Università
Ca'Foscari
Venezia

Corso di Laurea
in INFORMATICA

Tesi di Laurea Triennale

Accelerazione del Workflow di progettazione aerodinamica: Un Plugin per Alias basato sulle Radial Basis Functions

Relatore
Prof. Nicola Prezza

Correlatori
Prof. Marco Evangelos Biancolini
Dott. Marco Camponeschi

Laureando
Davide Zambon
Matricola 898103

Anno accademico
2024/2025

Abstract

Nel lavoro di progettazione automobilistica è comune alternare fasi di modellazione geometrica a fasi di simulazione aerodinamica. Tuttavia, ogni modifica geometrica richiede tipicamente una nuova simulazione, con un conseguente costo computazionale elevato, spesso di intere giornate.

Le Radial Basis Functions (RBF), abbinate ad un solutore adjoint, permettono di approssimare il risultato di una simulazione in presenza di piccole modifiche di forma, evitando la necessità di rieseguire l'intero calcolo CFD (Computational Fluid Dynamics).

Alias è un software di computer-aided design (CAD) utilizzato per modellare la forma delle automobili tramite superfici NURBS (Non-Uniform Rational Basis Spline). Queste superfici vengono successivamente discretizzate in una mesh, su cui si eseguono simulazioni fluidodinamiche. Ogni vertice della mesh porta con sé parametri che rappresentano l'interazione con l'aria.

Il plugin sviluppato in questo progetto si integra con Alias per catturare le modifiche geometriche effettuate dall'utente (tramite lo spostamento delle curve) e trasferirle alla mesh usando le RBF, aggiornando in tempo quasi reale anche le stime aerodinamiche.

Infine, si propone una futura estensione del sistema con una visualizzazione grafica dei risultati CFD sotto forma di mappa di colore direttamente sulla carrozzeria.

Desidero ringraziare tutte le persone che mi hanno accompagnato durante questo percorso.
Grazie alla mia famiglia, agli amici e alla mia ragazza per il sostegno costante che non mi hanno mai fatto mancare.
Ringrazio l'Università Ca' Foscari e il Prof. Nicola Prezza per il percorso formativo e il supporto ricevuto.
Un ringraziamento speciale a Giorgio Urso, che mi ha messo in contatto con RBF Morph, e al Prof. Marco Biancolini per avermi dato l'opportunità di lavorare su un progetto stimolante, cuore di questo elaborato.

Indice

Elenco delle figure

Capitolo 1

Introduzione

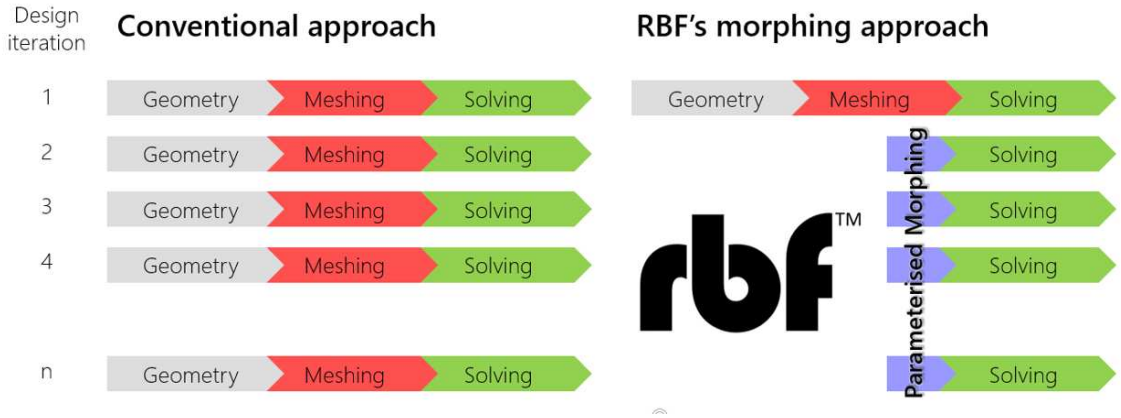


Figura 1.1: RBF Morph

Nell'ambito della progettazione automobilistica, la forma del veicolo deve essere valutata sia dal punto di vista estetico che funzionale. Ogni modifica, anche minima, comporta un compromesso tra design attraente e rispetto dei vincoli stringenti sulle prestazioni aerodinamiche.

Il lavoro del progettista si svolge quindi in equilibrio tra due esigenze: da un lato la volontà di ottenere una linea accattivante e distintiva, dall'altro la necessità di soddisfare i requisiti di efficienza fluidodinamica, determinanti per le prestazioni complessive del veicolo. Un approccio comune consiste nell'alternare fasi di modellazione con simulazioni CFD (Computational Fluid Dynamics), adattando progressivamente la geometria per ottimizzare forma e funzionalità.

Questo alternarsi di attività rende però molto oneroso e lungo il lavoro, in quanto una singola simulazione aerodinamica è computazionalmente molto costosa e richiede una quantità di tempo non indifferente, risultando quindi nella perdita di intere giornate di lavoro sul modello finale.

Il plugin qui presentato va a inserirsi proprio in questo contesto. L'idea, nata all'interno di RBF Morph, azienda per la quale lavoro, è quella di applicare le Radial Basis Functions (RBF), abbinate a un solutore adjoint, per stimare in modo rapido l'impatto aerodinamico di piccole modifiche geometriche, evitando la riesecuzione completa della simulazione CFD. Questo approccio consente una valutazione quasi istantanea dell'efficienza, riducendo drasticamente i tempi della fase iterativa.

Alias, software CAD sviluppato da Autodesk, è lo strumento scelto per l'integrazione del sistema. Pensato specificamente per la modellazione estetica delle carrozzerie, si basa su superfici NURBS e consente un controllo preciso della geometria.

Il funzionamento del plugin può essere riassunto nei seguenti passaggi:

1. Si parte da un modello CAD costituito da curve e superfici NURBS, e la sua discretizzazione in una mesh. Ad essi sono associati i risultati di una prima simulazione CFD, inclusa una matrice di sensibilità definita sui vertici della mesh;

2. Il plugin acquisisce uno snapshot della configurazione iniziale, costituito da un file `.step` (geometria) e da un file `.obj` (mesh);

3. Il progettista modifica la forma della geometria agendo direttamente sulle curve;

4. Un tool esterno proprietario calcola la deformazione corrispondente e genera un nuovo file `.obj` contenente la mesh aggiornata;

5. I parametri CFD associati ai vertici vengono aggiornati in base alla nuova configurazione; l'integrazione della matrice di sensibilità sulle nuove posizioni fornisce una stima approssimata, ma coerente, della variazione dell'efficienza aerodinamica.

Lo sviluppo futuro prevede la visualizzazione diretta dei risultati CFD sulla superficie del modello, tramite una mappa cromatica che evidenzia localmente i valori dei parametri.

La tesi verrà quindi organizzata con questa struttura:

1. Il **Capitolo 2** introduce la teoria dietro alle Radial Basis Functions, descrive brevemente i concetti fondamentali di fluidodinamica computazionale (CFD) e presenta le basi teoriche delle NURBS e delle mesh, con riferimento all'ambiente Alias;

2. Il **Capitolo 3** illustra l'architettura del plugin sviluppato, le sue componenti principali, le strutture dati, gli algoritmi usati e le scelte implementative;

3. Nel **Capitolo 4** si fornisce un esempio di utilizzo reale del plugin;

4. Infine, il **Capitolo 5** propone possibili implementazioni per gli sviluppi futuri del progetto.

Capitolo 2

Background teorico e matematico

In questo capitolo vengono presentati i concetti teorici e matematici che costituiscono la base del lavoro di tesi. L'obiettivo non è quello di fornire una trattazione esaustiva, ma di delineare i principi essenziali che permettono di comprendere:

1. come la geometria sia rappresentata all'interno di Alias mediante NURBS;
2. in che modo tale geometria venga discretizzata in una mesh per poter svolgere analisi numeriche;
3. i fondamenti della Computational Fluid Dynamics (CFD) e i parametri fisici di interesse (in particolare il *coefficiente di resistenza aerodinamica*, l'*efficienza* e la *matrice di sensibilità*);
4. l'utilizzo delle Radial Basis Functions come strumento per il morphing geometrico e la propagazione delle deformazioni sulla mesh.

Questi concetti, sebbene appartenenti a domini teorici distinti (geometria computazionale, analisi numerica e fluidodinamica), si intrecciano in modo naturale nello sviluppo del plugin: dalla rappresentazione CAD alla discretizzazione, dalla simulazione al controllo delle prestazioni aerodinamiche.

2.1 Non-Uniform Rational B-Spline

Le NURBS (Non-Uniform Rational B-Spline) sono una modellazione matematica usata in computer grafica per rappresentare curve e superfici, generalizzazione delle curve B-spline e delle curve di Bézier. Sono

un'ottima soluzione dal punto di vista informatico, in quanto per rappresentare delle superfici NURBS è richiesta una quantità di informazione complessiva molto minore rispetto alla rappresentazione delle stesse tramite altri metodi, come per esempio le mesh, e inoltre permettono una manipolazione molto più efficiente e di facile applicazione.

Lo studio ingegneristico delle NURBS ha inizio solo nel 1964 [1], la loro origine però è di gran lunga precedente, e aiuta anche a comprenderne il significato geometrico.

Quando i disegni tecnici erano fatti a mano, le curve libere non potevano essere disegnate con l'utilizzo degli strumenti classici come riga e compasso. Venivano utilizzati dei materiali flessibili ed elastici (alcuni legni in particolare), incastrati e tenuti in posizione da alcuni perni, detti nodi.

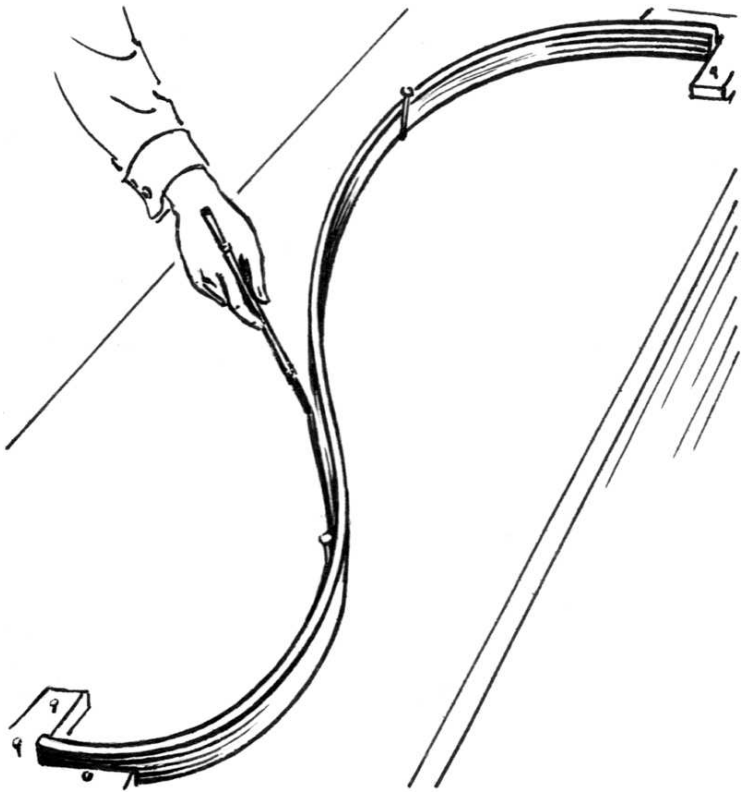


Figura 2.1: Spline in legno. Immagine tratta da Wikimedia Commons, archivio *Pearson Scott Foresman*, dominio pubblico. Per tracciare le curve, un materiale flessibile veniva incastrato tra dei perni.

Il materiale assumeva una curva continua e priva di discontinuità angolari.

A partire dalla formalizzazione del 1964, furono poi sviluppati algoritmi di computer grafica e rendering interattivo basati sulle NURBS, che oggi costituiscono lo standard nella Computer-Aided Geometric Design (CAGD) e sono utilizzate in tutti i principali software CAD, incluso Autodesk Alias. [2]

Una curva NURBS è caratterizzata da quattro elementi fondamentali: il grado p, i punti di controllo (CV) $P_0, \ldots, P_n$, il vettore dei nodi $U = \{u_0, \ldots, u_m\}$ e i pesi $w_0, \ldots, w_n$.

- Il grado p è un intero non negativo, tipicamente compreso tra 1 e 5 per motivi computazionali, ma in teoria non vi è alcun limite superiore. È concettualmente simile al grado dei polinomi: aumentando il grado si ottengono più gradi di libertà e quindi maggiore flessibilità. Una curva di grado 1 è una polilinea, mentre gradi più alti consentono curvature più lisce. È sempre possibile elevare il grado senza alterare la forma; l'operazione inversa non è in generale possibile se non introducendo approssimazioni;
- I punti di controllo $P_i \in \mathbb{R}^d$ sono $n+1$ (con $i = 0, \ldots, n$), e devono soddisfare la condizione $n+1 \geq p+1$. Ogni punto ha un peso $w_i > 0$ che influisce sull'andamento della curva;
- Il vettore dei nodi $U = \{u_0, \ldots, u_m\}$ contiene $m + 1 = n + p + 2$ valori non decrescenti. Non rappresentano punti geometrici visibili, ma determinano la porzione di curva influenzata da ciascun punto di controllo e la continuità locale. La molteplicità r di un nodo riduce la continuità a C^{p-r};
- La regola di valutazione è la formula matematica che, dato un parametro $u \in [u_p, u_{m-p}]$, calcola la posizione di un punto sulla curva usando grado, nodi, CV e pesi.

Le caratteristiche principali delle NURBS sono:

1. **Non-Uniform**: il vettore dei nodi può contenere valori ripetuti o

non equidistanti, permettendo di controllare continuità e influenza dei CV;

2. **Rational**: l'introduzione dei pesi w_i consente di rappresentare esattamente forme come cerchi, ellissi e archi, impossibili per sole B-spline polinomiali.

La forma generale di una curva NURBS è:

$$C(u) = \frac{\sum_{i=0}^{n} w_i \, P_i \, N_{i,p}(u)}{\sum_{i=0}^{n} w_i \, N_{i,p}(u)}, \quad u \in [u_p, u_{m-p}]$$

dove:

- P_i sono i punti di controllo;
- w_i i pesi;
- $N_{i,p}(u)$ le funzioni base B-spline di grado p;
- U il vettore dei nodi.

Le B-spline sono definite ricorsivamente:

$$N_{i,0}(u) = \begin{cases} 1, & u_i \leq u < u_{i+1}, \\ 0, & \text{altrimenti}, \end{cases}$$

e per $p \geq 1$:

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} \, N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} \, N_{i+1,p-1}(u),$$

con la convenzione che i termini con denominatore nullo valgono zero.

Dalla curva alla superficie il passo è breve. Le NURBS possono infatti essere estese al caso bidimensionale, descrivendo così superfici bidimensionali curve in uno spazio tridimensionale.

Nel contesto della tesi, le NURBS costituiscono il punto di partenza per la definizione delle deformazioni. La loro formulazione matematica è riportata a titolo informativo: essa viene infatti utilizzata internamente dal software CAD per la rappresentazione e il rendering grafico, ma non è manipolata direttamente all'interno del plugin.

L'interazione con le curve avviene tramite l'API (Application Pro-

gramming Interface) di Alias, che ne fornisce un'interfaccia astratta. In questo modo non è necessario implementare manualmente le formule delle B-spline o dei pesi razionali, ma è possibile accedere direttamente agli elementi fondamentali della curva, come:

- i punti di controllo (CV), con relative coordinate e pesi;
- il grado della curva;
- il vettore dei nodi;
- operazioni di valutazione (*eval*) della curva in corrispondenza di un parametro u;
- funzioni varie di interrogazione come per esempio per il calcolo della lunghezza.

In pratica, mentre la teoria delle NURBS definisce come una curva venga generata a partire da nodi, pesi e funzioni base, il plugin interagisce con queste entità attraverso le funzioni messe a disposizione dalla classe `AlCurve`[1]. L'attenzione non è quindi posta sulla ricostruzione analitica delle formule, ma sull'uso dei metodi di accesso e manipolazione forniti dal software CAD. Questo livello di astrazione consente di concentrarsi sulle operazioni necessarie al morphing senza dover gestire direttamente la complessità matematica sottostante.

2.2 Mesh e discretizzazione

La mesh è una rappresentazione discreta di una geometria continua. In termini semplici, significa trasformare curve e superfici, che sono oggetti matematici teoricamente infiniti e continui, in una collezione finita di elementi elementari. Questi elementi possono essere segmenti, triangoli,

[1]Dell'interfaccia delle classi verrà discusso più approfonditamente nella sezione 3.1

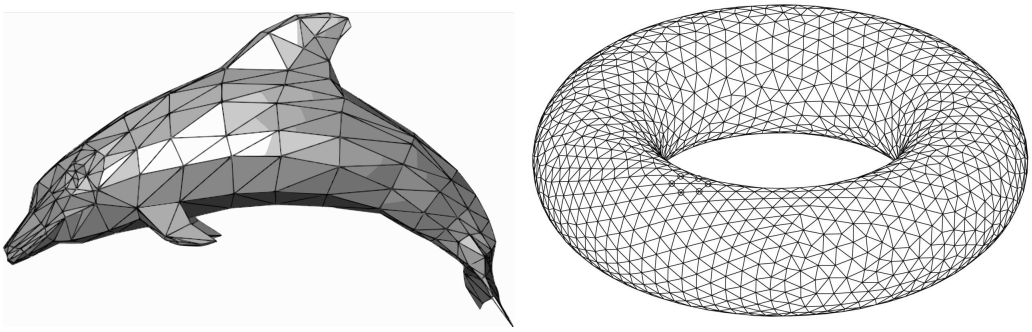


Figura 2.2: Due esempi di discretizzazione della superficie: la rappresentazione di un delfino e di un toroide. Da wikimedia: delfino e toroide. Rispettivamente `Public Domain` e `CC BY-SA 3.0` (autore: Ag2gaeh)

quadrilateri, tetraedri o esaedri, a seconda che si stia lavorando in due o tre dimensioni.

Il concetto di discretizzazione nasce dall'esigenza di svolgere calcoli numerici su oggetti geometrici. Un computer non può manipolare direttamente una superficie continua definita da un'equazione, ma può invece elaborare un insieme di punti e connessioni che ne approssimano la forma. Da qui deriva il termine "mesh", che letteralmente significa "rete": una griglia di nodi collegati tra loro che ricostruisce la geometria di partenza con un certo grado di approssimazione.

Storicamente, le prime mesh utilizzate nei metodi numerici erano strutturate, ovvero basate su una suddivisione regolare dello spazio in quadrati o cubi. Questo approccio semplificava notevolmente la gestione dei dati e i calcoli, ma si adattava male a geometrie complesse. Con il tempo si è diffuso l'uso delle mesh non strutturate, composte da triangoli, che si adattano molto meglio a forme arbitrarie. Oggi sono lo standard in gran parte delle simulazioni ingegneristiche. In alternativa, esistono mesh ibride, che combinano zone strutturate (per esempio lungo le direzioni principali del flusso) e zone non strutturate (per seguire contorni complessi). Le mesh sono usate come rappresentazione delle geometrie 3D nel contesto delle simulazioni fluidodinamiche, in quanto i software utilizzati per quest'ultime lavorano su insiemi di punti. Per questi fini,

non tutte le mesh sono uguali: la qualità della discretizzazione influisce direttamente sulla precisione e sull'efficienza della simulazione. Alcuni parametri fondamentali sono:

- la dimensione degli elementi, che deve essere sufficientemente piccola nelle regioni dove i fenomeni fisici variano rapidamente;
- il grado di distorsione degli elementi, che deve essere limitato per evitare instabilità numeriche;
- la transizione graduale tra aree fitte e aree più grossolane, che permette di ridurre il numero totale di nodi senza perdere accuratezza. [3]

2.3 Computational Fluid Dynamics

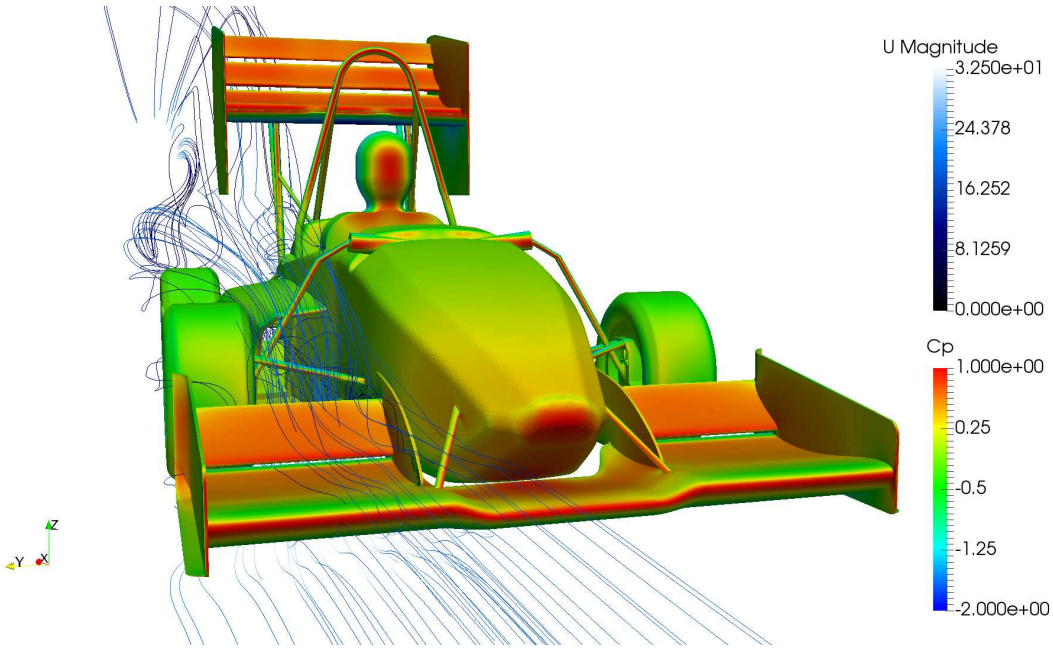


Figura 2.3: Formula Students Aerodynamics, licenza `CC BY-SA 4.0`, autore: theansweris27.com.

La **Computational Fluid Dynamics (CFD)** è il ramo della fluidodinamica che utilizza metodi numerici e algoritmi computazionali per analizzare e risolvere i problemi legati al moto dei fluidi. In altre parole, la CFD permette di "simulare" al computer il comportamento di un

fluido (aria, acqua, olio, ecc.) che interagisce con una geometria, senza dover ricorrere necessariamente a esperimenti fisici in galleria del vento o in laboratorio.

Per comprendere i concetti chiave della CFD è necessario introdurre, seppur in maniera semplificata, le leggi fondamentali che governano i fluidi e il modo in cui queste vengono tradotte in calcoli numerici. [4]

2.3.1 Le equazioni di base della fluidodinamica

Il comportamento di un fluido è descritto da un insieme di equazioni differenziali chiamate equazioni di Navier–Stokes, formulate nel XIX secolo da Claude-Louis Navier e George Gabriel Stokes. Esse derivano dall'applicazione dei principi fondamentali della meccanica:

1. **Legge di conservazione della massa** (equazione di continuità)

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0$$

   dove ρ è la densità del fluido e $\mathbf{u}$ è il vettore velocità. Questa equazione afferma che la massa non si crea né si distrugge: la quantità di fluido che entra in un volume di controllo deve essere uguale a quella che ne esce, tenendo conto di eventuali variazioni locali di densità;

2. **Conservazione della quantità di moto** (seconda legge di Newton applicata ai fluidi)

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} \right) = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f}$$

   dove p è la pressione, μ la viscosità dinamica del fluido, e $\mathbf{f}$ rappresenta forze esterne (ad esempio la gravità);

3. **Conservazione dell'energia**(primo principio della termodinamica)

$$\frac{\partial E}{\partial t} + \nabla \cdot ((E + p)\mathbf{u}) = \nabla \cdot (k \nabla T) + \Phi$$

dove E è l'energia totale per unità di volume, T la temperatura, k la conducibilità termica e Φ il termine di dissipazione viscosa.

Queste equazioni, combinate, sono in grado di descrivere fenomeni complessi come vortici, onde d'urto, turbolenza e separazione del flusso.

2.3.2 Metodi numerici in CFD

Sebbene siano note da quasi due secoli, le equazioni di Navier–Stokes non hanno una soluzione analitica generale. In effetti, dimostrare l'esistenza e la regolarità delle soluzioni in tre dimensioni è uno dei celebri Millennium Problems del Clay Mathematics Institute, tuttora irrisolto.

Per questo motivo, nella pratica ingegneristica si ricorre a metodi numerici: si discretizzano sia le equazioni sia il dominio spaziale (tramite una mesh), e si ottengono sistemi algebrici risolvibili al calcolatore. La CFD nasce proprio da questa esigenza: portare sul piano computazionale problemi altrimenti intrattabili con metodi teorici puri. [4]

Nel contesto della tesi, però, il metodo utilizzato per la simulazione CFD è del tutto trasparente. Il plugin infatti si basa su risultati pronti di simulazioni effettuate. I cambiamenti dei valori fluidodinamici successivi alle modifiche non richiedono la riesecuzione della simulazione, ma utilizzano una sorta di propagazione spaziale del risultato già ottenuto. I risultati CFD assumono quindi un duplice ruolo:

1. analitico, perché fornisce i valori delle prestazioni, che sono da mostrare a video e integrare in una texture di visualizzazione (argomento discusso nel capitolo 5);
2. computazionale, perché le modifiche geometriche operate in Alias si riflettono sulla mesh, che a sua volta influenza la simulazione CFD.

L'obiettivo finale è ottenere un ciclo di progettazione interattivo: il progettista modifica la geometria in ambiente CAD, la mesh si aggiorna in tempo reale e la CFD fornisce rapidamente un feedback quantitativo sulle prestazioni aerodinamiche.

2.3.3 Coefficienti aerodinamici

Le prestazioni aerodinamiche di una geometria sono sintetizzate da diversi **coefficienti**, grandezze adimensionali ottenute normalizzando forze e momenti rispetto a quantità di riferimento (tipicamente la pressione dinamica e un'area caratteristica). Ciò li rende indipendenti dalle unità fisiche e confrontabili tra configurazioni diverse.

Esempi rilevanti includono il coefficiente di resistenza C_d, il coefficiente di portanza C_L, i coefficienti di momento e l'efficienza aerodinamica, che esprimono in forma compatta l'effetto della forma geometrica sul flusso circostante. [5]

Non è necessario, in questa sede, entrare nel dettaglio delle singole definizioni: ciò che è rilevante ai fini di questa tesi è che tali coefficienti costituiscono l'**output di una simulazione CFD** e che il loro legame con la geometria può essere rappresentato mediante la **matrice di sensibilità**, di cui parliamo nella prossima sezione.

Il risultato finale del plugin non consiste nel ricalcolo diretto dei coefficienti, ma nella stima della loro **variazione** (Δ) in seguito alle modifiche di forma applicate alla geometria.

2.3.4 Matrice di sensibilità

In generale, la **matrice di sensibilità** o di *sensitività*[2] descrive come variazioni nei parametri di input di un modello influenzino le grandezze di output. Viene spesso usata in ambito CFD per collegare modifiche geometriche o condizioni al contorno a variazioni di più coefficienti aerodinamici (ad esempio C_d, C_L, momenti, etc).

Nel contesto di questa tesi il problema è però monofunzionale: l'output di interesse è un unico coefficiente (ad esempio il coefficiente di resistenza C_d). In questo caso, la sensibilità si riduce a un **gradiente**

[2]Italianizzazione del termine inglese sensitivity

che associa a ciascun vertice della mesh tre derivate parziali, una per ogni direzione spaziale.

Formalizzazione matematica. Dal punto di vista teorico, la variazione del coefficiente può essere formulata come un **integrale** di un campo di sensibilità distribuito sulla superficie della geometria:

$$\Delta f = \iint_{\Omega} \mathbf{s}(\mathbf{p}) \cdot \Delta\mathbf{p}\, dA,$$

dove $\mathbf{s}(\mathbf{p})$ rappresenta il vettore di sensibilità in un punto $\mathbf{p}$ della superficie, $\Delta\mathbf{p}$ è lo spostamento locale della geometria e A la superficie della geometria.

Nella pratica computazionale, tale integrale viene approssimato discretizzando la superficie in vertici di una mesh e sostituendo l'integrale con una somma sui vertici. La somma è quindi da intendersi come una **somma di Riemann discreta** che approssima l'integrale continuo.

Nel caso specifico del plugin, il dominio geometrico è rappresentato da una mesh con n vertici. Ogni vertice i ha tre gradi di libertà di spostamento $(\Delta x_i, \Delta y_i, \Delta z_i)$, così che l'intero stato geometrico può essere raccolto in un vettore

$$\Delta\mathbf{x} \in \mathbb{R}^{3n}.$$

La sensibilità del coefficiente $f(\mathbf{x}) = C_d(\mathbf{x})$ rispetto a questi spostamenti è descritta da un gradiente distribuito sui vertici, che implementativamente viene salvato come matrice $S \in \mathbb{R}^{n\times 3}$: ogni riga della matrice è definita come

$$S_i = \left(\frac{\partial f}{\partial x_i}, \frac{\partial f}{\partial y_i}, \frac{\partial f}{\partial z_i}\right).$$

La variazione del coefficiente si ottiene come

$$\Delta f \approx \sum_{i=1}^{n} S_i \cdot \Delta\mathbf{p}_i,$$

dove $\Delta\mathbf{p}_i = (\Delta x_i, \Delta y_i, \Delta z_i)$ è lo spostamento del vertice i. In forma compatta:

$$\Delta f \approx S \cdot \Delta\mathbf{x}, \qquad S \in \mathbb{R}^{1\times 3n}.$$

Rappresentazione dei dati. Per praticità implementativa, i dati sono forniti come tabella con n righe e 6 colonne: le prime tre sono le coordinate originali (x, y, z), utilizzate per allineare i vertici della mesh, e le ultime tre sono le componenti di sensibilità. Le coordinate non fanno parte della sensibilità vera e propria, che ha quindi dimensione $3n$.

Applicazione nel plugin. Nel ciclo Alias–CFD, il plugin:

1. legge da file la tabella della sensibilità precalcolata dal solver CFD;
2. riordina le righe per far corrispondere i vertici ai nodi della mesh di Alias;
3. calcola gli spostamenti $\Delta\mathbf{p}_i$ indotti dalle deformazioni nella geometria;
4. per ogni riga (per ogni vertice) esegue il prodotto scalare $\delta f_i = S_i \cdot \Delta\mathbf{p}_i$;
5. Somma tutti i contributi $S_i \cdot \Delta\mathbf{p}_i$ per ottenere la variazione Δf del coefficiente.

In questo modo la matrice di sensibilità funge da ponte tra deformazione geometrica e variazione stimata del coefficiente aerodinamico, consentendo un feedback quantitativo in tempo quasi reale.

2.4 Radial Basis Functions (RBF)

Le **Radial Basis Functions** (RBF) nascono come strumento matematico per l'interpolazione di dati sparsi (*scattered data interpolation*) e per la costruzione di campi scalari o vettoriali a partire da valori noti in un insieme di punti. La loro caratteristica principale è che il valore

della funzione dipende soltanto dalla distanza da un punto centrale (il *centro*). Formalmente:

$$\varphi(\mathbf{x}) = \varphi(\|\mathbf{x} - \mathbf{c}\|),$$

dove $\mathbf{c}$ è il centro della funzione radiale, φ la funzione utilizzata e $\|\cdot\|$ indica la norma euclidea.

Combinando più funzioni radiali centrate in punti diversi $\mathbf{p}_i$, si ottiene una combinazione lineare:

$$s(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i \, \varphi(\|\mathbf{x} - \mathbf{p}_i\|).$$

Qui:

- $\mathbf{x}$ è un punto qualsiasi nello spazio di cui si vuole determinare la nuova posizione;
- $\mathbf{p}_i$ sono i **punti di controllo**, i **centri** delle singole funzioni radiali. Ovvero posizioni nello spazio in cui si conosce o si impone il valore della deformazione;
- α_i sono dei coefficienti;
- N è il numero di punti di controllo (e, di conseguenza, il numero di termini della combinazione lineare).

I coefficienti α_i non sono scelti arbitrariamente, ma si determinano imponendo che la funzione interpolante $s(\mathbf{x})$ assuma i valori desiderati nei punti di controllo. Ovvero, si impone la condizione di interpolazione:

$$s(\mathbf{p}_i) = v_i \qquad \text{per ogni } i = 1, \ldots, N.$$

in cui v_i è lo **spostamento prescritto** nel punto $\mathbf{p}_i$. Questo porta a un sistema lineare di N equazioni con N incognite, la cui soluzione fornisce i coefficienti α_i. In tal modo, il campo di deformazione è l'unica combinazione lineare di funzioni radiali che soddisfa esattamente i vincoli imposti.

In questo modo, il campo di deformazione $s(\mathbf{x})$ riproduce esattamente

i vincoli noti nei punti di controllo e li propaga in maniera coerente a tutti gli altri punti della geometria. [6]

L'uso delle RBF è particolarmente efficace in problemi di deformazione geometrica perché:

1. dipendono solo da distanze euclidee, senza necessità di una parametrizzazione regolare della geometria;
2. garantiscono campi di deformazione *continui*. In altre parole, se un punto della geometria si muove leggermente, anche i punti vicini subiscono spostamenti coerenti e "morbidi". Matematicamente, la funzione $s(\mathbf{x})$ è continua, cioè non ha buchi o valori indefiniti;
3. garantiscono campi di deformazione *regolari*. Una funzione regolare non solo è continua, ma ha anche derivate prime (e spesso seconde) continue. Questo evita deformazioni con spigoli, pieghe o oscillazioni brusche. Per esempio, molte RBF sono C^2 o C^∞, cioè con derivate continue fino al secondo ordine o addirittura infinite;
4. possono riprodurre esattamente trasformazioni rigide (traslazioni e rotazioni) senza introdurre distorsioni spurie.

2.4.1 Ruolo del morphing di mesh nel flusso operativo del plugin

Dal punto di vista concettuale, le RBF permettono di trasferire una deformazione nota (imposta su una geometria CAD) a tutti i nodi di una mesh discreta, generando una nuova mesh coerente con la geometria modificata. Il vantaggio principale è che non è necessario rigenerare la mesh da zero: una mesh esistente può essere aggiornata in modo consistente e regolare.

Nel contesto di questa tesi, le RBF non vengono implementate né parametrizzate direttamente, ma sono trattate come un *tool esterno* che opera come una "black box", una funzione la cui implementazione è del tutto trasparente. Formalmente:

(CAD_originale, CAD_modificato, Mesh) $\longmapsto$ Mesh_deformata.

Nel gruppo di immagini 2.4, 2.5 e 2.6, un esempio visuale del processo.

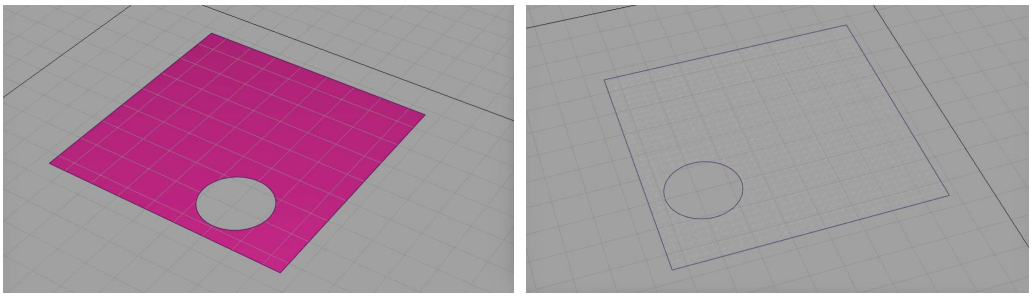


Figura 2.4: Si parte dalla geometria e dalla mesh corrispondente

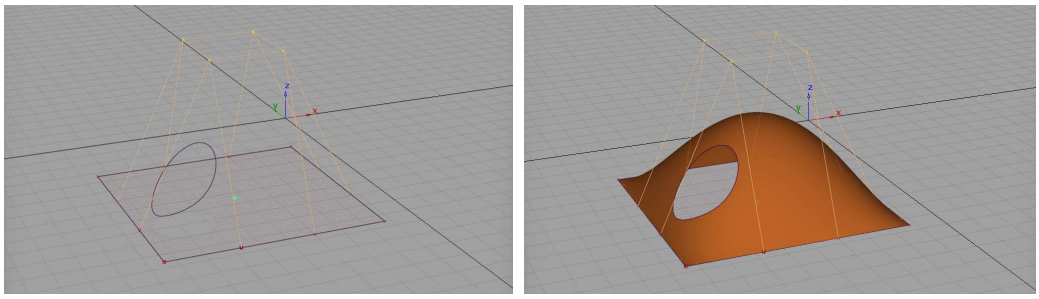


Figura 2.5: Si modifica la geometria

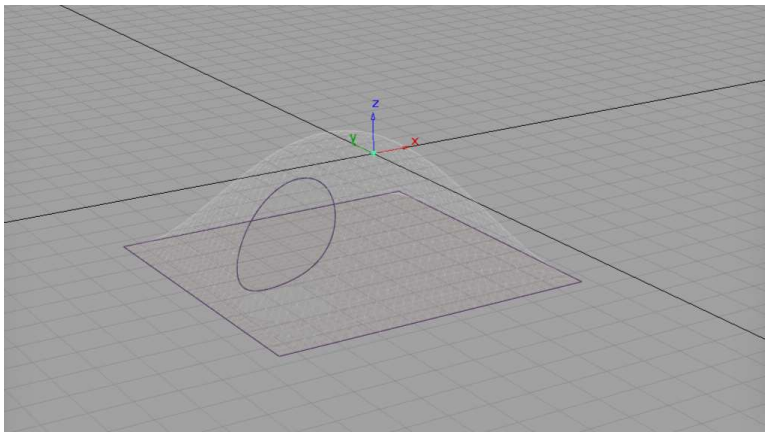


Figura 2.6: Le deformazioni alla geometria vengono propagate alla mesh

Per il plugin, è necessario utilizzare questo tool e non rigenerare la mesh ex novo ad ogni modifica della geometria. Un aspetto fondamentale, infatti, è che la **topologia della mesh** (cioè il numero e l'ordine

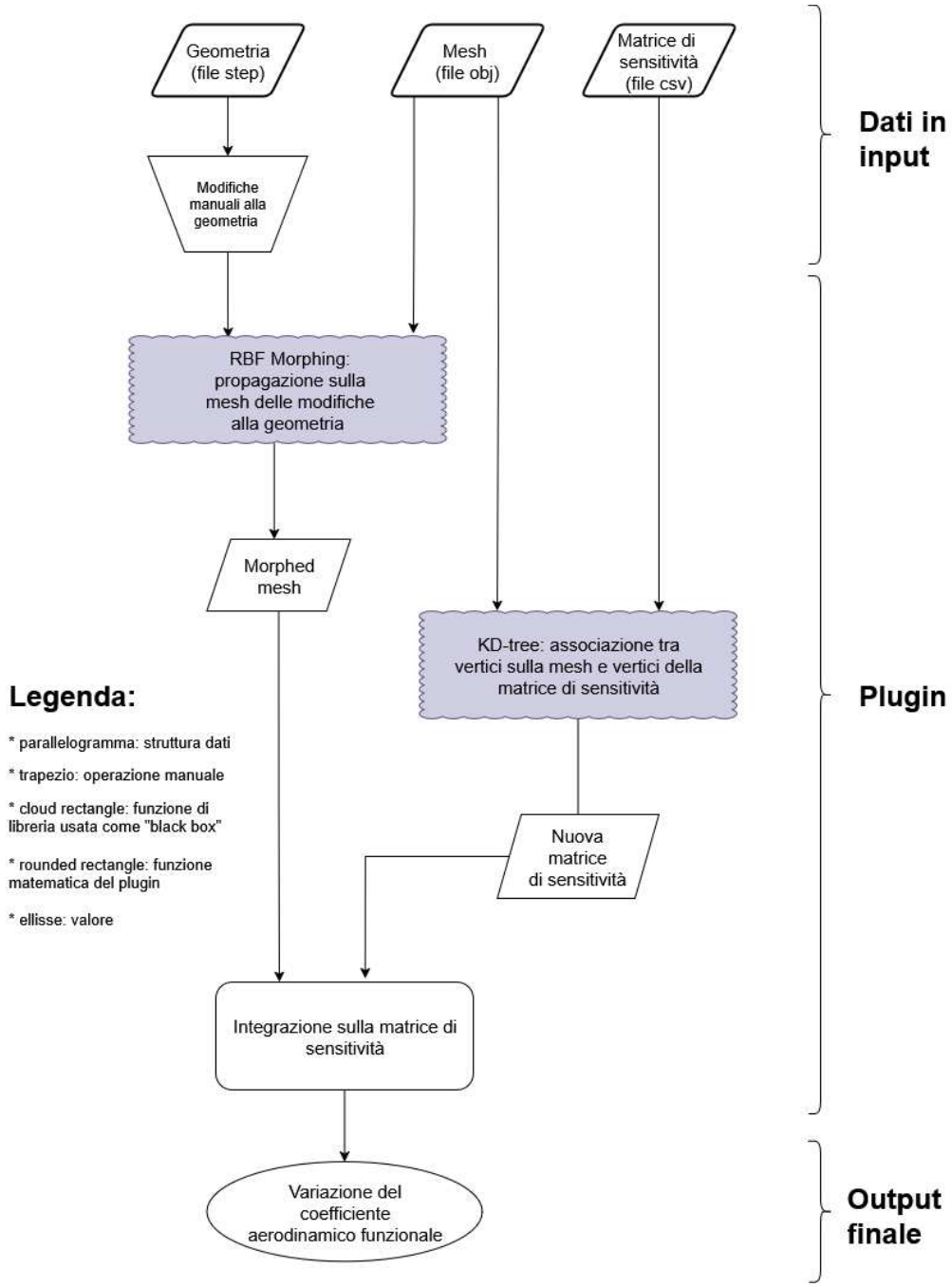
dei nodi) deve rimanere invariata. La deformazione non crea né elimina vertici, ma "muove" quelli esistenti. Questo vincolo è necessario perché la matrice di sensibilità è definita rispetto ai vertici originali: se la mesh fosse rigenerata da zero, l'associazione tra nodi e sensibilità andrebbe persa, rendendo impossibile stimare le variazioni aerodinamiche.

In questo schema, le RBF fungono da meccanismo di interpolazione che garantisce coerenza tra geometria CAD e mesh CFD, pur restando implementativamente nascoste all'utente. L'approccio consente di:

- evitare la rigenerazione della mesh a ogni modifica;
- mantenere la corrispondenza univoca tra nodi della mesh e righe della matrice di sensibilità;
- ottenere un ciclo di progettazione rapido: modifica della geometria $\rightarrow$ deformazione della mesh $\rightarrow$ valutazione immediata tramite sensibilità.

Capitolo 3

Il Plugin



Il processo inizia con tre file[1] distinti che costituiscono lo stato iniziale del progetto:

- **Geometria (file STEP)**: modello CAD del veicolo, descritto come insieme di superfici NURBS. Il formato *STEP* (ISO 10303, *Standard for the Exchange of Product model data*) è uno standard internazionale per lo scambio di dati geometrici tra diversi software di progettazione;
- **Mesh (file OBJ o STL)**: La discretizzazione della geometria in una rete di vertici, spigoli e facce. Deve essere quella utilizzata per la simulazione fluidodinamica. Il formato *OBJ* (originariamente di Wavefront) e il formato *STL* (*STereoLithography*) sono entrambi diffusi per rappresentare mesh triangolari;
- **Matrice di sensibilità (file CSV)**: file tabellare che contiene le sensibilità calcolate durante la simulazione CFD. Il formato CSV (*Comma-Separated Values*) rappresenta dati strutturati in righe e colonne ed è ampiamente usato per lo scambio di dati numerici. Esso è composto da tante righe quante i vertici della mesh, ognuna delle quali ha 6 valori (tre coordinate spaziali e 3 derivate), come definito nella sezione 2.3.4.

L'utente, lavorando nell'ambiente CAD, effettua delle modifiche manuali alla geometria. La geometria modificata viene passata, insieme alla mesh, a un componente esterno che utilizza le RBF (Radial Basis Functions), il quale, usato come una "black box", propaga la deformazione dalla geometria alla mesh, generando una *Morphed mesh*, cioè una nuova mesh che segue fedelmente le modifiche apportate dal designer sulla geometria.

Parallelamente è necessario associare i vertici della mesh a quelli della matrice di sensibilità (ricordando, come descritto nella sezione 2.4, che

[1] una panoramica sulle specifiche dei formati dei file viene data nella sezione 3.2

ogni successivo morphing della mesh mantiene intatta la corrispondenza dei vertici). Per farlo in modo efficiente, si utilizza una struttura dati chiamata k-d tree. Questo algoritmo permette di trovare rapidamente, per ogni vertice della matrice di sensibilità, il suo corrispettivo più vicino sulla mesh originale. L'output di questo step è una nuova matrice di sensibilità, con l'ordine dei vertici corrispondente a quello della mesh.

Infine, viene eseguita un'integrazione sulla nuova matrice: questa operazione aggrega le sensibilità locali per calcolare l'effetto complessivo delle modifiche, secondo la formula già introdotta nella sezione 2.3.4:

$$\Delta f \approx \sum_{i=1}^{n} S_i \cdot \Delta \mathbf{p}_i,$$

.

Il risultato di questo calcolo è la **variazione del coefficiente aerodinamico** funzionale: un singolo valore che stima di quanto la performance aerodinamica del veicolo sia migliorata o peggiorata a seguito delle modifiche. Questo feedback quasi istantaneo è l'obiettivo principale del progetto.

3.1 Alias SDK: un'introduzione

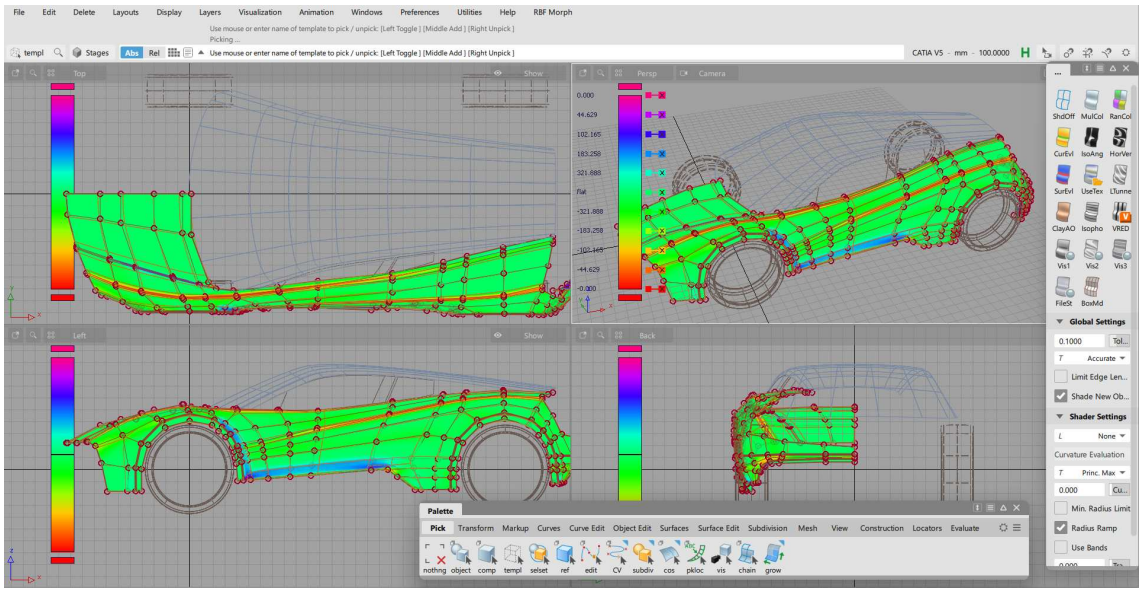


Figura 3.1: Interfaccia di Alias Learning Edition 2025

Autodesk Alias è un software di Computer-Aided Industrial Design (CAID), ampiamente utilizzato nell'industria automobilistica per la modellazione di superfici complesse, come per esempio le carrozzerie. La sua peculiarità risiede nella gestione avanzata e ad altissima precisione delle superfici NURBS, che garantisce un controllo preciso della continuità geometrica, oltre a un ambiente di visualizzazione avanzato, con strumenti per shading, texture e illuminazione, per la generazione di render fotorealistici.

Per poter creare un plugin che interagisca con Alias, è fondamentale comprendere come il software organizza internamente i dati geometrici e quali strumenti offre per manipolarli. Questa sezione introduce l'Alias SDK, il ponte tra l'interfaccia del software e la logica del plugin.

Per questo progetto è stata impiegata la versione *Learning Edition*, funzionalmente equivalente alla licenza completa ma con limitazioni nell'esportazione (disponibile solo nel formato nativo *.wire*). L'ambiente di lavoro permette di manipolare direttamente curve e superfici, applicare

trasformazioni interattive, anche scriptabili, e generare mesh di supporto per successive elaborazioni, come analisi CFD o rendering.

Per l'automazione e l'espansione delle funzionalità, Alias mette a disposizione un ricco SDK in C++, permettendo di automatizzare l'interazione con ogni tipo di entità, da quelle geometriche a quelle di rendering. Ogni oggetto è parte di una struttura a grafo aciclico diretto (DAG)[2], ed è esposto tramite API, che permettono di accedervi, modificare la scena, interagire con le curve, le superfici e gli altri oggetti, nonché di automatizzare flussi di lavoro.

Ci sono due componenti principali:

1. **OpenAlias**: Permette di accedere alla scena attiva, navigando il DAG, e interagire non solo con gli oggetti presenti, ma anche con le azioni dell'utente. Ad esempio, consente di definire azioni in risposta ai movimenti del mouse o alla pressione dei tasti, oltre a gestire input e output con l'utente;

2. **OpenModel**: Permette di gestire file *.wire* (formato nativo di Alias) senza che la scena sia aperta, dando quindi la possibilità di manipolare oggetti e automatizzare sequenze di costruzione direttamente a livello di file, o per operazioni batch.

Molte delle funzioni esposte nelle API sono comuni alle due componenti. Il plugin sviluppato richiede interazione con la scena e con l'utente, di conseguenza le funzioni usate sono quelle esposte in OpenAlias.

La struttura offerta al programmatore è ad oggetti, con gerarchie esplicite tra le classi, safe castings e soprattutto con una gestione intelligente dei puntatori, pensata per evitare memory leaks e sollevare il programmatore dalla gestione della memoria dei puntatori di cui Alias è responsabile.

[2]Struttura dati a grafo, in cui i nodi rappresentano gli oggetti e gli archi le relazioni gerarchiche. Descritto nella sezione 3.1.1

3.1.1 Traversing the DAG

Come riportato nella documentazione ufficiale di Alias[3], un *DAG node* fa parte di una struttura gerarchica simile a un albero, nota come grafo aciclico diretto o DAG. Tutti i nodi appartengono a un singolo DAG contenuto nell'universo corrente. La radice del DAG è accessibile invocando il metodo statico AlUniverse::firstDagNode(). Ogni nodo può essere collegato a un nodo precedente e ad uno successivo per formare una lista di nodi "fratelli". Questa può appartenere a un *group node*, e prende il nome di lista "figlia" del groupNode in questione.

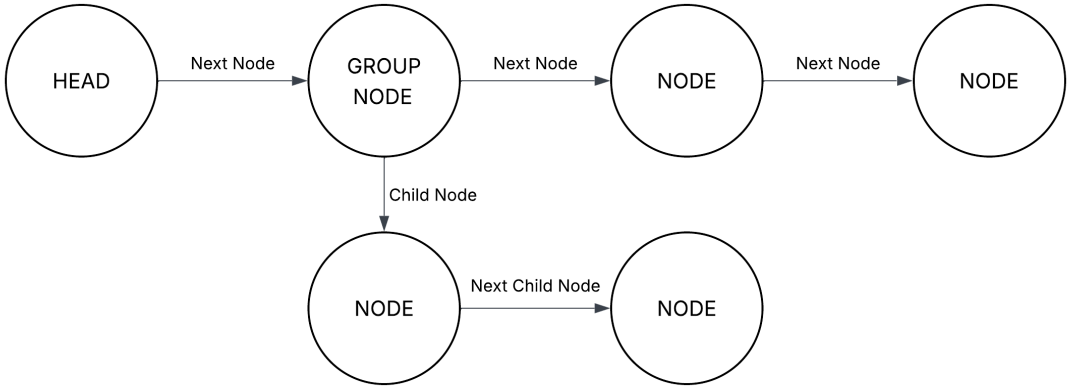


Figura 3.2: Schema del DAG

Come mostrato in figura 3.2, quindi, la struttura è una lista di nodi. Alcuni di questi nodi sono *group nodes*, e possono avere, oltre al fratello (il prossimo nodo in lista), anche dei figli. Questi ultimi sono a loro volta una lista di nodi e, ricorsivamente, hanno la stessa struttura del DAG principale.

Nel modello ad oggetti di Alias, il nome di ogni classe ha la struttura `Al-` (che sta per "Alias") `-classe`. Per esempio AlObject fornisce un'interfaccia generica a tutti gli elementi della scena. A partire da essa

[3]Traduzione libera a cura dell'autore da `https://help.autodesk.com/view/ALIAS/2025/ENU/?guid=GUID-A73FCABA-93DD-468B-8F06-98729CEADDD3`

si possono ottenere puntatori a classi più specifiche mediante metodi di cast dinamico, per esempio

- asDagNodePtr(): se l'oggetto è un *Dag Node*, restituisce il suo puntatore, NULL altrimenti;
- asGroupNodePtr(): se l'oggetto è un *Group Node*, restituisce il suo puntatore, NULL altrimenti;
- E così via, con la medesima struttura, per tutte le classi derivate.

Tramite i metodi childNode() e nextNode() è possibile ottenere il puntatore ai figli e ai fratelli, riuscendo così a navigare il DAG rispettivamente in senso verticale e orizzontale, come nello schema in figura 3.2. La funzione copyWrapper() crea una copia dell'oggetto gestita dal chiamante, che deve quindi essere liberata con delete.

Con questa interfaccia, è possibile esplorare ricorsivamente l'intero DAG a partire da un oggetto radice, così da poter accedere a tutti gli oggetti della scena, per poterli manipolare o per accedere alle strutture interne (nel plugin è necessario, ad esempio, accedere alla mesh per ottenere le coordinate dei punti interni, e traversare il DAG è il modo più immediato per farlo):

```
1  void traverseDag(const AlObject* root) {
2      if (!root) return;
3
4      // Ogni AlObject va "copiato" per poter essere manipolato
5      AlObject* current = root->copyWrapper();
6
7      while (current) {
8          // Qui è possibile elaborare il nodo corrente
9          // (ad esempio stampare informazioni sul nodo)
10
11         AlDagNode* dagNode = current->asDagNodePtr();
12         if (dagNode != NULL) {
13
```

```
14              // Se il nodo appartiene al DAG, controllo se è un
                ↪  gruppo
15              // (se lo è, il safe cast ritorna il puntatore,
                ↪  altrimenti NULL)
16              AlGroupNode* groupNode = current->asGroupNodePtr()
17              if (groupNode != NULL) {
18
19                  // Ricorsione sui figli
20                  AlObject *child = groupNode->childNode();
21                  traverseDag(child);
22                  delete child;
23              }
24
25              // Avanza al prossimo fratello
26              AlObject* next = dagNode->nextNode();
27              delete current;
28              current = next;
29          }
30      }
31  }
```

3.1.2 Struttura delle classi principali

Come detto, l'SDK è strutturato ad oggetti, in una gerarchia di classi che riflette la struttura della scena. Di seguito si presenta una panoramica delle classi principali effettivamente utilizzate nello sviluppo del plugin.

- **AlObject**: Classe base da cui derivano la maggior parte degli oggetti manipolabili. Fornisce metodi comuni ed ereditabili, per effettuare il safe-casting verso classi derivate, per ottenere informazioni generiche e per gestire i wrapper (es. `copyWrapper()`);
- **AlDagNode**: Classe rappresentante un nodo generico nel DAG, non ancora specializzato, superclasse di tutti i tipi di nodo. Ha metodi generici per la manipolazione della struttura dati;

- **AlCurveNode**, **AlSurfaceNode**, **AlMeshNode** etc: Derivate da `AlDagNode`, sono ognuna specializzata per un diverso tipo di entità geometrica. Forniscono accesso all'oggetto vero e proprio a cui il nodo fa riferimento (tramite `curve()`, `surface()`, `mesh()`, etc);

- **AlCurve**, **AlSurface**, **AlMesh**, **AlShader**, **AlTexture** etc: Queste classi rappresentano gli oggetti veri e propri: curve, superfici, mesh etc. Espongono metodi per accedere ai punti di controllo, per campionare punti intermedi, per calcolare bounding box o per esportare i dati in forma numerica. Per esempio, un oggetto di tipo `AlMesh` permette l'accesso diretto ai vettori rappresentanti i vertici;

- **AlUniverse**: Classe fuori dalla gerarchia delle altre, offre un contesto globale della scena attiva. Tramite questa classe è possibile accedere al nodo iniziale del DAG (`firstDagNode()`) e ad altre risorse globali dell'ambiente.

La separazione tra nodi e oggetti permette di avere una struttura ordinata: i nodi gestiscono la gerarchia e le trasformazioni; gli oggetti contengono i dati geometrici veri e propri. Tramite i nodi si naviga la scena, tramite gli oggetti si accede ai contenuti utili e alle geometrie.

Le classi elencate non sono esaustive, ma rendono chiara l'organizzazione generale dell'SDK. Per quelle non listate, i nomi sono autoesplicativi.

3.1.3 Gestione della memoria

La gestione della memoria è un aspetto critico nello sviluppo dei programmi in generale. Ancora di più quando si tratta di sviluppare un plugin, in quanto un errore si riflette anche su tutto l'ambiente del programma principale. Nel caso di Alias, la scelta di design effettuata da parte dei programmatori delle API è incentrata sugli oggetti *wrapper*. I puntatori si possono dividere principalmente in:

- puntatore ad oggetto (*AlMesh**, *AlCurve**, *AlSurface** ...);

- puntatore a un nodo (*AlMeshNode**, *AlCurveNode**, *AlSurfaceNode*...*).

Nel primo caso, quello che viene esposto in realtà è il puntatore a un Wrapper. Quest'ultimo fornisce accesso indiretto ai dati, astraendone la rappresentazione interna, e in questo caso è il programmatore a detenere la responsabilità della liberazione della memoria quando l'oggetto diventa non più necessario. Metodi come `AlObject::copyWrapper()` restituiscono un nuovo wrapper che fa riferimento agli stessi dati Alias sottostanti. Non vengono creati nuovi dati Alias, ma viene allocato un nuovo oggetto wrapper C++, che deve poi essere deallocato manualmente.

Nel secondo caso, al contrario, non viene *allocato* nulla, restituisce solamente il puntatore a un nodo già esistente che, essendo parte della struttura dati di Alias, è sotto totale responsabilità di quest'ultimo. Ugualmente, gli operatori di safe casting (come per esempio `AlObject::asDagNodePtr()`) sono puntatori ad oggetti esistenti e non devono essere deallocati manualmente.

3.2 Formati dei file

In questa sezione vengono brevemente presentati tre formati di file usati nel plugin, tutti compatibili con Alias per l'importazione e l'esportazione. Sebbene nella Learning Edition non sia concessa l'esportazione in questi formati, avendo accesso diretto alle geometrie interne è possibile aggirare il limite scrivendosi in proprio un plugin con dei metodi che espletino esattamente questa funzione.

3.2.1 STEP/STP

Il formato STEP, acronimo di *Standard for the Exchange of Product model data*, con estensioni `.step` o `.stp`, è uno standard ISO (ISO 10303) pensato per la memorizzazione di modelli CAD in 3D. Principalmente,

supporta geometrie solide, curve e superfici. Non ha supporto per le mesh.

Ha una struttura di tipo testuale, non binaria. Rimando alla definizione dello standard per le specifiche tecniche complete[4]. Tralasciando gli header, la sezione contenente le informazioni sulla geometria ha una struttura simile a questa:

```
#103= EDGE_CURVE('',#33,#36,#110,.T.);
#110= LINE('',#104,#111);
#111= VECTOR('',#112,1.);
#112= DIRECTION('',(0.,1.,0.));
```

Alias consente di importare file STEP e convertire le entità in superfici NURBS modificabili. Il plugin sviluppato non manipola direttamente questo tipo di file, ma è stato ampiamente usato in fase di testing, per importare nella scena geometrie già esistenti, ed è il formato di scambio delle informazioni sulla geometria con il tool esterno che si occupa del morphing.

3.2.2 OBJ

Il formato OBJ è uno standard, anch'esso testuale, per la rappresentazione di mesh poligonali. Ha estensione `.obj`. Il formato presenta le informazioni di vertici (tripletta di coordinate), facce/triangoli (tripletta di indici relativi ai vertici) e, facoltativamente, coordinate UV (coppia di coordinate) e versori delle normali (tripletta di coordinate). È molto semplice: vengono rappresentati per primi i vertici, uno per riga, identificati da una 'v' e dalle tre coordinate dimensionali; seguiti dal resto delle informazioni, rappresentato con la stessa struttura.

Di seguito, un esempio di file obj completo. Nello specifico un Tetraedro.

[4] `https://webstore.ansi.org/standards/iso/ISO10303212016`

```
v 1.0 1.0 1.0
v -1.0 -1.0 1.0
v -1.0 1.0 -1.0
v 1.0 -1.0 -1.0
f 1 2 3
f 1 2 4
f 1 3 4
f 2 3 4
```

Nel plugin sono direttamente manipolati con importazione, generazione ed esportazione. Sono parte integrante di una classe custom che rappresenta le mesh, contenente in realtà anche altre informazioni, come verrà illustrato più avanti.

3.2.3 STL

Il formato STL, acronimo di *Stereolithography* rappresenta oggetti tridimensionali come una collezione di triangoli, ciascuno definito da una normale e tre vertici. Di conseguenza è ottimo per la memorizzazione di mesh triangolari.

Il formato può essere sia testuale che binario. Di seguito un esempio in versione ASCII:

```
facet normal 0 0 1
  outer loop
    vertex 1 0 0
    vertex 0 1 0
    vertex 0 0 0
  endloop
endfacet
```

Nel plugin è fondamentale, in quanto la mesh a cui vengono associate le informazioni fluidoninamiche è memorizzata e importata in questo formato.

3.3 Struttura del Plugin

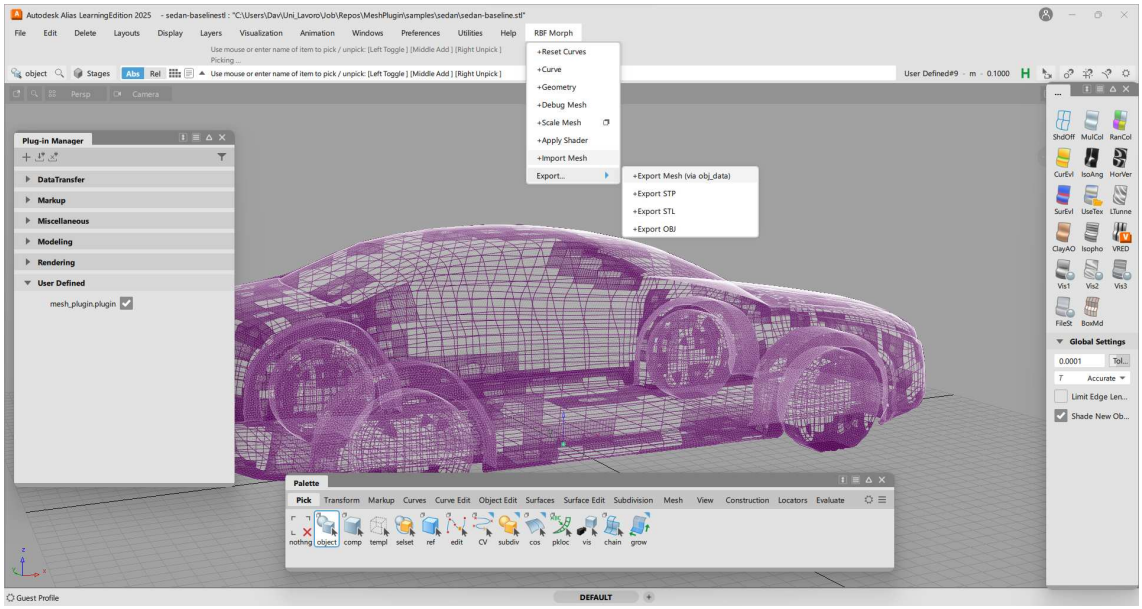


Figura 3.3: Interfaccia del plugin, con il menù a tendina

Prima di entrare nei dettagli, si riportano rapidamente i punti salienti delle funzionalità richieste, già introdotte nel capitolo 1:

1. acquisire inizialmente uno snapshot della geometria iniziale;
2. acquisirne un'altro dopo aver effettuato le modifiche;
3. propagare le modifiche alla mesh;
4. calcolare la variazione dei parametri aerodinamici di interesse.

3.3.1 Step precedenti

Lo sviluppo del progetto *MeshPlugin* è stato interamente svolto dall'autore di questa tesi e si è evoluto in più fasi. L'organizzazione attuale del codice, basata su una struttura modulare a classi e oggetti, è il risultato di un progressivo raffinamento. Nella fase iniziale, invece, il codice era monolitico e scritto in stile `C`: un unico file contenente tutte le funzioni, senza astrazioni né separazione delle responsabilità.

Questa impostazione, sebbene poco scalabile, era funzionale allo scopo della prima fase del lavoro: esplorare l'SDK di Alias e validare la fattibilità tecnica del progetto. Circa due mesi sono stati dedicati al setup dell'ambiente di sviluppo, alla comprensione di quanto esposto nel capitolo 2 e dei meccanismi di compilazione e linking, all'uso del debugger e allo studio della struttura interna del software (in particolare il DAG, *Directed Acyclic Graph*, e la gestione della memoria).

Il primo obiettivo implementativo era dimostrare che fosse possibile interagire con le mesh all'interno di Alias. A tal fine è stato sviluppato un prototipo in grado di:

- importare una mesh da file esterno;
- esportarla nuovamente in formato personalizzato;
- tentare la visualizzazione dei risultati di una simulazione CFD direttamente sulla mesh, sotto forma di mappa a colori (si usa un gradiente di colore per avere un riscontro visivo di quanto ogni vertice della mesh sia influente nella variazione del coefficiente fluidodinamico).

Quest'ultimo punto ha evidenziato subito una limitazione importante: l'SDK non espone alcun meccanismo per associare colori direttamente ai vertici (meccanismo chiamato *per-vertex shading*). Di conseguenza, non era possibile implementare la colorazione desiderata senza ricorrere a soluzioni indirette, tema che verrà approfondito nel capitolo 5.

Una volta verificata la possibilità di gestire le mesh, è stato affrontato il secondo passo: la validazione della capacità di deformare le coordinate dei nodi della mesh. Come test preliminare sono stati introdotti dei controlli grafici interattivi (*slider*), ciascuno associato a un fattore di scala indipendente lungo gli assi x, y e z. In questo modo l'utente poteva ingrandire o ridurre la mesh lungo una direzione specifica, dimostrando la fattibilità dell'aggiornamento in tempo reale della geometria.

Rimaneva infine da verificare la possibilità di navigare e interrogare le entità geometriche CAD interne (curve e superfici NURBS) tramite le

API. Solo dopo aver validato anche questa funzionalità è stato possibile procedere con un refactoring completo: il codice è stato riorganizzato in moduli coerenti, con l'introduzione di classi e oggetti, e sono state mantenute esclusivamente le componenti necessarie allo sviluppo del plugin finale. Questa è la base dell'architettura attuale, descritta nelle sezioni successive.

3.3.2 Organizzazione della directory

Alla radice del progetto si trovano:

1. i file di setup, per esempio (`MeshPlugin.vcxproj`, `MeshPlugin.sln`, `Makefile`);
2. il file `README.md`, utilizzato come diario di lavoro per annotare i vari passaggi;
3. il file `.gitignore`;
4. le cartelle `src/`, `include/`, `external/`, `doc/`, `samples/`, `obj/`, `x64/` e `debug/`.

I sorgenti sono suddivisi in `.cpp` (nella cartella `src/`) e `.h` o `.hpp` (in `include/`). La cartella `external/` ospita librerie di terze parti utilizzate, come `Eigen` [7] per l'algebra lineare e `nanoflann` [8] per la costruzione di k-d tree[5]. Le altre cartelle contengono documentazione, file di esempio (geometrie CAD, mesh, risultati CFD, ecc.), file compilati, output di debug e binari eseguibili.

La logica del plugin è suddivisa in componenti principali:

- **Logging**: modulo dedicato alla registrazione delle attività e al debugging;

[5]Struttura dati descritta nella sezione 3.3.7 usata per l'associazione tra i vertici della mesh e quelli della matrice di sensibilità

- **Inizializzazione e distruzione**: gestione del ciclo di vita del plugin, con registrazione e deregistrazione dei comandi in Alias;
- **Classe Obj_Data**: responsabile della gestione delle mesh, con metodi per leggere e scrivere file OBJ e per mantenere una rappresentazione interna arricchita;
- **Metodi per il morphing**: gestione delle deformazioni geometriche tramite curve e superfici NURBS, propagate ai nodi della mesh mediante RBF;
- **Sensitivity class**: implementazione e gestione della matrice di sensibilità;
- **UV mapping e texture**: modulo dedicato all'associazione di coordinate UV e alla preparazione della visualizzazione dei risultati CFD, approfondito nelle espansioni future (capitolo 5).

Un chiarimento è necessario riguardo al termine *UV mapping*. Nel contesto della grafica 3D, con UV mapping si intende il processo di associare a ogni vertice di una mesh tridimensionale una coppia di coordinate bidimensionali (u, v). Queste coordinate fungono da "indirizzo" in uno spazio 2D, tipicamente quello di un'immagine, e permettono di applicare texture o campi scalari sulla superficie. Nel caso di questo plugin, tale meccanismo viene sfruttato non per scopi estetici, ma per proiettare e visualizzare i risultati CFD (ad esempio distribuzioni di pressione) direttamente sulla geometria.

Questa struttura modulare consente di isolare le diverse responsabilità: ad esempio, le funzioni di logging sono completamente indipendenti dal morphing, mentre la classe di sensibilità si appoggia a `Eigen` e `nanoflann` senza dipendere dai dettagli del caricamento della mesh. Il risultato è un'architettura flessibile, in cui ogni componente è chiaramente documentato e può essere riutilizzato o esteso in futuro.

3.3.3 Logging

Il modulo di *logging* ha lo scopo di registrare le operazioni svolte dal plugin e di fornire uno strumento di tracciamento per errori e messaggi diagnostici.

Si è trattato, inoltre, di uno dei primi problemi affrontati nello sviluppo del progetto. Essendo il plugin integrato in Alias, non era possibile utilizzare lo standard output della console (`std::cout`) e il prompt interno di Alias non risultava adeguato: in caso di errori gravi del software stesso, non era garantito alcun canale affidabile per la scrittura di messaggi. La soluzione scelta è stata quindi quella di sostituire qualsiasi istruzione di `print` o messaggio a schermo — sia di debug, sia informativo, sia legato a eccezioni — con una scrittura diretta su file di log. L'implementazione si trova nei file `include/logging.h` e `src/logging.cpp`. Il cuore del sistema è la funzione `logMessage`, che riceve in ingresso:

- il livello di log (`INFO`, `WARNING`, `ERROR`, `DEBUG`);
- il nome della funzione chiamante, ottenuto tramite macro;
- il messaggio testuale da registrare;
- il file sorgente e la linea da cui proviene la chiamata.

Ogni messaggio viene arricchito con un timestamp leggibile, e scritto su file tramite uno stream globale `std::ofstream`. Ciascuna riga del log contiene quindi:

1. l'istante temporale in cui è avvenuta l'operazione;
2. il livello del messaggio;
3. la funzione, il file e la linea sorgente;
4. il contenuto testuale fornito dal programmatore.

Per semplificare l'utilizzo del logger, sono state definite alcune macro che incapsulano la chiamata a `logMessage`:

- `LOG_INFO(msg)`;
- `LOG_WARN(msg)`;
- `LOG_ERROR(msg)`;

- `LOG_DEBUG(msg)`.

Queste macro consentono di inserire messaggi di log in modo immediato e leggibile, senza dover specificare manualmente file, linea e funzione. La macro `LOG_DEBUG`, in particolare, è attiva solo se la costante `DEBUG_ACTIVE` è definita: questo permette di escludere i messaggi di debug in compilazioni rilasciate all'utente finale, mantenendoli invece durante lo sviluppo.

Il risultato è un file `output.log` che, durante l'esecuzione del plugin, raccoglie informazioni come l'esempio seguente:

```
1 [2025-08-13 20:17:57] [DEBUG] [Surface_Points::Surface_Points]
  ↪ (\include\curve_points.hpp:298) index: 18. Is this point
  ↪ valid? true. real u, v: (1.178097, 0.000000)
2 [2025-08-13 20:17:57] [ERROR] [recursive_curve_explorer]
  ↪ (\src\plugin_methods.cpp:796) SurfaceException thrown.
  ↪ ErrorCode: AliasSide. Msg: Error while retrieving the word
  ↪ coordinates of the surface point at position (9, 0).
  ↪ StatusCode: 9
```

Questa infrastruttura consente a ogni funzione critica del plugin di lasciare tracce della propria attività e, in caso di crash o comportamento inatteso, il log diventa uno strumento essenziale per analizzare passo passo le operazioni svolte e i dati elaborati.

3.3.4 Inizializzazione e distruzione

La fase di inizializzazione del plugin è cruciale: è in questo momento che Alias "scopre" le funzionalità aggiuntive messe a disposizione, carica i menu e associa ciascun comando alla relativa callback. In maniera simmetrica, la distruzione deve garantire che tutte le risorse siano correttamente rilasciate, evitando memory leak o crash di Alias dopo la rimozione del plugin. Queste operazioni sono interamente contenute in `src/main.cpp`, che come ci si può aspettare è il "contenitore" della struttura del plugin, l'inizializzatore delle varie funzionalità.

Helper per la creazione di comandi Per ridurre la ridondanza è stata scritta una funzione di supporto, `create_function_helper`, che si occupa della creazione sia della `AlMomentaryFunction` (il vero e proprio comando eseguibile), sia del relativo `AlFunctionHandle` (il puntatore che permette di aggiungerlo a menu o sub-menu di Alias):

```
statuscode s; // variabile globale
void create_function_helper(AlMomentaryFunction& func,
                            AlFunctionHandle& handle,
                            const std::string& funcName,
                            const std::string& handleName,
                            void (*callback)()) {
    s = func.create(funcName.c_str(), callback);
    LOG_INFO(funcName + " function creation status: "
             + std::to_string(s));
    s = handle.create(handleName.c_str(), &func);
    LOG_INFO(handleName + " handle creation status: "
             + std::to_string(s));
}
```

Questo consente di ridurre l'inizializzazione di un nuovo comando a una singola riga, invece di ripetere la stessa sequenza di istruzioni per ciascuna funzione.

Funzione `plugin_init` Il punto di ingresso è rappresentato dalla funzione `plugin_init`, chiamata automaticamente da Alias al momento del caricamento del plugin. Qui vengono eseguite le seguenti operazioni principali:

1. apertura del file di log e stampa delle prime informazioni diagnostiche;
2. controllo di licenza per la libreria `RBF Morph`;
3. inizializzazione di `AlUniverse`, l'ambiente interno di Alias;

4. dichiarazione delle funzioni e associazione alle callback del plugin (es. `import_mesh`, `scale_mesh`, `curve_state`);

5. creazione dell'editor per gli *slider* di deformazione, tramite la classe `AlEditor`;

6. creazione del menu principale `RBF Morph` e del relativo submenu `Export...`.

Un estratto della funzione è riportato qui di seguito:

```
1  extern "C" PLUGINAPI_DECL int plugin_init(const char* dirName) {
2      // apertura file di log
3      if (!outFile.is_open()) {
4          AlPrintf(kPrompt, "Error opening log file."); // prompt
           ↪  direttamente nella GUI di Alias
5          return 1;
6      }
7      LOG_INFO("outfile opened with success ");
8
9      // inizializzazione universo
10     if (AlUniverse::isInitialized()) {
11         LOG_WARN("Universe already initialized. skipping.");
12     } else {
13         s = AlUniverse::initialize();
14         if (s != sSuccess) {
15             outFile.close();
16             return 1;
17         }
18     }
19
20     // creazione di funzioni e handle
21     create_function_helper(importMeshFunc, importMeshHandle,
22                            "import_mesh", "Import Mesh",
23                            importMeshIntoAlias);
24
25     create_function_helper(scaleMeshFunc, scaleMeshHandle,
```

```
26                              "scale_mesh", "Scale Mesh",
27                              scale_mesh);
28
29      // creazione editor slider
30      sliderEditor = new AlEditor("Scale Mesh Slider Editor",
31                                  "scale_mesh");
32      sliderEditor->addDoubleSlider("lambda_h", 1.0, 0.8, 1.2,
33                                    lambda_h_slider_callback);
34      sliderEditor->addButton("Morph!", scale_mesh, false);
35      sliderEditor->create();
36
37      // creazione menu principale
38      rbf_menu = new AlMainMenu();
39      rbf_menu->create("RBF Morph");
40
41      // associazione al menu
42      importMeshHandle.appendToMenu("RBF Morph");
43      scaleMeshHandle.appendToMenu("RBF Morph");
44
45      AlPrintf(kPrompt, "RBF-Morph plug-in installed");
46      return 0;
47  }
```

Grazie a questo meccanismo, Alias presenta all'utente un nuovo menu `RBF Morph` con voci che richiamano direttamente le callback del plugin, come si può osservare in figura 3.3.

Funzione `plugin_exit` In maniera speculare, al momento della disinstallazione del plugin Alias richiama la funzione `plugin_exit`, che rimuove menu, sottomenu, handle e funzioni, dealloca le risorse dinamiche (es. editor e oggetti grafici), e chiude il file di log:

```
1   extern "C" PLUGINAPI_DECL int plugin_exit(void) {
2       LOG_INFO("exiting the plugin... ");
3
```

```
4       // rimozione editor
5       delete sliderEditor;
6       sliderEditor = nullptr;
7
8       // rimozione handle dai menu
9       importMeshHandle.removeFromMenu();
10      scaleMeshHandle.removeFromMenu();
11
12      // distruzione oggetti
13      importMeshHandle.deleteObject();
14      importMeshFunc.deleteObject();
15      scaleMeshHandle.deleteObject();
16      ScaleMeshFunc.deleteObject();
17
18      // rimozione menu
19      rbf_menu->remove();
20      delete rbf_menu; rbf_menu = nullptr;
21
22      outFile.close();
23      AlPrintf(kPrompt, "RBF-Morph plugin removed");
24      return 0;
25  }
```

Questa distinzione tra fase di inizializzazione e fase di distruzione garantisce che Alias possa caricare e scaricare il plugin più volte nella stessa sessione senza problemi. Inizialmente si erano verificati crash dovuti alla distruzione prematura di oggetti ancora referenziati: per questo, la sequenza di `deleteObject()` e `removeFromMenu()` è stata curata con particolare attenzione.

Queste due funzioni quindi definiscono il ciclo di vita del plugin all'interno di Alias.

3.3.5 Importazione ed esportazione delle mesh

Uno dei primi obiettivi del progetto è stato implementare un meccanismo robusto per importare ed esportare mesh, in modo da avere un formato intermedio che permettesse di:

- leggere una geometria esistente dalla scena Alias;
- salvare su file la mesh in un formato standard e facilmente modificabile;
- ricaricare la stessa mesh in Alias dopo eventuali modifiche o elaborazioni.

Per questo è stata implementata la classe `Obj_Data`, che si occupa della lettura e scrittura di file in formato OBJ. Si è scelto OBJ perché è un formato testuale semplice, ampiamente supportato, e adatto a descrivere mesh triangolari o poligonali senza perdita di informazione.

La classe `Obj_Data` Si occupa di mantenere in memoria:

- i vertici della mesh;
- le normali (versori perpendicolari alle superfici locali, utilizzati per calcolare l'orientamento e l'illuminazione della geometria);
- le facce (insiemi di indici che definiscono i triangoli);
- le **coordinate texture** (u, v), che associano a ciascun vertice una posizione in uno spazio bidimensionale per permettere il cosiddetto UV mapping;
- i colori associati a ogni vertice (estensione rispetto al formato standard).

Oltre a questi dati "geometrici", la classe fornisce anche funzioni per:

- caricare una mesh, tramite il parsing di file OBJ esterno;
- scrivere su disco una mesh costruita in Alias;
- convertire la rappresentazione interna di Alias in un formato lineare, adatto a successive elaborazioni (es. applicazione di RBF).

Integrazione con Alias Una volta definita la classe `Obj_Data`, è stato possibile integrarla con Alias scrivendo due funzioni di callback:

- `importMeshIntoAlias`: legge un file OBJ e costruisce la mesh nella scena Alias;
- `exportMeshFromAlias`: prende una mesh presente in Alias e la scrive su file tramite `Obj_Data`.

Entrambe le funzioni sono registrate al momento dell'inizializzazione del plugin (sezione 3.3.4), e compaiono quindi come voci di menu sotto la voce principale `RBF Morph`. Per esempio, l'importazione è gestita da due funzioni concettualmente simili alle seguenti:

```
1  void fromOBJtoAlMesh(const Obj_Data& obj, AlMesh* mesh) {
2
3          // Extract obj_vertices and obj_faces
4          const auto& obj_vertices = obj.AccessVertices();
5          const auto& obj_faces = obj.AccessFaces();
6
7          s = mesh->create(
8                  obj_vertices.size(),
9                  obj_vertices.data(),
10                 obj_faces.size(),
11                 obj_faces.data(),
12         );
13 }
14
15 void importMeshIntoAlias(string path) {
16     Obj_Data mesh_obj;
17     AlMesh* mesh = new AlMesh();
18     mesh_obj.importFromFile(path))
19     fromOBJtoAlMesh(mesh_obj, mesh);
20 }
```

Curiosamente, questa funzione si è rivelata una delle più problematiche da implementare. La classe `AlMeshNode`, che rappresenta nel DAG il col-

legamento tra una mesh e la scena, non espone un costruttore pubblico, a differenza della maggior parte delle altre classi `AlNode`. In un primo momento è stato quindi necessario adottare un workaround, duplicando il nodo di una mesh già presente e sostituendone i dati geometrici. Solo in seguito è emersa un'eccezione dell'SDK: a differenza di altri oggetti, le mesh non richiedono un'inserzione esplicita nel DAG. La creazione di un oggetto `AlMesh` genera infatti automaticamente il relativo nodo `AlMeshNode` e lo collega al grafo della scena, senza interventi manuali.

Questa parte ha avuto una doppia utilità: da un lato testare le capacità di Alias di interagire con file esterni, dall'altro definire un "ponte" di scambio con il resto delle librerie. Grazie a questo modulo, è possibile sia manipolare una mesh in Alias, sia potenzialmente salvarla su disco, processarla con metodi esterni e reimportarla nella scena. Si tratta quindi del mattone fondamentale per tutte le operazioni successive di morphing e valutazione CFD.

3.3.6 Morphing

Per ragioni di riservatezza non è possibile entrare nei dettagli implementativi né presentare frammenti di codice relativi al morphing. L'intero processo viene quindi trattato come una *black box*. Il morphing rappresenta tuttavia il nucleo del lavoro svolto, poiché consente di trasferire in maniera consistente le modifiche locali applicate dall'utente all'intera mesh di simulazione.

Lo strumento utilizzato si basa su *Radial Basis Functions* (RBF), che propagano gli spostamenti geometrici imposti in un sottoinsieme di vincoli a tutti i nodi della mesh. In questo modo, la deformazione risulta continua e coerente, senza introdurre distorsioni locali.

Nel flusso implementativo, il software acquisisce come input la mesh

e la B-rep[6] nelle condizioni iniziali. Una classe dedicata mantiene uno *snapshot* della geometria originale alla prima esecuzione. Ad ogni successiva modifica essa viene fornita come input al sistema di morphing, uno strumento esterno proprietario ottimizzato per gestire mesh con milioni di nodi, insieme alla B-rep nella sua configurazione aggiornata e alla mesh originale. Il sistema propaga la trasformazione applicata alla B-rep alla mesh iniziale, garantendo che al termine del processo la nuova mesh risulti perfettamente sovrapposta alla B-rep aggiornata. Il risultato è una nuova mesh che conserva la stessa topologia della precedente, ma con i vertici riposizionati secondo il campo di deformazione.

Poiché i vertici della mesh sono associati a quelli della matrice di sensibilità, le modifiche geometriche non si limitano all'aspetto visibile, ma si riflettono direttamente anche sulle quantità aerodinamiche di interesse, rendendo possibile un aggiornamento in tempo reale delle prestazioni aerodinamiche associate al modello.

3.3.7 Sensitivity Class

La `Sensitivity` è la classe che realizza il collegamento tra le deformazioni della geometria e la valutazione dei coefficienti aerodinamici, chiamati in questo frangente *funzionali*, come ad esempio il drag o l'efficienza. Il punto di partenza è la *matrice di sensibilità* introdotta nella sezione 2.3.4: per ogni vertice della superficie essa contiene le derivate parziali del coefficiente rispetto agli spostamenti nelle tre direzioni. Grazie a questa informazione è possibile stimare in maniera rapida la variazione del funzionale a partire dagli spostamenti geometrici prodotti dal morphing.

La classe si occupa di tre compiti fondamentali:

- importare da file CSV i dati di sensibilità;

[6]Boundary Representation, è il modello specifico utilizzato in questo contesto per descrivere la geometria attraverso le superfici NURBS di confine

- riallineare le sensibilità all'ordinamento dei vertici della mesh corrente;
- calcolare il valore integrato della variazione del coefficiente fluidodinamico.

```
class Sensitivity {
public:
        struct PointCloud {
                std::vector<Eigen::Vector3d> points;

                inline size_t kdtree_get_point_count() const {
                ↪  return points.size(); }

                inline double kdtree_get_pt(const size_t idx,
                ↪  const size_t dim) const {
                        return points[idx][dim];
                }
                template <class BBOX>
                bool kdtree_get_bbox(BBOX&) const { return
                ↪  false; }
        };
        struct PointCloudAdaptor {
                const Sensitivity::PointCloud& cloud;

                PointCloudAdaptor(const Sensitivity::PointCloud&
                ↪  cloud_) : cloud(cloud_) {}

                inline size_t kdtree_get_point_count() const {
                ↪  return cloud.points.size(); }

                inline double kdtree_get_pt(const size_t idx,
                ↪  const size_t dim) const {
                        return cloud.points[idx][dim];
                }

```

```
25                  template <class BBOX>
26                  bool kdtree_get_bbox(BBOX&) const { return
                     ↪  false; }
27          };
28          double calculateIntegratedValue(
29                  const std::vector<float>& original_vertices,
30                  const std::vector<VECTOR(float, 3)>&
                     ↪  morphed_vertices
31          ) const;
32          Sensitivity(const std::string& filename);
33          ~Sensitivity() = default;
34          Eigen::MatrixXd getSensitivityMatrix() const;
35          Eigen::MatrixXd getOrderedSensitivityMatrix()
             ↪  const;
36          Eigen::Vector3d getVertexSensitivity(int vertex_index)
             ↪  const;
37          Eigen::Vector3d getOriginalVertex(int vertex_index)
             ↪  const;
38          void reorderMatrixByMesh(const std::vector<float>&
             ↪  mesh_vertices);
39          double getFunctionalValue(int vertex_index) const;
40  private:
41          Eigen::MatrixXd m_original_sensitivity_matrix;
42          Eigen::MatrixXd m_ordered_sensitivity_matrix;
43          std::shared_ptr<std::map<int, Eigen::Vector3d>>
             ↪  mesh_vertices;
44          std::ifstream m_file;
45          std::string m_filename;
46  };
```

Il file CSV utilizzato ha 6 (in alcuni casi 7) colonne: le prime tre rappresentano le coordinate originali del vertice, le tre successive il gradiente di sensibilità $\frac{\partial J}{\partial x}, \frac{\partial J}{\partial y}, \frac{\partial J}{\partial z}$. La settima colonna, quando presente, contiene il valore funzionale locale associato al vertice. Quest'ultimo

tuttavia non viene utilizzato, poiché per ogni vertice il contributo viene ricalcolato come

$$\delta J_i = \mathbf{s}_i^\top \Delta \mathbf{v}_i,$$

dove $\mathbf{s}_i$ è il vettore delle tre derivate parziali e $\Delta \mathbf{v}_i$ lo spostamento del vertice. Il valore funzionale locale fornito dal solver risulta dunque ridondante, e si può ignorare, in quanto può sempre essere ottenuto a partire dal gradiente e dagli spostamenti.

K-d tree per il riallineamento delle sensibilità

Come discusso nella Sezione 2.3.4, la matrice di sensibilità è fornita dal solver CFD sotto forma di tabella: ad ogni vertice della mesh originale (coordinate (x, y, z)) è associato un vettore di sensibilità. Tuttavia, l'ordinamento dei vertici nella mesh di Alias non coincide necessariamente con quello del file CSV. Se i dati fossero semplicemente accoppiati per indice, la corrispondenza risulterebbe errata: a un vertice della mesh verrebbero attribuite le sensibilità di un punto geometrico completamente diverso.

È quindi necessario associare correttamente, ad ogni vertice della mesh corrente, il vettore di sensibilità del vertice corrispondente nella matrice. Il problema si può trattare come una ricerca del punto più vicino: per ogni vertice della mesh in Alias, bisogna trovare quale riga del CSV corrisponde alla sua posizione originale. È per questo motivo che nella rappresentazione dei dati nel file csv per ogni vertice i primi 3 valori sono le coordinate spaziali: servono a trovare la corrispondenza coi vertici nella mesh

Il k-d tree. Per risolvere efficientemente questo problema si utilizza una struttura dati detta **k-d tree** (*k-dimensional tree*), introdotta negli anni '70 da Jon Bentley [9]. Si tratta di un albero binario di partizionamento dello spazio progettato per organizzare un insieme di punti in uno

spazio a k dimensioni (nel nostro caso, $k = 3$), ottimizzando la ricerca dei punti più vicini a una data query.

La costruzione di un k-d tree avviene tramite una suddivisione ricorsiva dello spazio euclideo, partendo dall'insieme di tutti i punti. Ad ogni passo, lo spazio (e il corrispondente insieme di punti) viene partizionato da un iperpiano allineato a uno degli assi cartesiani. Il criterio di suddivisione segue due regole fondamentali:

1. L'asse di suddivisione viene scelto ciclicamente. Al livello 0 (la radice) si usa l'asse x, al livello 1 l'asse y, al livello 2 l'asse z, per poi ricominciare con l'asse x al livello 3, e così via, secondo la regola $a = p \mod k$ (dove a è l'asse e p la profondità);

2. Una volta selezionato un asse, l'insieme di punti viene partizionato scegliendo la mediana delle coordinate dei punti lungo quell'asse. Il punto corrispondente alla mediana diventa il nodo corrente dell'albero. Tutti i punti con una coordinata inferiore (o uguale) sulla dimensione scelta vengono inseriti nel sottoalbero sinistro, mentre gli altri vengono inseriti nel sottoalbero destro.

La costruzione di un k-d tree bilanciato può essere fatta in tempo compreso tra $O(k \cdot n \log n)$ e $O(n \log n)$, a seconda delle scelte implementative. [10, 11].

La ricerca del k-NN di un punto q in un k-d tree (detta *query*) è un processo che inizia dalla radice dell'albero e procede ricorsivamente attraverso i nodi, prendendo decisioni sulla direzione da seguire (sinistra o destra) in base alla distanza dei punti rispetto al punto di query:

1. Si confronta la coordinata del punto di query con quella del nodo corrente lungo la dimensione selezionata. In base al confronto, si decide se proseguire nel sottoalbero a sinistra o a destra;

2. Una volta raggiunto un nodo foglia, quel punto viene assunto come il "miglior candidato" corrente e viene calcolata la sua distanza da q. L'algoritmo risale quindi l'albero (*backtracking*) e per ogni nodo

verifica se il sottoalbero opposto può contenere punti che potrebbero essere più vicini al punto di query rispetto al più lontano dei k-NN attuali. Se la distanza minima possibile tra il piano di divisione e q è inferiore alla distanza dell'attuale k-esimo vicino, si esplora il sottoalbero opposto.

Fissato un albero con n nodi, e considerando la media sul numero di query (cioè su tutti i possibili punti q nello spazio), la ricerca di un singolo vicino più prossimo ha una complessità media di $O(\log n)$, mentre per k vicini più prossimi, in generale, la complessità media è $O(k \log n)$. Nel caso di albero poco bilanciato o in presenza di alta dimensione, la complessità può degenerare al worst-case $O(n)$ [12, 9].

In questo contesto, la dimensionalità dello spazio è molto bassa, la distribuzione dei punti si può considerare bilanciata e la query riguarda la ricerca di un singolo vicino più prossimo (1-NN), facendo quindi ricadere il problema nel caso più efficiente, in cui la query media ha costo computazionale pari a $O(\log n)$.

Implementazione nel plugin. L'algoritmo di riallineamento procede come segue:

1. si legge il file CSV e si costruisce una `PointCloud`, cioè una lista di punti tridimensionali con le loro sensibilità associate;
2. si costruisce su questi punti il k-d tree, usando la libreria `nanoflann`;
3. per ciascun vertice della mesh di Alias si effettua una query sul k-d tree per trovare il vertice più vicino nella `PointCloud`;
4. il vettore di sensibilità della riga corrispondente del CSV viene copiato in una nuova matrice ordinata, che ha quindi la stessa disposizione dei vertici della mesh corrente.

```
1 // Riallineamento con k-d tree
2 void Sensitivity::reorderMatrixByMesh( const std::vector<float>&
  ↪ mesh_vertices) {
3     // Costruzione point cloud
4     PointCloud cloud;
```

```
5       for (int i = 0; i < m_original_sensitivity_matrix.rows();
        ↪  i++) {
6           cloud.points.push_back(getOriginalVertex(i));
7       }
8       PointCloudAdaptor pc_adaptor(cloud);
9
10      // Creazione dell'indice k-d tree
11      nanoflann::KDTreeSingleIndexAdaptor<
12          nanoflann::L2_Simple_Adaptor<double, PointCloudAdaptor>,
13          PointCloudAdaptor, 3
14      > tree(3, pc_adaptor, {32});
15      tree.buildIndex();
16
17      // Per ogni vertice della mesh, trova quello più vicino
18      for (size_t i = 0; i < num_vertices; ++i) {
19          Eigen::Vector3d query_point(...);
20          size_t result_index; double dist2;
21          nanoflann::KNNResultSet<double> resultSet(1);
22          resultSet.init(&result_index, &dist2);
23          tree.findNeighbors(resultSet, query_point.data());
24
25          // Riallineamento
26          m_ordered_sensitivity_matrix.row(i) =
27              getVertexSensitivity(result_index);
28      }
29  }
```

In questo modo, dopo il riallineamento, il i-esimo vertice della mesh e la i-esima riga della matrice di sensibilità corrispondono sempre allo stesso punto geometrico.

Strutture di supporto. Per permettere a `nanoflann` di interagire con i dati senza ricopiarli in un nuovo formato, sono stati definiti:

- una classe `PointCloud`, che memorizza le coordinate (x, y, z) e i vettori di sensibilità letti dal CSV;

- un `PointCloudAdaptor`, che fornisce a `nanoflann` le funzioni per accedere a una coordinata di un punto e per calcolare la distanza euclidea.

In questo modo la libreria può trattare la `PointCloud` come una struttura nativa, senza duplicare i dati in memoria.

Integrazione numerica sulla superficie

Una volta riallineati i dati, è possibile stimare la variazione del **funzionale aerodinamico** di interesse, cioè il coefficiente $f(\mathbf{x})$ calcolato dal solver CFD.

Dal punto di vista teorico, la variazione si scrive come integrale di superficie:

$$\delta J = \int_S \mathbf{s}^\top \Delta\mathbf{v}\, dS,$$

dove S è la superficie della geometria, $\mathbf{s}$ è il vettore di sensibilità e $\Delta\mathbf{v}$ lo spostamento locale della geometria[7].

Nella discretizzazione a vertici, l'integrale è approssimato da una sommatoria. Affinché tale approssimazione sia corretta, è necessario tenere conto del contributo dell'elemento di superficie dS associato a ogni vertice. La modalità con cui questo avviene dipende dalla definizione del vettore di sensibilità. Si possono delineare due strategie:

- Se le sensibilità s_i sono definite per unità di superficie, il contributo di ogni nodo deve essere esplicitamente pesato per la sua area di pertinenza A_i (ad esempio, tramite un vettore area nodale);
- In alternativa, le sensibilità sono fornite come una grandezza finita che incorpora già il contributo dell'area locale. In questo caso, il vettore s_i rappresenta direttamente il contributo sulla superficie di pertinenza del nodo.

[7]Questa forma è una notazione alternativa ma del tutto equivalente a quella con l'integrale doppio esplicito scritta nella sezione 2.3.4

In questo lavoro la sensibilità in input adotta la seconda strategia: lo spostamento del vertice i-esimo è

$$\Delta \mathbf{v}_i = \mathbf{v}_i^{\text{morph}} - \mathbf{v}_i^{\text{orig}},$$

e il contributo puntuale è

$$\delta J_i = \mathbf{s}_i^\top \Delta \mathbf{v}_i.$$

La somma su tutti i vertici approssima l'integrale:

$$\delta J \approx \sum_{i=1}^{N} \delta J_i.$$

L'implementazione concreta di questa somma è così strutturata:

```
1  double Sensitivity::calculateIntegratedValue(
2          const std::vector<float>& original_vertices,
3          const std::vector<VECTOR(float, 3)>& morphed_vertices)
           ↪  const {
4
5          double total_integrated_value = 0.0;
6
7          for (int i = 0; i < m_ordered_sensitivity_matrix.rows();
           ↪  ++i) {
8                  // Get the ordered sensitivity vector for the
                   ↪  current vertex
9                  Eigen::Vector3d sensitivity_vector =
                   ↪  m_ordered_sensitivity_matrix.row(i);
10
11                 // Calculate the displacement (delta) for the
                   ↪  current vertex
12                 const double delta_x = morphed_vertices[i][0] -
                   ↪  original_vertices[(i) * 3 + 0];
13                 const double delta_y = morphed_vertices[i][1] -
                   ↪  original_vertices[(i) * 3 + 1];
```

```
14                  const double delta_z = morphed_vertices[i][2] -
                    ↪  original_vertices[(i) * 3 + 2];
15
16                  // Element-wise multiplication and add to the
                    ↪  total sum
17                  total_integrated_value += (sensitivity_vector[0]
                    ↪  * delta_x);
18                  total_integrated_value += (sensitivity_vector[1]
                    ↪  * delta_y);
19                  total_integrated_value += (sensitivity_vector[2]
                    ↪  * delta_z);
20          }
21
22          return total_integrated_value;
23  }
```

Questo approccio consente di calcolare in tempo reale la variazione del coefficiente aerodinamico, collegando direttamente le modifiche geometriche imposte dall'utente con il loro impatto stimato sulle prestazioni.

Capitolo 4

Esempio di applicazione

Per illustrare il funzionamento del plugin è utile presentare un caso d'uso, tratto da una sessione di prova. Nel caso in analisi, il modello utilizzato è l'ala anteriore di un veicolo da formula 1, di cui si hanno il CAD e la mesh a esso associata, come visibile in figura 4.1

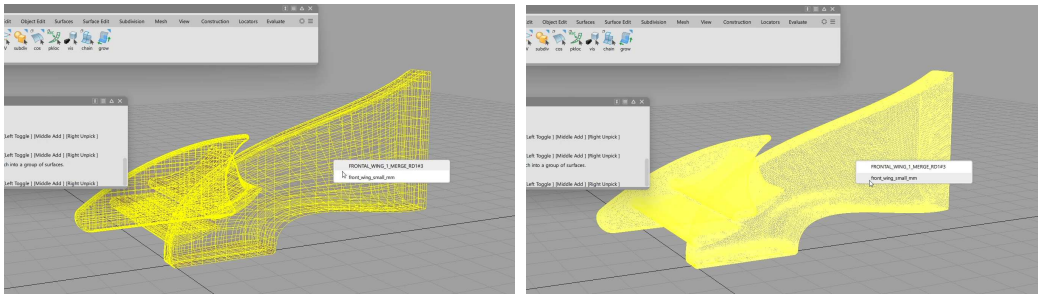


Figura 4.1: CAD e Mesh originali

Nella fase iniziale l'utente seleziona la mesh e avvia il comando *curve* dal menu del plugin. In questo momento viene registrato lo stato di riferimento: sia la geometria CAD che la mesh vengono memorizzate come configurazione iniziale, e ciascun vertice della mesh viene associato al corrispondente vettore di sensibilità letto dal file CFD. Nel prompt di Alias viene confermata l'avvenuta inizializzazione con il messaggio `sensitivity initialized, baseline impact = 0.0` (figura 4.2)

Una volta eseguito lo snapshot della geometria e impostata la baseline, l'utente può intervenire liberamente sulla geometria CAD. In questa demo, vengono alterate le proporzioni e le dimensioni di uno dei due lati dell'ala. La mesh non viene alterata manualmente: essa è già stata salvata nello snapshot iniziale e verrà aggiornata automaticamente dal plugin. Dopo la modifica, il comando *curve* viene richiamato una seconda

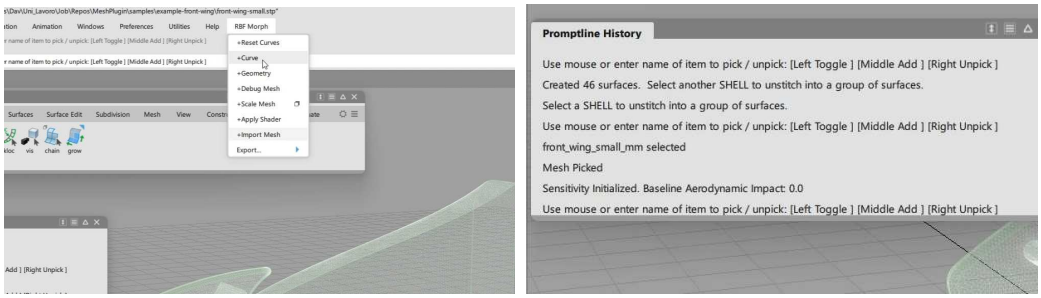


Figura 4.2: Inizializzazione delle geometrie e baseline dell'impatto aerodinamico

volta (immagine 4.3). In questo momento il sistema di morphing basato su RBF elabora la differenza tra la geometria attuale e quella di riferimento, propagandola alla mesh CFD. Il risultato è la generazione di una nuova mesh, che aderisce alla geometria modificata mantenendo la stessa connettività dell'originale, come si può vedere dall'immagine 4.4. Contestualmente, nel prompt compare un aggiornamento dell'impatto aerodinamico calcolato tramite la matrice di sensibilità, ad esempio `updated aerodynamic impact: -0.556`, visibile nell'immagine 4.5.

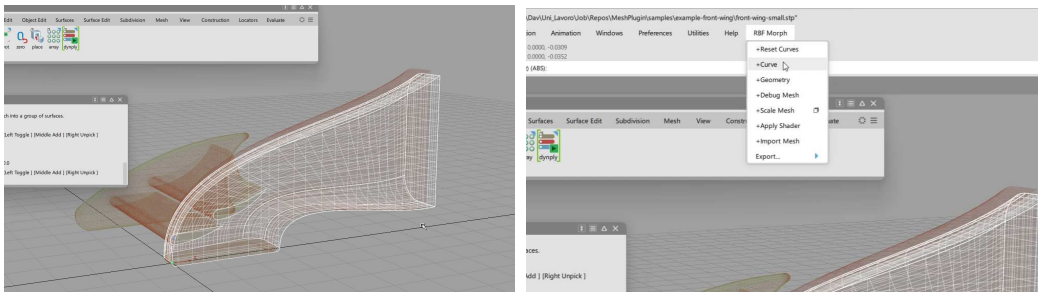


Figura 4.3: La geometria viene modificata e viene nuovamente invocata la funzione `curve`

Il processo può essere ripetuto più volte: a ogni nuova modifica della geometria, il plugin aggiorna la mesh e ricalcola la variazione stimata delle prestazioni. In questo modo diventa possibile esplorare diverse varianti progettuali in rapida sequenza, ottenendo sempre un feedback quantitativo sull'effetto aerodinamico delle modifiche apportate.

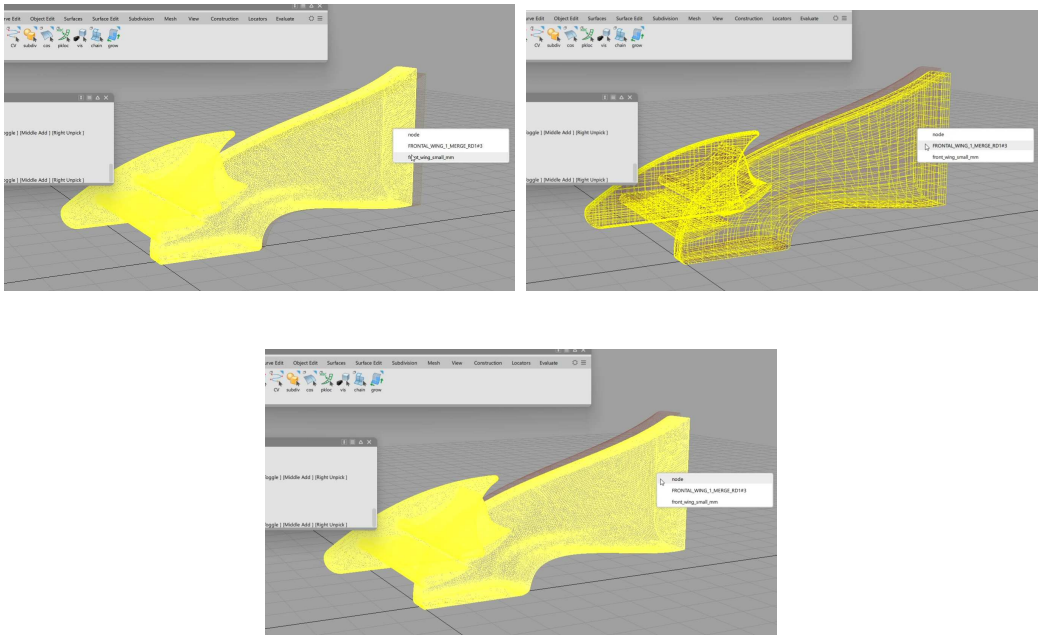


Figura 4.4: Mesh originale, CAD dopo le modifiche e mesh risultante dopo il morphing

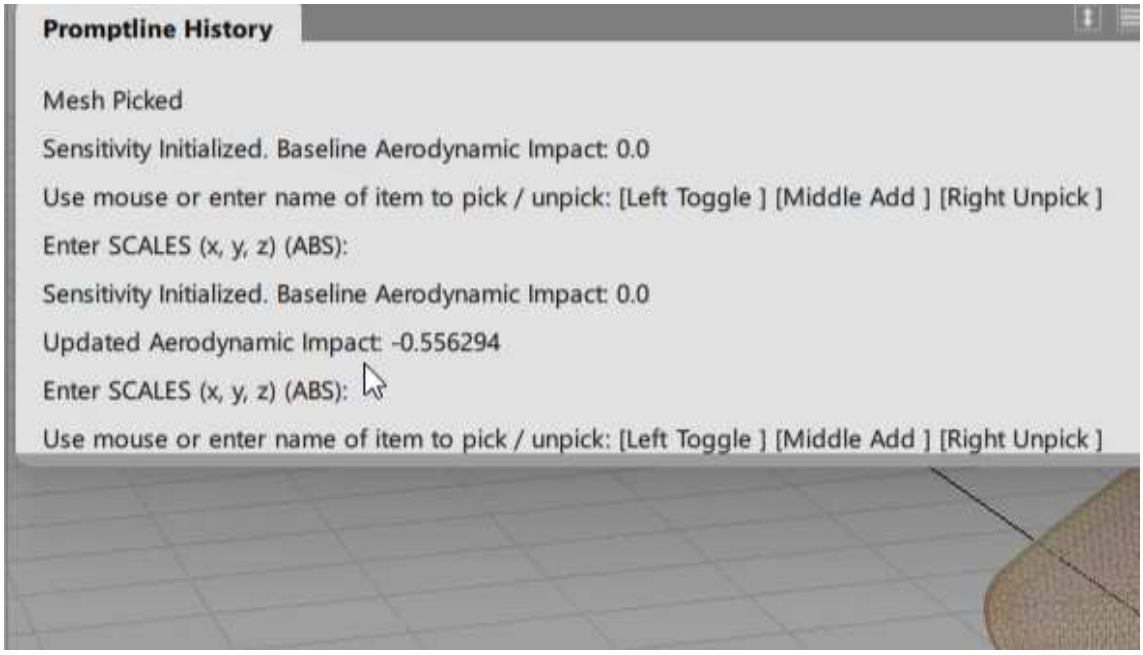


Figura 4.5: Stima della variazione del coefficiente aerodinamico

Capitolo 5

Espansione Futura: visualizzazione dei risultati

Un passo naturale nello sviluppo del plugin riguarda la possibilità di visualizzare in Alias i risultati delle analisi CFD direttamente sulla superficie della mesh, sotto forma di mappe di colore, che in ambito ingegneristico è lo standard per comunicare in maniera immediata e visiva grandezze come pressione, velocità o distribuzione del coefficiente di attrito sulla superficie.

5.1 Shaders

Gli *shader* sono piccoli programmi eseguiti direttamente sulla scheda grafica (GPU), il cui compito è determinare l'aspetto visivo degli oggetti renderizzati in una scena 3D.

La tecnica più semplice per ottenere questo risultato sarebbe l'utilizzo di *shaders per-vertex*, cioè programmi grafici in grado di associare ad ogni vertice della mesh un colore calcolato a partire da un valore scalare (ad esempio la pressione). In ambiente OpenGL[1] o DirectX[2] questo si realizza facilmente tramite uno *shader di frammento* che interpola i valori tra i vertici.

[1]OpenGL (`https://www.opengl.org`) è un'API multipiattaforma e multilinguaggio per il rendering di grafica vettoriale 2D e 3D

[2]DirectX (`https://en.wikipedia.org/wiki/DirectX`) è un insieme di API sviluppate da Microsoft per la gestione di grafica, suono e input in applicazioni multimediali, in particolare nei giochi per Windows.

L'idea è quella di utilizzare colormap standard della visualizzazione scientifica, come:

- **Jet**: blu → verde → giallo → rosso, utile per grandezze positive che variano tra minimo e massimo;
- **Seismic**: blu per valori negativi, bianco per lo zero e rosso per valori positivi, adatta a grandezze con segno;
- **Viridis**: una colormap percettivamente uniforme dal viola al giallo, che garantisce leggibilità anche in stampa in scala di grigi.

Queste mappe permettono di tradurre valori numerici in gradienti cromatici facilmente interpretabili. Ad esempio, la distribuzione di pressione su un'ala può essere resa immediatamente leggibile colorando in blu le zone a bassa pressione e in rosso quelle ad alta pressione, facilitando l'analisi visiva dei risultati CFD direttamente dentro Alias.

5.2 Il problema

All'interno di Alias, il termine *shader* non indica un piccolo programma eseguito sulla GPU, ma un modello matematico che definisce come la luce interagisce con la superficie di un oggetto, determinandone l'aspetto visivo. Gli shader in Alias sono dunque responsabili delle proprietà ottiche dei materiali, quali riflessione, diffusione e trasparenza, e vengono utilizzati per simulare la resa visiva di un oggetto 3D in modo realistico o stilizzato, tramite per esempio i modelli d'ombreggiatura di Phong, Lambert, Blinn etc.[3]

Idealmente, si vorrebbe disporre di uno shader "per-vertex" che permetta di associare un colore specifico a ciascun nodo della mesh in base a un valore numerico (che sarebbe quindi il valore di pressione, velocità, drag o altri coefficienti derivanti dalla matrice di sensibilità).

[3]Per approfondire questo aspetto: `https://it.wikipedia.org/wiki/Ombreggiatura`

Purtroppo, le API di Alias non mettono a disposizione shader personalizzabili di questo tipo. La documentazione ufficiale[4] mostra come gli shader integrati consentano soltanto un numero limitato di parametri, come colore diffuso, trasparenza, riflessione, mappatura tramite texture, etc: non esiste alcun meccanismo diretto per associare ad ogni vertice della mesh un colore arbitrario calcolato a runtime.

In realtà, sono presenti degli shader nativi che utilizzano mappe di colore per enfatizzare caratteristiche geometriche come curvature o parametri di superficie (chiamati *diagnostic shaders* all'interno del software) solo che non sono esposti tramite API. In linea teorica, quindi, Alias sarebbe perfettamente in grado di gestire anche la visualizzazione scientifica per-vertex, ma la limitazione tecnica dell'SDK ne impedisce l'uso diretto.

Per questo motivo, il plugin deve adottare soluzioni alternative (*workaround*) per ottenere effetti simili, quantomeno finché un modello di shading analogo al per-vertex non venga esposto nelle API.

5.3 Workaround

Il primo approccio esplorato per visualizzare dati CFD su una mesh in Alias è stato l'utilizzo del formato FBX, standard proprietario di Autodesk che supporta nativamente a rappresentazione di mesh 3D con informazioni di colore *per-vertex*. L'idea era sfruttare un canale di import/export già previsto, senza sviluppare estensioni dedicate. A tal fine è stato creato in Blender[5] un file FBX contenente una mesh con colori associati a ciascun vertice, successivamente importato in Alias. Il tentativo è fallito: le informazioni di colore non vengono interpretate da

[4] `https://help.autodesk.com/view/ALIAS/2025/ENU/?guid=GUID-B49FFD5F-B11E-47D8-87DF-90F3AE6BF230`

[5] `https://www.blender.org/`, software open source per la modellazione 3D

Alias in fase di importazione. Questo conferma che Alias non gestisce il colore per-vertex, neppure tramite un formato che lo supporta.

La seconda soluzione, tuttora in fase di sviluppo, consiste in un **workaround basato sul mapping UV e sulle texture**. L'idea è di tradurre i valori CFD in un'immagine bidimensionale e applicarla come texture alla mesh, aggirando così la mancanza di supporto per-vertex. Il processo concettuale è:

1. calcolare i valori CFD da visualizzare (ad esempio pressione, coefficiente locale o sensibilità);
2. assegnare a ciascun vertice una coppia di coordinate (u, v) che ne determinano la posizione nel dominio bidimensionale della texture, in modo coerente con la topologia della superficie 3D;
3. trasformare i valori CFD in colori attraverso una colormap (ad esempio *jet* o *seismic*);
4. generare un'immagine raster in cui ogni pixel rappresenta un colore codificato dai valori CFD;
5. associare questa immagine alla mesh tramite le coordinate UV e visualizzarla in Alias mediante gli shader già presenti nel software.

Criticità. Questa procedura introduce tre principali difficoltà tecniche:

1. **Associazione valore-colore:** è necessario definire una funzione di mappatura che, dato un valore CFD, restituisca univocamente un colore nella colormap scelta;
2. **Parametrizzazione UV:** per passare da una superficie tridimensionale a un dominio bidimensionale è indispensabile costruire una mappa UV. Questo richiede un algoritmo di *unwrapping*, cioè di appiattimento della superficie su un piano, minimizzando distorsioni e garantendo che vertici vicini nello spazio 3D rimangano vicini anche nello spazio (u, v);
3. **Generazione della texture:** i dati CFD campionati sui vertici devono essere interpolati e rasterizzati, cioè trasformati in una gri-

glia di pixel, in un'immagine bidimensionale (ad esempio nei formati BMP o PNG), in modo che possano essere caricati come texture da Alias.

Tra questi, la costruzione della mappa UV è il passaggio più complesso: non esiste una parametrizzazione unica per una superficie arbitraria, e la qualità della visualizzazione dipende fortemente dalla scelta della proiezione e dal controllo delle distorsioni introdotte.

5.3.1 Color mapping

La prima fase è la conversione dei valori CFD in colori. Per questo è stata implementata una funzione di *remapping* su un intervallo normalizzato, e due mappe cromatiche di esempio: una *seismic* (blu → bianco → rosso) e una *jet* (blu → ciano → verde → giallo → rosso).

```
float remapToRange (float value, float oldMin, float oldMax,
                    float newMin, float newMax) {
    return newMin + (value - oldMin) * (newMax - newMin) /
     ↪  (oldMax - oldMin);
}

RGB valueToColorSeismic(float value, float minval, float maxval)
 ↪  {
    float t = remapToRange(value, minval, maxval, -1.0f, 1.0f);
    uint8_t r, g, b;
    if (t < 0) {
        // valori negativi: blu -> bianco
        r = g = static_cast<uint8_t>((1.0f + t) * 255);
        b = 255;
    } else {
        // valori positivi: bianco -> rosso
        r = 255;
        g = b = static_cast<uint8_t>((1.0f - t) * 255);
    }
    return RGB{ r, g, b};
```

```
}

RGB valueToColorJet(float value, float minValue, float maxValue) {
    float t = remapToRange(value, minValue, maxValue, 0.0f, 1.0f);
    float r = std::clamp(1.5f - std::abs(4.0f * t - 3.0f), 0.0f, 1.0f);
    float g = std::clamp(1.5f - std::abs(4.0f * t - 2.0f), 0.0f, 1.0f);
    float b = std::clamp(1.5f - std::abs(4.0f * t - 1.0f), 0.0f, 1.0f);
    return RGB{ static_cast<uint8_t>(r * 255),
                static_cast<uint8_t>(g * 255),
                static_cast<uint8_t>(b * 255) };
}
```

5.3.2 UV mapping

Per poter proiettare la texture sulla superficie, ogni vertice della mesh deve essere associato a una coordinata bidimensionale (u, v) nello spazio della texture. Sono stati implementati due approcci sperimentali:

- **Spherical mapping**: adatto a geometrie "globose", come ali o fusoliere. In questo metodo, ogni vertice $\mathbf{v} = (x, y, z)$ viene proiettato su una sfera circoscritta alla mesh. Le coordinate UV sono calcolate come: $u = \frac{\phi}{2\pi}, \quad v = \frac{\theta}{\pi}$, dove $\phi = \arctan 2(y, x)$ è l'angolo azimutale e $\theta = \arccos(z/r)$ è l'angolo polare, con $r = \|\mathbf{v}\|$. Questo metodo preserva ragionevolmente la continuità sulla superficie, ma può introdurre distorsioni vicino ai poli della sfera.;
- **Pixel (grid) mapping**: ogni vertice viene assegnato a una cella di una griglia quadrata di dimensione $\lceil\sqrt{N}\rceil \times \lceil\sqrt{N}\rceil$, dove N è il numero totale di vertici della mesh. In pratica, i vertici vengono ordinati secondo un criterio arbitrario (ad esempio l'indice nella

lista dei vertici) e mappati sulla griglia in ordine raster (da sinistra a destra e dall'alto in basso). Le coordinate UV sono quindi normalizzate tra 0 e 1:

$$u = \frac{i \bmod W}{W - 1}, \quad v = \frac{\lfloor i/W \rfloor}{H - 1},$$

dove i è l'indice del vertice, $W = H = \lceil \sqrt{N} \rceil$ sono le dimensioni della griglia. Questo metodo garantisce una copertura completa della texture, ma non tiene conto della geometria reale della superficie, quindi può generare distorsioni significative.

Confronto tra Spherical Mapping e Pixel Mapping

Il primo approccio considerato è stato lo **spherical mapping**, con cui si assegna ad ogni vertice le coordinate UV ricavate dalle corrispondenti coordinate sferiche (θ, ϕ). Il vantaggio è la semplicità: si ottiene una parametrizzazione continua e coerente, utile soprattutto per superfici "chiuse" come sfere, ellissoidi o forme affini. Tuttavia, in presenza di geometrie complesse o prive di una simmetria regolare, la distorsione risulta inevitabile e la texture non si distribuisce in maniera uniforme sulla superficie, invalidando così totalmente l'obiettivo iniziale di associare ad ogni vertice un'informazione di colore.

```
1  void applySphericalMapping(Obj_Data& mesh) {
2      std::vector<float>& vertices = mesh.AccessVertices();
3      std::vector<float>& uvs = mesh.AccessUVs();
4      uvs.resize(mesh.NumberOfVertices() * 2);
5      for (size_t i = 0; i < vertices.size(); i += 3) {
6          float x = vertices[i], y = vertices[i+1], z =
           ↪  vertices[i+2];
7          float theta = std::atan2(z, x);
8          float r = std::sqrt(x*x + y*y + z*z);
9          float phi = std::acos(y / r);
10         size_t idx = i / 3;
11         uvs[2*idx]     = (theta + PI_F) / (2.0f * PI_F);
```

```
12          uvs[2*idx + 1] = phi / PI_F;
13      }
14  }
```

Per ovviare a queste difficoltà, è stato sperimentato un secondo approccio, informalmente chiamato **pixel mapping**. L'idea iniziale era di associare direttamente ogni vertice della mesh a un pixel univoco di una texture quadrata, così che ad ogni vertice corrispondesse esattamente un colore. In questo modo la speranza era che la geometria diventasse "trasparente" rispetto alla visualizzazione, e che fosse sufficiente mantenere l'associazione uno-a-uno *vertice* $\rightarrow$ *pixel*.

```
1  void applyPixelMapping(Obj_Data& mesh) {
2          std::vector<float>& uvs = mesh.AccessUVs();
3
4          uvs.clear();
5          uvs.resize(mesh.NumberOfVertices() * 2);
6          int texture_size = static_cast<int>(
7          std::ceil(std::sqrt(mesh.NumberOfVertices())));
8          for (size_t i = 0; i < mesh.NumberOfVertices(); i++) {
9                  size_t u1 = i % texture_size;
10                 size_t v1 = i / texture_size; // integer
                   ↪  division
11                 auto u = uvs[2 * i] = static_cast<float>(u1 /
                   ↪  float(texture_size));                  // u
12                 auto v = uvs[2 * i + 1] = static_cast<float>(v1
                   ↪  / float(texture_size));        // v
13         }
14 }
```

Questa ipotesi si è però dimostrata errata. Il motivo è che le coordinate UV non servono solo a identificare un vertice, ma devono garantire la *continuità* della mappatura sulla superficie: i vertici vicini nello spazio 3D devono risultare vicini anche nello spazio UV, così che la texture appaia coerente e continua. Con il pixel mapping questo non avviene,

perché i vertici vengono disposti arbitrariamente in una griglia bidimensionale senza tenere conto della topologia della mesh. Alias, interpretando la texture come immagine continua, genera quindi interpolazioni incoerenti e la visualizzazione risulta spezzata o addirittura casuale. In definitiva:

- lo **spherical mapping** produce una mappatura coerente, ma introduce distorsioni significative per geometrie complesse;
- il **pixel mapping** preserva l'univocità dei dati per-vertex, ma non è adatto alla resa grafica poiché non mantiene la coerenza topologica della superficie.

Questa fase di *trial and error* è stata comunque utile per chiarire un aspetto fondamentale: la generazione di coordinate UV non può essere trattata come un semplice problema di indexing, ma richiede un vero e proprio processo di **unwrapping parametrico**, tipico della computer graphics. È quindi necessario sviluppare o integrare un metodo di mappatura UV più sofisticato per ottenere una visualizzazione corretta e fedele dei dati CFD sulla geometria, problema ancora non definitivamente risolto.

Nell'ultima sezione, 5.3.4 verrà esposta una possibile soluzione, ancora da affinare e totalmente da implementare. Di seguito, invece, la trattazione prosegue con la pipeline generale

5.3.3 Generazione della texture

Una volta determinati i colori dei vertici e le loro coordinate UV, il passo successivo è la costruzione di una vera e propria immagine bitmap. Per semplicità è stato scelto il formato BMP, in quanto privo di compressione e facile da generare: è sufficiente scrivere l'header e poi un array di pixel RGB.

```
1 #pragma pack(push, 1)
2 // pragma pack avoids compiler optimization and aligning
```

```
3  struct BMPFileHeader {
4      // File type always BM which is 0x4D42
5      uint16_t fileType{ 0x4D42 };
6      uint32_t fileSize{ 0 };
7      uint16_t reserved1{ 0 };
8      uint16_t reserved2{ 0 };
9      uint32_t offsetData{ 0 };
10 };
11 struct BMPInfoHeader {
12     uint32_t size{ 0 };
13     int32_t width{ 0 };
14     int32_t height{ 0 };
15     uint16_t planes{ 1 };
16     uint16_t bitCount{ 0 };
17     uint32_t compression{ 0 };
18     uint32_t sizeImage{ 0 };
19     int32_t xPelsPerMeter{ 0 };
20     int32_t yPelsPerMeter{ 0 };
21     uint32_t clrUsed{ 0 };
22     uint32_t clrImportant{ 0 };
23 };
24 #pragma pack(pop)
25 bool generateBitmapImage(const char* filename,
26                          int width, int height,
27                          std::vector<RGB>& pixels) {
28     int paddingAmount = ((4 - (width * 3) % 4) % 4);
29     int rowSize = width * 3 + paddingAmount;
30     int fileSize = 14 + 40 + rowSize * height;
31     BMPFileHeader fileHeader;
32     fileHeader.fileSize = fileSize;
33     fileHeader.offsetData = 14 + 40;
34     BMPInfoHeader infoHeader;
35     infoHeader.size = 40;
36     infoHeader.width = width;
37     infoHeader.height = height;
```

```
38      infoHeader.bitCount = 24;
39
40      std::ofstream file(filename, std::ios::binary);
41      file.write((char*)&fileHeader, sizeof(fileHeader));
42      file.write((char*)&infoHeader, sizeof(infoHeader));
43
44      const uint8_t paddingData[4] = {0,0,0,0};
45      for (int y = 0; y < height; y++) {
46          for (int x = 0; x < width; x++) {
47              const RGB& pixel = pixels[y * width + x];
48              file.write((char*)&pixel, 3);
49          }
50          file.write((char*)paddingData, paddingAmount);
51      }
52      return true;
53  }
```

A questo punto, si hanno disponibili la mesh con le relative coordinate parametriche u, v e la texture colorata. È quindi sufficiente assegnare lo shader alla mesh per visualizzare i risultati nella GUI.

5.3.4 Soluzione proposta per l'UV unwrapping

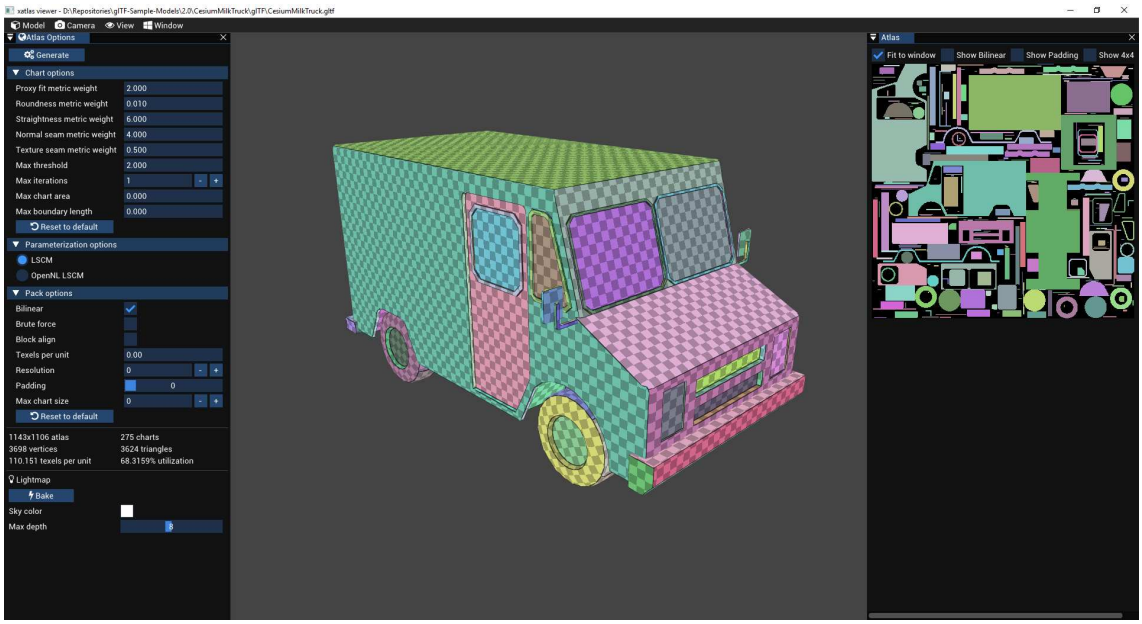


Figura 5.1: Esempio di unwrapping con xatlas, licenza `MIT`, autore: Jonathan Young

La tecnica descritta non è da considerarsi un contributo originale: un'implementazione analoga è già disponibile nella libreria open-source *Xatlas*[6], che si intende riutilizzare come base, evitando così di doverla reimplementare integralmente da zero.

L'idea di fondo è semplice: preservare (per quanto possibile) la vicinanza locale tra punti nello spazio 3D anche nello spazio 2D delle texture: si parte dalla geometria 3D, si spezza in regioni "appiattibili", e si cerca di minimizzare distorsioni quando si passa allo spazio 2D. Questo garantisce che la mappa di colori CFD, una volta "stesa" sulla superficie, risulti continua e interpretabile.

Costruzione del *grafo di vicinato* (kNN / r-ball). Si parte dal solo insieme dei vertici della mesh, per ognuno dei quali si identifica un

[6] `https://github.com/jpcy/xatlas`

piccolo insieme di vicini.

- **kNN** (k-Nearest Neighbors): per ciascun vertice si prendono i k punti più vicini in termini di distanza euclidea. "k" è un intero piccolo (tipicamente 8–16) che controlla quante relazioni locali si vogliono mantenere;
- **r-ball**: in alternativa (o in aggiunta), si considerano come vicini tutti i punti entro un certo raggio r dalla posizione del vertice. Il raggio si sceglie in funzione della scala media della mesh (per esempio 2–3 volte la lunghezza media degli spigoli).

Per rendere queste ricerche efficienti si usa il k-d tree, discusso nella sezione 3.3.7. Idealmente, quindi, si può ridurre l'overhead se si riescono a sfruttare gli stessi calcoli sia per la matrice di sensibilità che per questo obiettivo. Il risultato di questa fase è un *grafo di vicinato*, che cattura l'organizzazione locale della superficie in 3D.

Stima delle normali e del frame tangente locale (PCA). Per costruire una parametrizzazione UV stabile è necessario conoscere, in corrispondenza di ciascun vertice, l'orientamento locale della superficie. Questo si ottiene stimando:

- la **normale** locale, cioè la direzione perpendicolare alla superficie;
- un **frame tangente**, costituito da due direzioni ortogonali che giacciono sul piano tangente e completano una terna ortonormale con la normale.

Una tecnica standard per calcolare queste quantità è la **Principal Component Analysis** (PCA) locale. Il procedimento è il seguente:

1. Per ogni vertice v, si seleziona un intorno $N(v)$ costituito dai vertici adiacenti (ricavati dalla connettività del grafo della mesh).
2. Si costruisce la matrice di covarianza dei punti in $N(v)$, traslati rispetto a v:

$$C = \frac{1}{|N(v)|} \sum_{u \in N(v)} (u - v)(u - v)^\top.$$

3. Si calcolano gli autovalori e gli autovettori di C. Gli autovettori corrispondono alle direzioni principali di variazione dei punti vicini, ordinati per varianza decrescente;
4. La direzione associata all'autovalore minimo identifica la **normale locale n**, cioè la direzione lungo la quale i punti variano meno;
5. Le altre due direzioni ortogonali (relative agli autovalori maggiori) giacciono invece sul piano tangente e costituiscono il **frame tangente** $\{\mathbf{t}_1, \mathbf{t}_2\}$.

In questo modo ogni vertice è dotato di un sistema di riferimento locale ortonormale $\{\mathbf{t}_1, \mathbf{t}_2, \mathbf{n}\}$. Questa informazione è cruciale per due motivi: consente di "stendere" localmente la superficie in 2D, facilitando la costruzione di una mappa UV coerente, e riduce il rischio di introdurre distorsioni geometriche indesiderate (come twist o pieghe) nella parametrizzazione.

Segmentazione in *patch* mediante region growing. La superficie non viene mappata tutta in un colpo solo, ma viene divisa in regioni più semplici, dette **patch**. La **segmentazione** avviene con un algoritmo di **region growing** (crescita di regione):

- si sceglie un vertice come *seme* e si aggiungono progressivamente i suoi vicini coerenti, poi i vicini dei vicini e così via;
- la *coerenza* è verificata imponendo una **deviazione angolare** massima tra le normali (ad esempio, accetto un vicino se l'angolo tra le due normali è minore di una data soglia, e un limite di distanza (derivato dal grafo r-ball o kNN);
- quando non si trovano più vicini "compatibili", la patch è completa; si ripete con un nuovo seme sui vertici rimasti.

Così si ottengono patch che seguono la geometria, senza "saltare" tra parti lontane o separate da pieghe marcate. I contorni delle patch (i *seam*) saranno gli unici punti dove è accettabile una discontinuità UV controllata.

Proiezione delle patch su un piano (parametrizzazione locale). Per ogni patch, si costruisce una mappa 3D→2D stabile:

- si esegue una PCA sull'intera patch per ottenere due direzioni principali (assi) che definiscono un **piano di proiezione**;
- ogni punto della patch viene *proiettato* su quel piano, ottenendo coordinate 2D (u, v) locali;
- si normalizzano (u, v) della patch in un quadrato $[0, 1] \times [0, 1]$, rispettando le distanze locali per preservare la continuità.

In questo modo, si preserva l'idea di base: all'interno della patch, i vertici vicini in 3D devono restare vicini anche in UV.

Costruzione dell'*atlas* e *packing*. L'insieme delle patch mappate in 2D costituisce un **atlas**. Occorre ora "impacchettare" (**packing**) le patch nello stesso spazio UV globale $[0, 1]^2$ senza sovrapporle:

- ad ogni patch si assegna un riquadro dello spazio UV (anche una semplice griglia iniziale va bene);
- si scala e si trasla la patch dentro il suo riquadro, lasciando un piccolo margine (padding) per evitare sbavature nelle interpolazioni.

Il risultato sono coordinate UV *globali* continue all'interno di ciascuna patch, con discontinuità solo sui bordi (accettabili e controllate).

Dati CFD → colori → *texture*. L'obiettivo è trasformare i valori scalari CFD calcolati nei vertici della mesh (ad esempio pressione, sensibilità locale, coefficiente di attrito) in una mappa di colori 2D che Alias possa interpretare come *texture*. Il flusso logico è il seguente:

1. **Scalari → colori.** Si sceglie una funzione di mappatura cromatica (*colormap*), ad esempio *jet* o *seismic*, che associa a ciascun valore scalare s_i un colore RGB $c_i = (r_i, g_i, b_i)$. Questa fase produce quindi un *colore per vertice*;

2. **Mesh → spazio UV.** Ogni vertice della mesh possiede coordinate (u_i, v_i) nel dominio parametrico bidimensionale. In questo spazio i triangoli della mesh tridimensionale sono "stesi" come triangoli 2D con gli stessi vertici, ma descritti in UV;
3. **Rasterizzazione per triangoli.** Per ciascun triangolo (i, j, k), si considerano le coordinate UV dei suoi vertici e i rispettivi colori (c_i, c_j, c_k). Il triangolo viene rasterizzato in un'immagine bidimensionale: i pixel interni sono colorati per **interpolazione baricentrica**, che garantisce una variazione continua del colore all'interno della faccia e coerenza tra triangoli adiacenti;
4. **Generazione della texture.** Una volta rasterizzati tutti i triangoli, si ottiene un'immagine 2D completa che rappresenta la distribuzione spaziale dei valori CFD. Questa immagine è salvata in un formato standard (es. PNG, JPEG) e può essere applicata alla mesh in Alias come texture.

In alternativa, ma con minore accuratezza, è possibile usare la tecnica dello *splatting*: ogni vertice colora un piccolo disco di pixel attorno alla sua posizione UV, con intensità decrescente verso i bordi. Questa soluzione è più semplice da implementare, ma non garantisce la continuità tra triangoli adiacenti e può introdurre disomogeneità visive.

La rasterizzazione triangolare resta quindi la scelta preferibile, perché sfrutta direttamente la connettività della mesh e assicura che la texture risultante sia continua e priva di artefatti da campionamento sparso.

Filtri, continuità e gestione dei *seam*. I visualizzatori (Alias incluso) applicano spesso filtri (per es., bilineare) sulle texture. Per ridurre artefatti:

- si mantiene un minimo di padding tra patch nel packing;
- si ammorbidiscono i colori vicino ai seam (blending minimo) o si allineano, dove possibile, le direzioni delle patch su bordi adiacenti;
- si valuta "quanto spazio UV" assegno a zone più dettagliate, per non perdere risoluzione visiva in regioni critiche.

Integrazione nel plugin. Operativamente, il flusso nel plugin sarebbe:

1. costruzione del k-d tree e del grafo di vicinato (dal mesh loader);
2. stima di normali e frame tangente (PCA locale);
3. segmentazione in patch (region growing con soglie);
4. parametrizzazione per patch (proiezione e normalizzazione);
5. packing dell'atlas UV;
6. scelta del colormap e rasterizzazione in UV dei triangoli con i valori CFD/sensibilità;
7. esportazione della texture e associazione alla mesh tramite le UV appena calcolate.

Tutte le fasi sono deterministicamente ripetibili e sfruttano le UV e le texture, cioè ciò che Alias gestisce nativamente per lo shading.

Capitolo 6

Conclusioni

Il progetto descritto in questa tesi si colloca in un ambito fortemente trasversale, che ha richiesto di mettere in relazione metodi matematici, concetti fisici e problematiche grafiche. Le *Radial Basis Functions* (RBF) hanno fornito il supporto teorico e pratico per propagare le deformazioni geometriche; la *matrice di sensibilità* ha reso possibile collegare tali deformazioni a variazioni delle prestazioni aerodinamiche; infine, le tecniche di *UV unwrapping* e di texture mapping hanno permesso di rappresentare in modo visivamente intuitivo i risultati CFD.

Il risultato è un plugin per *Autodesk Alias* che integra strumenti e librerie originariamente indipendenti, trasformandoli in un sistema coerente e funzionale. Questa integrazione ha permesso di ottenere una soluzione industriale originale e utile, in grado di ridurre sensibilmente i tempi del ciclo di progettazione aerodinamica.

Il contributo principale è stato l'impiego delle RBF come meccanismo di morphing della mesh. Questo approccio ha reso possibile propagare le deformazioni locali dalla geometria CAD alla mesh CFD senza rigenerare la discretizzazione, preservando la corrispondenza tra i nodi della mesh e le righe della matrice di sensibilità, e consentendo di stimare in tempo reale la variazione di un coefficiente aerodinamico. Ne risulta uno strumento interattivo di supporto al progettista, che può così ottenere un feedback quantitativo immediato sulle proprie scelte di design.

Dal punto di vista implementativo, il lavoro ha richiesto la definizione di un'architettura modulare e la scelta accurata delle strutture dati, per garantire efficienza in operazioni fondamentali come il riallineamento dei vertici tramite k-d tree e il calcolo numerico delle variazioni prestazionali.

Un ulteriore contributo è stato lo studio delle limitazioni di Alias nella visualizzazione. Poiché non è disponibile un supporto nativo al colore per-vertex, è stato progettato un workaround basato su mapping UV e texture, che costituisce la base per futuri sviluppi e rappresenta un passo fondamentale per integrare manipolazione geometrica e rappresentazione qualitativa dei risultati CFD nello stesso ambiente.

Oltre al valore tecnico, il progetto ha avuto anche un grande valore formativo. L'attività ha richiesto di acquisire da zero competenze in ambiti tra loro molto distanti — dalle RBF alla sensibilità aerodinamica, fino ai problemi di UV unwrapping — e di integrarli in un unico sistema coerente. Ciò ha rappresentato un percorso di apprendimento intensivo, in cui la necessità di affrontare problemi reali ha guidato l'approfondimento teorico e lo sviluppo di soluzioni pratiche.

Guardando al futuro, lo strumento potrà evolvere verso un sistema di visualizzazione più avanzato, con gestione accurata delle mappe UV e colormaps percettivamente uniformi; potrà inoltre essere esteso alla valutazione di più coefficienti aerodinamici e a scenari fluidodinamici complessi, fino a integrarsi con metodi di ottimizzazione e tecniche di apprendimento automatico.

In conclusione, il lavoro mostra come l'unione di strumenti matematici, concetti fisici e soluzioni grafiche possa tradursi in un sistema innovativo, capace di accelerare in modo significativo il workflow di progettazione aerodinamica. Questo risultato rappresenta un primo passo verso una più stretta integrazione tra modellazione interattiva e simulazione scientifica, aprendo prospettive concrete per lo sviluppo di sistemi di ottimizzazione in tempo reale.

Bibliografia

[1] I. J. Schoenberg. «Spline Functions and the Problem of Graduation». In: *Proceedings of the National Academy of Sciences of the United States of America* 52.4 (1946). PubMed Central, pp. 947–950. DOI: 10.1073/pnas.52.4.947. URL: https://doi.org/10.1073/pnas.52.4.947.

[2] Les A. Piegl e Wayne Tiller. *The NURBS Book*. 2nd. Springer, 1997.

[3] Gianluca Magrì. «Ottimizzazione Aerodinamica di un velivolo mediante CAD Parametrico e Mesh Morphing». Università degli studi di Padova, 2023.

[4] Andrea Lopez. «Ottimizzazione di flussi esterni ed interni mediante metodi CFD adjoint e Mesh Morphing». Tesi di laurea mag. Tor Vergata, 2020.

[5] Filippo Ricci. «Ottimizzazione aerodinamica di una MotoGP mediante CFD e Mesh Morphing». Tor Vergata, 2023.

[6] Martin D. Buhmann. «Radial Basis Functions: Theory and Implementations: Preface». In: *Radial Basis Functions: Theory and Implementations*. 2003. URL: https://api.semanticscholar.org/CorpusID:122445786.

[7] *Eigen C++ template library for linear algebra*. URL: http://eigen.tuxfamily.org.

[8] *Nanoflann: A C++11 header-only library for K-d Trees / Nearest-Neighbor search*. URL: https://github.com/jlblancoc/nanoflann.

[9] Jon Louis Bentley. «Multidimensional binary search trees used for associative searching». In: *Communications of the ACM* 18.9 (1975), pp. 509–517. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007.

[10] Ingo Wald e Vlastimil Havran. «On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)». In: *Proceedings of the Eurographics Symposium on Rendering*. 2006, pp. 275–286. URL: https://doi.org/10.2312/EGSR/EGSR06/275-286.

[11] Russell A. Brown. «Building a Balanced k-d Tree in $O(kn \log n)$ Time». In: *Journal of Computer Graphics Techniques (JCGT)* 4.1 (mar. 2015), pp. 50–68. ISSN: 2331-7418. URL: http://jcgt.org/published/0004/01/03/.

[12] Jerome H. Friedman, Jon L. Bentley e Raphael A. Finkel. «An Algorithm for Finding Best Matches in Logarithmic Expected Time». In: *ACM Transactions on Mathematical Software (TOMS)* 3.3 (1977), pp. 209–226. DOI: 10.1145/355744.355745. URL: https://doi.org/10.1145/355744.355745.

[13] *Sito ufficiale dell'azienda Rbf-Morph*. URL: http://rbf-morph.com.

[14] *xatlas: A small C++11 library for generating texture coordinates for lightmap baking*. Consultata per le metodologie principali di UV-unwrapping. URL: https://github.com/jpcy/xatlas.

[15] *Autodesk Alias SDK Documentation*. URL: https://help.autodesk.com/view/ALIAS/2025/ENU/.

[16] Ubaldo Cella. «Setup and Validation of High Fidelity Aeroelastic Analysis Methods Based on RBF Mesh Morphing». Tesi di dott. Tor Vergata, 2015.

[17] Marco Gozzi. «DLR-F6 design optimization by means of Radial Basis Functions mesh morphing». Tesi di laurea mag. Tor Vergata, 2013.