibili in scena, preferenze, o all'attuale area di lavoro.

- **Data Access** (`bpy.data`): probabilmente il modulo più importante tra tutti, il quale viene utilizzato per la comunicazione tra Blender e Python. Qualsiasi informazione relativa agli oggetti in scena, geometrica o meno, è contenuta nei datablock, accessibile e modificabile mediante l'interprete proprio grazie a questo specifico modulo.

- **Message Bus** (`bpy.msgbus`): viene utilizzato per ricevere notifiche nel momento in cui le proprietà di un datablock vengono modificate mediante il Data Access. Una grossa limitazione al Message Bus è che alcuni tipi di aggiornamento – quali il movimento degli oggetti nella 3D Viewport, e i cambiamenti effettuati mediante il sistema di animazione – non possono essere notificati.

- **Operators** (`bpy.operators`): concede l'opportunità di richiamare e utilizzare operatori scritti in C, in Python o delle macro. Solo tre argomenti chiave possono essere utilizzati per definire un determinato operatore; inoltre, nessun operatore ha un valore di ritorno, in quanto ognuno di essi ritorna un `set()` così definito:

  ```
  {'RUNNING_MODAL', 'CANCELLED', 'FINISHED', 'PASS_THROUGH'}
  ```

  I valori di ritorno più comuni sono {`'FINISHED'`} e {`'CANCELLED'`}.
  Un esempio di operatore potrebbe essere il seguente:

  ```
  bpy.ops.test.operator(override_context, execution_context,
  undo)
  ```

  I tre argomenti, tutti opzionali, sono di seguito descritti:

  - `override_context` è di tipo **`dict`** e permette di modificare i membri al context visti dall'operatore;
  - `execution_context` è di tipo **`str` (enum)** e definisce il context nel quale l'operatore viene eseguito;
  - `undo` è di tipo **`bool`**.

- **Types** (`bpy.types`): qui sono raccolti tutti i tipi di dato utilizzati da Blender.

- **Utilities** (`bpy.utils`): contiene diverse funzioni specifiche di Blender ma non strettamente legate ai dati.

- **Path Utilities** (`bpy.path`): molto simile al modulo `os.path` di Python, contiene funzioni per gestire i path in Blender.

- **Application Data** (`bpy.app`): contiene valori sul software che restano invariati nel corso della sua esecuzione.
- **Property Definitions** (`bpy.props`): definisce proprietà per estendere i dati interni a Blender. Queste funzioni sono usate per assegnare proprietà a classi registrate direttamente mediante Blender, e non possono essere utilizzate in maniera diretta.

Capitolo 2

I componenti aggiuntivi realizzati

La finalità di questo progetto di Tesi è quella di ricavare informazioni geometriche su un gruppo di modelli 3D, così che successivamente possa essere effettuata un'analisi che, a seconda dello specifico caso applicativo, possa fornire dei risultati.
Dapprima, verrà descritto cosa effettivamente fanno ambedue i programmi lato back-end, su cosa operano, quali sono i dati grezzi di input e quali sono i loro effettivi prodotti finali, e quindi, qual è il loro comportamento generale.
Nei successivi due capitoli, verrà invece fatta una rassegna più precisa di ciò che computazionalmente elaborano i due componenti aggiuntivi, descrivendo parti di codice e le basi matematiche e geometriche che hanno portato alla scrittura quei frammenti, e di come è stato prodotto il front-end che costituisce i due add-on finali.

In entrambi i casi, si opera da un file con estensione `.blend`, contenente una serie di modelli che possono trovarsi non necessariamente all'interno di una collezione (potrebbero trovarsi semplicemente nella scena, pur non essendo contenuti da una collezione).
Tali modelli, sono costituiti da una serie di "**dati grezzi**", che in qualche modo devono essere prelevati, ordinati e messi da qualche parte, così che in seguito possa essere effettuata un'analisi di un certo tipo.

Per semplicità di esplicazione, d'ora in poi verrà supposto che tutte operazioni vengano effettuate su una singola mesh.

2.1 Point Cloud Stats

Questo primo add-on è stato realizzato allo scopo di prelevare informazioni di carattere geometrico da modelli, ottenuti mediante la fotogrammetria, rappresentanti rocce di una parete antica.
È giusto precisare che ognuna delle operazioni caratterizzanti questo software vengono effettuate con un certo grado di approssimazione, derivante anche dal processo di trasposizione da oggetto reale e tangibile, a fotografia, a modello tridimensionale.

Tutti i dati raccolti e/o calcolati potranno successivamente essere esportati su un file, per poi essere importati su un database sul quale effettuare analisi di machine learning per poter quanto meno stimare, per esempio, il periodo di costruzione della suddetta parete.

2.1.1 Scansione del modello e raccolta dei dati grezzi

I dati che vengono raccolti e/o calcolati sono:

- posizione e numero dei vertici della mesh nello spazio;
- normali e numero di facce;
- media e scarto quadratico della posizione dei vertici;
- media e scarto quadratico delle normali;
- raggio approssimativo della sfera minima contenente la mesh;
- tensore di inerzia della mesh;
- indice di planarità;
- normale media per ottante.

Il software analizza la struttura dati che la definisce e "cattura" le sue caratteristiche geometriche principali, quali le posizioni nello spazio 3D dei vertici che compongono la mesh e le sue normali. Questi, sono i dati grezzi che inizialmente il software ha a sua disposizione.
Naturalmente posizione, numero di vertici, normali e numero di facce sono le generalità più immediate che possono essere ricavate direttamente dalla mesh, senza effettuare alcun calcolo. Tutti gli altri dati, invece, vengono ricavati indirettamente mediante dei calcoli.

Bisogna precisare che ognuno dei valori citati finora è costituito da tre componenti, uno su ogni asse x, y e z.

2.1.2 Calcolo degli indici di tendenza centrale

I valori raccolti nel passo precedente stanno alla base di tutti i calcoli che vengono effettuati dal software per analizzare le caratteristiche geometriche del modello in questione.
Altri dati devono essere dedotti a partire da quelli già a disposizione.

La media aritmetica dei vertici è utile affinché possa essere calcolato il **punto medio** del modello (che definisce quindi la posizione di quest'ultimo nello spazio). Lo scarto quadratico medio è invece utile affinché si possa avere una stima della **dispersione** della posizione di tutti i vertici della mesh.
Inoltre, moltiplicando le tre componenti della deviazione standard dei vertici, è possibile avere una stima approssimata del volume del modello, dato che sarà utile in seguito.

La stessa operazione di media e di calcolo della deviazione standard viene effettuata anche sulle normali, così da avere degli indici di dispersione che riguardano le superfici del modello.

2.1.3 Stima di scala e rotazione della mesh

Fine ultimo questa analisi, è avere anche una misura dell'**ingombro** e dell'**orientamento** che il modello ha nello spazio.

L'ente matematico in grado di dare informazioni circa queste caratteristiche, che un corpo rigido nello spazio possiede, è il **tensore di inerzia**, una matrice rappresentata sinteticamente di seguito.

$$T = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \tag{2.1}$$

Successivamente verrà effettuata una descrizione analitica di tale matrice e del suo significato. Per il momento, ai fini della comprensione del comportamento del software, basti sapere che i vari $I_{\alpha\beta}$ con $\alpha, \beta = x, y, z$ sono i **momenti di inerzia** del corpo rigido – in questo caso della mesh poligonale – lungo la linea attraverso il centro di massa e parallela ai vari assi di rotazione.

Calcolando autovettori e autovalori che compongono questa matrice, è possibile ricavare una stima della rotazione e della dimensione di ingombro del modello in uno spazio tridimensionale.

Rispettivamente:

- gli **autovalori** esprimono la dimensione dell'ingombro;
- gli **autovettori** esprimono la rotazione dell'ingombro.

2.1.4 Indice di planarità e istogrammi delle normali

Due ulteriori funzioni sono rispettivamente quella di calcolo di un approssimativo **indice di planarità** e di un **istogramma delle normali** sugli otto quadranti nello spazio di Blender.

Il primo indice sorge dalla necessità di sapere quanto una mesh si discosta da un piano.
Mediante questo calcolo, infatti, si ottiene un indice in grado di suggerire quanto la pietra è vicina all'avere una superficie (si intende in questo caso la superficie visibile) regolare.

Il secondo nasce invece dalla necessità di avere ulteriori informazioni sulla forma della pietra nello spazio: per cui, vengono raggruppate le normali del solido secondo l'ottante di appartenenza, e per ognuno di essi viene calcolata una normale media: il risultato sarà un istogramma con otto bin – uno per ottante – da tre valori ciascuno (uno per ogni direzione).

2.1.5 Salvataggio dei valori raccolti e visualizzazione grafica

Per il momento, tutti i dati raccolti sono contenuti in delle semplici variabili interne alle procedure Python che compongono il software.
Il metodo standard per "salvare" questi dati in modo che vengano poi analizzati in un'altra sede, è quello di creare un file con valori separati da virgole (***CSV*** o ***Comma Separated Values***, nel gergo). Esso consiste in una tabella, le cui righe, o **record**, indicano i singoli modelli, mentre le colonne, o **attributi**, indicano le caratteristiche di questi, quali media, scarto quadratico medio, numero di vertici, ecc. Una volta raccolti tutti i dati, questi verranno inseriti in un apposito file CSV mediante lo script stesso, in modo che questo possa essere successivamente spostato altrove e letto, indipendentemente dal software.

È utile dare anche una visualizzazione grafica all'utente, in modo tale che egli possa capire immediatamente – senza dover osservare i valori numerici e dover leggere e comprendere il file CSV sopracitato – quali siano le caratteristiche geometriche delle mesh in scena. È ovvio che per dei semplici cubi sia banale capire – dando un semplice sguardo – quali possano essere le caratteristiche geometriche, ma tale approccio risulta essere molto utile per mesh complesse e irregolari (come il presente caso delle pietre che compongono una parete).
Tale visualizzazione grafica è il prodotto dei dati raccolti nei passi precedenti. Per evitare un **visual cluttering** (="disordine visivo") piuttosto serio, è stato stabilito di utilizzare delle sfere deformate in modo da sintetizzare tutte le caratteristiche geometriche delle mesh alle quali esse fanno riferimento.
In sostanza, per ogni modello, una procedura dell'add-on crea una sfera, la quale ha volume proporzionale a quello della mesh, e dimensioni e rotazione sui tre assi che vengono applicate a partire dagli autovalori ed autovettori del tensore di inerzia di quest'ultima.
Tali sfere – che da qui in poi verranno definite "***pills***" a causa della loro forma – verranno poi inserite in una collezione apposita, così che possa poi essere esportata altrove o analizzata separatamente ai modelli iniziali.

Per fornire all'utente più esperto una maggiore immediatezza nella lettura degli effettivi valori calcolati, è stata inoltre fornita un'interfaccia basata su un pannello, presente direttamente in 3D Viewport, nel quale vengono presentati tutti i dati – praticamente i medesimi esportati sul file CSV – raggruppati in vari tab, uno per categoria di dato.

Infine – ultimo aspetto da menzionare, ma non meno importante rispetto ai precedenti – è stata fornita all'utente la possibilità di interfacciarsi direttamente con Blender, implementando un campo di inserimento manuale di una proprietà (sempre accessibile dalla prima scheda suddetto pannello), la quale può brevemente descrivere la mesh in questione. Anche questo dato, naturalmente, verrà esportato direttamente sul file CSV.

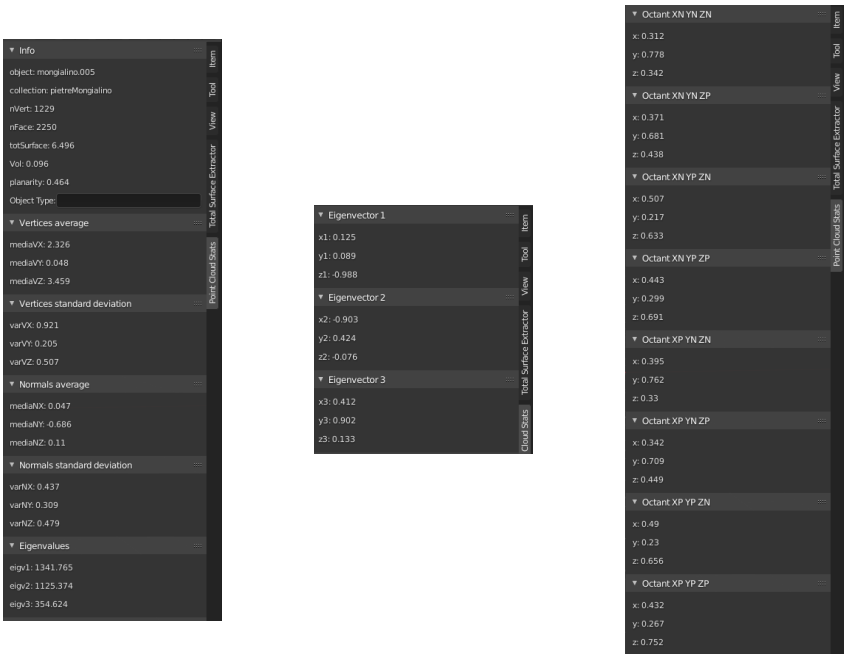
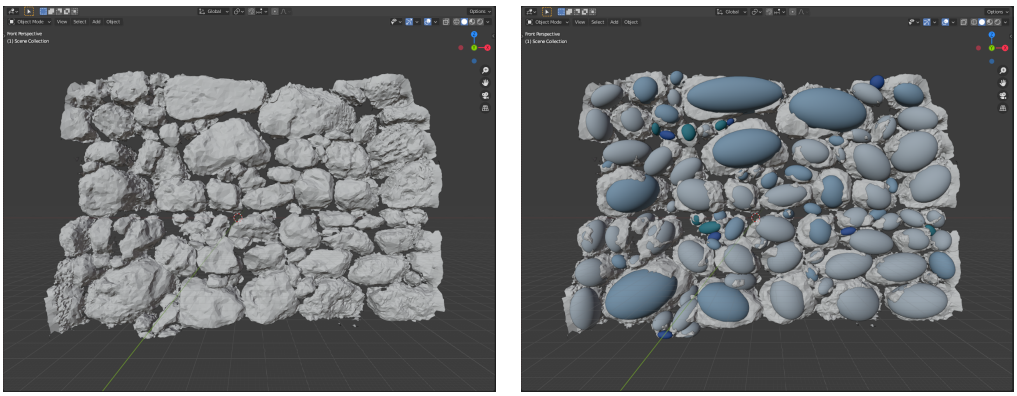


Figura 2.1: Pannello di visualizzazione dei dati raccolti; notare il campo di inserimento manuale “Object Type”

Di seguito, si presentano due esempi degli elementi in scena, prima e dopo l’esecuzione del software, ed il sopracitato pannello di raccolta dei dati.



(a) **(b)**

Figura 2.2: Esempio di una scena Blender di un muro pietre prima **(a)** e dopo **(b)** l’esecuzione dell’add-on

2.2 SurfEx

Questo specifico add-on è stato sviluppato pensando al bisogno, in campo edile, di un'estensione per Blender che permetta a ingegneri e architetti di poter effettuare in maniera rapida delle misure e soprattutto di poter raggruppare gli elementi presenti in un rilievo secondo diverse classi di oggetti, a seconda dei risultati di questi calcoli.

2.2.1 Scansione del modello e raccolta dei dati grezzi

I dati che vengono raccolti e/o calcolati sono:

- numero dei vertici della mesh nello spazio;
- numero di facce;
- superficie totale del solido.

Il meccanismo principale di raccolta dei dati è praticamente il medesimo adottato nella realizzazione del precedente add-on; in questo caso, i dati direttamente disponibili sono il numero di vertici della mesh poligonale ed il numero di facce, mentre la superficie totale del solido viene indirettamente calcolata partendo da queste ultime.

2.2.2 Salvataggio dei valori raccolti e visualizzazione grafica

In questo specifico caso, non è stata pensata nessuna particolare forma grafica di sintesi dei valori attenuti; tuttavia, l'utente sarà anche qui in grado di esportare in un file CSV i dati raccolti e calcolati, di visualizzare tutte le informazioni in un pannello implementato nella 3D Viewport, e soprattutto di aggiungere manualmente una proprietà aggiuntiva che possa caratterizzare il singolo modello analizzato.

Di seguito si presentano una serie di risultati[2], ottenuti da Sebastiano Russo Forcina, facendo uso di questa specifica estensione per Blender nell'analisi dei modelli poligonali dei Fontanoni dell'Acquanova, locati in Caltagirone.

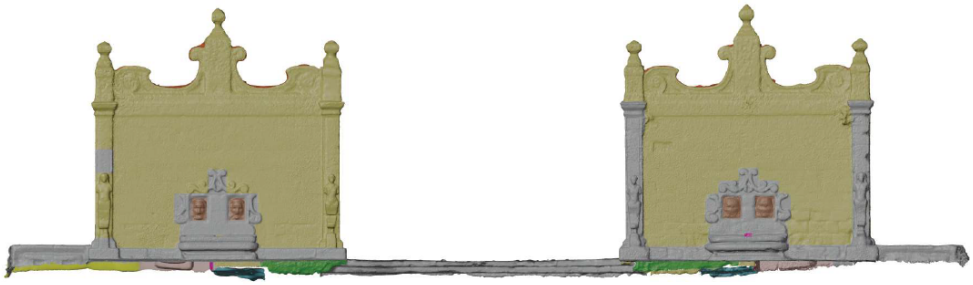


Figura 2.3: Prospetto anteriore in falsi colori

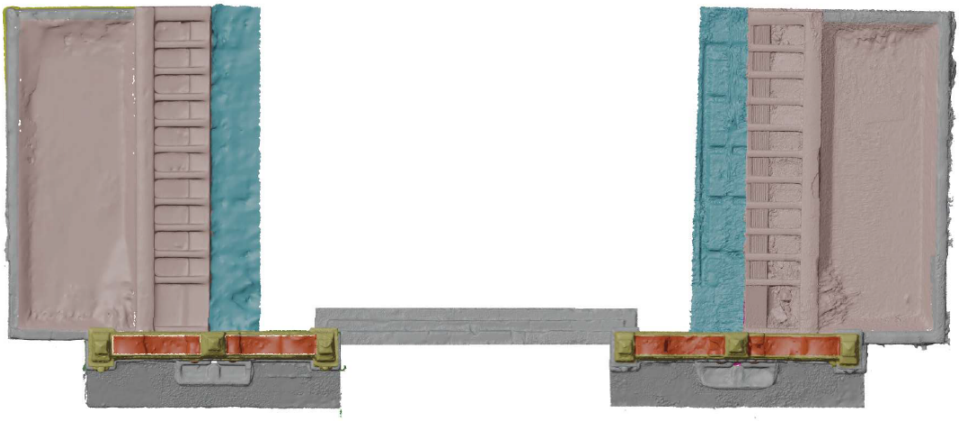


Figura 2.4: Planimetria in falsi colori

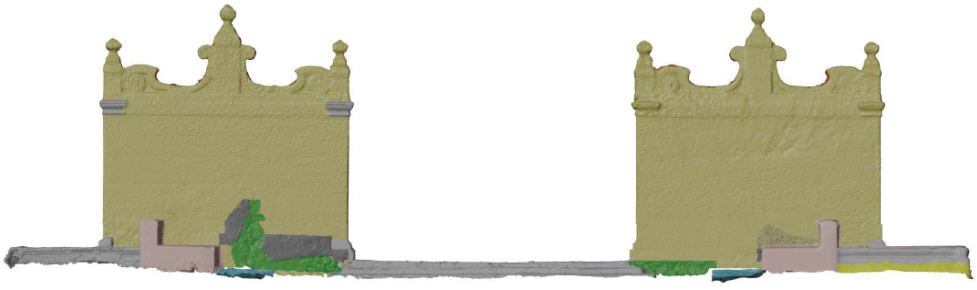


Figura 2.5: Prospetto posteriore in falsi colori

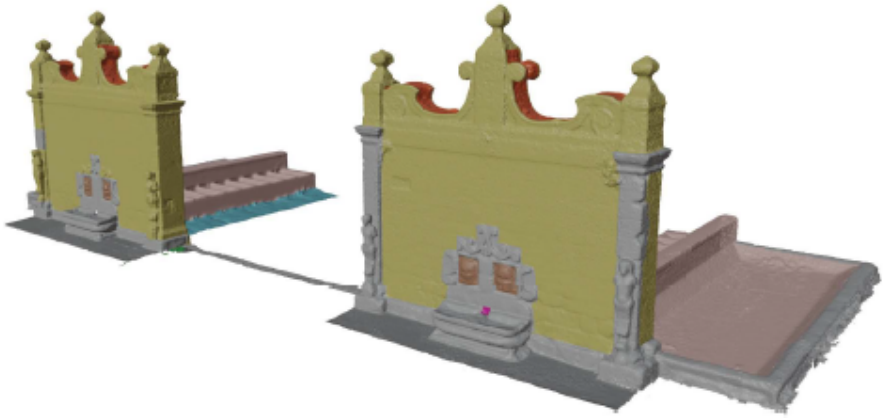


Figura 2.6: Vista prospettica in falsi colori

	collection	nVert	nFace	totSurface [m^2]
Arenaria	Materiali	885069	1759262	135,6
Calcarenite	Materiali	530931	1044319	65,36
Marmo	Materiali	7256	12940	1,3
Malta cementizia	Materiali	5920	11096	2,38
Cocciopesto	Materiali	22196	42108	8,52
Pietrame con malta retro	Materiali	4225	7903	1,95
Tonachina	Materiali	230262	457762	166,42
Conci si calcarenite e basalto sbozzati	Materiali	38765	75651	25,83
Laterizio e pietrame misto con malta	Materiali	12662	23308	2,79
Basalto lavico	Materiali	64130	125088	13,4
Elemento metallico	Materiali	1644	3037	0,84
Totale		1803060	3562474	424,41

Figura 2.7: Patina biologica sul prospetto principale

Figura 2.8: Degradi sui prospetti principali

oggetto		N° Vertici	N° Facce	Superficie [m^2]	
Fontana Nord		683170	1361847	103.521	
Fontana Sud		725096	1446813	104.498	
Lavatoi e abbeveratoi		343590	682314	236.695	
Scala		203253	402622	14.767	
Superficie totale del Modello				459.481	
D_Patina biologica		69252	128749	170.264	37,1%
D_Mancanza		2751	4341	3.393	0,7%
D_Vegetazione		4419	7711	6.753	1,5%
D_Fessurazione		114	103	0,053	0,0%
D_graffiti atti vandalici		564	954	1.365	0,3%
D_alterazione cromatica		1957	3513	2.751	0,6%
D_Erosione		8428	15834	7.671	1,7%
D_efflorescenza		2342	3923	8.038	1,7%
D_Dilavamento		17402	32703	35.903	7,8%

Capitolo 3

Back-end in dettaglio: Point Cloud Stats

Di seguito si descrive l'intero itinerario computazionale e matematico attraversato durante lo sviluppo di questa prima estensione per Blender, utilizzata al fine di analizzare i modelli, ottenuti mediante un processo di trasposizione fotogrammetrica, di rocce costituenti una parete antica.

3.1 Scansione dei modelli poligonali

Ogni singola operazione viene effettuata esclusivamente sugli oggetti visibili nella 3D Viewport di Blender, così da consentire all'utente di eventualmente escludere dall'analisi le mesh indesiderate, semplicemente utilizzando l'apposita funzione *"Hide"* già presente.
Ciò naturalmente significa che bisogna mantenere un riferimento a tutti quegli oggetti di tipo mesh (oggetti come eventuali telecamere e fonti di luce verranno pertanto escluse) visibili in scena.

Tra i membri della sezione "Screen" del modulo `bpy.context`, ossia quei membri relativi a ciò che viene visto su schermo dall'utente, ve ne è presente uno progettatto specificatamente per questo genere di situazioni, noto come **`bpy.context.visible_objects`**.
Esso altro non è che una lista contenente dei `bpy.types.Object` (detto in parole povere, contiene tutti i tipi di oggetti, mesh e non, visibili sulla scena). Ognuno di questi oggetti, inoltre, possiede un attributo che ne specifica il tipo mediante una stringa (enum). I possibili valori di ritorno sono:

```
['MESH', 'CURVE', 'SURFACE', 'META', 'FONT', 'HAIR',
'POINTCLOUD', 'VOLUME', 'GPENCIL', 'ARMATURE', 'LATTICE',
```

`'EMPTY', 'LIGHT', 'LIGHT_PROBE', 'CAMERA', 'SPEAKER']`

sebbene il tipo interessato in questa sede sia il **`'MESH'`**.

Di ogni mesh, nello specifico, devono essere considerati sia i vertici che le normali alle facce, elementi fondamentali per poter effettuare i vari calcoli sulla composizione geometrica di ogni singolo oggetto in scena.
Ambedue questi blocchi di informazioni sono accessibili mediante il modulo `bpy.data`.
È importante assicurarsi che i cambiamenti effettuati sulle mesh, quali spostamenti nello spazio, rotazioni, aggiustamenti di scala e quant'altro, siano stati effettivamente applicati, in quanto tutti i cambiamenti apportati vengono salvati sul datablock solo una volta che viene selezionata la voce *"Apply changes"*, altrimenti richiamabile mediante l'operatore

```
bpy.ops.object.transform_apply(location, rotation, scale)
```

I suoi argomenti (opzionali) **`location`**, **`rotation`** e **`scale`** sono tre **`bool`** che permettono all'utente di scegliere quale informazione deve essere sovrascritta sul datablock.

Codice 3.1: Salvataggio del datablock di ogni mesh visibile nella 3D Viewport

```
1 import bpy
2
3 objs = bpy.context.visible_objects.copy()
4
5 for obj in objs:
6      if obj.type == "MESH":
7           bpy.ops.object.transform_apply(location=True,
               rotation=True, scale=True)
8           obj_data = obj.data
```

3.1.1 Vertici

Il datablock di ogni singola mesh, possiede il membro **`vertices`**, che ritorna un **`MeshVertices`**, il quale è una collezione **`bpy_prop_collection`** di **`MeshVertex`**. I singoli vertici, sarebbero quindi i **`MeshVertex`** facenti parte di questa collezione, e ognuno dei quali possiede un attributo **`co`**, che restituisce le tre coordinate x, y e z del vertice preso in considerazione.
Tuttavia, è doveroso menzionare che queste tre coordinate sono **locali**, e quindi relative allo specifico sistema di riferimento della mesh presa in considerazione, e non all'intero mondo di Blender.

Tra le API non esiste funzione o membro che permetta di ricavare nativamente le coordinate **globali**, pertanto è necessario effettuare un prodotto riga per colonna tra due diverse matrici.
Ogni oggetto di Blender, è infatti caratterizzato da una matrice di trasformazione relativa al sistema di riferimento del mondo di Blender, nota come **`matrix_world`**.
Nel caso in cui l'oggetto in questione fosse posizionato nell'origine del sistema di riferimento, essa sarebbe caratterizzata come una matrice diagonale 4×4.

$$\texttt{matrix\_world} = \begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix} \tag{3.1}$$

Essa possiede informazioni sia sulla posizione che sulla rotazione di ogni oggetto "immerso" nel mondo di Blender.
Supponendo che l'oggetto sia indicato con `obj` e che `co.x`, `co.y` e `co.z` siano le sue coordinate locali, queste sarebbero rappresentate dalla seguente matrice 3×1.

$$\texttt{obj.co} = \begin{pmatrix} \texttt{co.x} \\ \texttt{co.y} \\ \texttt{co.z} \end{pmatrix} \tag{3.2}$$

Mediante l'operatore di prodotto matriciale `@` integrato in Python, si riuscirà così ad ottenere la matrice 3×1 delle coordinate globali dei vertici che compongono la mesh.
Ognuno di questi tre valori, viene quindi salvato in un'apposita lista, una per coordinata; nello specifico, le coordinate vengono salvate rispettivamente in `x_verts`, `y_verts` e `z_verts`.

Codice 3.2: Inserimento delle coordinate globali dei vertici in apposite liste separate

```
1  import bpy
2
3  objs = bpy.context.visible_objects.copy()
4
5  for obj in objs:
6       if obj.type == "MESH":
7            obj_data = obj.data
8
9            obj_verts = obj_data.vertices
10           m_world = obj.matrix_world
11           nVerts = len(obj_verts)
12
13           x_verts = []
```

```
14          y_verts = []
15          z_verts = []
16
17          for vert in obj_verts:
18               w_vert = m_world @ vert.co
19               x_verts.append(w_vert.x)
20               y_verts.append(w_vert.y)
21               z_verts.append(w_vert.z)
```

3.1.2 Normali

Per raccogliere le normali alle facce di una mesh, bisogna fare anzitutto riferimento alle superfici che la compongono. All'interno dei datablock, i dati ad esse relativi sono raccolte sotto il membro **`polygons`**. Similmente a quanto già visto con i vertici, tale membro ritorna un **`MeshPolygons`**, ossia una collezione **`bpy_prop_collection`** di **`MeshPolygon`**; le singole superfici che compongono la mesh sarebbero quindi i vari singoli `MeshPolygon`, ognuno dei quali possiede, tra i tanti, i seguenti attributi:

- **`area`**: restituisce il valore (in sola lettura) dell'area del poligono (un `float` in [0, inf]);
- **`normal`**: restituisce (in sola lettura) i tre versori x, y e z che compongono la normale al poligono (una lista di tre `float` in $[-1, 1]$).

Ognuno di questi tre valori, viene quindi salvato in un'apposita lista, una per coordinata; nello specifico, le coordinate vengono salvate rispettivamente in `x_normals`, `y_normals` e `z_normals`, mentre il numero dei poligoni e la somma delle loro aree sono rispettivamente salvati in `nFaces` e `totSurface`.

Codice 3.3: Inserimento dei versori delle normali in apposite liste separate, e calcolo del numero di facce e della superficie totale della mesh

```
1  import bpy
2
3  objs = bpy.context.visible_objects.copy()
4
5  for obj in objs:
6       if obj.type == "MESH":
7            obj_data = obj.data
8            obj_polygons = obj_data.polygons
9            nFaces = len(obj_polygons)
10
11           x_normals = []
12           y_normals = []
```

```
13          z_normals = []
14          totSurface = 0
15
16          for p in obj_polygons:
17               totSurface += p.area
18
19          for p in obj_polygons:
20               normals = p.normal
21               x_normals.append(normals[0])
22               y_normals.append(normals[1])
23               z_normals.append(normals[2])
```

3.2 Indici di posizione e dispersione

Come già anticipato nel paragrafo 2.2.1, i dati raccolti dalle mesh e discussi nel precedente capitolo, nella loro attuale forma grezza, non sono sufficientemente esplicativi per poter effettuare un'analisi statistica. È necessario utilizzare degli indicatori di posizione e dispersione, quali **media** e **deviazione standard,** per poter così ricavare delle informazioni che sono utili per poter descrivere ogni modello poligonale esaminato.

3.2.1 Media

Per tutte le implementazioni descritte qui di seguito è stata utilizzata una banale media aritmetica.

Codice 3.4: Funzione di calcolo della media aritmetica dei valori presenti in una lista

```
1 def media(L):
2      if len(L) != 0: return sum(L)/len(L)
3      else: return 0.0
```

Il frammento di codice in sovrimpressione calcola una media aritmetica dei valori presenti in una qualunque lista di valori numerici, e restituisce un valore pari a 0.0 nel caso in cui quest'ultima fosse vuota.

Nello specifico, tale funzione è stata utilizzata per calcolare il **vertice medio** e la **normale media** delle mesh.
Il primo, coinciderà esattamente con il **punto centrale** del modello poligonale (ed è quindi utile per ricavarne la posizione nello spazio, approssimando l'intero solido ad un punto), mentre la seconda corrisponde alla normale della superficie media in grado di sintetizzare l'intera mesh.
Questi valori, sono presentati mediante due terne:

- **vertsAvg = (x_vertsAvg, y_vertsAvg, z_vertsAvg)**, dove:
  - **x_vertsAvg** corrisponde alla media delle coordinate x di tutti i vertici della mesh;
  - **y_vertsAvg** corrisponde alla media delle coordinate y di tutti i vertici della mesh;
  - **z_vertsAvg** corrisponde alla media delle coordinate z di tutti i vertici della mesh;
- **normalsAvg = (x_normalsAvg, y_normalsAvg, z_normalsAvg)**, dove:
  - **x_normalsAvg** corrisponde alla media dei versori x di tutte le normali alle facce della mesh;
  - **y_normalsAvg** corrisponde alla media dei versori y di tutte le normali alle facce della mesh;
  - **z_normalsAvg** corrisponde alla media dei versori z di tutte le normali alle facce della mesh.

Codice 3.5: Calcolo del vertice e della normale media di ogni mesh visibile

```
1  import bpy
2
3  objs = bpy.context.visible_objects.copy()
4
5  for obj in objs:
6       if obj.type == "MESH":
7            obj_data = obj.data
8            obj_polygons = obj_data.polygons
9            nFaces = len(obj_polygons)
10
11           obj_verts = obj_data.vertices
12           m_world = obj.matrix_world
13           nVerts = len(obj_verts)
14
15           x_verts = []
16           y_verts = []
17           z_verts = []
18
19           x_normals = []
20           y_normals = []
21           z_normals = []
22           totSurface = 0
23
24           for vert in obj_verts:
```

```
25                w_vert = m_world @ vert.co
26                x_verts.append(w_vert.x)
27                y_verts.append(w_vert.y)
28                z_verts.append(w_vert.z)
29
30           for p in obj_polygons:
31                totSurface += p.area
32
33           for p in obj_polygons:
34                normals = p.normal
35                x_normals.append(normals[0])
36                y_normals.append(normals[1])
37                z_normals.append(normals[2])
38
39           x_vertsAvg = media(x_verts)
40           y_vertsAvg = media(y_verts)
41           z_vertsAvg = media(z_verts)
42
43           x_normalsAvg = media(x_normals)
44           y_normalsAvg = media(y_normals)
45           z_normalsAvg = media(z_normals)
46
47           vertsAvg = (x_vertsAvg, y_vertsAvg, z_vertsAvg)
48           normalsAvg = (x_normalsAvg, y_normalsAvg,
                z_normalsAvg)
```

3.2.2 Scarto quadratico medio

Per le implementazioni descritte in questo paragrafo, è stata utilizzata la seguente funzione di calcolo della deviazione standard.

Codice 3.6: Funzione di calcolo della deviazione standard dei valori presenti in una lista

```
1 def stDev(L):
2      m = media(L)
3      L = [(x-m)*(x-m) for x in L]
4      return pow(sum(L)/len(L),1/2)
```

Il frammento di codice sopra presentato descrive una funzione che calcola la deviazione standard (o scarto quadratico medio) di tutti i valori presenti in una lista, facendo uso delle funzioni `pow()` e `sum()` di Python per calcolare rispettivamente la potenza (in questo caso, la radice quadrata) e la somma dei valori presenti nella lista.

Nello specifico, il calcolo della deviazione standard è utile affinché si possa ottenere un indice affidabile della dispersione dei dati raccolti (da questo

punto di vista, la media già calcolata in precedenza non sarebbe infatti sufficiente). In questo modo, si ha a disposizione un indice che aiuti l'utente a capire in che modo variano localmente le coordinate dei vertici e delle normali di ogni mesh.
Esattamente come per i medi, anche questi valori sono presentati mediante due terne:

- **vertsStDev = (x_vertsStDev, y_vertsStDev, z_vertsStDev)**, dove:
  - **x_vertsStDev** corrisponde alla deviazione standard delle coordinate x di tutti i vertici della mesh;
  - **y_vertsStDev** corrisponde alla deviazione standard delle coordinate y di tutti i vertici della mesh;
  - **z_vertsStDev** corrisponde alla deviazione standard delle coordinate z di tutti i vertici della mesh;
- **normals_StDev = (x_normalsStDev, y_normalsStDev, z_normalsStDev)**, dove:
  - **x_normalsStDev** corrisponde alla deviazione standard dei versori x di tutte le normali alle facce della mesh;
  - **y_normalsStDev** corrisponde alla deviazione standard dei versori y di tutte le normali alle facce della mesh;
  - **z_normalsStDev** corrisponde alla deviazione standard dei versori z di tutte le normali alle facce della mesh.

Codice 3.7: Calcolo dello scarto quadratico medio dei vertici e delle normali di ogni mesh visibile

```
1  import bpy
2
3  objs = bpy.context.visible_objects.copy()
4
5  for obj in objs:
6       if obj.type == "MESH":
7            obj_data = obj.data
8            obj_polygons = obj_data.polygons
9            nFaces = len(obj_polygons)
10
11           obj_verts = obj_data.vertices
12           m_world = obj.matrix_world
13           nVerts = len(obj_verts)
14
```

```
15          x_verts = []
16          y_verts = []
17          z_verts = []
18
19          x_normals = []
20          y_normals = []
21          z_normals = []
22          totSurface = 0
23
24          for vert in obj_verts:
25               w_vert = m_world @ vert.co
26               x_verts.append(w_vert.x)
27               y_verts.append(w_vert.y)
28               z_verts.append(w_vert.z)
29
30          for p in obj_polygons:
31               totSurface += p.area
32
33          for p in obj_polygons:
34               normals = p.normal
35               x_normals.append(normals[0])
36               y_normals.append(normals[1])
37               z_normals.append(normals[2])
38
39          x_vertsStDev = stDev(x_verts)
40          y_vertsStDev = stDev(y_verts)
41          z_vertsStDev = stDev(z_verts)
42
43          x_normalsStDev = stDev(x_normals)
44          y_normalsStDev = stDev(y_normals)
45          z_normalsStDev = stDev(z_normals)
46
47          vertsStDev = (x_vertsStDev, y_vertsStDev,
               z_vertsStDev)
48          normals_StDev = (x_normalsStDev, y_normalsStDev,
               z_normalsStDev)
```

3.3 Tensore di inerzia

L'ente matematico più consono per descrivere le più importanti caratteristiche geometriche di un corpo solido, quali dimensione e rotazione, è il **tensore di inerzia**.
La potenza e l'efficacia di questo ente, in contesti di questo tipo, stanno nel fatto che mediante esso non solo si è in grado di ricavare questi due tipi di trasformazione, ma applicando un procedimento di decomposizione, si riesce anche ad avere una stima più o meno grossolana di un solido di rotazione,

detto **elissoide di inerzia**, in grado di rappresentare lo spazio di ingombro dell'intero solido.
Come qualsiasi altro tensore, esso altro non è che una "matrice multidimensionale", che in questo caso raccoglie tutti i momenti di inerzia di un qualsiasi solido definito in uno spazio e dotato di un centro di massa.

3.3.1 Momento di inerzia

Un singolo **momento di inerzia** misura l'**inerzia** (ossia, la resistenza alle variazione dello stato di moto) di un corpo al variare della sua velocità angolare attorno ad uno specifico asse.

Supponiamo di considerare un asse di rotazione $\hat{z}$ e un sistema di n punti materiali. Con r_i si indicano le distanze di ognuno di questi punti dall'asse e con m_i le rispettive masse (naturalmente, per $i = 1, \ldots, n$).
Allora il momento di inerzia rispetto all'asse, si definisce come:

$$I_{\hat{z}} = \sum_{i=1}^{n} m_i r_i^2 \tag{3.3}$$

Bisogna tuttavia considerare che lo spazio tridimensionale è un insieme $\mathbb{R}^3$, basato su tre assi cartesiani x, y e z, che possono essere considerati alla stregua dell'asse di rotazione $\hat{z}$. Per cui, precisando che un solido generico può avere diversi momenti di inerzia (a causa della non simmetria di quest'ultimo), possono essere presi in esame i seguenti momenti:

- I_{xx} = momento di inerzia lungo la retta che attraversa il centro di massa e parallela all'asse delle x;
- I_{yy} = momento di inerzia lungo la retta che attraversa il centro di massa e parallela all'asse delle y;
- I_{zz} = momento di inerzia lungo la retta che attraversa il centro di massa e parallela all'asse delle z.

Ognuno dei momenti si può definire come:

$$I_{ij} = \sum_{l} m_l((x_l)_k(x_l)_k\delta_{ij} - (x_l)_i(x_l)_j) \tag{3.4}$$

dove l indica la l-esima componente della distribuzione di masse e δ_{ij} è il **delta di *Kronecker***[3], ossia una funzione booleana così definita:

$$\delta_{ij} = \begin{cases} 1 & \text{se } i = j \\ 0 & \text{altrimenti} \end{cases} \tag{3.5}$$

3.3.2 Calcolo della matrice

Considerando i tre assi cartesiani x, y e z come assi di rotazione, e facendo utilizzo della formula (3.4), si ottiene così la matrice (2.1).
Infatti:

$$\bar{\bar{I}} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} = \sum_{i=1}^{n} m_i \begin{bmatrix} (y_i^2 + z_i^2) & -x_i y_i & -x_i z_i \\ -x_i y_i & (x_i^2 + z_i^2) & -y_i z_i \\ -x_i z_i & -y_i z_i & (x_i^2 + y_i^2) \end{bmatrix} \tag{3.6}$$

In questo specifico caso, dovendo solo calcolare le caratteristiche geometriche (e non fisiche, quindi) di ogni singolo modello poligonale, si suppone che per ognuno di essi la massa sia **unitaria**, e che il centro di massa coincida col **centro geometrico** del solido; per cui si ottiene:

$$\bar{\bar{I}} = \sum_{i=1}^{n} \begin{bmatrix} (y_i^2 + z_i^2) & -x_i y_i & -x_i z_i \\ -x_i y_i & (x_i^2 + z_i^2) & -y_i z_i \\ -x_i z_i & -y_i z_i & (x_i^2 + y_i^2) \end{bmatrix} \tag{3.7}$$

Per calcolare tale matrice, sono state implementate due diverse funzioni: una per il calcolo dei momenti di inerzia sulla **diagonale principale**, e una per il calcolo sulle **triangolari strettamente superiore ed inferiore**.

Codice 3.8: Funzione di calcolo dei momenti di inerzia sulla diagonale principale

```
1 def momentoInerziaDiagonale(V,W):
2     V = [x*x for x in V]
3     W = [x*x for x in W]
4     return sum(V)+sum(W)
```

Il frammento di codice sopra mostrato, prende infatti in input due liste `V` e `W`, ognuna contenente in serie una singola coordinata x, y o z dei due vertici ai quali esse fanno riferimento, ne calcola il quadrato (in questo caso, si applica la (3.4) con $\delta_{ij} = 1$), per poi restiturne la somma.

Codice 3.9: Funzione di calcolo dei momenti di inerzia sulle triangolari strettamente superiore ed inferiore

```
1 def momentoInerziaMisto(V,W):
2     inerzia = 0
3     for i in range(len(V)):
4         inerzia = inerzia - (V[i]*W[i])
5     return inerzia
```

La funzione in sovrimpressione, richiamata per il calcolo dei momenti di inerzia fuori dalla diagonale principale, applica lo stesso tipo di operazione; tuttavia, si ricorda che in questo caso nella (3.4) si ha $\delta_{ij} = 0$, per cui, per ogni elemento presente nelle due liste, si ritorna l'opposto del loro prodotto.

Di seguito si mostra il codice, scritto nel flusso di esecuzione principale dello script, per l'ottenimento del tensore di inerzia.
Per questioni di brevità, si consideri il seguente codice indentato nel ciclo `for` del **Codice 3.1** e che tutti gli altri dati, quali coordinate di vertici e normali, siano già stati raccolti.

Codice 3.10: Calcolo del tensore di inerzia

```
# tolgo l'offset del centro dell'oggetto
x_v=[x-x_vertsAvg for x in x_verts]
y_v=[y-y_vertsAvg for y in y_verts]
z_v=[z-z_vertsAvg for z in z_verts]

I_xx = momentoInerziaDiagonale(y_v, z_v)
I_yy = momentoInerziaDiagonale(x_v, z_v)
I_zz = momentoInerziaDiagonale(x_v, y_v)
I_xy = momentoInerziaMisto(x_v, y_v)
I_xz = momentoInerziaMisto(x_v, z_v)
I_yz = momentoInerziaMisto(y_v, z_v)

T_matrix = [[I_xx, I_xy, I_xz],\
            [I_xy, I_yy, I_yz],\
            [I_xz, I_yz, I_zz]]
```

Come è possibile notare dal frammento di codice in sovrimpressione, onde evitare imprecisioni nei calcoli dovute al sistema di riferimento scelto (il globale rispetto al locale, o viceversa), è giusto togliere l'offset dal centro della mesh, le cui coordinate, si ricorda, sono contenute nelle liste `x_vertsAvg`, `y_vertsAvg` e `z_vertsAvg`. Per cui, le coordinate corrette sono inserite nelle tre nuove liste **`x_v`**, **`y_v`** e **`z_v`**.
Successivamente, i singoli membri della matrice (i vari momenti di inerzia) vengono calcolati e poi inseriti in una lista di tre liste da tre elementi ciascuna chiamata **`T_matrix`**, la quale è l'equivalente della matrice (3.7).

3.3.3 Calcolo degli autovalori e autovettori

Nella sua attuale forma, il tensore in sé non fornisce informazione alcuna sulla geometria della singola mesh presa in considerazione, se non i momenti di inerzia presi in riferimento alle varie combinazioni di assi di rotazione.
Osservando tuttavia la (3.7), e soprattutto, considerando il sistema di riferimento degli assi cartesiani, ci si può rendere conto che secondo il **teorema spettrale**[4], applicando un processo di **decomposizione**, è possibile

ricondurla alla seguente matrice diagonale:

$$\bar{\bar{I}} = \begin{bmatrix} I_1 & 0 & 0 \\ 0 & I_2 & 0 \\ 0 & 0 & I_3 \end{bmatrix} \tag{3.8}$$

Nello specifico, bisogna trovare una **base ortonormale**, ossia una base di vettori ortogonali di norma uno rispetto al prodotto scalare definito sullo spazio vettoriale, e tale base può essere considerata il sistema di riferimento individuato dai tre assi x, y e z nel mondo di Blender.
In tal modo, si riusciranno a ottenere gli autovalori e gli autovettori della matrice (3.7), i quali saranno particolarmente utili in seguito per trovare un solido di rotazione, un elissoide, in grado di approssimare l'ingombro dell'intera mesh.

Per calcolare questi due enti, è stata utilizzata la funzione **`utu(M)`**, la quale effettuando k decomposizioni QR, restituisce due ulteriori matrici:

- la matrice U (3×3), le cui colonne sono gli **autovettori** della matrice M presa in input;
- la matrice T (3×3), i cui elementi sulla diagonale principale sono invece gli **autovalori** della matrice M di input.

Codice 3.11: Funzione principale di decomposizione di una matrice

```
1  def utu(M):
2       U = [[1,0,0], [0,1,0], [0,0,1]]
3       T = M.copy()
4       k = 5
5       for i in range(k):
6            qr = decomposizioneQR(T)
7            A = prodotto(qr[1],qr[0])
8            U1 = U.copy()
9            U = prodotto(U1,qr[0])
10      return (T,U)
```

Relativamente alla funzione sopra presentata, è importante notare una serie di aspetti.
Essa richiama k volte nel suo corpo le due funzioni **`decomposizioneQR(A)`** e **`prodotto(A,B)`**, le quali verranno discusse in seguito; la scelta k non è casuale: all'aumentare del valore di tale variabile, aumenta infatti la **precisione** di decomposizione della matrice, a discapito del tempo impiegato nell'esecuzione della funzione; solitamente si pone $k = 10$, ma poiché il contesto attuale non richiede una elevatissima precisione, ma piuttosto una buona

velocità di esecuzione, come miglior valore di compromesso si è scelto $k = 5$, che garantisce una buona precisione, ma allo stesso tempo brevi tempi di calcolo.
Nel suo ciclo `for` eseguito cinque volte, viene anzitutto richiamata – come già accennato in precedenza – la funzione `decomposizioneQR(A)`.

Codice 3.12: Funzione di decomposizione QR

```
def decomposizioneQR(A):
     # 1
    m1 = modulo([A[0][0], A[1][0], A[2][0]])
    q00 = A[0][0]/m1
    q10 = A[1][0]/m1
    q20 = A[2][0]/m1

     # 2
    s = A[0][1]*q00 + A[1][1]*q10 + A[2][1]*q20

    a01 = A[0][1] - s*q00
    a11 = A[1][1] - s*q10
    a21 = A[2][1] - s*q20
    m2 = modulo([a01,a11,a21])
    q01 = a01/m2
    q11 = a11/m2
    q21 = a21/m2

     # 3
    s0 = A[0][2]*q00 + A[1][2]*q10 + A[2][2]*q20
    s1 = A[0][2]*q01 + A[1][2]*q11 + A[2][2]*q21

    a02 = A[0][2] - s0*q00 - s1*q01
    a12 = A[1][2] - s0*q10 - s1*q11
    a22 = A[2][2] - s0*q20 - s1*q21
    m3 = modulo([a02, a12, a22])
    q02 = a02/m3
    q12 = a12/m3
    q22 = a22/m3

     # 4
    r10 = 0
    r20 = 0
    r21 = 0
    r00 = m1
    r11 = m2
    r22 = m3
    r01 = s
    r02 = s0
    r12 = s1

```

```
42      Q=[[q00,q01,q02], [q10,q11,q12], [q20,q21,q22]]
43      R=[[r00,r01,r02], [r10,r11,r12], [r20,r21,r22]]
44      return [Q,R]
```

Questa funzione, presa in input una qualunque matrice quadrata M, applica la seguente scomposizione:

$$M = QR \tag{3.9}$$

dove:

- Q è una **matrice ortonormale**;
- R è una **matrice triangolare superiore**.

Il corpo di tale funzione può essere descritto secondo i seguenti punti.

1. Viene calcolata la **prima colonna** della matrice Q, la quale è uguale alla prima colonna della matrice A, ma normalizzata;
2. Viene calcolata la **seconda colonna** della matrice Q: nello specifico, viene prima calcolata la lunghezza s della proiezione della seconda colonna della matrice A sulla prima colonna di Q; poi alla seconda colonna della matrice A viene sottratto un vettore lungo s e parallelo alla prima colonna di Q. Ciò che resta è invece perpendicolare alla prima colonna della matrice Q, e viene normalizzato.
3. Viene calcolata la **terza colonna** della matrice Q secondo il medesimo procedimento di prima.
4. Vengono costruite le matrici Q ed R secondo i valori ottenuti.

Per effettuare la normalizzazione, è stata implementata invece la seguente funzione:

Codice 3.13: Funzione di normalizzazione

```
1 def modulo(v):
2     w = [x*x for x in v]
3     return (sum(w))**(1/2)
```

Nello specifico, essa prende in input una lista `v` e per ogni valore al suo interno calcola il quadrato, per poi ritornarne la radice quadrata della somma.

Una volta trovate le due matrici Q ed R, viene calcolato il loro prodotto mediante la funzione **`prodotto(A,B)`** (descritta qui di seguito), il quale viene inserito nella variabile `U`.
Vine infine ritornata la coppia `(T,U)` delle matrici T ed U.

Codice 3.14: Funzione di prodotto tra due matrici

```
def prodotto(A,B):
    # A e B sono due liste di liste
    # che descrivono due matrici (MxN) e (NxK)
    rowA = len(A)                  # righe di A
    colA = len(A[0])               # colonne di A
    rowB = len(B)                  # righe di B
    colB = len(B[0])               # colonne di B
    C = []
    for i in range(rowA):
        C.append([0 for x in range(colB)])
         # calcoliamo i prodotti
    for i in range(rowA):
        for j in range(colB):
            for k in range(rowA):
                C[i][j] = C[i][j] + A[i][k]*B[k][j]
    # restituiamo C
     return C
```

3.3.4 Diagonali e rotazione dell'elissoide

Come già accennato nei paragrafi precedenti, lo scopo del calcolo del tensore di inerzia, dei suoi autovalori e dei suoi autovettori, è quello di ottenere – per ogni singola mesh visibile in 3D Viewport – un solido di rotazione, detto **elissoide di inerzia**, che ne possa al meglio approssimare lo spazio di ingombro, e che soprattutto ne possa costituire un rapido indice di valutazione agli occhi di un utente meno interessato ai valori numerici.

Mediante la decomposizione della matrice (3.7), sono stati ottenuti dei sistemi di autovalori e autovettori, utili al fine ultimo di generazione dell'elissoide. Tuttavia, quest'operazione è preceduta da una serie di calcoli che permetteranno di ottenere le variabili prese in input dalla funzione ad essa dedicata.
In particolare:

- gli autovalori possono essere utilizzati per ricavare le **diagonali** sui tre assi dell'elissoide;
- gli autovettori possono essere raccolti in una matrice che ne descrive la **rotazione** nello spazio.

Attraverso la matrice diagonale (3.8) è possibile ottenere i tre autovalori I_1, I_2 e I_3, dove:

$$I_1 \leq I_2 \leq I_3$$

In realtà, essi sono inversamente proporzionali alle tre diagonali del suddetto elissoide. Infatti, supponendo che la lunghezza delle tre digonali su ogni singolo asse sia rispettivamente espressa come D_x, D_y e D_z, si ha:

$$\begin{aligned} D_x &= 1/\sqrt{I_1} \\ D_y &= 1/\sqrt{I_2} \\ D_z &= 1/\sqrt{I_3} \end{aligned} \tag{3.10}$$

È importante tuttavia ricordare che i valori ottenuti mediante i calcoli sulle coordinate dei vertici sono molto piccoli, approssimativamente vicini allo zero. Pertanto, è necessario attuare un **aumento in scala** ed una successiva **normalizzazione**.
Di seguito, si riporta il frammento di codice che esegue le operazioni descritte.

Codice 3.15: Calcolo delle diagonali dell'elissoide di inerzia

```
1 eigvals, eigvecs = utu(T_matrix)
2 autovalori = [eigvals[0][0], eigvals[1][1], eigvals[2][2]]
3 scaleFac = 3
4 diags = [(abs(autovalori[0]))**(-1/2), (abs(autovalori[1]))
      **(-1/2), (abs(autovalori[2]))**(-1/2)]
5 norm_diags = normalize(diags)
```

In particolare, prima viene effettuata la decomposizione della matrice `T_matrix` generata secondo quanto visto nel **Codice 3.10**; gli autovalori e gli autovettori vengono salvati nelle rispettive variabili **`eigvals`** ed **`eigvecs`** facendo uso della funzione `utu(M)` (**Codice 3.11**). Gli autovalori vengono ulteriormente salvati nella lista **`autovalori`** per una più rapida accessibilità.
Le diagonali vengono poi calcolate (secondo la formula (3.10)) e salvate in valore assoluto (mediante la funzione `abs()` di Python) nella lista **`diags`**.
Successivamente, la lista delle diagonali viene normalizzata attraverso la funzione **`normalize(eig)`** presentata qui di seguito.

Codice 3.16: Funzione di normalizzazione degli autovalori

```
1 def normalize(eig):
2     tot = sum([abs(x) for x in eig])
3     eig = [x/tot for x in eig]
4     return eig
```

Dopo una serie di test, è stato verificato che il miglior fattore di scala di rappresentazione degli elissoidi è **`scaleFac=3`**; valori maggiori o minori avrebbero infatti portato a elissoidi troppo grandi o troppo piccoli.
Per aumentare ancora di più l'accuratezza di rappresentazione del solido di rotazione, è stato inoltre calcolato il **raggio della sfera minima**, centrata

nel baricentro (il quale coincide in questo caso con il centro geometrico), che include tutti i vertici della mesh. Questo verrà successivamente passato alla funzione di generazione della mesh sferica che verrà descritta nel prossimo paragrafo.

Codice 3.17: Stima del raggio della sfera minima centrata nel baricentro che include tutti i vertici della mesh

```
raggi = []
for i in range(len(x_v)):
    raggi.append(x_v[i]**2 + y_v[i]**2 + z_v[i]**2)
raggioSfera = max(raggi)**(1/2)
```

Nello specifico, in una lista denominata **`raggi`** vengono inserite, per ogni mesh, le somme dei quadrati di ogni singola coordinata dei vertici. Di questa lista, viene poi presa la radice quadrata del massimo.

Viene infine costruita la **matrice di rotazione**, basata sugli autovettori precedentemente calcolati. Essa fornirà un chiaro indice di come ogni singola mesh è ruotata nello spazio.

Codice 3.18: Costruzione della matrice di rotazione

```
blenderT_Matrix = mathutils.Matrix((eigvecs[0], eigvecs[1],
    eigvecs[2]))
```

Essa viene costruita, nel codice in sovrimpressione, mediante la funzione **`mathutils.Matrix()`**. Essa altro non è che una funzione appartenente al modulo standalone **`mathutils`** di Blender, il quale non è strettamente legato alle sue API di base, ma del quale il software fa largo utilizzo per l'implementazione di matrici e vettori.
In particolare, la funzione `mathutils.Matrix()` richiama il costruttore alla classe **`Matrix`** del modulo e, prese in input le tre liste `eigvecs[0]`, `eigvecs[1]` ed `eigvecs[2]`, restituisce una matrice, la quale non viene implementata come lista di liste per questioni di compatibilità con la matrice di rotazione delle mesh di Blender.

3.3.5 Generazione dell'elissoide in 3D Viewport

È adesso possibile generare, direttamente sulla 3D Viewport, l'elissoide di inerzia, il quale sarà una mesh sferica avente come centro il centro della mesh di partenza (la pietra in questione), e opportunamente trasformata mediante i sistemi di autovalori ed autovettori discussi e ottenuti nei precedenti paragrafi.

Codice 3.19: Generazione della sfera deformata

```
1 bpy.ops.mesh.primitive_uv_sphere_add(radius = raggioSfera *
      scaleFac, location=vertsAvg, scale=(norm_diags[0],
      norm_diags[1],norm_diags[2]))
2 bpy.ops.object.shade_smooth()
3 pill = bpy.context.active_object
4 pill.name = obj.name + '_pill'
5 pill.rotation_euler = blenderT_Matrix.to_euler('XYZ')
```

Nel frammento di codice soprastante viene utilizzato l'operatore

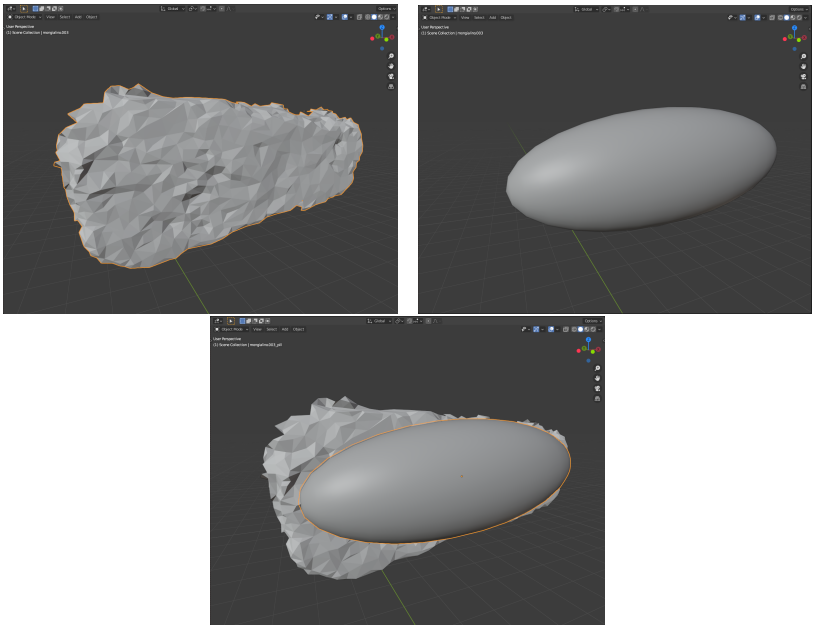
`bpy.ops.mesh.primitive_uv_sphere_add(radius,location,scale)`

per poter generare una mesh primitiva di tipo "UV Sphere". I tre parametri sono opzionali, ma per velocizzare il processo di deformazione sono stati passati alla funzione operatore i seguenti valori:

- **`radius=raggioSfera*scaleFac`**: grazie a questo parametro, la sfera avrà raggio pari a quello della sfera minima calcolata nel **Codice 3.17**, moltiplicata per il fattore di scala `scaleFac=3`;
- **`location=vertsAvg`**: posiziona la sfera nello stesso punto dello spazio in cui è centrata la mesh alla quale fa riferimento;
- **`scale=(norm_diags[0],norm_diags[1],norm_diags[2])`**: deforma la sfera ponendo la lunghezza delle sue diagonali sui tre assi pari agli autovalori normalizzati calcolati come visto nel **Codice 3.15**.

La mesh viene poi salvata in una variabile **`pill`** mediante la chiamata **`bpy.context.active_object`** (essa è infatti la mesh attiva in questo istante di esecuzione). Il salvataggio della mesh nella variabile, è utile più che altro al fine di applicare ulteriori cambiamenti su di essa: innanzitutto viene rinominata usando il nome della mesh alla quale fa riferimento e concatenando a tale stringa il suffisso "**`_pill`**". In secondo luogo, essa viene ruotata usando la matrice di rotazione generata nel frammento di **Codice 3.18**. Per la conversione in matrice di rotazione euleriana è stata usata la funzione **`to_euler('XYZ')`**. Il parametro passato permette di scegliere un sistema di riferimento (in questo caso quello basato sugli assi cartesiani x, y e z) piuttosto che un altro.
Viene inoltre usato l'operatore **`bpy.ops.object.shade_smooth()`** per smussare la mesh e renderla visivamente più uniforme.

(a) **(b)**



(c)

Figura 3.1: Esempio di generazione di una "pill" su una mesh rocciosa: nella **Figura (a)** la singola mesh di partenza; nella **Figura (b)** l'elissoide generato; nella **Figura (c)** il paragone tra le due mesh

3.4 Indice di planarità

Un altro valore estremamente utile al fine di classificazione della pietra potrebbe essere un indice che approssimativamente esprima quanto la superficie visibile di ogni singola pietra è vicina all'essere "piana": per tale motivo, è stato pensato un **indice di planarità**, il quale è compreso in un intervallo $[0, 1]$: naturalmente, quanto è più grande, tanto la superficie frontalmente visibile del modello tridimensionale della pietra è vicina all'essere regolare; viceversa, una pietra più lontana ad una forma planare avrà un indice più basso.

È importante considerare come questo indice potrà essere utilizzato per fornire un ulteriore aiuto visivo all'utente, mediante una colorazione – secondo una palette di colori ben definita – delle *pill* create in precedenza.

3.4.1 Calcolo dell'indice

Per il calcolo di questo indice, è stato scelto di fare utilizzo della **distanza** (in valore assoluto) tra un generico **piano perpendicolare** alla normale media della mesh (già ricavata in precedenza), e di normalizzare il valore risultante.

Codice 3.20: Creazione di un piano perpendicolare alla normale media della mesh

```
1 bpy.ops.mesh.primitive_plane_add(size = 1.0, location =
      vertsAvg)
2 bpy.ops.object.origin_set(type='ORIGIN_GEOMETRY', center='
      MEDIAN')
3 plane = bpy.context.active_object
4 plane.hide_set(True)
5 plane.name = obj.name + '_plane'
6 plane.rotation_euler[0] = math.radians(90)
7 plane.location = plane.location + mathutils.Vector(normalsAvg)
```

In maniera abbastanza simile al **Codice 3.19**, mediante il codice sopra presentato viene generato nello spazio di Blender una mesh di un piano di dimensione unitaria (secondo il sistema di riferimento adottato dal software) e centrato nel centro del modello poligonale della pietra. Successivamente, viene salvato il riferimento a quest'ultimo così da poter essere in grado di apportare le modifiche necessarie al calcolo dell'indice.
Essendo per nulla importante ai fini della visualizzazione grafica, il piano viene immediatamente **nascosto** dalla 3D Viewport (onde evitare visual cluttering), sebbbene tutte le sue informazioni spaziali siano accessibili – e modificabili – attraverso il suo datablock. Si ricordi inoltre, che per ogni mesh, queste righe di codice vengono eseguite in un arco di tempo così breve che l'utente non si rende conto che questi piani vengono generati.
Successivamente, il piano viene opportunamente ruotato e spostato facendo uso della variabile `normalsAvg`, così da renderlo perpendicolare alla normale media.

Codice 3.21: Calcolo della distanza media tra la mesh ed il piano perpendicolare alla sua normale media

```
1 dists = []
2 for vert in obj_verts:
3       w_vert = m_world @ vert.co
4       dist = distEuclide(w_vert, plane.location)
5       dists.append(dist)
6
7 avgDist = media(dists)
8 planarity = avgDist / raggioSfera
9
10 bpy.data.objects.remove(plane, do_unlink=True)
```

Una volta generato il piano, viene calcolata la **distanza euclidea** (secondo le coordinate globali) tra tutti i punti della mesh della pietra, ed il centro del piano generato. Di seguito si presenta la funzione implementata per il calcolo della distanza.

Codice 3.22: Funzione di calcolo della distanza euclidea

```
def distEuclide(vert_source, vert_dest):
    xS = vert_source[0]
    yS = vert_source[1]
    zS = vert_source[2]

    xD = vert_dest[0]
    yD = vert_dest[1]
    zD = vert_dest[2]

    return abs(((xD-xS)**2 + (yD-yS)**2 + (zD-zS)**2)**1/2)
```

Di tutte queste distanze, viene trovata la media, la quale viene poi normalizzata e inserita nella variabile **`planarity`**.

Di seguito vengono riportati dei test che dimostrano l'effettiva coerenza dei risultati ottenuti.

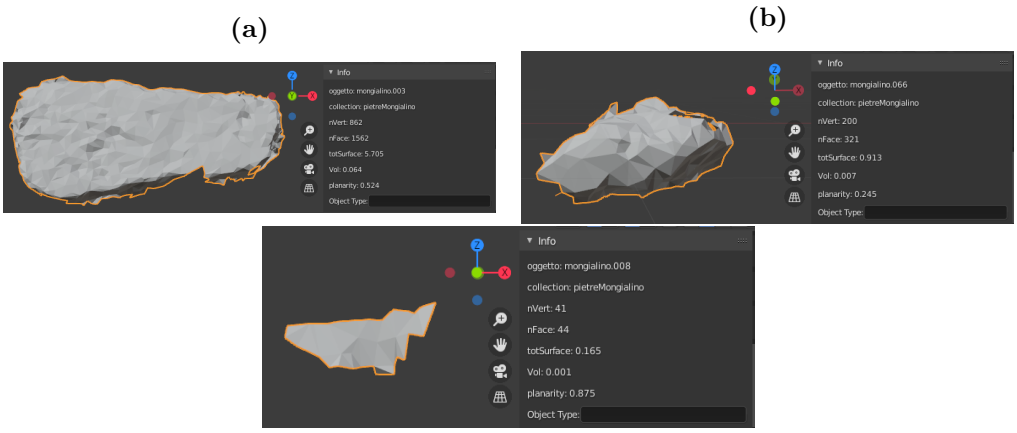


Figura 3.2: Diversi indici di planarità per tre pietre della parete del castello di Mongialino: **(a)** 0.525, **(b)** 0.245, **(c)** 0.875

Come è possibile notare, all'aumentare della "pizzutaggine" del modello poligonale della roccia, l'indice di planarità diminuisce, sintomo che la superficie visibile della pietra è poco piana.

Un ottimo valore di normalizzazione è risultato essere il **raggio della sfera minima** includente la mesh della pietra, già calcolato attraverso il **Codice 3.17**: esso, essendo effettivamente proporzionale ad un ipotetico raggio della mesh, ha mostrato ottimi risultati nella normalizzazione.
Infine, il piano viene **scollegato** e **rimosso** dalla scena in quanto non verrà più utilizzato.

3.4.2 Colorazione delle mesh

Per fornire un'ulteriore informazione visiva all'utente, è stato stabilito di colorare le *pill* ottenute attraverso l'indice di planarità.
Ciò consente l'immediata comprensione di quale pietra ha un alto indice e quale invece ne ha uno più basso.
Naturalmente, nel contesto di Blender non è possibile colorare delle mesh se non assegnando ad ognuna di esse uno **shader**, ossia un algoritmo in grado di conferire ad un modello tridimensionale un materiale virtuale corrispondente ad un materiale realmente esistente nel mondo reale.

Tra le tante tipologie di shader, è stato scelto il ***Diffuse BSDF***, scelta dettata dalla compatibilità diretta con la 3D Viewport in *Solid Mode*. Tale modalità, infatti, non consentirebbe la visualizzazione dei materiali (ad essa sono dedicate, in base alla necessità, le modalità *Material Preview* e *Rendered*); tuttavia, è possibile utilizzare lo shader *Diffuse BSDF* per visualizzare delle mesh banalmente colorate (senza alcun particolare tipo di effetto fisico) direttamente in 3D Viewport.

Secondo l'approccio utilizzato, è stata realizzata una palette di cinque colori: ad ogni colore corrisponde un intervallo di appartenenza dell'indice di planarità. Ciò consente la presenza di soli cinque materiali da creare e utilizzare, piuttosto che uno per ogni mesh.

Palette colori

R:204	G:204	B:229	—	planarity $\in [0, 0.2)$
R:153	G:178	B:204	—	planarity $\in [0.2, 0.4)$
R:75.5	G:127.5	B:178.5	—	planarity $\in [0.4, 0.6)$
R:0	G:76.5	B:102	—	planarity $\in [0.6, 0.8)$
R:0	G:25.5	B:127.5	—	planarity $\in [0.8, 1]$

Come si può notare, si è deciso di utilizzare una **scala lineare** basata su cinque intervalli di ampiezza equivalente, piuttosto che una logaritmica.

Inoltre è stato scelto il seguente approccio basato su due diverse funzioni: una funzione implementata per la **generazione** dei cinque materiali necessari, in base al gradiente di colori prestabilito, e una implementata per l'**assegnazione** dello specifico materiale ad ogni mesh *pill*, in base all'indice di planarità del singolo modello poligonale della roccia alla quale ognuna di esse fa riferimento.

Codice 3.23: Funzione di generazione dei materiali

```
1 def generateMaterial():
2     mat1 = bpy.data.materials.new(name = "000_020_MAT")
3     mat2 = bpy.data.materials.new(name = "020_040_MAT")
4     mat3 = bpy.data.materials.new(name = "040_060_MAT")
5     mat4 = bpy.data.materials.new(name = "060_080_MAT")
6     mat5 = bpy.data.materials.new(name = "080_100_MAT")
7
8     mat1.roughness = mat2.roughness = mat3.roughness = mat4.
          roughness = mat5.roughness = 1
9
10     #R:204         B:204        B:229
11    mat1.diffuse_color = (0.8,0.8,0.9,1)
12     #R:153         B:178        B:204
13    mat2.diffuse_color = (0.6,0.7,0.8,1)
14     #R:75.5   B:127.5    B:178.5
15    mat3.diffuse_color = (0.3,0.5,0.7,1)
16     #R:0           B:76.5       B:102
17    mat4.diffuse_color = (0.0,0.3,0.4,1)
18     #R:0           B:25.5       B:127.5
19    mat5.diffuse_color = (0.0,0.1,0.5,1)
20
21    matList = [mat1, mat2, mat3, mat4, mat5]
22
23    return matList
```

In particolare, si può notare che la funzione **`generateMaterial()`** contiene tutte le istruzioni necessarie alla crezione dei cinque materiali.
Per la crezione di ognuno di essi, è stata infatti richiamata la funzione **`bpy.data.materials.new(name)`**, passando come parametro `name` il nome del materiale che verrà poi visualizzato direttamente dall'interfaccia di Blender; per la nomenclatura di quest'ultimo, è stato scelto un approccio basato sull'intervallo dell'indice di planarità al quale il colore di quello shader fa riferimento. Ognuno dei materiali viene salvato in una variabile **`mat<X>`** (con `<X>`$= 1, \ldots, 5$) così da poterne ottenere dei riferimenti, utili per apportare successive modifiche e per poter avere un effettivo output.

Successivamente, per ogni shader generato, vengono infatti impostate l'opacità **roughnes=1** e i gradienti di colore in base agli intervalli già discussi. Infine, viene ritornata la lista **matlist** contenente i cinque materiali creati.

Codice 3.24: Funzione di assegnazione dei materiali

```
def assignMaterialLinear(pillObj, parameter, materials):
    if parameter >= 0.00 and parameter < 0.20:
        pillObj.data.materials.append(materials[0])
    elif parameter >= 0.20 and parameter < 0.40:
        pillObj.data.materials.append(materials[1])
    elif parameter >= 0.40 and parameter < 0.60:
        pillObj.data.materials.append(materials[2])
    elif parameter >= 0.60 and parameter < 0.80:
        pillObj.data.materials.append(materials[3])
    elif parameter >= 0.80 and parameter =< 1:
        pillObj.data.materials.append(materials[4])
```

La funzione **assignMaterialLinear()** implementata e sopra presentata prende in input i seguenti valori:

- **pillObj**: il riferimento alle singole mesh *pill*;
- **parameter**: l'indice di planarità per ognuna dei modelli poligonali delle rocce;
- **materials**: la lista dei cinque materiali generati.

Nello specifico è possibile notare che, in base all'intervallo di appartenenza, il materiale prestabilito viene assegnato alle singole *pill* richiamando la funzione **pillObj.data.materials.append(material[i])**.

Naturalmente, queste due funzioni vengono richiamate nel corpo di esecuzione principale dell'intero script.

Codice 3.25: Generazione degli shader nel flusso di esecuzione principale

```
objs = bpy.context.visible_objects.copy()
pillsColors = generateMaterial()
```

La funzione `generateMaterials()` viene richiamata immediatamente dopo la copia delle mesh visibili. L'output della funzione viene salvato in una variabile **pillsColors**.

Codice 3.26: Assegnazione degli shader nel flusso di esecuzione principale

```
bpy.ops.mesh.primitive_uv_sphere_add(radius = raggioSfera *
    scaleFac, location=vertsAvg, scale=(norm_diags[0],
    norm_diags[1],norm_diags[2]))
```

```
2 bpy.ops.object.shade_smooth()
3 pill = bpy.context.active_object
4 pill.name = obj.name + '_pill'
5 assignMaterialLinear(pill, planarity, pillsColors)
6 blenderT_Matrix = mathutils.Matrix((eigvecs[0], eigvecs[1],
      eigvecs[2]))
7 pill.rotation_euler = blenderT_Matrix.to_euler('XYZ')
```

La funzione `assignMaterialLinear()` è stata poi richiamata nel sezione di creazione delle singole *pill* descritto nel **Codice 3.19**.

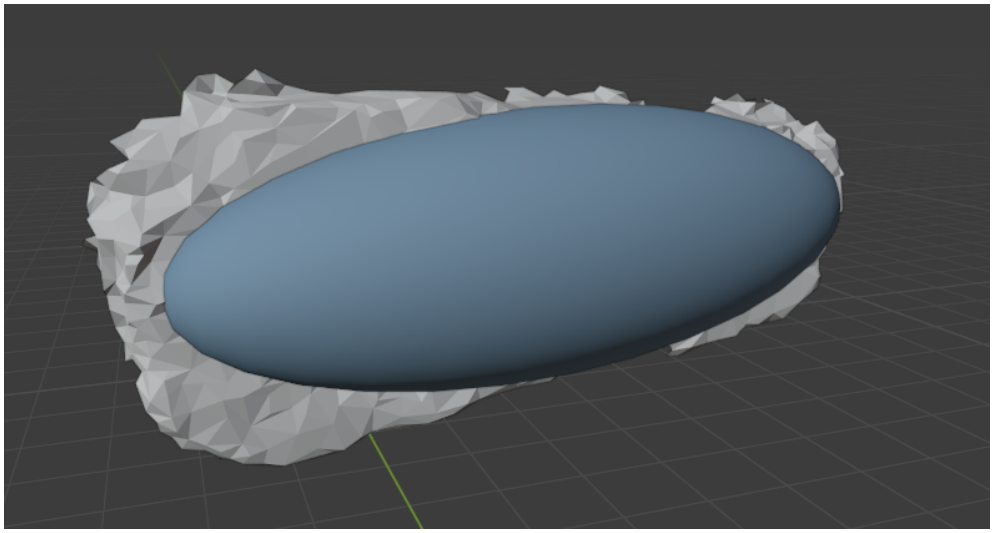


Figura 3.3: *Pill* colorata in base all'indice di planarità della pietra a cui fa rifarimento

3.5 Istogramma delle normali

Nonostante l'indice di planarità possa essere di aiuto nell'analisi della superficie frontalmente visibile delle rocce da analizzare, esso non fornisce alcuna informazione sulle restanti superfici.
Ogni mesh è infatti "immersa" in uno spazio costituito da otto **quadranti** individuati dalle direzioni degli assi x, y e z e dai rispettivi versi positivo e negativo, e ognuna di esse è caratterizzata da un numero di poligoni e vertici non indifferente: per cui, per meglio comprendere la forma di ogni mesh, oltre al tensore ed il suddetto indice di planarità, potrebbe essere altrettanto utile costruire un **istogramma** che descriva la normale media su ogni **ottante (locale)** nel quale ogni sezione della mesh è iscritto. Questo istogramma potrebbe infatti fornire un valido strumento per la costruzione di un classificatore in machine learning.

Ogni mesh, pur essendo localizzata in un certo punto dello spazio globale di Blender, è anche centrata in un sistema di riferimento locale sempre basato su otto quadranti e avente come origine il centro geometrico della mesh stessa. Per cui, è possibile verificare in quale ottante ogni superficie della mesh sia individuato, tenere il riferimento ad ognuna di esse e calcolare per ogni quadrante la normale media, per poi costruire un istogramma.
In particolare, l'istogramma realizzato per ogni mesh sarà caratterizzato da otto **bin** – uno per ottante – da tre valori ciascuno, uno per coordinata.

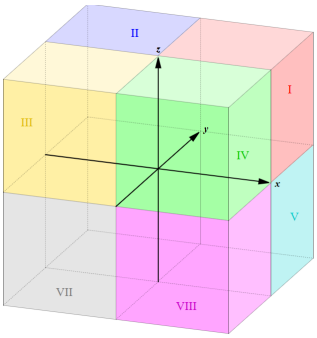


Figura 3.4: Spazio tridimensionale suddiviso in ottanti

3.5.1 Raccolta delle normali

Prendendo come riferimento una singola mesh, per individuare le normali su ogni ottante è necessario fare riferimento ai versi delle tre componenti che caratterizzano ognuno di questi versori. Se per ognuno degli otto quadranti si deve tenere il riferimento a dei versori con tre componenti, una per coordinata, allora si dovranno utilizzare 24 liste.

Codice 3.27: Liste delle normali su ogni ottante

```
1  XN_YN_ZN_x = []
2  XN_YN_ZN_y = []
3  XN_YN_ZN_z = []
4
5  XN_YN_ZP_x = []
6  XN_YN_ZP_y = []
7  XN_YN_ZP_z = []
8
9  XN_YP_ZN_x = []
10 XN_YP_ZN_y = []
11 XN_YP_ZN_z = []
12
13 XN_YP_ZP_x = []
14 XN_YP_ZP_y = []
15 XN_YP_ZP_z = []
16
17 XP_YN_ZN_x = []
18 XP_YN_ZN_y = []
19 XP_YN_ZN_z = []
20
21 XP_YN_ZP_x = []
22 XP_YN_ZP_y = []
23 XP_YN_ZP_z = []
```

```
24
25 XP_YP_ZN_x = []
26 XP_YP_ZN_y = []
27 XP_YP_ZN_z = []
28
29 XP_YP_ZP_x = []
30 XP_YP_ZP_y = []
31 XP_YP_ZP_z = []
```

Ognuna delle liste, presenta una nomenclatura secondo il seguente schema: **X<Verso>_Y<Verso>_Z<Verso>_<componente>**, dove <Verso> è **P** se positivo e **N** se negativo.

Di seguito si presenta il frammento di codice implementato per individuare l'ottante di appartenenza di ognuna delle normali che caratterizzano la mesh. La funzione fornita dalle API di Blender per ottenere le normali è la stessa utilizzata nel **Codice 3.3**.

Codice 3.28: Raccolta delle normali su ogni ottante

```
1 for p in obj_polygons:
2      if isNegative(p.normal[0]) and
3         isNegative(p.normal[1]) and
4         isNegative(p.normal[2]):
5                XN_YN_ZN_x.append(p.normal[0])
6                XN_YN_ZN_y.append(p.normal[1])
7                XN_YN_ZN_z.append(p.normal[2])
8
9      elif isNegative(p.normal[0]) and
10          isNegative(p.normal[1]) and
11          isPositive(p.normal[2]):
12               XN_YN_ZP_x.append(p.normal[0])
13               XN_YN_ZP_y.append(p.normal[1])
14               XN_YN_ZP_z.append(p.normal[2])
15
16     elif isNegative(p.normal[0]) and
17          isPositive(p.normal[1]) and
18          isNegative(p.normal[2]):
19               XN_YP_ZN_x.append(p.normal[0])
20               XN_YP_ZN_y.append(p.normal[1])
21               XN_YP_ZN_z.append(p.normal[2])
22
23     elif isNegative(p.normal[0]) and
24          isPositive(p.normal[1]) and
25          isPositive(p.normal[2]):
26               XN_YP_ZP_x.append(p.normal[0])
27               XN_YP_ZP_y.append(p.normal[1])
28               XN_YP_ZP_z.append(p.normal[2])
29
```

```
30      elif isPositive(p.normal[0]) and
31           isNegative(p.normal[1]) and
32           isNegative(p.normal[2]):
33                XP_YN_ZN_x.append(p.normal[0])
34                XP_YN_ZN_y.append(p.normal[1])
35                XP_YN_ZN_z.append(p.normal[2])
36
37      elif isPositive(p.normal[0]) and
38           isNegative(p.normal[1]) and
39           isPositive(p.normal[2]):
40                XP_YN_ZP_x.append(p.normal[0])
41                XP_YN_ZP_y.append(p.normal[1])
42                XP_YN_ZP_z.append(p.normal[2])
43
44      elif isPositive(p.normal[0]) and
45           isPositive(p.normal[1]) and
46           isNegative(p.normal[2]):
47                XP_YP_ZN_x.append(p.normal[0])
48                XP_YP_ZN_y.append(p.normal[1])
49                XP_YP_ZN_z.append(p.normal[2])
50
51      elif isPositive(p.normal[0]) and
52           isPositive(p.normal[1]) and
53           isPositive(p.normal[2]):
54                XP_YP_ZP_x.append(p.normal[0])
55                XP_YP_ZP_y.append(p.normal[1])
56                XP_YP_ZP_z.append(p.normal[2])
```

Per semplicità, per individuare il verso di ognuna delle coordinate, sono state implementate le due banali funzioni **`isPositive()`** e **`isNegative()`**, presentate di seguito per completezza.

Codice 3.29: Funzione di individuazione di un versore positivo

```
1 def isPositive(num):
2      if num >= 0: return True
3      else: return False
```

Codice 3.30: Funzione di individuazione di un versore negativo

```
1 def isNegative(num):
2     if num < 0: return True
3     else: return False
```

Esse semplicemente restituiscono un valore booleano a seconda della positività o negatività del parametro **`num`** passato.

3.5.2 Calcolo della normale media per ottante

Per la realizzazione di questo istogramma, vengono utilizzati 24 valori, ossia le componenti di ogni normale media su ogni ottante.

Codice 3.31: Calcolo della normale media per ottante

```
avg_XN_YN_ZN_x = media(XN_YN_ZN_x)
avg_XN_YN_ZN_y = media(XN_YN_ZN_y)
avg_XN_YN_ZN_z = media(XN_YN_ZN_z)

avg_XN_YN_ZP_x = media(XN_YN_ZP_x)
avg_XN_YN_ZP_y = media(XN_YN_ZP_y)
avg_XN_YN_ZP_z = media(XN_YN_ZP_z)

avg_XN_YP_ZN_x = media(XN_YP_ZN_x)
avg_XN_YP_ZN_y = media(XN_YP_ZN_y)
avg_XN_YP_ZN_z = media(XN_YP_ZN_z)

avg_XN_YP_ZP_x = media(XN_YP_ZP_x)
avg_XN_YP_ZP_y = media(XN_YP_ZP_y)
avg_XN_YP_ZP_z = media(XN_YP_ZP_z)

avg_XP_YN_ZN_x = media(XP_YN_ZN_x)
avg_XP_YN_ZN_y = media(XP_YN_ZN_y)
avg_XP_YN_ZN_z = media(XP_YN_ZN_z)

avg_XP_YN_ZP_x = media(XP_YN_ZP_x)
avg_XP_YN_ZP_y = media(XP_YN_ZP_y)
avg_XP_YN_ZP_z = media(XP_YN_ZP_z)

avg_XP_YP_ZN_x = media(XP_YP_ZN_x)
avg_XP_YP_ZN_y = media(XP_YP_ZN_y)
avg_XP_YP_ZN_z = media(XP_YP_ZN_z)

avg_XP_YP_ZP_x = media(XP_YP_ZP_x)
avg_XP_YP_ZP_y = media(XP_YP_ZP_y)
avg_XP_YP_ZP_z = media(XP_YP_ZP_z)
```

Le variabili contenenti i valori medi hanno una nomenclatura realizzata secondo il seguente schema: **avg_X<Verso>_Y<Verso>_Z<Verso>_<componente>**. I valori ottenuti verranno poi stampati sul file CSV di output, nelle 24 colonne ad essi dedicati.

Di seguito si presentano i risultati ottenuti da alcuni test effettuati prendendo come riferimento una mesh appositamente realizzata e due mesh di due rocce costituenti una parete antica del castello di Mongialino.

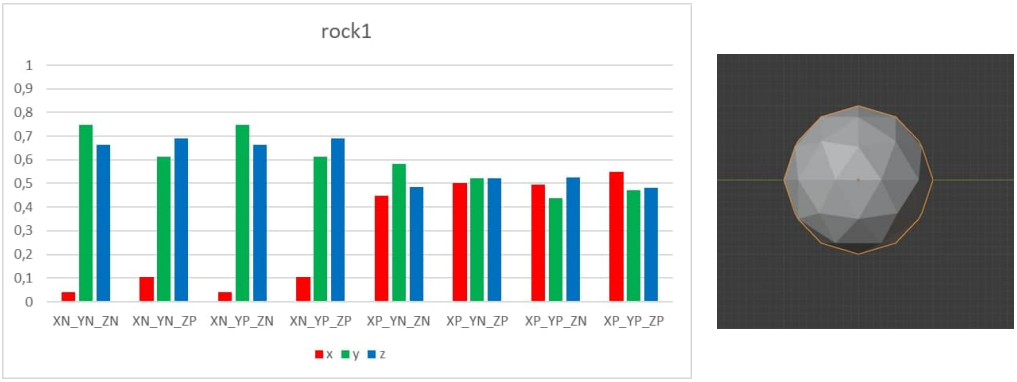


Figura 3.5: Istogramma risultante da una semi-icosfera

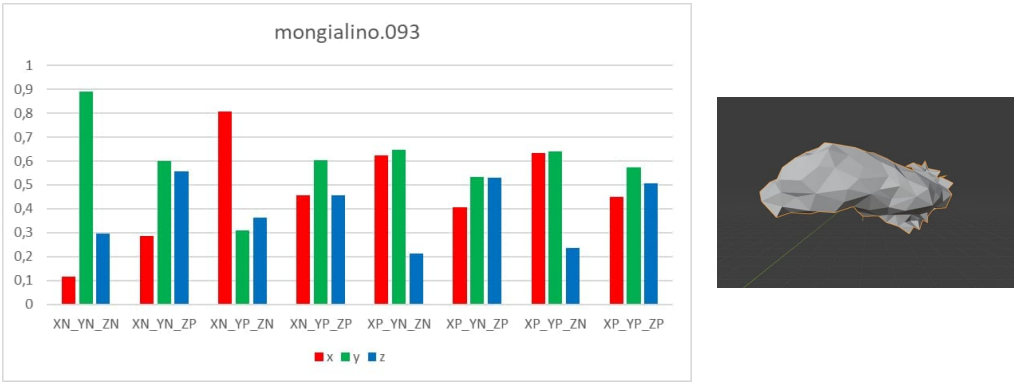


Figura 3.6: Istogramma risultante da una pietra della parete del castello di Mongialino

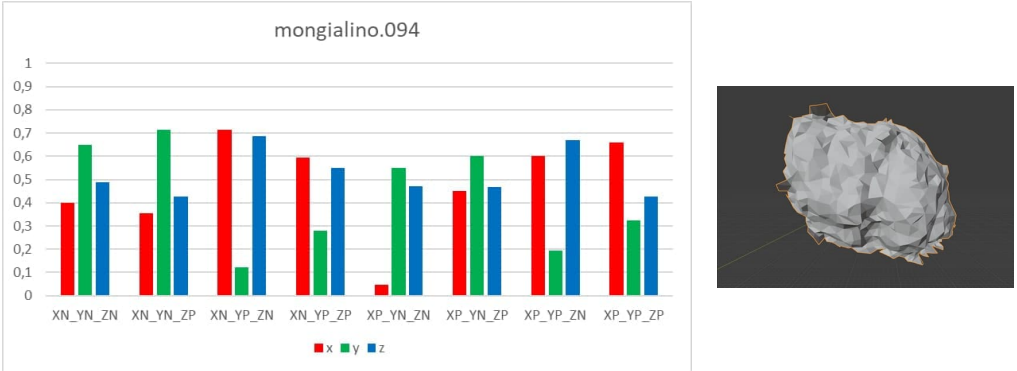


Figura 3.7: Istogramma risultante da un'altra pietra della parete del castello di Mongialino

3.6 Gestione delle Blender Collection

Importante implementazione, dedita ad una maggior "user-friendliness" di questo componente aggiuntivo, è sicuramente quella relativa alla gestione delle **Blender Collection**.
Infatti, l'utente potrebbe voler esportare le varie mesh ottenute mediante il calcolo del tensore di inerzia (le cosiddette "`pill`"), in modo da poterle analizzare separatamente ai modelli poligonali iniziali. Pertanto è stato stabilito di porre, una volta generate, queste mesh all'interno di una Blender Collection, così che possano essere più facilmente esportate ed importate altrove (o per semplice questione di ordine).

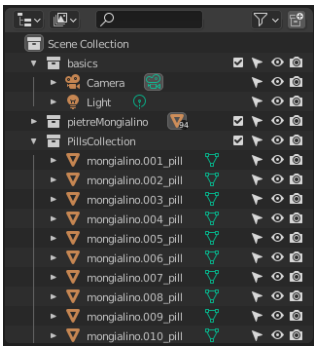


Figura 3.8: Outliner di un file `.blend`: si noti la "PillsCollection"

3.6.1 Creazione della nuova collezione

Innanzitutto, è importante che venga creata una nuova collezione prima che la scansione delle mesh e i successivi calcoli vengano effettuati.

Codice 3.32: Creazione o sovrascrittura della PillsCollection

```
1 cols = bpy.data.collections
2 for col in cols:
3       if col.name == "PillsCollection":
4             bpy.data.collections.remove(col)
5 myCol = bpy.data.collections.new("PillsCollection")
```

Questo frammento di codice è stato inserito all'inizio dell'intero flusso di esecuzione, quindi prima ancora di entrare nel ciclo rappresentato nel **Codice 3.1**.
Viene anzitutto salvata la lista di tutte le collezioni (mediante il membro **`bpy.data.collections`**) in una variabile denominata **`cols`**. Nel successivo ciclo `for`, viene presa una piccola precauzione nel caso in cui l'utente avesse fatto qualche modifica alle mesh di partenza e volesse rieseguire il calcolo: l'eventuale collezione **`PillsCollection`** viene infatti rimossa se presente; ciò permette una sovrascrittura della stessa, in quanto anche il suo contenuto viene totalmente scollegato e rimosso dalla scena. Infine, se non presente, essa viene ricreata vuota e salvata nella variabile **`myCol`**.

3.6.2 Inserimento delle mesh nella collezione

Naturalmente, dopo la generazione degli elissoidi, questi dovranno inseriti nella nuova collezione, così da garantire una più rapida esportazione.

Codice 3.33: Inserimento delle mesh appena create nella collezione

```
1 old_pill_coll = pill.users_collection
2 myCol.objects.link(pill)
3 for ob in old_pill_coll:
4       ob.objects.unlink(pill)
```

Questo frammento di codice, a differenza del precedente, è indentato al ciclo `for` del **Codice 3.1**, ed è scritto immediatamente dopo la generazione dell'elissoide di inerzia mostrata nel **Codice 3.19**.
Blender crea automaticamente gli oggetti in una "**master collection**", ma si vuole che gli oggetti stiano nella collection creata in precedenza.
Se questa situazione non venisse gestita, le `pill` sarebbero collegate contemporaneamente a più collection, pertanto prima vengono salvate tutte le vecchie collection alle quali la mesh **`pill`** è collegata mediante l'attributo **`users_collection`**, questa viene poi alla nuova collection `myCol`, e infine scollegata dalle vecchie.

Capitolo 4

Back-end in dettaglio: SurfEx

Esattamente come per il precedente add-on, di seguito si descrive l'itinerario di sviluppo di questa seconda estensione per Blender, la quale è stata specificatamente progettata al fine di estrarre informazioni sulla superficie del modelli, in modo che l'utente possa poi scegliere di suddividere quei modelli, in base a questa e mediante la parte front-end (la quale verrà ampiamente discussa durante il successivo capitolo) in diverse categorie di oggetti.

Questo secondo add-on, decisamente più semplice del primo (in quanto non vengono effettuati particolari tipi di calcolo), prende in realtà ispirazione da un altro add-on, noto come ***BlenderBIM*** [5], il quale fa utilizzo dello standard ***BIM*** (**B**uilding **I**nformation **M**odelling) al fine di estendere le funzionalità di Blender in modo da fornire un utile strumento di calcolo a ingegneri, architetti, geometri e tutte quelle figure che gravitano nel campo dell'edilizia.

4.1 Scansione dei modelli poligonali

Come visto per il precedente add-on, le operazioni di calcolo e manipolazione dei dati sono anzitutto precedute da un salvataggio in variabile delle informazioni contenute nel datablock al quale la mesh fa riferimento.

Codice 4.1: Salvataggio delle principali informazioni presenti sul datablock

```
1 objs = bpy.context.visible_objects.copy()
2 for obj in objs:
3     if obj.type == "MESH":
4         bpy.ops.object.transform_apply(location=True, rotation
              =True, scale=True)
5         obj_data = obj.data
6         obj_verts = obj_data.vertices
```

```
7         obj_polygons = obj_data.polygons
```

In questo caso non si nota nessuna variazione rispetto al metodo già adottato e visto nel capitolo precedente. Le operazioni vengono infatti effettuate in un ciclo `for`; per cui, per ogni oggetto di tipo mesh visibile in 3D Viewport, innanzitutto vengono applicati sul datablock gli ultimi cambiamenti effettuati dall'utente, e in seguito vengono salvate in **`obj_data`**, **`obj_verts`** e **`obj_polygons`** rispettivamente il **datablock** relativo alla mesh, l'informazione (la lista) sui **vertici** e l'informazione (anche qui, la lista) sulle **facce** del modello poligonale.

4.2 Calcolo della superficie totale del solido

Anche per il calcolo dei valori in output, questo specifico add-on si discosta veramente poco dal precedente.
I valori calcolati sono decisamente meno rispetto a prima: per cui, nessun particolare indice statistico, nessun tensore; naturalmente ciò, nell'implementazione della parte front-end, porterà ad una struttura dati di output molto più semplice e meno ricca di valori.

Codice 4.2: Calcolo della superficie totale

```
1 nVerts = len(obj_verts)
2 nFaces = len(obj_polygons)
3 totSurface = 0
4 for p in obj_polygons:
5     totSurface += p.area
```

Nello specifico, dal codice in alto, indentato nel ciclo descritto nel **Codice 4.1**, è possibile notare che vengono salvati:

- in una variabile **`nVerts`** il numero dei vertici, ottenuto mediante la lunghezza della lista `obj_verts`;
- in una variabile **`nFaces`** il numero delle facce, ottenuto mediante la lunghezza della lista `obj_polygons`;
- in una variabile **`totSurface`** la superficie totale della specifica mesh, ottenuta sommando le superfici delle singole facce ritornate dall'attributo **`area`** del `Polygon`.

Capitolo 5

Front-end: comunicazione con l'utente

Passaggio chiave di entrambi i software sviluppati nel corso di questo progetto di Tesi, è sicuramente lo sviluppo della parte **front-end**. In questa sede, per "front-end" si fa riferimento sia al suo significato nell'**analisi dati** (ossia la visualizzazione e il fornimento all'utente dei dati calcolati e raccolti), sia al significato più generico che ha nell'ambito della **progettazione software**, ossia l'interazione con l'utente mediante un'interfaccia grafica.
In fondo, quello che si è visto finora, è solo l'itinerario di sviluppo di due "banali" script in Python eseguibili direttamente dall'interprete personalizzato che lo stesso Blender fornisce.

In questo capitolo si parlerà invece, oltre che delle strutture dati utili alla comunicazione con l'utente, e alla produzione del file CSV, anche del processo che ha portato i due script ad evolversi in due vere e proprie **estensioni** per Blender.

Una precisazione in merito: da questo punto di vista la differenza tra i due add-on è **minima**, per cui la seguente descrizione varrà per ambedue le estensioni sviluppate: verrà tuttavia fatta menzione delle piccole differenze (quando presenti) che interessano l'implementazione di questo layer.

5.1 Dizionario in output

Anzitutto, è necessaria un'efficiente struttura dati, di tipo **chiave-valore**, che possa contenere tutte le informazioni raccolte **categorizzate** in base al loro nominativo, il quale poi dovrà essere visualizzato sia sul file CSV che

sull'interfaccia grafica di Blender.
La struttura dati migliore da questo punto di vista, e nativamente presente in Python, è sicuramente la **`dict`**. Per cui, in entrambi gli add-on è stato implementato nel flusso principale di esecuzione un `dict`, chiamato **`meshInfo`**, il quale contiene tutte le informazioni raccolte nel corso dell'esecuzione del software.
Di seguito, si presentano i due diversi `meshInfo`:

Codice 5.1: Dizionario implementato in *Point Cloud Stats*

```
meshInfo = {'oggetto': obj.name, 'collection': colName,
            'nVert': nVerts, 'nFace': nFaces,
            'totSurface': round(totSurface,3),
            'Vol': round(volume,3),
            'planarity': round(planarity,3),
            'objType': obj['Object Type'],
            'mediaVX': round(x_vertsAvg,3),
            'mediaVY': round(y_vertsAvg,3),
            'mediaVZ': round(z_vertsAvg,3),
            'varVX': round(x_vertsStDev,3),
            'varVY': round(y_vertsStDev,3),
            'varVZ': round(z_vertsStDev,3),
            'mediaNX': round(x_normalsAvg,3),
            'mediaNY': round(y_normalsAvg,3),
            'mediaNZ': round(z_normalsAvg,3),
            'varNX': round(x_normalsStDev,3),
            'varNY': round(y_normalsStDev,3),
            'varNZ': round(z_normalsStDev,3),
            'eigv1': round(autovalori[0],3),
            'eigv2': round(autovalori[1],3),
            'eigv3': round(autovalori[2],3),
            'x1': round(eigvecs[0][0],3),
            'y1': round(eigvecs[0][1],3),
            'z1': round(eigvecs[0][2],3),
            'x2': round(eigvecs[1][0],3),
            'y2': round(eigvecs[1][1],3),
            'z2': round(eigvecs[1][2],3),
            'x3': round(eigvecs[2][0],3),
            'y3': round(eigvecs[2][1],3),
            'z3': round(eigvecs[2][2],3),
            'XN_YN_ZN_x': abs(round(avg_XN_YN_ZN_x,3)),
            'XN_YN_ZN_y': abs(round(avg_XN_YN_ZN_y,3)),
            'XN_YN_ZN_z': abs(round(avg_XN_YN_ZN_z,3)),
            'XN_YN_ZP_x': abs(round(avg_XN_YN_ZP_x,3)),
            'XN_YN_ZP_y': abs(round(avg_XN_YN_ZP_y,3)),
            'XN_YN_ZP_z': abs(round(avg_XN_YN_ZP_z,3)),
            'XN_YP_ZN_x': abs(round(avg_XN_YP_ZN_x,3)),
            'XN_YP_ZN_y': abs(round(avg_XN_YP_ZN_y,3)),
            'XN_YP_ZN_z': abs(round(avg_XN_YP_ZN_z,3)),
```

```
40              'XN_YP_ZP_x': abs(round(avg_XN_YP_ZP_x,3)),
41              'XN_YP_ZP_y': abs(round(avg_XN_YP_ZP_y,3)),
42              'XN_YP_ZP_z': abs(round(avg_XN_YP_ZP_z,3)),
43              'XP_YN_ZN_x': abs(round(avg_XP_YN_ZN_x,3)),
44              'XP_YN_ZN_y': abs(round(avg_XP_YN_ZN_y,3)),
45              'XP_YN_ZN_z': abs(round(avg_XP_YN_ZN_z,3)),
46              'XP_YN_ZP_x': abs(round(avg_XP_YN_ZP_x,3)),
47              'XP_YN_ZP_y': abs(round(avg_XP_YN_ZP_y,3)),
48              'XP_YN_ZP_z': abs(round(avg_XP_YN_ZP_z,3)),
49              'XP_YP_ZN_x': abs(round(avg_XP_YP_ZN_x,3)),
50              'XP_YP_ZN_y': abs(round(avg_XP_YP_ZN_y,3)),
51              'XP_YP_ZN_z': abs(round(avg_XP_YP_ZN_z,3)),
52              'XP_YP_ZP_x': abs(round(avg_XP_YP_ZP_x,3)),
53              'XP_YP_ZP_y': abs(round(avg_XP_YP_ZP_y,3)),
54              'XP_YP_ZP_z': abs(round(avg_XP_YP_ZP_z,3))}
```

Codice 5.2: Dizionario implementato in *SurfEx*

```
1 meshInfo = {'oggetto': obj.name, 'collection': colName,
2             'nVert': nVerts, 'nFace': nFaces,
3             'totSurface': round(totSurface, 3),
4             'objType': obj['Obj Type']}
```

Si noti la differenza, espressa in quantità di coppie chiave-valore, tra i due dizionari. Inoltre, per avere una migliore visualizzazione dei valori a virgola mobile, in entrambi i casi si è optato per un **arrotondamento** alla terza cifra decimale (strategia adottata dallo stesso Blender nella sua visualizzazione grafica) mediante la funzione di Python **`round()`**.
Inoltre, un elemento presente in entrambi casi nella struttura dati è rappresentato dalla coppia

```
'objType': obj['Object Type']
```

la quale verrà discussa successivamente, nella sezione relativa all'interfaccia grafica. Per attuale comprensione del lettore, basti sapere per il momento che tale campo viene utilizzato al fine di inserire una proprietà personalizzata ad ogni singola mesh, in modo che l'utente possa ulteriormente categorizzarle.

5.2 Esportazione dei dati su CSV

Come già accennato nel precedente paragrafo, in realtà lo scopo della struttura dati creata è quello di avere a portata di mano un riferimento a tutti i dati raccolti e calcolati nel corso dell'esecuzione del software. Per cui, tale riferimento può essere utile in tal senso per fornire un output (su file o visibile su interfaccia) di quanto fatto in precedenza.

Verrà in questa sede discussa la modalità di implementazione dell'esportazione dei dati in un file CSV, a partire dal dizionario presentato in precedenza. La funzionalità di questo file è presto detta: a causa della sua struttura, esso può essere utilizzato come colonna portante di costruzione di una base di dati sempre a portata di mano, sulla quale, volendo, sarebbe anche possibile effettuare delle operazioni di machine learning.

Si ricorda, tuttavia, che per utilizzare le funzioni di manipolazione dei file con estensione `.csv` è necessario anzitutto importare sullo script Python il modulo `csv`.

5.2.1 Creazione del file

Per la creazione del file, è stata implementata una funzione dedicata; ciò permetterà una maggiore facilità nell'inglobare questa funzione in un Blender Operator, il quale verrà poi utilizzato per lo sviluppo della GUI.

Codice 5.3: Funzione di creazione del file CSV

```
def createCSV(filepath):
    global csvFile
    global writer

    csvFile = open(filepath, 'w', newline='')

    csvFile.write("sep=,\n")

    fieldnames = [# valori di heading]

    writer = csv.DictWriter(csvFile, fieldnames=fieldnames)
    writer.writeheader()

    return csvFile, writer
```

Si riportano qui di seguito le due diverse liste **`fieldnames`**:

Codice 5.4: Lista `fieldnames` per l'add-on *Point Cloud Stats*

```
fieldnames = ['oggetto', 'collection', 'nVert', 'nFace',
              'totSurface', 'Vol', 'planarity', 'objType',
              'mediaVX', 'mediaVY', 'mediaVZ',
              'varVX', 'varVY', 'varVZ',
              'mediaNX', 'mediaNY', 'mediaNZ',
              'varNX', 'varNY', 'varNZ',
              'eigv1', 'eigv2', 'eigv3',
              'x1', 'y1', 'z1',
              'x2', 'y2', 'z2',
              'x3', 'y3', 'z3',
```

```
11              'XN_YN_ZN_x', 'XN_YN_ZN_y', 'XN_YN_ZN_z',
12              'XN_YN_ZP_x', 'XN_YN_ZP_y', 'XN_YN_ZP_z',
13              'XN_YP_ZN_x', 'XN_YP_ZN_y', 'XN_YP_ZN_z',
14              'XN_YP_ZP_x', 'XN_YP_ZP_y', 'XN_YP_ZP_z',
15              'XP_YN_ZN_x', 'XP_YN_ZN_y', 'XP_YN_ZN_z',
16              'XP_YN_ZP_x', 'XP_YN_ZP_y', 'XP_YN_ZP_z',
17              'XP_YP_ZN_x', 'XP_YP_ZN_y', 'XP_YP_ZN_z',
18              'XP_YP_ZP_x', 'XP_YP_ZP_y', 'XP_YP_ZP_z']
```

Codice 5.5: Lista `fieldnames` per l'add-on *SurfEx*

```
1 fieldnames = ['oggetto', 'collection', 'nVert', 'nFace',
2               'totSurface', 'objType']
```

Il funzionamento di **`createCSV()`** in realtà è piuttosto semplice, perché vengono effettuate solamente due operazioni standard di generazione di un file CSV, preso in input il **`filepath`** desiderato.

Innanzitutto vengono dichiarate **`global`** le due variabili **`csvFile`** e **`writer`**, così da poter mantenere il medesimo riferimento in maniera globale per tutto lo script.

- `csvFile` mantiene il riferimento al file CSV, creato mediante la funzione `open()` di Python con parametri:
  - **`filepath`** (ossia il percorso desiderato);
  - **`'w'`** (per permettere la sua scrittura);
  - **`newline=''`** (per fare in modo che la scrittura venga fatta dall'inizio del file);
- `writer` mantiene il riferimento ad un `DictWriter`, ossia un oggetto del modulo `csv` usato per mappare liste e dizionari in righe di un file Python; i parametri in input sono:
  - il file `csvFile`
  - la lista di heading `fieldnames`.

La lista `fieldnames` è necessaria per la corretta funzionalità del file. Essa contiene infatti i valori di **heading**, ossia quei valori che costituiscono la prima riga del file stesso; tali valori sono utili affinché i valori di ogni riga (da qui in poi nominate **record**) vengano suddivisi per attributi, i quali poi corrispondono ai nominativi dei valori raccolti.
Nel codice si può inoltre notare la scrittura della prima riga sul file:

```
csvFile.write("sep=,\n")
```

ciò permette una migliore tabulazione mediante *Excel* e *LibreOffice Calc*: i file separati da virgola mobile, infatti non sono nativamente tabulati; per cui, mediante questa riga di codice, verrà designata la virgola come separatore di tabulazione, e ad ogni attributo sarà associata una colonna del foglio di calcolo.

Successivamente, una volta terminata l'operazione di scrittura dell'header mediante la funzione **`writer.writeheader()`**, vengono ritornati in output le due variabili di riferimento `csvFile` e `writer`.

5.2.2 Scrittura del file

Per la scrittura del file è stata implementata la seguente funzione, la quale si presenta in maniera identica in ambedue gli add-on.

Codice 5.6: Funzione di scrittura del file CSV

```
1 def writeCSV(filepath):
2     csvFile, writer = createCSV(filepath)
3 
4     try:
5         for obj in meshesData:
6             writer.writerow(meshesData[obj])
7     except:
8         print("Error in writer.writerow()")
9 
10    try:
11        csvFile.close()
12    except:
13        print("Error in csv.close()")
```

In questa funzione, la quale prende in input il `filepath` desiderato, è possibile vedere come i riferimenti vengono creati mediante la funzione `createCSV()` descritta nel precedente paragrafo.
Successivamente in due blocchi **`try - except`** vengono effettuate prima la scrittura del singolo record, facendo uso della funzione `writerow()` del `writer` alla quale viene passata una ulteriore struttura dati di tipo `dict` chiamata **`meshesData`** con chiave **`obj`**, e poi la funzione di chiusura del file **`close()`**. Questa struttura, non ancora trattata nel corso della descrizione, viene dichiarata come `global` nel flusso di esecuzione principale dello script, in modo da tenere un riferimento verso tutti gli oggetti presenti in scena. In particolare, ad ogni chiave `obj`, la quale fa riferimento ad uno dei singoli oggetti di tipo mesh presenti sulla 3D Viewport, è associato il dizionario `meshInfo`.
In tal modo, si ha a portata di mano una struttura dati che contiene, dopo

aver effettuato i vari calcoli, tutte le coppie di tipo chiave-valore che fanno riferimento ai singoli oggetti e ai dizionari `meshInfo` ad essi associati. Di seguito, il frammento di codice inserito nel flusso principale di esecuzione, per la creazione e il riempimento della struttura dati `meshData`.

```
1       meshesData[obj.name] = meshInfo
```

Naturalmente, questa riga viene inserita dopo il riempimento del dizionario descritto nel **Codice 5.1 / 5.2**.
Per cui, ogni qual volta la funzione verrà eseguita, ad ogni "passata" del ciclo `for` presentato nel **Codice 3.1**, una riga nel file CSV verrà scritta.

Il file CSV finale sarà il seguente, nei due diversi casi.

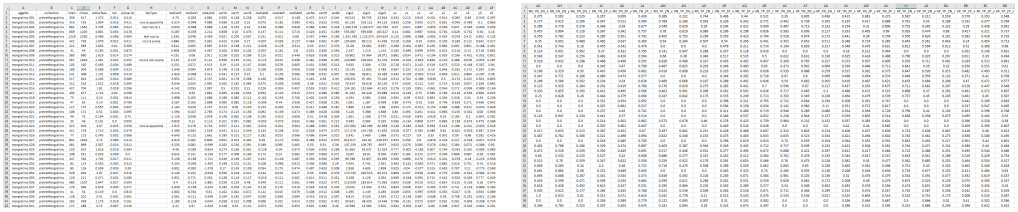


Figura 5.1: File CSV di output per usando l'add-on *Point Cloud Stats*

	A	B	C	D	E	F
1	oggetto	collection	nVert	nFace	totSurface	objType
2	mongialino.001	pietreMongialino	359	617	1.473	
3	mongialino.002	pietreMongialino	416	735	1.954	mediamente esteso
4	mongialino.003	pietreMongialino	862	1562	5.705	molto esteso
5	mongialino.004	pietreMongialino	669	1205	4.801	
6	mongialino.005	pietreMongialino	1229	2250	6.496	molto molto esteso
7	mongialino.006	pietreMongialino	422	739	1.979	mediamente esteso
8	mongialino.007	pietreMongialino	222	369	1.053	
9	mongialino.008	pietreMongialino	41	44	0.165	poco esteso
10	mongialino.009	pietreMongialino	383	690	1.619	
11	mongialino.010	pietreMongialino	567	1029	2.461	

Figura 5.2: File CSV di output per usando l'add-on *SurfEx*

Si noti come in entrambi casi, sotto l'attributo `objType`, alcune righe siano state riempite manualmente (sempre facendo utilizzo del software sviluppato, e non del foglio di calcolo) per dare un'ulteriore caratterizzazione al singolo oggetto. Questo procedimento verrà descritto in seguito.

5.3 Conversione in add-on e interfaccia grafica

Vengono adesso discusse tutte le implementazioni effettuate sui due script affinché da tali potessero realmente fungere da estensione per Blender, così da poterne ampliare le funzionalità e fornendo ad esso nuovi elementi grafici in grado di operare sulla scena e di visualizzare su schermo quanto descritto fino a questo punto.

5.3.1 Dizionario meta-info

Il primo passo consiste nell'includere all'inizio dell'intero file `.py` un dizionario **`bl_info`** che include tutte le informazioni relative all'add-on. Esso verrà automaticamente letto da Blender, il quale, all'installazione dell'estensione, visualizzerà come informazioni testuali tutti i valori immessi nel dizionario. I possibili tipi di chiave sono:

- **`name`** (`string`): nome dello script, visualizzato nel menù degli add-on;
- **`description`** (`string`): breve descrizione dell'add-on;
- **`author`** (`string`): nome dell'autore;
- **`version`** (`tuple of integers`): versione dello script, utilizzato per permettere all'utente di distinguere una versione da un'altra;
- **`blender`** (`tuple of 3 integers`): minima versione di Blender supportata dall'add-on;
- **`location`** (`string`): descrive dove l'utente può trovare la funzionalità aggiunta, per esempio "Scena > Point Cloud Stats";
- **`warning`** (`string`): utilizzato per informare l'utente di un bug noto al rilascio;
- **`support`** (`string`) descrive il tipo di supporto; sebbene `COMMUNITY` sia quello di default, è possibile inserire i seguenti tipi:
  - `OFFICIAL`: per gli add-on ufficialmente supportati dalla Blender Foundation;
  - `COMMUNITY`: per gli add-on mantenuti dalla community di sviluppatori;
  - `TESTING`: per gli add-on neonati esclusi dal rilascio stabile;

- **`wiki_url`** (`string`): link alla documentazione dell'add-on;
- **`tracker_url`** (`string`): campo opzionale per specificare un bug tracker diverso da quello di default;
- **`category`** (`string`): definisce il gruppo di appartenenza dell'add-on.

Di seguito si presentano due esempi di `bl_info` dei due script realizzati, nella loro forma al momento della stesura di questa Tesi:

Codice 5.7: Dizionario meta-info per *Point Cloud Stats*

```
1 bl_info = {
2     "name": "Point Cloud Stats",
3     "author": "nebuchadnezzar",
4     "version": (1, 5),
5     "blender": (2, 90, 0),
6     "location": "Scene > Point Cloud Stats",
7     "description": "Retrieves meshes informations, saves them
          in a CSV file and creates a collection of 'Pills' that
           summarizes them.",
8     "warning": "",
9     "doc_url": "",
10    "category": "Add Mesh",
11 }
```

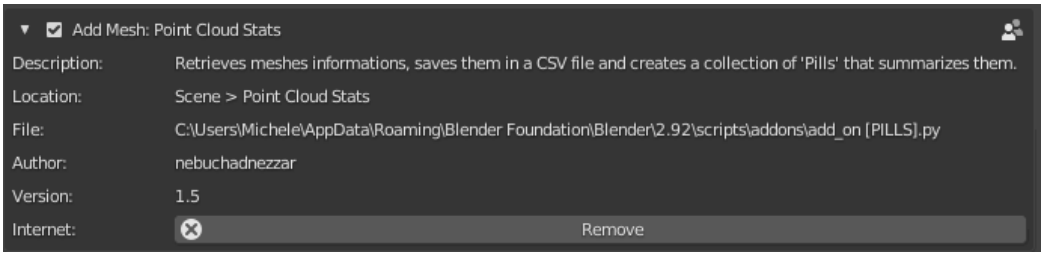


Figura 5.3: Visualizzazione delle meta-info per l'add-on *Point Cloud Stats* su Blender

Codice 5.8: Dizionario meta-info per *SurfEx*

```
1 bl_info = {
2     "name": "SurfEx",
3     "author": "nebuchadnezzar",
4     "version": (1, 1),
5     "blender": (2, 90, 0),
6     "location": "Scene > SurfEx",
7     "description": "Calculates the total surface of a mesh and
          puts it in a CSV file.",
```

```
8       "warning": "",
9       "doc_url": "",
10      "category": "Add Mesh",
11  }
```

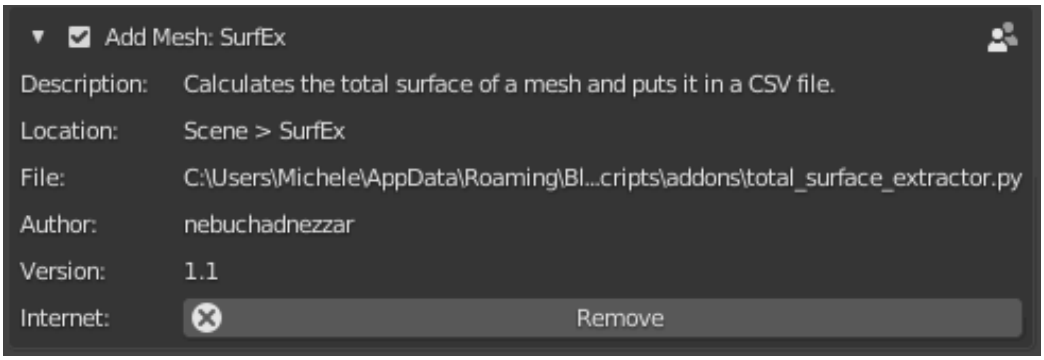


Figura 5.4: Visualizzazione delle meta-info per l'add-on *Point Cloud Stats* su Blender

5.3.2 Conversione di funzioni in Blender Operators

Per lo sviluppo dell'interfaccia grafica, è anzitutto necessario inglobare le funzioni principali eseguite nel corso dello script in dei **Blender Operators**: infatti, la GUI altro non fa che richiamare questi operatori ed eseguire le funzioni che li costituiscono. Passo fondamentale di questa fase, per cui, è quello di spostamento dell'intero back-end dei due add-on (descritto rispettivamente nel **Capitolo 3** e nel **Capitolo 4**) in una funzione chiamata banalmente **`main()`**. Affinché poi essa possa esser eseguita mediante un operatore, viene passato un valore `context`.

Per quanto riguarda il processo di creazione e scrittura del file CSV, non bisogna invece fare altro (se non inglobare all'interno del rispettivo operatore), in quanto tali operazioni vengono già effettuate mediante due funzioni.

Per costruire un Blender Operator, è necessario anzitutto costruire una nuova classe che erediti da **`bpy.types.Operator`**.
Tale classe, presenta tutta una serie di attributi che lo sviluppatore può inserire e personalizzare a propria necessità. Nello specifico, si descrivono quelli utilizzati per la costruzione dei principali operatori:

- **`bl_idname`**: identificativo dell'operatore attraverso il quale esso verrà richiamato alla sua esecuzione;

- **`bl_label`**: breve descrizione dell'operatore (visualizzata nei Button);
- **`bl_description`**: descrizione più dettagliata dell'operatore;

Inoltre, in ogni operatore possono essere definite una o più funzioni – secondo lo standard di Blender – che ne consentono l'esecuzione.
Si descrivono di seguito le funzioni utilizzate.

- **`Operator.execute`**: esegue la funzione inglobata e ritorna `'FINISHED'` (vedasi **Capitolo 1.2**). Prende in input i seguenti valori:
  - `self`;
  - `context`.
- **`Operator.invoke`**: utilizzata per assegnare proprietà poi utilizzate in `execute()`. Prende in input i seguenti valori:
  - `self`;
  - `context`;
  - `event`.

In particolare, sono state costruite due diverse classi operatore: una per l'esecuzione del back-end dei due add-on, ed uno per l'esportazione dei dati in un file CSV.

Si comincia innanzitutto col descrivere l'operatore che ingloba la funzione `main()`, dedita all'esecuzione dell'intero back-end dei due add-on.

Codice 5.9: Operatore di esecuzione del back-end

```
1 class MainFlowOperator(bpy.types.Operator):
2     bl_idname = "scene.main_flow"
3     bl_label = "Calcola!"
4     bl_description = "Calcola le principali caratteristiche
          geometriche delle mesh"
5
6     def execute(self, context):
7         main(context)
8         return {'FINISHED'}
```

In questo caso l'operatore **`MainFlowOperator`** si presenta in maniera abbastanza semplice: l'unica cosa che fa è eseguire la funzione `main()` il cui corpo è caratterizzato dal back-end già descritto in precedenza.

Di seguito si presenta invece l'operatore **`CreateCSVOperator`**, il quale – come il nome suggerisce – è dedito all'esportazione dei dati in formato `.csv`.

Codice 5.10: Operatore di esportazione del file CSV

```
1 class CreateCSVOperator(bpy.types.Operator):
2     bl_idname = "scene.select_dir"
3     bl_label = "Esporta su file CSV"
4     bl_description="Esporta i file analizzati su un file CSV"
5     filepath = bpy.props.StringProperty(subtype="DIR_PATH")
6 
7     def execute(self, context):
8         writeCSV(self.filepath)
9         return {'FINISHED'}
10 
11     def invoke(self, context, event):
12         context.window_manager.fileselect_add(self)
13         return {'RUNNING_MODAL'}
```

Oltre che dalla funzione `Operator.execute`, la quale invoca la funzione `writeCSV(filepath)` descritta nel **Codice 5.6**, questo operatore è caratterizzato dalla funzione `Operator.invoke`: esso, mediante la chiamata al context del **Window Manager** di Blender, permette di selezionare mediante interfaccia grafica un percorso di destinazione del file e di salvarlo come **`StringProperty`** di tipo **`DIR_PATH`** – ossia, un tipo di proprietà che definisce i percorsi – in una variabile `filepath`. In tal modo, il file verrà creato nel percorso selezionato dall'utente.

Si noti che i due operatori hanno identificatori rispettivamente uguali a **`scene.main_flow`** e **`scene.select_dir`**.

5.3.3 Pulsanti di esecuzione

Terminate le opportune precisazioni sui Blender Operator, è ora possibile trattare l'implementazione della GUI tramite la quale l'utente può effettivamente eseguire quanto descritto fino a questo punto.

Tra i vari tipi di elementi grafici presenti in Blender, è stato scelto un **`Panel`**, ossia un pannello in grado di contenere una serie di **`Button`** che eseguono degli `Operator`. La costruzione di un `Panel` è molto simile a quella di un `Operator`: anche in questo caso è necessario costruire una classe che erediti da un'altra (**`bpy.types.Panel`**), anche in questo caso si ha a disposizione una serie di attributi da poter personalizzare in base alle necessità e anche in questo caso la classe costruita definisce una funzione secondo lo standard definito da Blender.
Oltre agli attributi precedenti, se ne presentano tuttavia altri tre:

- **bl_space_type**: lo spazio visibile di Blender nel quale il pannello verrà visualizzato (per esempio la 3D Viewport piuttosto che l'Image Editor, o altri);
- **bl_region_type**: la regione nella quale il pannello verrà utilizzato. È un `enum` in

  `['WINDOW', 'HEADER', 'CHANNELS', 'TEMPORARY', 'UI', 'TOOLS', 'TOOL_PROPS', 'PREVIEW', 'HUD', 'NAVIGATION_BAR', 'EXECUTE', 'FOOTER', 'TOOL_HEADER']`

  con `'WINDOW'` come valore di default;
- **bl_context**: definisce il `context` al quale il pannello appartiene.

Inoltre, è impossibile non citare la funzione **draw()**, la quale definisce gli elementi da visualizzare sul pannello.
In particolare, per realizzare un pannello visibile direttamente dal Layout è stata realizzata una classe **LayoutPanel** come segue.

Codice 5.11: Classe che definisce un pannello con Button cliccabili

```
1  class LayoutPanel(bpy.types.Panel):
2      bl_label = # Label dell'add-on
3      bl_idname = "SCENE_PT_layout"
4      bl_space_type = 'PROPERTIES'
5      bl_region_type = 'WINDOW'
6      bl_context = "scene"
7
8      def draw(self, context):
9          layout = self.layout
10
11         scene = context.scene
12
13         row = layout.row()
14         column = layout.column()
15         column.scale_y = 2.0
16         row.operator("scene.main_flow")
17         row.operator("scene.select_dir")
```

Nello specifico, in una riga vengono visualizzati i due Button in grado di eseguire rispettivamente i due operatori `MainFlowOperator` e `CreateCSVOperator`.

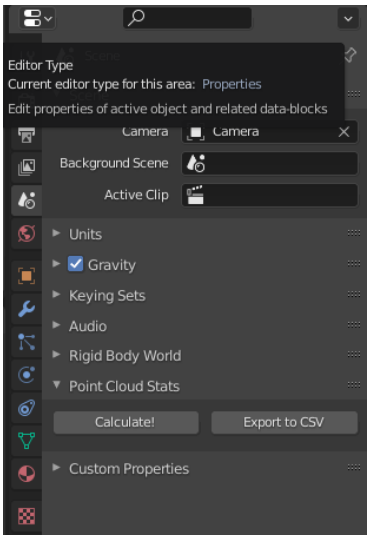


Figura 5.5: Pannello con i due Button

5.3.4 Pannello di visualizzazione

Non è detto però che l'utente non voglia che i dati raccolti siano sempre disponibili e **visibili** direttamente dall'interfaccia di Blender: ciò gli permetterebbe di evitare di controllare il CSV ogni volta lo si richieda, e di delegare al citato file l'unico utilizzo per il quale è stato pensato, ossia quello di esportazione dei dati su un supporto esterno, come una base di dati.
Proprio per questo motivo, è stato pensato di fornire all'utente un'ulteriore visualizzazione mediante un **pannello** visibile direttamente sulla 3D Viewport.
In base all'add-on realizzato, vi sono alcune differenze grafiche dipendenti dalla quantità di dati mostrata a video, ma il metodo di implementazione è pressoché identico. Difatti, in entrambi i casi il pannello presenta un elenco di dati, stampati direttamente dal dizionario `meshesData`, contenente i vari `meshInfo` per ogni chiave `obj`.

Ogni qual volta un oggetto viene selezionato, viene preso il suo riferimento e vengono estratte dal dizionario ad esso relativo le informazioni calcolate in precedenza, le quali poi verranno stampate sul pannello, suddivise in varie **Tab**.
Ciò, naturalmente comporta l'implementazione di due diversi elementi grafici e, per cui, due diversi tipi di classe: la prima, denominata **`View3DPanel`**, la quale implementa il pannello da mostrare in 3D Viewport, e la seconda **`<X>Tab`** (dove a `<X>` va sostituita la sua utilità, in base ai dati mostrati), la

quale implementa il Tab nel quale i dati vengono effettivamente mostrati. Naturalmente, le due tipologie di classe sono caratterizzate dagli stessi attributi e metodi descritti nei paragrafi precedenti.

Codice 5.12: Classe del pannello visualizzato in 3D Viewport

```
1 class View3DPanel:
2     bl_space_type = 'VIEW_3D'
3     bl_region_type = 'UI'
4     bl_category = # Nome add-on
5 
6     @classmethod
7     def poll(cls, context):
8         return (context.object is not None)
```

I vari attributi, come si può notare, sono stati riempiti in modo da visualizzare il pannello in 3D Viewport. L'attributo **bl_category** permette invece di mostrare il nome dell'add-on nella "linguetta" visibile sulla destra dell'interfaccia di Blender.
Da notare il **@classmethod poll()**: esso, preso in input il `context` e una classe, visualizza l'oggetto se non nullo.

A scopo rappresentativo (poiché da Tab a Tab l'implementazione non varia) viene di seguito mostrata la classe dedita alla visualizzazione dei dati in un Tab. Ogni Tab, come accennato in precedenza, viene visualizzato direttamente sul pannello.

Codice 5.13: Classe che identifica il Tab di visualizzazione dei dati

```
1 class InfoTab(View3DPanel, bpy.types.Panel):
2     bl_idname = "VIEW3D_PT_test_1"
3     bl_label = "Info"
4 
5     def draw(self, context):
6         global meshesData
7         obj = bpy.context.active_object
8         if obj.name in meshesData:
9             meshDict = meshesData[obj.name]
10            meshDict = dict(itertools.islice(meshDict.items(),
                  5))
11            for k, v in meshDict.items():
12                self.layout.label(text = k + ": " + str(v))
13            row = self.layout.row()
14            row.prop(obj, '["Obj Type"]')
15            meshesData[obj.name]['objType'] = obj['Obj Type']
```

Com'è possibile notare dalla funzione `draw()`, viene utilizzato il dizionario `meshesData`, contenente tutte le informazioni su ogni oggetto presente in

scena e dichiarato come `global` per mantenere il medesimo riferimento nel corso di tutto lo script.
In **`obj`** viene poi inserito il riferimento all'oggetto attualmente selezionato: se il nome di quell'oggetto è presente (come chiave) sul dizionario, allora il dizionario ulteriormente contentuto in `meshesData`, ossia `meshInfo`, viene "**tagliato**" mediante la funzione **`islice()`** del modulo **`itertools`**. Ciò permette la visualizzazione dei soli dati interessati in quello specifico Tab. La sezione tagliata viene poi salvata in una variabile **`meshDict`**.
La funzione che ne permette la visualizzazione è **`self.layout.label()`**, la quale prende la chiave e il valore del dizionario `meshDict` e li visualizza sul pannello.
Di seguito si mostrano i risultati finali di queste implementazioni.

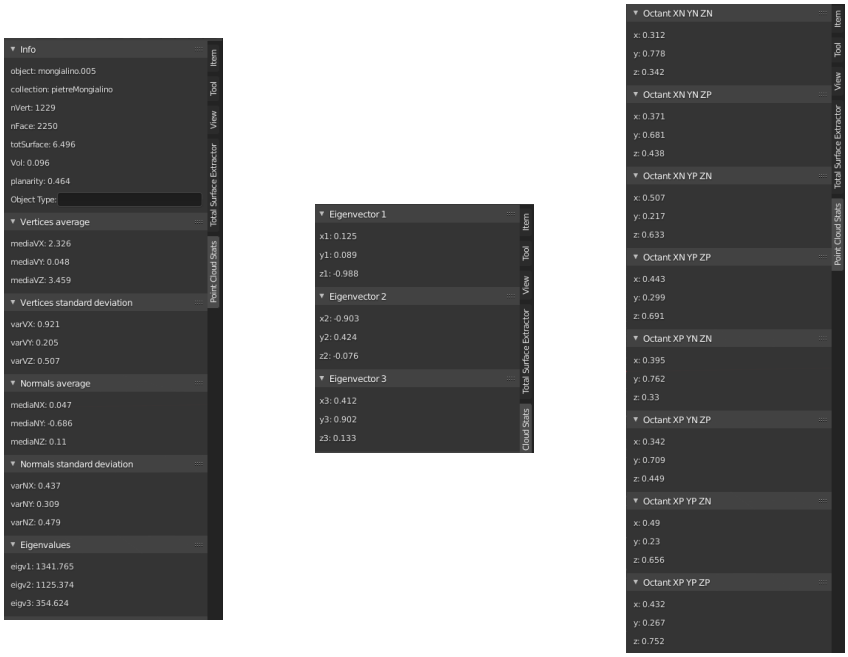


Figura 5.6: Pannello di visualizzazione dei dati per l'add-on *Point Cloud Stats*

Figura 5.7: Pannello di visualizzazione dei dati per l'add-on *SurfEx*

Le ultime due righe del **Codice 5.13** verranno infine analizzate nel paragrafo successivo.

5.3.5 Campo di inserimento e Custom Properties

Fino ad adesso si è rimandata, per questioni di ordine, la descrizione relativa al campo di inserimento manuale presente in ambedue gli add-on.

Nelle ultime due righe del **Codice 5.13** è possibile notare l'implementazione di tale **campo di testo**. Il suo funzionamento è presto detto: ogni oggetto di Blender, mesh o meno che sia, ha a disposizione tutta una serie di proprietà definibili direttamente dall'utente e note come **Custom Properties**: il tipo di una singola Custom Property può variare a seconda della necessità dell'utente, per cui si può passare da un `bool`, ad un `float` ad una `string`. In questo caso si è optato per il tipo **`string`**.
Innanzitutto, è stato necessario aggiungere un nuovo campo di proprietà ad ogni singolo oggetto visibile in scena: ciò viene fatto nella funzione `main()`, la quale descrive il back-end del software, attravero la seguente riga di codice:

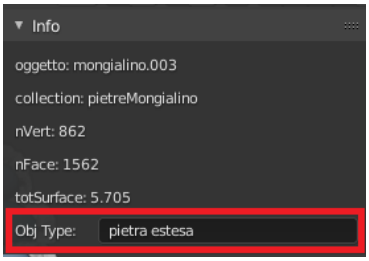
```
obj['Obj Type'] = ""
```

Come si può notare, l'oggetto funziona alla stregua di un **dizionario** con una serie di chiavi; viene per cui aggiunta una nuova chiave **`'Obj Type'`** che

ha come valore una stringa inizialmente vuota. È necessario, tuttavia, che questo valore venga modificato dalla GUI fornita: in realtà, Blender fornisce nativamente la possibilità di aggiungere e modificare una Custom Property, ma integrare la possibilità di modifica nell'add-on ne aumenta la semplicità d'uso.

Mediante la penultima riga del **Codice 5.13**, viene consentita la **visualizzazione** del campo di testo direttamente sul pannello implementato e descritto nel precedente paragrafo. Naturalmente se `obj` è l'oggetto selezionato, il campo di testo farà riferimento a quello specifico oggetto per la chiave `'Obj Type'`.
Mediante l'ultima riga, viene invece consentita la **sovrascrittura** del valore presente sul dizionario `meshesData` alla modifica di tale campo di testo: ciò comporta la scrittura di tale valore sul dizionario interno `meshInfo` che fa riferimento ad `obj`, comportandone la scrittura anche sul file CSV alla sua esportazione.

(a)



(b)

mongialino.003	pietreMongialino	862	1562	5.705	pietra estesa

Figura 5.8: Campo di testo riempito (a) e valore scritto sul file CSV (b)

Questa implementazione è stata pensata per quegli utenti che vogliono ulteriormente **categorizzare**, in maniera manuale, gli oggetti in scena in base alle caratteristiche calcolate nel corso dell'esecuzione dei due script, permettendo così una suddivisione degli stessi in diverse **classi**.

5.3.6 Registrazione delle classi

Ultimo passaggio, ma non per questo meno importante, è quello di **registrazione** delle classi implementate. Infatti, è necessario che le classi vengano effettivamente istanziate, affinché gli operatori e gli elementi grafici possano essere usati e visualizzati mediante Blender in modo del tutto nativo: in questa maniera, sembrerà che le funzioni fornite dall'add-on facciano parte del software.

Per istanziare le classi viene utilizzata la funzione **`register()`**: essa richiama la funzione **`bpy.utils.register_class()`** per ogni classe (operatore o grafica che sia) implementata nell'add-on. Si presenta di seguito un esempio.

Codice 5.14: Collegamento delle classi implemente

```
1 def register():
2     bpy.utils.register_class(MainFlowOperator)
3     bpy.utils.register_class(CreateCSVOperator)
4     bpy.utils.register_class(LayoutPanel)
5     regTabs()
```

La funzione **`regTabs()`** è stata implementata per avere maggior ordine nel codice: essa, in particolare, registra ogni classe `<X>Tab`, il cui numero varia a seconda dell'add-on.

È necessario definire anche lo **scollegamento** di tali classi, naturalmente. In questo caso, si utilizza la funzione **`unregister()`**, la quale, in maniera abbastanza simile alla sua "gemella" vista precedentemente, richiama la funzione **`bpy.utils.unregister_class()`** per ogni classe implementata nell'add-on.

Codice 5.15: Scollegamento delle classi implementate

```
1 def unregister():
2     bpy.utils.unregister_class(MainFlowOperator)
3     bpy.utils.unregister_class(CreateCSVOperator)
4     bpy.utils.unregister_class(LayoutPanel)
5     unregTabs()
```

In maniera del tutto identica, è stata inoltre implementata una funzione **`unregTabs()`**, la quale scollega ogni classe `<X>Tab` dello specifico add-on.

Conclusioni

Dalle analisi di carattere geometrico effettuate mediante i due add-on, è facile comprendere come Blender, mediante lo scripting, si possa prestare non solo all'utilizzo "artistico" nel mondo dell'animazione, ma – con gli adeguati strumenti – anche a quello dell'analisi dei dati di modelli 3D virtuali di oggetti esistenti nel mondo reale.
Infatti, da questo punto di vista, lo sviluppo degli add-on di terze parti apre un ampio panorama di possibilità: Blender, unito alla potenza di Python, fornisce un efficace software per l'analisi di dati ottenibili direttamente da modelli 3D contenuti in una scena e ottenuti, per esempio, per fotogrammetria.

I due add-on costituenti questo progetto di Tesi, oltre ad offrire già nella loro attuale forma un valido aiuto alle personalità che gravitano nelle due realtà alle quali ognuno di essi si rivolgono, potrebbero naturalmente evolvere e divenire un ben più potente strumento di analisi e sintesi dei dati.

Si pensi infatti ad un utilizzo insieme con un classificatore in machine learning che, con i dati ottenuti mediante l'add-on *Point Cloud Stats*, possa aiutare gli archeologi a costruire una base di dati indicizzata in base alla forma delle pietre che costituiscono una specifica parete e, da qui, di stimare un'ipotetica età di costruzione dell'opera muraria.
Interessanti sono anche gli utilizzi nel rilievo architettonico: l'add-on *SurfEx* potrebbe infatti essere la base di un sistema **HBIM** (**H**eritage **B**uilding **I**nformation **M**odeling).

Si pensi infine a tantissime altre realtà che potrebbero usufruire di questo efficace strumento di analisi dei dati, partendo da una "semplice" conversione fotogrammetrica di un oggetto presente nel mondo reale.

Bibliografia

[1] **GALLO G., BUSCEMI F., FERRO M., FIGUERA M., RIELA P.M.** *Abstracting stone walls for visualization and analysis.* Pattern Recognition. ICPR International Workshops and Challenges, 2021.

[2] **RUSSO FORCINA S.** *Rilievo digitale e Computer Grafica per la conservazione dei beni architettonici e culturali: i Fontanoni dell'Acquanova in Caltagirone.* Laurea in Ingegneria Edile-Architettura, Università degli Studi di Catania, 2021.

[3] **WIKIPEDIA**, *Delta di Kronecker.* `https://it.wikipedia.org/wiki/Delta_di_Kronecker`, ultima visita il 14/03/2021.

[4] **WIKIPEDIA**, *Teorema spettrale.* `https://it.wikipedia.org/wiki/Teorema_spettrale`, ultima visita il 14/03/2021.

[5] *BlenderBIM Add-on.* `https://blenderbim.org/`, ultima visita il 19/03/2021.

[6] **BLENDER FOUNDATION**, *Blender 2.92.0 Python API.* `https://docs.blender.org/api/2.92/`, ultima visita il 13/03/2021.

[7] **WIKIPEDIA**, *Momento di inerzia.* `https://it.wikipedia.org/wiki/Momento_di_inerzia`, ultima visita il 13/03/2021.

[8] **POINSOT L.** *Outlines of a New Theory of Rotatory Motion.* R. Newby, Cambridge, 1834.

[9] **LANDAU L.D., LIFSHITZ E.M.** *Mechanics.* Butterworth-Heinemann, London, 3rd edition, 1976.